Self-Timed Field Programmable Gate Array Architectures

Robert Payne

Doctor of Philosophy
University of Edinburgh
1997

Abstract

Dynamic hardware systems exploit the in-system reconfigurability of Field Programmable Gate Arrays (FPGAs), but are currently limited by the delay properties of synchronous FPGA architectures. Synchronous circuits are difficult to manipulate dynamically, since this alters their internal delays. The speed-independent properties of self-timed circuits overcome this problem, thus allowing the full benefits of dynamic reconfiguration to be exploited. The general properties of self-timed systems, such as modularity, low power and data dependent delays also provide benefits to less dynamic FPGA systems as well.

This thesis introduces a model for self-timed FPGA architectures called STACC (Self-Timed Array of Configurable Cells). STACC architectures replace the global clock of an FPGA with an array of timing cells that provide local self-timed control to a region of logic blocks. STACC differs from previous self-timed FPGA architectures in that it does not disrupt the structure of the logic blocks.

The STACC model is used to produced a self-timed version of the Xilinx XC6200 FPGA. Example circuits for the self-timed XC6200 demonstrate the benefits of self-timing for implementing dynamic hardware systems. Evaluation of the architecture shows that the implementation overhead of the timing array is reasonable, and that the self-timed XC6200 has the potential to out-perform the synchronous XC6200 through use of data dependent delays.

Acknowledgements

I would like to thank Gordon Brebner for his advice, relaxed style of supervision and his patient hyphen-eradication in the drafts. Also, Iain Lindsay for feedback on the electronic engineering aspects of the work. Vinod Rebello and Robert Mullins provided various tips and advice concerning asynchronous circuit design. Thanks to Anne for proof-reading.

Finally, thanks to the various players of badminton and flingers of frisbees, residers in Liberton and brewers of coffee, bakers of cakes and office-mates who kept me as a reasonably well balanced and adjusted person during the course of writing this thesis.

Declaration

I declare that this thesis was composed by me and that the work contained therein is my own, except where stated otherwise in the text. Parts of the work have previously been presented in the following:

[i] R.E.Payne. Self-Timed FPGA Systems. In *Proceedings of the 5th International Workshop on Field Programmable Logic and Applications*, LNCS 975, September 1995.

[ii] R.E.Payne. Asynchronous FPGA Architectures. In *IEE Proceedings on Computers and Digital Techniques, Special Issue on Asynchronous Processors*, September 1996.

[iii] R.E.Payne. Self-Timed Reconfigurable Elements. In *Proceedings of the First U.K. Asynchronous Forum*, December 1996.

[iv] R.E.Payne. Run-time Parameterised Circuits for the Xilinx XC6200. To be published in *Proceedings of the 7th International Workshop on Field Programmable Logic and Applications*, September 1997.

(Robert Payne)

Table of Contents

Chapter 1

Introduction

1.1 Reconfigurability and Self-Timing

Field Programmable Gate Arrays (FPGAs) are a form of programmable logic; they are devices designed to implement a wide range of different logic circuits. The key property of programmable logics that differentiates them from custom hardware is their reconfigurability. Such devices cannot compete with a custom hardware implementation in terms of density or speed, but their reconfigurability allows hardware designs to be created and changed rapidly, thus reducing time-to-market and costs over custom hardware.

Traditionally, programmable logics have been configured in special programmers that are external to the host system. However, many current FPGAs have SRAM configuration memories, which can be programmed in-system. Thus, a configuration can be loaded into the FPGA and run, just like a software program, but with performance closer to that of dedicated hardware. Dynamic hardware systems attempt to exploit the software-like reconfigurability of SRAM FPGAs. For example, such systems can be used to implement circuits larger than the size of the FPGA, by swapping parts of the circuitry to and from the FPGA; or circuits can be customised for a particular problem on the fly.

However, the exploitation of reconfigurability in dynamic hardware systems is limited by the delay properties of the FPGA architecture. Changing the environment or shape of a circuit alters the delay properties of the circuit, which means it can fail to meet the global clock in synchronous systems. The starting point for this thesis is the proposal to utilise the speed-independence of self-timed circuits to allow the rapid manipulation of circuits in dynamic hardware systems. Such an approach promises to allow the full dynamic reconfigurability of FPGAs to be exploited.

Current synchronous-oriented FPGA architectures pose problems for the implementation of self-timed circuits. The assumptions made in self-timed communication protocols are often not maintained by synchronous FPGA architectures. The approach taken in this thesis is to develop self-timed FPGA architectures to overcome the problems with current FPGAs.

The thesis introduces a new model for self-timed FPGA architectures called STACC (Self-Timed Array of Configurable Cells). In STACC, the global clock of a synchronous FPGA architecture is replaced with an array of timing cells. These timing cells provide local timing control to regions of logic blocks, which are left unaltered from the original synchronous FPGA architecture. The clear split between timing cells and data cells (logic blocks) in STACC reflects the split in self-timed bundled-data protocols between control path and data path.

To demonstrate the viability of STACC, the STACC model is applied to the Xilinx XC6200 FPGA architecture. The Xilinx XC6200 was chosen since it is a recent architecture (first silicon in 1995), and that it includes features for the use of dynamic hardware. The self-timed XC6200 architecture is used to construct circuits parameterised at run-time, which demonstrate the benefits of self-timing for dynamic hardware systems.

1.2 Thesis Structure

The thesis consists of four main parts. The introductory chapters present background material on FPGAs and self-timed systems, and outline the potential benefits of self-timed FPGA systems. The second part of the thesis introduces the STACC model for self-timed FPGA architectures, and develops the circuit elements required for the construction of STACC architectures. The third part of the thesis concerns the application of the STACC model to the Xilinx XC6200 FPGA architecture. Finally, the thesis concludes with a summary of the main results and a discussion of possible future work.

Figure 1.1 summaries the structure of the thesis and shows the relationship between the chapters. Below, a chapter by chapter summary for each part of the thesis is given.

1.2.1 Introduction

Chapter 2 covers background material on FPGAs. A key part of this chapter is the discussion of dynamic hardware systems, virtual hardware, and run-time

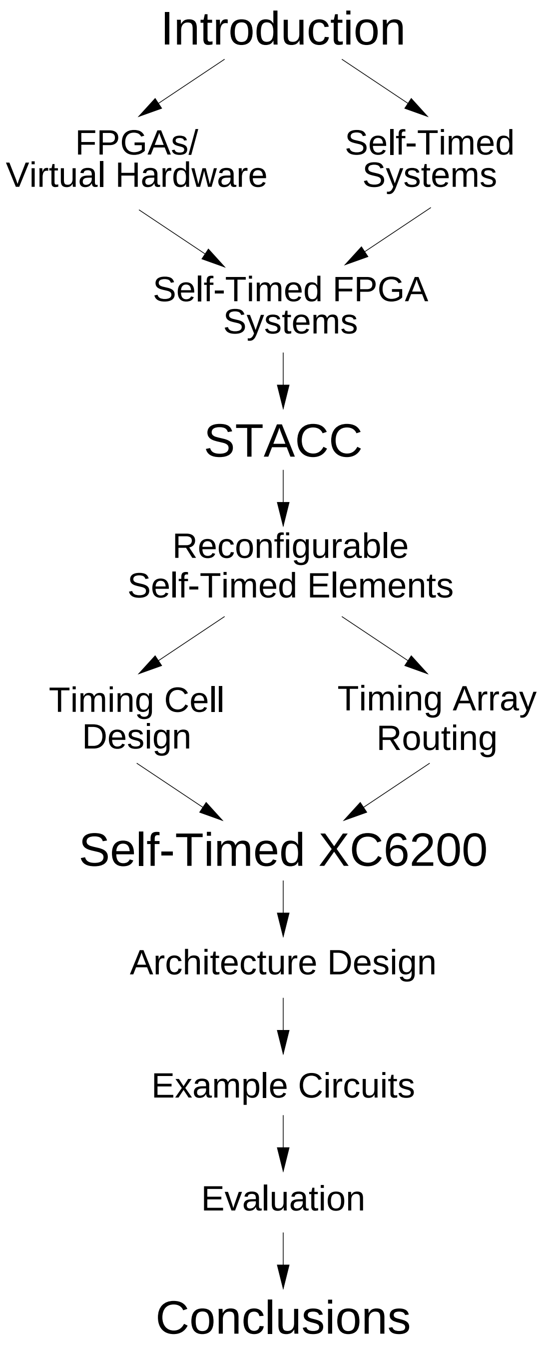


Figure 1.1: Thesis Structure

parameterised circuits. **Chapter 3** introduces self-timed systems and includes a detailed discussion of bundled-data systems.

Chapter 4 is central to the rest of the thesis. It considers the potential benefits and drawbacks of self-timing for FPGA systems in general, and dynamic hardware in particular. The chapter also reviews the current research on self-timed circuits using synchronous FPGAs and proposed self-timed FPGA architectures.

1.2.2 STACC

Chapter 5 introduces a model for self-timed FPGA architectures: STACC (Self-Timed Array of Configurable Cells). The STACC model involves replacing the global clock in a synchronous FPGA with an array of timing cells that provide local clock signals.

The next three chapters focus on the implementation of the timing array in STACC architectures. **Chapter 6** introduces several new self-timed elements, such as the Q-Merge/Select pair and the reconfigurable C-Muller gate, which are used as building blocks for the timing array. **Chapter 7** describes in detail the design of timing cells for STACC. The timing cell is developed from a basic reconfigurable C-Muller gate into a timing cell capable of selective communication and arbitration. **Chapter 8** concerns routing structures for handshaking signals in the STACC timing array. These structures are based on another development of the reconfigurable C-Muller gate: the handshaking crossbar.

1.2.3 A Self-Timed XC6200

Chapters 9 to 11 concern the application of the STACC model to a contemporary FPGA architecture: the Xilinx XC6200. **Chapter 9** introduces the current XC6200 architecture, concentrating on the features useful for dynamic hardware, and then presents the design of a self-timed XC6200 using the STACC model. **Chapter 10** contains a case study of the use of the self-timed XC6200 for implementing dynamic hardware systems. The example circuits are run-time parameterised circuits for finite field operations with application to Reed-Solomon error correction. Finally, **Chapter 11** compares the self-timed XC6200 relative to the synchronous XC6200, considering its delay performance and the implementation overhead of the timing array.

1.2.4 Conclusions

Chapter 12 summarises the main results of the work. The chapter ends with a discussion of possible directions for future research.

1.3 Contributions

This thesis makes original contributions in a number of areas. The main contributions are listed below. These points are expanded upon in Chapter 12, which summarises the conclusions of the thesis.

Self-timed Dynamic Hardware: A key contribution of the thesis is the identification of the benefits of self-timed circuits for implementing dynamic hardware systems. Previous work on self-timed circuits for FPGAs have concentrated only on the prototyping of self-timed systems.

STACC: is a new model for creating self-timed reconfigurable architectures. Unlike previous self-timed FPGA architectures (MONTAGE, PGA-STC), STACC-based architectures do not alter the structure of the logic blocks for self-timing.

Self-timed Reconfigurable Elements: The thesis introduces a number of new self-timed elements, potentially of wider use in self-timed design: the Q-Merge/Select Pair, the reconfigurable C-Muller gate, the STACC timing cell and handshaking crossbars.

Self-timed XC6200: Using the STACC model, this thesis presents the design, simulation and evaluation of a new self-timed FPGA architecture based on the Xilinx XC6200.

Run-Time Parameterised Circuits: The circuits developed for run-time parameterisation on the self-timed XC6200 are of note, due to the hierarchy of parameterisation, and the benefits arising from self-timing. Some of the design techniques developed, such as the abstract block size, are applicable to XC6200 designs in general.

Current Sensing Completion Detection (CSCD): The thesis provides insight into the potential benefits of the CSCD delay scheme and proposes the use of CSCD for meta-stability resolution.

Chapter 2

Field Programmable Gate Arrays

2.1 Background

FPGAs are the successors to earlier forms of programmable logic such as PLAs and PALs. The initial need for programmable logics has been to integrate a number of SSI (Small Scale Integration) or MSI (Medium Scale Integration) parts on a single chip, without the expense or time of building a custom part. A key difference of FPGAs from these previous forms of programmable logic is the size of circuit that can be implemented on one device. Whilst earlier programmable logic devices could replace a small number of SSI or MSI parts, FPGAs can implement VLSI parts (over 10,000 gate equivalents) within a single programmable device. The implementation of larger circuits within FPGAs has necessitated a change in architecture. Earlier programmable logic devices, such as PLAs (see Figure 2.1), implemented circuits as a two-level logic function, i.e., a boolean sum of products. Such devices had two planes of logic: an AND plane that produced the product terms from the inputs and an OR-plane that summed the products to produce the outputs.

However, the use of two-level logic functions becomes cumbersome for larger circuits, since the size of each plane increases more rapidly than the complexity of the circuit. A solution is to factorise the two-level logic functions and implement the circuits using multi-level logic functions. To implement multi-level logic functions, FPGA architects have drawn on the design of Mask Programmable Gate Arrays (MPGAs). MPGAs are a semi-custom implementation style for ASICs (Application Specific Integrated Circuits). MPGAs save cost on expensive custom masks, by having a fixed set of masks defining a collection of basic building block gates (such as NAND gates), and then use a few custom masks to define the routing between them. The key difference between MPGAs and FPGAs is the method of configuration. In MPGAs, the config-

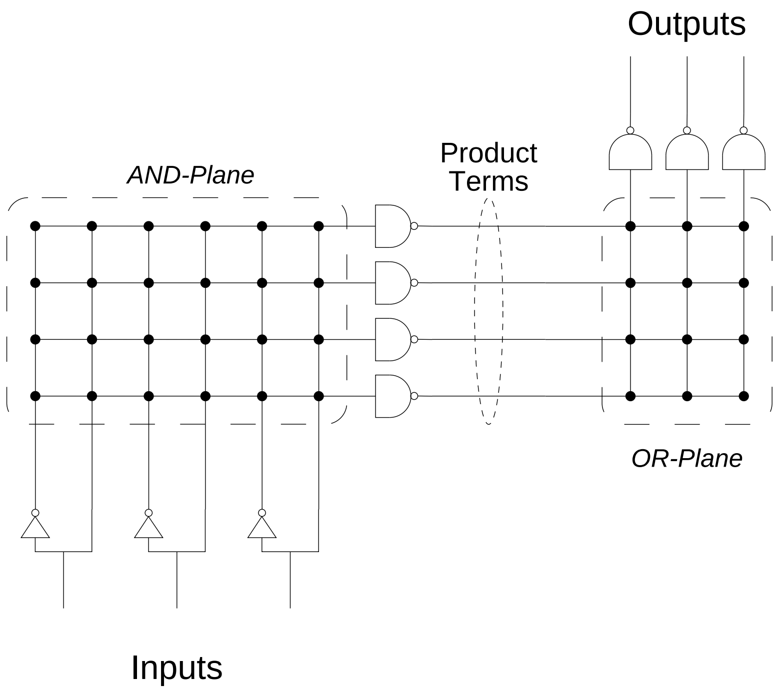


Figure 2.1: PLA Architecture

uration is defined by the routing masks whilst, in FPGAs, it is defined by a configuration memory. MPGAs and FPGAs are sufficiently related that FPGA designs can be migrated directly to MPGAs with similar architectures. For example, CLA is an MPGA version of the Algotronix CAL architecture [65], and the Xilinx HARDWIRE architecture is an MPGA version of the XC4000 FPGA [124].

2.2 Elements of an FPGA Architecture

Figure 2.2 shows an idealised FPGA architecture. The architecture consists of an array of function blocks. Each function block can be configured to implement a variety of basic gates and a basic memory element, such as a D-type flip-flop. The function blocks are wired together to form a circuit using reconfigurable interconnect. Switchboxes connect a function block's inputs and outputs to the interconnect. Typically, architectures provide separate interconnect for routing local and non-local signals. At the edge of the array, special input/output switchboxes are provided to connect to external signals. The circuit implemented by the FPGA is determined by values stored in the FPGA's configuration memory. The configuration memory determines the functions implemented by the function blocks and the routing implemented

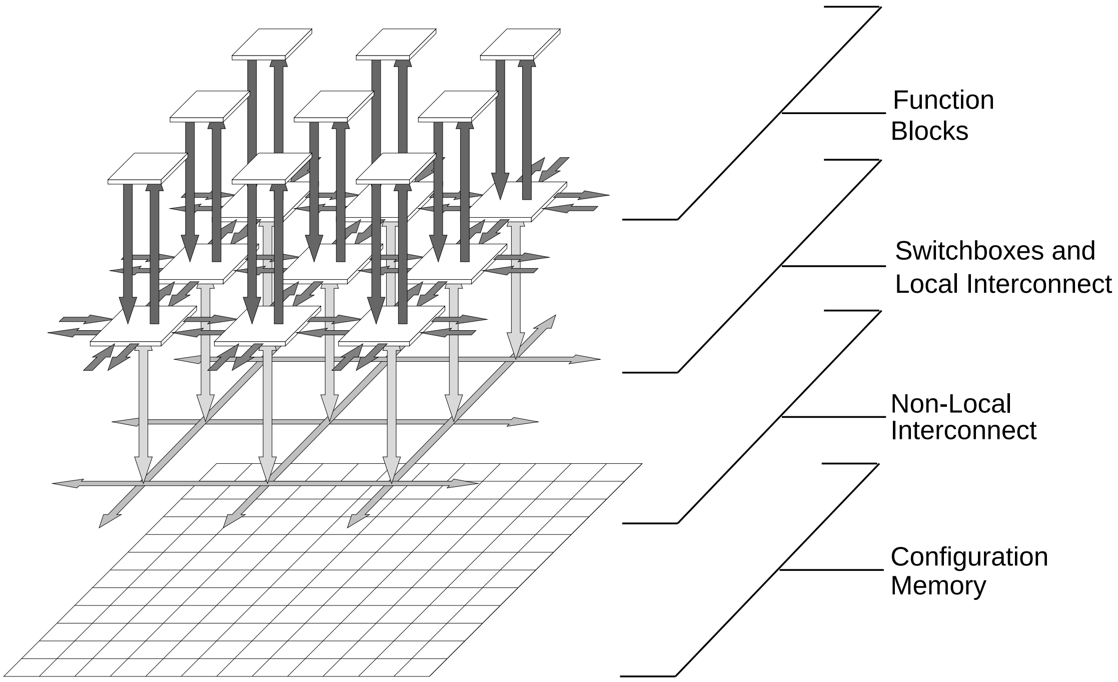


Figure 2.2: Elements of an FPGA Architecture

by the switchboxes.

Currently, a large number of FPGA architectures are available commercially, with little agreement on a common design. The following sections examine the decisions made in current FPGA architectures. They discuss the design of the function blocks (Section 2.3), reconfigurable interconnect (Section 2.4) and configuration memory (Section 2.5). Some desirable properties of how these elements are assembled to create an FPGA architecture are discussed in Section 2.6. The use of FPGAs in dynamic hardware applications is discussed in Section 2.7.

2.3 Function Blocks

Current architectures use a wide variety of function blocks. A basic requirement of the function blocks is that any logic function can be constructed given a sufficient supply of them. Two-input NAND or NOR gates are sufficient for this purpose, however most FPGA architectures choose to use function blocks that can implement any boolean function of between 2-5 boolean variables.

Current FPGA architectures use five basic styles of function block: primitive gates, LUTs (Look-Up Tables), multiplexors, PALs (Programmable Array Logic) and CAMs (Content Addressable Memories). These styles are described below. In addition, most architectures embellish the basic choice of function block with additional features to improve the implementation of certain functions, such as dedicated carry logic for adders and counters.

Primitive Gate Function Blocks

The simplest function block possible for an FPGA is to supply a primitive gate such as two-input NAND or NOR gate. All other logic functions can be built given a sufficient supply of these gates. This approach has been adopted successfully in 'sea of gates' MPGAs. The advantage of using primitive gates is that the function block is easy to design and small, so can be replicated in large numbers. The main drawback of using such basic elements for FPGAs is that it requires a large amount of slow reconfigurable interconnect. Most FPGA architectures use more complex cells, with mostly fixed internal, routing for speed. At the time of writing, the only FPGA architecture to use a primitive gate function block has been the GEC-Plessey ERA architecture [33, 45].

Look-Up Table Function Blocks

Figure 2.3 shows the design of a Look-Up Table (LUT) function block. The multiplexor is used to select an output value from a configuration memory. Essentially, each function block acts as a small ROM (Read Only Memory), whose output is selected by the input signals. To provide one boolean function of N input variables requires 2^N configuration bits. Architectures such as the Xilinx XC4000 [124] and AT&T ORCA [9] allow the LUT to be split into sub-LUTs to provide more functions, but of fewer variables. For example, the Xilinx XC4000 [124] architecture allows a function block to provide two boolean functions of four variables each or one boolean function of five variables.

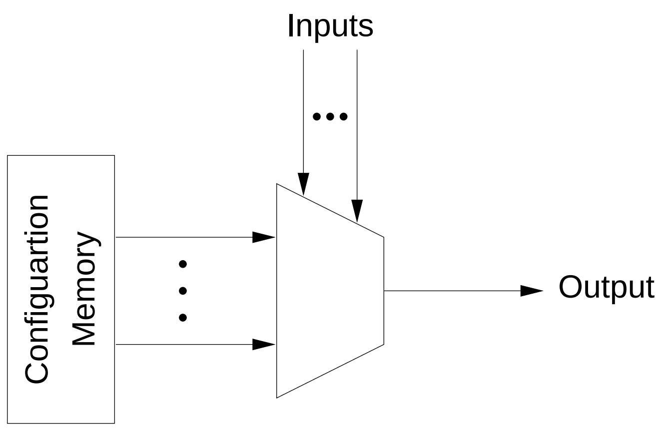


Figure 2.3: Look-Up Table Function Block

Multiplexor Based Function Blocks

Figure 2.4 shows the function block of a multiplexor based FPGA. A multiplexor with N select inputs is capable of implementing all boolean functions of $N+1$ input variables. In contrast to a LUT based function block, a multiplexor based function block is not directly configured by a configuration memory. Instead, the function is determined by configuring routing to the select and data inputs of the multiplexor. Configuring this routing becomes costly for large multiplexors, so typically FPGA architectures, such as the Algotronix CAL1024 [3] and Actel Act1000 [2] use small multiplexors with only one or two select inputs.

Many FPGA architectures add additional logic to the multiplexor. The Act1000 FPGA [2] includes a two input NOR gate on one of the select inputs of the multiplexor. The Cypress pASIC380 [24] includes a number of wide input AND gates that can be tapped separately as outputs. Architectures like the Atmel AT6000 [10] add so much additional logic to the multiplexor that they can be regarded as being a separate type of 'complex gate' function block.

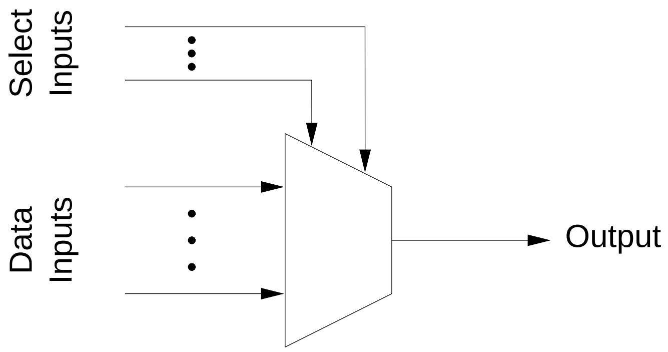


Figure 2.4: Multiplexor Function Block

PAL Based Function Blocks

PAL based architectures can be considered as an evolutionary step from older two-level programmable logics to current FPGA architectures. FPGAs like the Altera MAX series [4], can be considered as 'Mega-PALs' where a small number of traditional PALs are placed on the same chip with limited reconfigurable interconnect to join them.

Figure 2.5 shows a PAL (Programmable Array Logic) based function block. Each output is implemented as a boolean sum of products. Product terms are generated by wide input AND gates and then summed together using a fixed OR gate; in Figure 2.5, De Morgan's rule allows NAND gates to be used in place of AND and OR gates. The functions implementable by a PAL based function block are limited by the number of available product terms. Certain functions, in particular XOR like functions, use a large number of product terms, so map poorly to PAL based function blocks. However, most logic functions need considerably less product terms than the worst case. Furthermore, PAL based designs lead to very dense implementation when used with fuse based configuration memories (see Section 2.5).

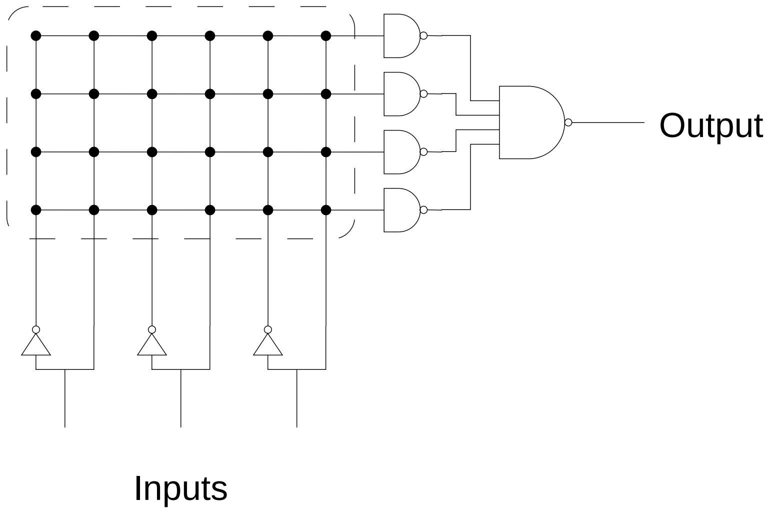


Figure 2.5: PAL Function Block

Content Addressable Memory Function Block

Figure 2.6 shows a novel Content Addressable Memory (CAM) function block that has been designed by Oxford Parallel [110]. The CAM based function block can act in two modes. In the first mode, the cell can act as a Random Access Memory (RAM). A word of the RAM is selected with the word select lines. Data can be read or written using the data out and data in signals in conjunction with a read/write signal. When reading in this mode, the CAM cell acts in a similar way to a LUT based function block. In the second mode, the cell compares the contents of the data input with the RAM contents. If the data matches, then the appropriate match line goes high. In this mode, the CAM cell acts in a similar way to the product terms in a PAL based function unit, which only go high when they match a particular data input pattern. The advantage of the CAM based design over the other function blocks is the variety of structures it can implement. As well as being able to implement LUT and PAL structures, it can easily implement dense RAM and CAMs in the FPGA.

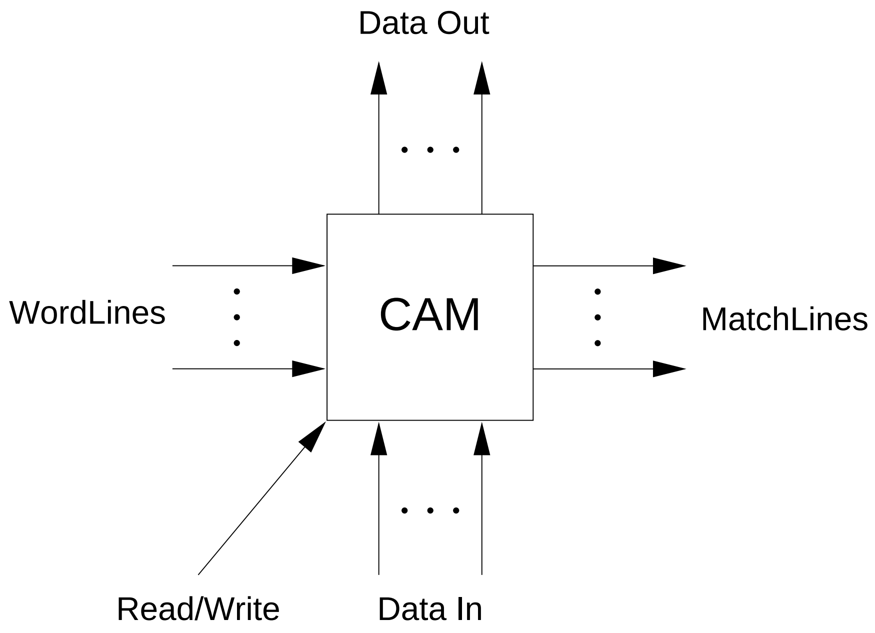


Figure 2.6: CAM based Function Block

2.3.1 Memory Elements

Most FPGA architectures provide a dedicated memory element as part of the function block. Exceptions are the Actel Act1000 [2], and GEC-Plessey ERA [33, 45] architectures, which require the memory elements to be implemented from the basic function block. Typically, architectures have D-type registers or latches as the basic memory elements. In architectures with more than one output from the function block, usually only some of the outputs are connected to memory elements. In all architectures, the configuration can choose to bypass the memory element if the output of the logic block is not registered.

Most FPGAs are poor at implementing dense memory structures such as RAM. The Xilinx XC4000 [124] architecture overcomes this by allowing the configuration memory of the LUT to be used alternatively as a block of SRAM. The Oxford Parallel FPGA [110] also allows dense memory structures to be implemented using its CAM based function blocks.

2.4 Reconfigurable Interconnect

Current FPGAs use a wide range of interconnect architectures. The architectures can be classified according to the type of basic interconnection resource provided and how the basic interconnect elements are joined together to form the routing network. Most architectures also provide special routing resources for signals, such as reset and clock signals.

2.4.1 Interconnect Elements

FPGAs use a number of basic elements to interconnect their routing resources. These are described below.

Bidirectional Interconnect

In FPGAs using fuse based configuration (see Section 2.5), two wire segments can be joined simply by blowing a fuse. The connections are bidirectional, since signals can flow in either direction through the fuse. Crossbar switches can be constructed from a grid of overlap wires with fuses at the intersections. In FPGAs using SRAM configuration memories, bidirectional connections require pass transistors controlled by configuration bits. These interconnect elements require more silicon area than in fuse based architectures. Also, pass transistors have a higher electrical resistance than fuses, so bidirectional buffering is required. The direction of buffering must be determined by additional configuration bits, as in the Xilinx XC3000 and XC4000 architectures [124].

Unidirectional Interconnect

To avoid implementing bidirectional buffers, many FPGAs with SRAM configuration memories, such as the Algotronix CAL [3], constrain wires to having one fixed driver. This constrains the direction of signal flow along wires within the architecture to be unidirectional. Unidirectional signalling leads to less flexibility in the use of the routing resources, but single direction drivers avoid the possibility of driver conflicts where multiple drivers drive signals to opposing values.

Open Collector and Tristate Interconnect

Architectures, such as the XC4000 [124] and Oxford Parallel [110], include interconnect with open collector drivers. The advantage of open collector drivers is that wide input OR and AND gates can be constructed using wired logic. Indeed, older two-level programmable logics used wired logic to implement the bulk of their circuitry. The Xilinx XC3000 and XC4000 [124] architectures include tristate drivers, to allow bus-like structures to be built on the FPGA. When using tristate drivers, the designer has the responsibility of avoiding driver conflicts, arising from more than one tristate driver driving the signal at any one time.

2.4.2 Routing Networks

Current FPGA architectures provide two basic styles of interconnection network based on separate routing channels or integrated routing and function blocks. The choice of interconnect network is strongly influenced by whether signals are unidirectional or bidirectional. Unidirectional architectures favour the point to point links used in integrated architectures, whilst bidirectional architectures favour the use of separate routing channels.

Separate Routing Channels

FPGAs that use a channel based interconnect network (such as Actel [2], Altera [4] and Xilinx [124]), emulate the style of routing in many MPGAs. In channel based routing topologies, routing channels run horizontally and vertically through the architecture. At the intersection of the channels, switchboxes allow signals to move from horizontal to vertical channels. Other switchboxes allow the function blocks to connect to the routing channels. Channel based routers usually supply a variety of different lengths of wire segments in the routing channel for local and non-local routing. Typically, many short segments are provided for local interconnect, with fewer medium length and full length wire segments for non-local routing.

Integrated Routing and Logic

Architectures such as the Algotronix CAL [3] have integrated routing and logic blocks. There is no clear differentiation between function block and routing as there is with channel based routing. Typically, routing in these architectures is based on a nearest neighbour mesh. A problem with nearest neighbour meshes is the lack of non-local routing. Earlier architectures, such as the CAL1024 [3], chose not to supply any at all. Later architectures such as the Xilinx XC6200 [123] architecture include a hierarchy of non-local routing structures.

Hybrid Routing Schemes

The TRIPTYCH [55] and Atmel AT6000 [10] architectures combine aspects of both channel and integrated routing schemes. In TRIPTYCH, most connections use local point-to-point links along a nearest diagonal neighbour mesh.

Additionally, function blocks are connected to vertical routing channels for non-local signals. In the AT6000 architecture, vertical and horizontal buses are provided in addition to the nearest neighbour mesh.

2.4.3 Clock Routing

Currently, all commercial FPGA architectures are geared towards the design of synchronous systems. Most architectures have dedicated clock routing to allow the distribution of a global clock signal with minimum skew across the FPGA. Typically, a choice of several global clock signals is given to allow two and four phase synchronous clocking schemes to be implemented. Most architectures also allow clocks to be driven from the local routing, but this option is rarely used, as it lacks the low skew characteristics of the dedicated clock routing. Furthermore, a number of different clocks operating asynchronously to each other creates interfacing problems.

The alternative to implementing synchronous circuits on FPGAs is to use self-timed or asynchronous circuits, which do not require a global clock signal. Implementing self-timed circuits on current FPGAs is discussed in Chapter 4, and the design of dedicated FPGA architectures for implementing self-timed circuits is the subject of the rest of the thesis.

2.4.4 Input/Output Interface

At the edge of an FPGA, special blocks are needed to allow the input and output of signals to and from the FPGA. Most commercial FPGA architectures try to limit the number of input and output pins to save on packaging costs. Several schemes are used to minimise the number of pins.

The simplest is only to provide some of the inputs/outputs from the array as pins. Another alternative is to share input/output pins with the configuration interface, since in most FPGA architectures, once the FPGA is programmed these pins are not used. However, this requires additional circuitry to implement the switching from configuration mode to input/output mode. An approach adopted in the Algotronix CAL1024 [3] architecture is to use ternary signalling. Ternary signalling uses special circuitry to share an input and an output on the same pin. The illegal state when the output is being driven to a different value from the input is detected by special circuitry, and the value being received can be reconstructed from knowledge of the signal being driven on the line. FPGAs using addressable SRAM as configuration memory, such as

CAL [3], provide another option for input/output. Data can be read and written into the array using the SRAM interface. Potentially, using this interface could alleviate the need for other input/output pins altogether, but currently no FPGAs have adopted this approach.

A conflicting objective to minimising the number of pins is to allow the array to be naturally extended, which requires all input and outputs to be provided as pins. This is extremely costly; of all the FPGAs discussed, only CAL [3] manages this, and this is through its use of ternary signalling. Even providing all the necessary extensions, an array of FPGAs cannot be treated as a uniform array due to the magnitude of off-chip delays.

One method of providing a large array of FPGA chips, which is as close as possible to one uniform array of function blocks is WSI (Wafer Scale Integration). An example of this approach is the Teramac [108] system built by HP Laboratories, which integrates several FPGAs on one MCM (Multi-Chip Module). Also Isshiki et al [64] have built a MCM with 12 Xilinx XC3042 chips and an Aptix FPID (Field Programmable Interconnect Device) as additional interconnect.

2.5 Configuration Memory

Current FPGAs use two basic types of configuration memories: fuse based and SRAM based. The key difference between these two types of configuration memory is that SRAM based designs have the the potential to be reconfigured in-system, whilst fuse based designs need to be programmed externally to the system in a special programmer. However, fuses can be implemented more compactly, which leads to a different style of architecture from SRAM based ones, where the configuration memory is relatively expensive to implement. These two types of configuration memory are discussed below.

2.5.1 Fuse based Configuration Memory

Fuse based FPGAs use the same configuration method as the older two-level forms of programmable logic, such as PALs. In fuse based FPGAs, the configuration is determined by the pattern of blown and unblown fuses. The principal advantage of fuse based configuration over SRAM, is that fuses are efficient to implement on silicon. At the silicon level, a fuse can be created at the crossing point of two wires. Special processing steps are used to make a thin layer of semi-conductor between the two wires that can be made non-conducting (blown) by the application of high voltages. Many manufacturers [24, 4] use anti-fuses instead. An anti-fuse is the opposite of a fuse: it is non-conducting until a high voltage is applied to it. Other types of fuses allow the configuration to be erased electrically or using ultra-violet light. Compact arrays of fuses can be created using a grid of wires as in PLA-type architectures.

The configuration of fuse based FPGAs requires a special programmer to generate the high voltages required to blow the fuses. The configuration interface of fuse based FPGA allows the programmer to apply these voltages to each fuse individually in the architecture by having each fuse at the crossing point of a row and column to which the appropriate voltage can be applied.

2.5.2 Static RAM Configuration Memory

A major development of some FPGA architectures from older programmable logics is the use of SRAM for the configuration memory. The benefit of SRAM is that the configuration of the FPGA can be altered in-system, rather than needing the device to be removed from the system to a special programmer. A drawback of using SRAM is that it requires more silicon area since a SRAM cell implementation requires several transistors and associated wiring, whilst a fuse can be created simply at the crossing point of two wires. Another drawback is that the configuration is volatile, so needs to be reloaded every time the system is powered up.

Two methods are used for configuring SRAM FPGAs. One option is to configure the SRAM serially, as in the Xilinx XC4000 architecture [124] by providing all the configuration data to configure the FPGA in sequence. In serial access SRAM FPGAs, the configuration memory of the FPGA is implemented as a very long shift register. The use of the term SRAM (Static Random Access Memory) by such FPGA manufacturers in this case is a misnomer, since clearly the access is not random. The advantage of a serial interface is that there is no need to supply address signals to the FPGA, to indicate which part of the SRAM is to be programmed. This leads to a saving in silicon area and pins required for the configuration interface.

The alternative to serial access is a normal addressable SRAM interface as used in the Algotronix [3] and Atmel [10] FPGA architectures. The advantage of a true random access interface is that parts of the chip can be selectively reconfigured. In addition, the interface can be used to read back results from the array. The Xilinx XC6200 architecture [23] extends the basic SRAM access by allowing the use of 'wild cards' in the address given to the SRAM, so that arrays of repeated circuit elements in the array can be configured in one operation.

2.6 Repetition, Hierarchy and Symmetry

The previous sections have concentrated on the basic elements and structures used in FPGA architectures. This section focusses on some desirable higher level properties of an FPGA architecture, which arise from the way elements are put together. These features are particularly important to FPGA design tools which are used to generate the configuration of the FPGA.

A fundamental requirement of an FPGA architecture is that the basic cell of the architecture can be replicated to form a regular array. A result of this repetition is that a design placed at one point in the array can generally be transposed to another position without change. All current FPGAs are based on a rectangular repeat pattern, but other shapes that give a regular covering of the silicon could be used, such as hexagons or equilateral triangles. Some FPGAs build the architecture as a hierarchy of elements, rather than using a simple repeating structure. The top level of the hierarchy is repeated across the silicon. For instance the Altera Flex 8000 [4] architecture consists of Logic Element Blocks grouped together into Logic Array Blocks.

Another desirable property of an FPGA architecture is symmetry, which allows designs to be rotated and flipped. This property is desirable for placement and routing software, since it allows more options for placement and routing of designs. Many architectures display reflective symmetry in one direction (such as the Cypress [24]) and some in two directions (such as TRIPTYCH [55]). Some also have rotational symmetry, for example, the Oxford Parallel [110] architecture has rotational symmetry of order two, whilst the AT6000 [10] has rotational symmetry of order four. Another useful property, related to symmetry, is for the architecture to provide function blocks with interchangeable inputs. For example, LUT based function blocks allow any permutation of inputs to be used. This flexibility allows routing software more options in how to route signals to a function block.

Many architectures add irregular features to their function blocks to improve the implementation of certain functions. For example Xilinx [124] and Altera [4] provide special carry generators to improve implementation of counters and adders. With the emergence of the PREP [1] benchmarks as an industrial standard, such features may increase with manufacturers striving to improve their benchmark performance. However, less regular features are more difficult for synthesis tools to use. For example the XC4000 [124] carry logic can only be used by a special XBLOX generator program rather than the standard placement and routing software. This is a similar observation to that made by RISC processor designers: complex features are difficult for compilers to use effectively.

2.7 Dynamic Hardware Systems

A key difference of many FPGA architectures over older forms of programmable logic is the use of SRAM for the configuration memory. Rather than needing a special programmer to be reconfigured, such devices can be reconfigured in-system. The reconfigurability of SRAM FPGAs is more akin to software than hardware: a configuration file can be loaded into the FPGA's configuration memory and then run in a similar way to software. In other words, SRAM FPGAs can act as 'soft-hardware'. Systems that exploit the reconfigurability of SRAM FPGAs are often referred to as *dynamic hardware systems*.

Current dynamic hardware systems can be classified into two groups dependent on the system architecture. The first class of dynamic hardware system consists of an FPGA and microprocessor with the FPGA being used as a co-processor. The second class of dynamic hardware system consists of a large array of FPGAs connected by a routing network, similar in structure to current parallel computers.

2.7.1 Co-processor Dynamic Hardware Systems

Co-processor dynamic hardware systems consist of a closely coupled system of FPGA and microprocessor. Computation is shared between the microprocessor and the FPGA. In such systems, the FPGA is configured with a set of instructions adapted to the application problem. Good candidates for migration from software to the FPGA are inner loops of program code. Work at UMIST [85] is examining the automatic and user guided migration of target code from software to hardware.

For co-processor systems to show significant performance gains, the performance gain of the FPGA must outweigh the additional communication cost of going to the co-processor. Hence, it is preferable in co-processor dynamic hardware systems to place the microprocessor and FPGA on the same local bus, as in the HARP board [93] and EVC [21]. Both Page [94] and DeHon [28] argue that the natural progression is for FPGA co-processors to be integrated on the same piece of silicon, very much as floating point units have migrated from being co-processors to being an integrated part of microprocessors.

2.7.2 Large Array Dynamic Hardware Systems

Large array dynamic hardware systems resemble parallel systems in many ways, and in particular massively parallel systems, such as the Connection Machine [58] or DAP [59] . Both consist of a large array of processing elements joined by a routing network, and are loosely coupled to a host computer that deals with input/output and reconfiguration. The main difference between massively parallel computers and large array dynamic hardware systems is that the dynamic hardware systems do not have a global instruction issue. Instead, in dynamic hardware systems, the program is hardwired by the configuration. Another difference is that large array dynamic hardware systems have a more flexible interconnect architecture, but again it is fixed by the configuration.

Both massively parallel computers and large array dynamic hardware systems have been targeted at similar application domains. Large array dynamic hardware systems have demonstrated superior performance on several problems compared to far more expensive parallel systems. The Splash and Splash2 systems [6] have shown considerable speed-up on the searching of genetic databases [98] and the travelling salesman problem [49]. The PAM architecture [12] has recorded the fastest implementation of the RSA cryptography algorithm [79]. The SPACE machine [89] has been used for road traffic simulations. Cellular automata applications have been implemented by a number of researchers [90, 65, 63]. However, the lack of dedicated floating point units in large array dynamic hardware systems make them a poor match for many high performance computing applications.

2.7.3 Models of Dynamic Hardware Systems

In the previous section, current dynamic hardware systems were classified broadly into two groups based on the system architecture. Some machines do not fit well into either group. For example, systems such as ArMen [99] and CM2X [101] combine aspects of the co-processor and parallel system approaches. Both these machines are parallel computers where the processors have FPGA co-processors attached.

Boloski et al [13] and Guccione [53] have suggested alternative methods of classifying dynamic hardware systems, to the one adopted here. Both authors introduce a broad definition of reconfigurability. For example, an ALU in a normal processor can be regarded as a reconfigurable unit, with a small number of reconfiguration bits that define which arithmetic function that it performs.

Boloski et al [13] use this idea to compare SIMD (Single Instruction Multiple Data) parallel computers and FPGAs. They consider FPGAs as a class of ELIW (Extremely Long Instruction Word) architecture. Since the instruction is so long, systems either load the instruction infrequently, as in FPGAs, or shorten the instruction by sending the same instructions to all parts of the array, as in SIMD parallel computers. Boloski et al argue for a hybrid architecture, which consists of a local configuration as in a FPGA, along with a global instruction issue as in a SIMD array.

Guccione [53] uses the concept of reconfigurable units to propose a classification scheme similar in spirit to Flynn's classification of parallel systems. He chooses to classify dynamic hardware systems by whether they have large or small reconfigurable units, and whether they include on-board memory. However, the difference between what constitutes a large or small reconfigurable unit is unclear, and does not lead to a clear classification scheme.

2.7.4 Virtual Hardware

Dynamic hardware systems introduce a new resource into computing architectures, namely reconfigurable hardware. Like other resources within a computer, reconfigurable hardware is limited and often the need for more reconfigurable hardware than is available will arise. *Virtual hardware systems* attempt to give the illusion of more reconfigurable hardware than is actually available. Conceptually, virtual hardware is analogous to virtual memory. Virtual memory emulates a much larger memory space by swapping pages of memory between a much smaller physical memory space and a backing store. In virtual hardware, a much larger area of reconfigurable hardware is emulated by swapping configuration data between reconfigurable hardware and a backing store.

The term 'virtual hardware' is used by many authors to refer to any system composed of reconfigurable FPGAs. In this thesis, virtual hardware is used to refer to the class of system which involves swapping parts of a circuits to and from the FPGA during the operation of the circuit. The term dynamic hardware system is used to refer to the more general class of system that utilises the reconfigurability of SRAM based FPGAs.

A key consideration in the design of virtual hardware systems is the time taken to reconfigure the array. If the reconfiguration takes longer than the time to perform the operations in software then there is no performance gain. Several researchers, such as Ling [72] and DeHon [28], have suggested minimising configuration times by having FPGA architectures with more than one configuration memory, so that one configuration can be changed whilst another configuration is in use. The benefits of such an approach are debatable, since configuration memory represents a large proportion of the silicon real estate of an FPGA architecture. It may be as effective to provide more reconfigurable logic, rather than provide extra configuration memories with additional circuitry to switch between different configuration memories.

2.7.5 Current Virtual Hardware Systems

Most implementations of virtual hardware to date have been limited to a fixed pattern of swapping circuits to and from known locations on the reconfigurable hardware. The advantage of such systems is that, as the pattern of swapping is fixed and the location of the circuits known, then each configuration of the reconfigurable hardware can be simulated to check for correct operation. However, this approach requires algorithms that have clear boundaries between different stages.

Several such applications have been in the domain of neural networks. The RRANN system [32, 31] divided the circuit between the different phases of the neural network, so different circuits were loaded for the back propagation and forward propagation stages of the algorithm. Lysaght et al [76, 75] adopt a different approach to implementing neural networks by swapping in different circuits for each layer of the neural network.

More general virtual hardware systems have been built where the pattern of swapping and location of circuits within the virtual hardware is less limited. French et al [34] proposed a co-processor dynamic hardware system where the FPGA is used as instruction cache: instructions not already in the FPGA are loaded in when required. Writhlin and Hutchings [122] have implemented such a scheme called DISC (Dynamic Instruction Set Computer). Instructions may be dynamically loaded at any position in a one-dimensional space on the FPGA. An interesting feature of this system is that the microprocessor has been removed from the system and instead a small processor is configured on the FPGA itself. A dynamic paging system has been developed by Brebner and Gray [18] for a fax decoding circuit. In this system, pages of the fax decoding circuit are loaded on demand when the circuit indicates a page fault. Work by Brebner [16, 17] has also examined virtual hardware operating systems.

Simulation work by Ling [72] has investigated the idea of performing the whole of a computation using virtual hardware. This complicates the architecture as circuits can communicate between pages of the virtual hardware. A mechanism must be provided for transferring results between two hardware pages, when potentially one of the hardware pages is not loaded into the reconfigurable hardware.

2.7.6 Run-Time Parameterised Circuits

Parameterisation of circuits is now a common part of many FPGA design tools. For instance, many graphical design tools allow the definition of a bit sliced component of arbitrary width, such as a N-bit wide adder. More comprehensive parameterisation can often be achieved through use of a Hardware Design Language, for example, VHDL [62] or Ruby [74].

However, the parameterisation of these designs is fixed at compile time. Often, it would be useful for an application to specify the parameterisation of a hardware accelerator at run time rather than compile time. For example, a constant multiplier circuit is quicker and more compact than a general multiplier circuit. If a large number of data values are to be multiplied by a constant value, then it is beneficial to configure an instance of a parameterised circuit at run time, rather than using a general purpose multiplier circuit.

Similar concepts are being explored in the context of partial evaluation in functional programming languages by Singh et al [107]. An important property of run-time parameterised circuits is that they have the potential to outperform a dedicated hardware implementation. This arises since the dedicated hardware is optimised for solving a class of problems, whilst a run-time parameterised circuit is optimised to solve a particular instance of a problem.

Run-time parameterised circuits share many of the same problems with virtual hardware systems. The central issue is that the generation of the configuration must be done quickly, otherwise the speed-up of using virtual hardware is lost in the time taken to generate the configuration and then reconfigure the FPGA. The central challenge to designers of these systems is ensuring that the configuration works as expected without having time to use complex place and route algorithms and delay analysis algorithms that are used in design tools.

2.8 Summary

This chapter has discussed the basic elements of an FPGA architecture, and described the wide variety of architectures currently available. The latter part of the chapter described how the in-system reconfigurability of FPGAs with SRAM configuration memories is being used in dynamic hardware systems. Virtual hardware systems and run-time parameterised circuits were identified as classes of dynamic hardware system which present particularly challenging problems to researchers, since the FPGA configuration is often determined on the fly.

Chapter 3

Self-Timed Systems

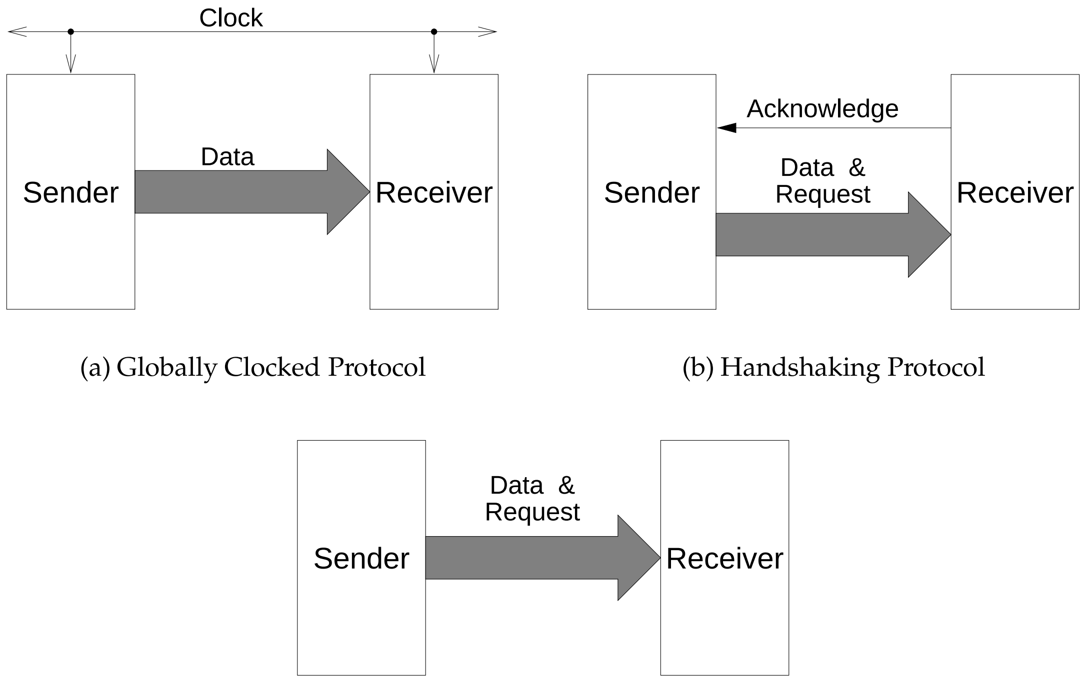
3.1 Background

Today, most digital systems are built synchronously. The synchronous approach has not always been dominant. Machines such as ORDVAC (1951) and MU5 (1969) were built asynchronously [35]. The synchronous design style has come to dominate for a variety of reasons, principally to do with ease of design and ease of testing. In the meantime, asynchronous design has been relegated to a niche academic discipline. However, the problems of clock distribution and power dissipation as clock frequencies increase are bringing the future dominance of synchronous systems into doubt. These problems coupled with improved asynchronous design styles, have led to a resurgence of interest in asynchronous design from academia over the last few years. Industry is now taking an interest in asynchronous design with Phillips [116], Intel [120] and Sun [109] funding research.

3.2 Synchrony, Asynchrony and Self-Timing

The terms synchronous and asynchronous are used in a variety of different ways and different contexts in both hardware and software communities. In this section, the definitions used in this thesis are introduced. At the systems level, a *synchronous system* is one where all the communication actions are synchronised, typically by a global clock signal. In contrast, each communication within an *asynchronous system* is independent of any other; there is no global synchronisation of the whole system. However, within an asynchronous system, individual communications may be synchronised locally depending on the form of communication protocol used.

Figure 3.1(a) illustrates the communication protocol used in synchronous



(a) Globally Clocked Protocol

(b) Handshaking Protocol

(c) Unsynchronised Communication

Figure 3.1: Communication Protocols

systems. Data is transfered from sender to receiver on the tick of a global clock signal. Every communication in the system is synchronised by the global clock.

Figure 3.1(b) shows another form of synchronised communication: a *handshaking* protocol. The sender transmits data to the receiver together with an implicit or explicit request signal. Having received the request signal and the data, the receiver indicates receipt using the acknowledge signal. This *request/acknowledge handshake* synchronises the communication; the sender cannot send more data until it has received an acknowledge from the receiver. Though the communication is synchronised, a system built using handshaking protocols is asynchronous, since each communication is independent.

An unsynchronised communication protocol is shown in Figure 3.1(c). The only difference from the handshaking protocol is the lack of acknowledge signal, but as a result the communication is not synchronised: the sender sends data when it wants, without waiting for the receiver.

These communication protocols make a variety of different assumptions about the delays in the communication channel between sender and receiver. The globally clocked communication protocol assumes the data from the sender will be valid at the receiver before the next clock tick. This requirement places a rigid limit on the delay of all modules in the system; every module's delay must always be less than the clock period. For the asynchronous communication protocol of Figure 3.1(c), the assumption is that the receiver must always be able to process communications as fast as the sender can produce them, since there is no way for the receiver to regulate the incoming data flow. Thus, in the globally clocked protocol, the speed of the module is determined by the clock, and in the unsynchronised protocol, the speed of a module is determined by the rate of communications from the sender.

In contrast to the other two protocols, the handshaking protocol of Figure 3.1(b) is *speed-independent* : the protocol places constraints on the ordering of signals, but not on the time taken to produce or consume the communications by a module. Systems composed using handshaking protocols are known as *self-timed* ; each part of the system proceeds at its own pace, rather than having its pace determined externally by a clock signal or by the arrival of data. Most modern asynchronous systems are self-timed and the terms are often used synonymously in the literature.

Though self-timed protocols make no assumptions about the time taken to produce or consume communications, different protocols do make different assumptions concerning the delays within the communication channel. *Delay-sensitive* protocols make an assumption of similar wiring delays, so that an ordering in time of signals at the sender will arrive in the same order at the receiver. Seitz [104] refers to a region within a system where this assumption can be made as an *equi-potential region* . No such assumption is made by *delay-insensitive* protocols: such protocols assume that any signal can be arbitrarily delayed, so an ordering in time of signals at the sender may arrive in a different order at the receiver.

3.3 A Comparison of Self-Timed and Synchronous Systems

The differences in the communication protocols described above lead to a wide range of differences in the systems composed using them. Below the benefits and drawbacks of self-timed systems relative to synchronous systems are considered. In the comparison, systems that use unsynchronised communication (i.e. asynchronous but not self-timed) are excluded. Such systems have properties of both self-timed and synchronous systems, but do not possess some of the key advantages of self-timed systems such as robustness and modularity.

Asynchronous communication comes into its own when communicating over a distance, so that waiting for the acknowledge signal to return, or distributing a clock incurs a significant performance penalty. Asynchronous communication protocols are considered in [113].

3.3.1 Advantages of Self-Timed Systems

Modularity

Self-timed modules may easily be composed into a working system. For delay-insensitive modules, the modules can simply be joined together and the composition will work. For delay-sensitive modules, the modules must be connected using similar wiring, so that signals are not re-ordered. The modularity of self-timed systems allows incremental change; a module can be replaced by one with similar functionality, but different performance and the composition will still work. In contrast, synchronous systems are less modular, since the designer is always concerned with whether each module and the wiring will meet the global clock constraint.

Robustness

Synchronous systems are generally dependent on an external clock source for timing. Localised effects, such as temperature and operating voltage, that affect the delays within the system do not affect the external clock source. Hence, a synchronous system may fail at higher temperatures, or at low voltages when the internal delays exceed the clock period.

Self-timed systems are more robust since they rely on internal sources of timing for their operation, so all the delays in the system are scaled by the environmental effects. As a result, self-timed systems have the potential for increased performance over synchronous systems where the worst case environmental conditions have to be assumed.

Delay-insensitive self-timed systems are particularly robust with respect to environmental effects, since there are no assumptions made about signal delays except for isochronic forks (see Section 3.4.3). Such systems are robust to large changes in temperature and voltage [83]. Also, delay-insensitive circuits can easily be migrated to different implementation technologies (such as Gallium Arsenide [112]).

No Global Clock

In synchronous systems, much effort has to be spent in ensuring that the global clock signal goes reliably to all parts of the circuit. Avoiding clock skew becomes very costly in terms of power, and clock routing area as a synchronous system becomes larger and the clock frequency becomes higher. It is reported that the DEC 21064 Alpha [30] uses six levels of clock buffering in distributing the clock signal, and has a 30W power requirement. In self-timed circuits, the timing information only has to be consistent locally rather than globally, which is much simpler to ensure.

Average Case Performance

Self-timed modules generate their own timing signals, so can pass on results at their own speed, instead of having to wait for the next global clock change. This allows self-timed circuits to utilise the average case performance of the circuit rather than being limited by the worse case performance as synchronous systems are.

This fact also allows the use of area efficient implementations, which are impractical in synchronous systems because the worse case delay is so much larger than the average case delay. For example, with an N-bit adder, a circuit where the carry ripples up the circuit can be used, as the worst case will only occur on average once in 2^N additions (see [44] for a more detailed analysis of self-timed adders).

Low Power

In CMOS circuits, the static power consumption is almost zero. However, the constant changing of a global clock signal in a synchronous system causes transitions to be continually passing through the system, resulting in dynamic power consumption even when no valid data is passing through. In self-timed systems, transitions are only produced when data is passing through the system; there is no dynamic power consumption when the system is idle.

Meta-stability

All synchronous systems have the potential for failure when interfacing with their external environment. The problem arises when sampling an external signal that is not synchronised to the global clock. The signal may be changing when sampled, which can lead to the sampling flip-flop entering a *meta-stable* state: a state where it is unable to decide whether the sample was a logic one or zero. This meta-stable state will eventually resolve to logic one or logic zero, but takes an arbitrary amount of time to do so.

Circuits can be built that detect the end of a meta-stable state, but the signal that indicates the end of meta-stable state is itself not synchronised to the clock, so the problem repeats itself. However, in a self-timed system, there is no clock to synchronise to, so the self-timed system can wait for the resolution of the meta-stable state before continuing. Seitz [104] discusses meta-stability in more detail.

Flow Control

Many systems need to regulate the flow of data between different parts of the system, for example, to prevent a buffer over filling or to share a resource between different processes. Within synchronous systems, flow control has to be implemented using additional circuitry which mimics the handshaking protocols of self-timed systems. In self-timed systems, flow control comes for free as part of the communication protocol.

3.3.2 Disadvantages of Self-Timed Systems

Extra Circuitry

Self-timed circuitry is generally larger than equivalent synchronous circuitry as they must generate their own timing signals. The degree of overhead depends on the communication protocol used. Bundled-data protocols (see Section 3.4.2) only require two extra signals, a request and an acknowledge, so are efficient to implement if the data bundle is relatively large. Simple delay-insensitive protocols (see Section 3.4.3) use two wires to transmit one data bit so have greater area overheads.

Testing

Testing self-timed systems for defects is harder than in the synchronous case. In a synchronous system there is a clear transition from one global state to another on the tick of the global clock. The clock can be used to single step the system so that there is time to ascertain the current system state. Additionally, if a manufacturing defect results in increased internal delays within a synchronous system, then the system can simply be run at a slower clock speed. In a delay-sensitive self-timed system, the part may be useless, unless some method for altering the internal timing delays is provided.

Difficult to Design

Asynchronous circuits have a reputation for being difficult to design. Partly this is because they lack the clean change of global state that occurs in synchronous circuits, and partly it is due to the possibility of race and hazard conditions. This is one of the most frequently quoted reasons why most circuits are built in a synchronous fashion.

The actual difficulty of asynchronous design depends on the style chosen. Many early asynchronous design styles [114, 71] used unsynchronised communication protocols rather than self-timed protocols. Such circuits were dependent on the fact that the feedback loop to establish the new state of the circuit was faster than the time taken for a new input to arrive (known as *fundamental* and *input/output mode* circuits). Such design styles needed careful design to avoid races and hazards.

Most modern asynchronous circuits use self-timed protocols. Bundled-data systems (see Section 3.4.2) are similar to synchronous systems in that the same data path is used. Once the different timing style has been understood, they are no more difficult to design then synchronous systems. Delay-insensitive protocols are more difficult to design because the designer must still be conscious of possible races and hazards. However, synthesis tools (such as [82, 119]) have been developed that automate this translation process, eliminating race and hazard conditions from designs. On a larger scale, self-timed circuits become easier to design due to other properties such as modularity and robustness.

3.4 Self-Timed Communication Protocols

So far, only the general nature of self-timed communication protocols has been discussed. This section examines in detail the wide range of self-timed communication protocols that are used. The protocols can be classified at several levels. At the lowest level, protocols have to attach significance to the transitions of individual signals within the protocol. Once the significance of individual transitions is determined, the protocols need to encode the data and request signals. The encoding of the data and request signals determine whether the protocol is delay-sensitive or delay-insensitive. Finally, the protocols need to determine the conventions for the ordering of the handshaking signals. These issues are discussed below. Hauck [54] gives further background on self-timed communication protocols.

3.4.1 Signalling Conventions

In self-timed protocols, transitions on wires are often more important than the actual level. Most self-timed communication protocols come in several varieties that differ in which transitions are significant within the protocol. Protocols where all transitions on wires are significant are termed *two-phase*, *transition*, *event* or *non-return-to-zero (NRZ)* protocols. The term two-phase protocols refers to the two transitions used in the request/acknowledge handshake. Protocols where transitions in one direction only (logic zero to one or logic one to zero) are significant are termed *return-to-zero (RZ)* or *four-phase* protocols. Four-phase signalling requires an additional redundant return-to-zero or *recovery* transition before the next significant transition, thus four transitions are involved in the request/acknowledge handshake. In four-phase signalling, the choice of which transition is significant can be different for different wires within a system.

Both two-phase and four-phase signalling have advantages. Two-phase signalling involves fewer transitions on a wire so can be faster and use less power than four-phase signalling, which requires two redundant transitions. However, the redundant transitions in four-phase protocols can be overlapped with the computation phase within a module. Hence, two-phase signalling is only significantly faster when communication time for the handshake is longer than computation within the module. An advantage of four-phase signalling is that the circuitry is often much simpler as the signals are in the same state at the end of the handshake, which can lead to performance advantages merely because of smaller circuit size [95]. Another option is to use the redundant transitions in a four-phase protocol to indicate another sequential event, for example, acknowledge and release signals in four-phase arbiters can be combined.

An alternative to two-phase and four-phase signalling is *single-track handshaking* [115]. Single-track handshake circuits use the same wire for request and acknowledge signals. The sender pulls the wire high to indicate a request and the receiver pulls it back low to indicate an acknowledge. A benefit of this approach is that it only uses a single wire with two phases and returns the signal to its initial state. However, complex driving and detection circuits are required for the single-track approach.

3.4.2 Bundled-Data Protocols

In *bundled-data* protocols (Figure 3.2), a data bundle is passed to the receiver together with a separate request signal. A transition on the request signal signifies valid data on the data bundle. Hence the request must only be asserted after the data is valid; this is known as the *bundling constraint*. Bundled-data protocols are delay-sensitive, since even though the request wire is asserted after the data at the sender, arbitrary delays in wires could result in this ordering not holding at the receiver.

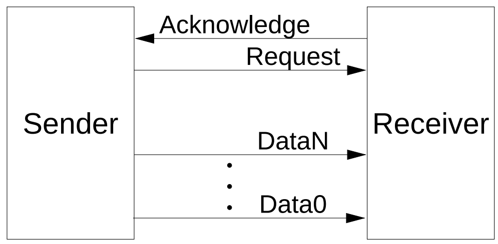


Figure 3.2: Bundled-Data Protocol

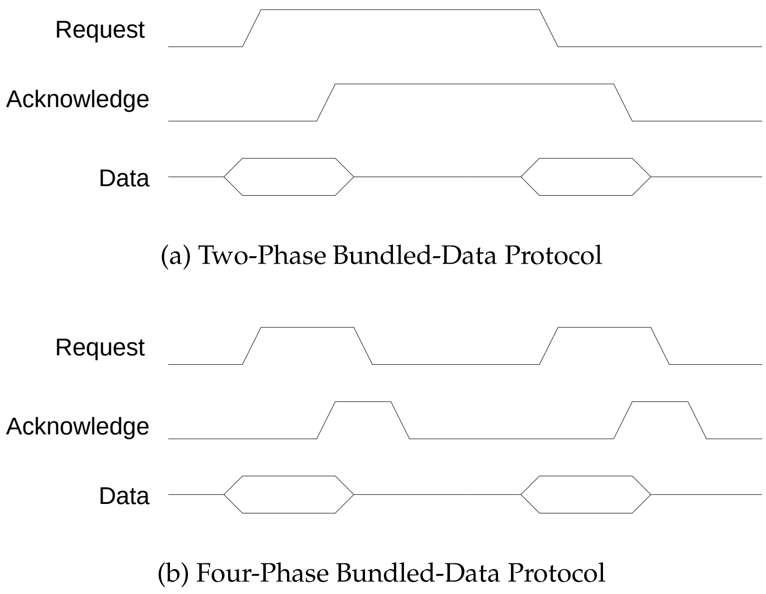


Figure 3.3: Bundled-Data Protocol Timing

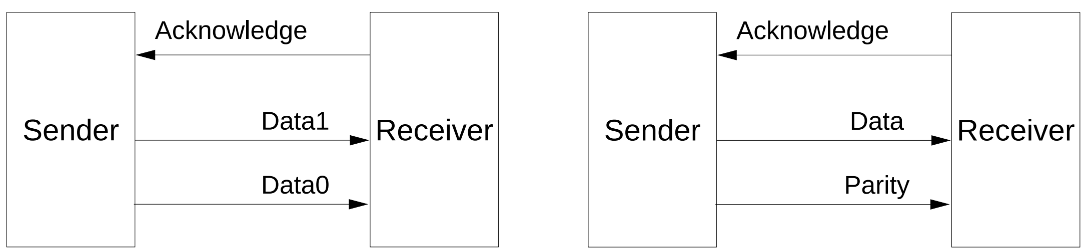
Figure 3.3 illustrates the timing of two-phase and four-phase bundled-data protocols. In both protocols, a transition on the request signal signifies that the data is valid. When the receiver has captured the data, it asserts the acknowledge signal to signify to the sender that it can change the values on its data lines. In the two-phase protocol of Figure 3.3(a), this completes the handshake, and the protocol is repeated for transitions on the handshaking signals in opposite directions. In contrast, the four-phase bundled-data protocol (Figure 3.3(b)) has an additional recovery phase during which the request followed by the acknowledge signals are reset to their original states.

3.4.3 Delay-insensitive Protocols

Delay-insensitive protocols make the assumption that arbitrary delays can be introduced on any signal. Hence an ordering in time of signals at the sender is not necessarily preserved at the receiver.

Martin [81] shows that only a very restricted class of circuit can be made delay-insensitive at the transistor level. He argues that the best compromise to delay-insensitivity that can be made is the use of *isochronic forks* . An isochronic fork places a one-sided bound on certain delay paths of transitions from a forking (fanning out) signal. By using isochronic forks at the transistor level, gates with delay-insensitive interfaces can be built, so at higher levels of abstraction, delay-insensitive systems can be built.

Due to the arbitrary signal delay, a single request signal that indicates the validity of the data cannot be used, since the request signal may be re-ordered, so it arrives before the data signals. Instead, delay-insensitive protocols combine the data and request signals into a codeword. The codewords are arranged such that any subset of the transitions composing a codeword are not codewords themselves. Hence, when a receiver receives a codeword, it knows that it has received all the transitions from the sender, and can safely return an acknowledge signal. Verhoeff [117] discusses the mathematics of delay-insensitive codes in more detail.



(a) Dual Rail Code

(b) Data/Parity Code

Figure 3.4: Delay-Insensitive Communication of One Bit

The simplest delay-insensitive protocols transmit one bit of data, and hence require two codewords. The most common one-bit code is the *dual-rail code*

(Figure 3.4(a)). In the dual rail-code a transition on one wire indicates the value one and a transition on the other indicates the value zero. An alternative is to use a *data/parity* code (Figure 3.4(b)). The data/parity code uses a data signal that signals a change in the data value, and a parity wire that signals when there is no change in the data value. An advantage of this code as a two-phase encoding scheme (as used by McAuley [84]) is that the data value is always available on the data line. Such codes where the data is available without the need for encoding/decoding, are known as *systematic codes*.

A variety of codes exist to transmit more than one bit of data. The simplest technique is simply to encode each bit using the one-bit protocols discussed previously. However, this requires two wires for each data bit, which is costly. Delay-insensitive codes that use less than two wires per bit require more complex encoding/decoding. An example are *k-out-of-n* protocols. Each codeword involves k transitions from n wires, thus k-out-of-n protocols allow n choose k values to be communicated.

Sperner codes, a class of k-out-of-n code, are the optimal delay-insensitive code, in the sense that they maximise the number of codewords for a given number of wires, but are difficult to encode/decode. *Knuth codes* are a subset of Sperner codes which are easier to encode/decode. A disadvantage of both Sperner and Knuth codes are that they are non-systematic. An alternative systematic encoding that uses less than two wires per bit are *Berger codes*. In a Berger code the data is transmitted along with a binary number (known as the parity) indicating the number of zeros in the data.

3.4.4 Handshaking Conventions

Several options exist within handshaking protocols, concerning whether the sender or receiver initiates the handshake. Protocols initiated by the sender are known as *push* handshaking protocols (the sender 'pushes' the data to the receiver). Receiver initiated handshaking protocols are known as *pull* protocols (the receiver 'pulls' the data from the sender). Another option within the handshaking protocol is to implement a *two-way* data transfer. Data is transfered on both the request and acknowledge phases of the handshake.

3.5 Self-Timed Circuit Implementation

The preceding discussion focussed on the protocols themselves rather than on their implementation. This section examines some basic circuit elements used in the implementation of self-timed circuits, which will be used in later chapters.

In particular, the C-Muller gate is described, which forms the basic synchronisation element in a large number of self-timed implementation styles. Building on the C-Muller gate, self-timed pipelines are discussed using the example of Sutherland's Micropipelines [111]. Also, Sutherland's choice of control blocks is examined, since these are typical of the control blocks used in a variety of self-timed circuit design styles.

3.5.1 The C-Muller Gate

The *C-Muller gate* or *Rendezvous element* (see Figure 3.5) forms the basic synchronisation element in many self-timed circuits. In event-based (two-phase) signalling, the C-Muller gate can be thought of as acting as an AND gate for events: the C-Muller gate will not generate an output event until events have occured on all of its inputs.

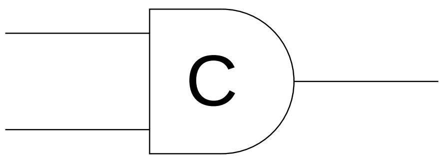


Figure 3.5: Two-Input C-Muller Gate

The statement that the C-Muller gate acts as an AND gate for events requires two caveats. Firstly it is costly to implement a true event-based AND gate where the direction of transition is completely irrelevant. It is much easier to implement a gate if the initial level of the input and output signals is known. Typically, it is assumed that initially the C-Muller gate's output is logic zero and that all inputs are logic one. Given these initial conditions, the behaviour of the C-Muller gate can be described easily as a two state finite state machine (Table 3.1).

Inputs	**Output**
all logic 0	logic 0
dissimilar	no change
all logic 1	logic 1

Table 3.1: C-Muller Gate Next State Table

The second aspect of the C-Muller gate's behaviour that differs from the AND gate for events model concerns the effect of multiple events on an input whilst there is no output event. Ideally, an AND gate for events would allow multiple events to queue up, but it is impossible for a C-Muller gate implementation to store an unbounded number of events. Hence, the C-Muller implementation restricts the inputs such that only one event may occur on each input before an output event occurs. If a second event does occur then they cancel each other out.

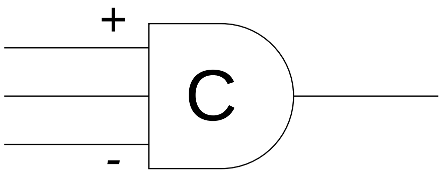


Figure 3.6: Asymmetric C-Muller Gate

In four-phase signalling protocols, transitions in only one direction are significant. This leads to circuits where synchronisation is only required on transitions to one particular logic value of a signal. *Asymmetric C-Muller gates* allow synchronisation on transitions in one direction only. Figure 3.6 shows a three-input asymmetric C-Muller gate. The inputs marked with + and - signs, only synchronise on transitions to logic '1' and logic '0' respectively. The unmarked input acts as a normal C-Muller gate input and synchronises on transitions to both levels.

3.5.2 Micropipelines

Micropipelines are a style of two-phase bundled-data pipeline introduced by Sutherland [111]. Figure 3.7 illustrates a generic bundled-data pipeline. The data path is similar to a synchronous pipeline, except that the latches are connected to a local timing control block, rather than being connected to a global clock. The timing control block deals with generating the request and acknowledge signals for the self-timed protocol and provides the necessary capture and pass signals for the latches. A Micropipeline is a specific example of a self-timed pipeline that uses a two-phase bundled-data protocol.

Figure 3.8 shows the basic timing block for Micropipelines, and a complete Micropipeline is shown in Figure 3.9. The behaviour required of the timing control block is that it will not capture data until there is a request event from the previous stage and an acknowledge event from the next stage. Synchronisation on these two events is done by the two-input C-Muller gate which forms the basis of the timing block. The pass signal is generated directly from

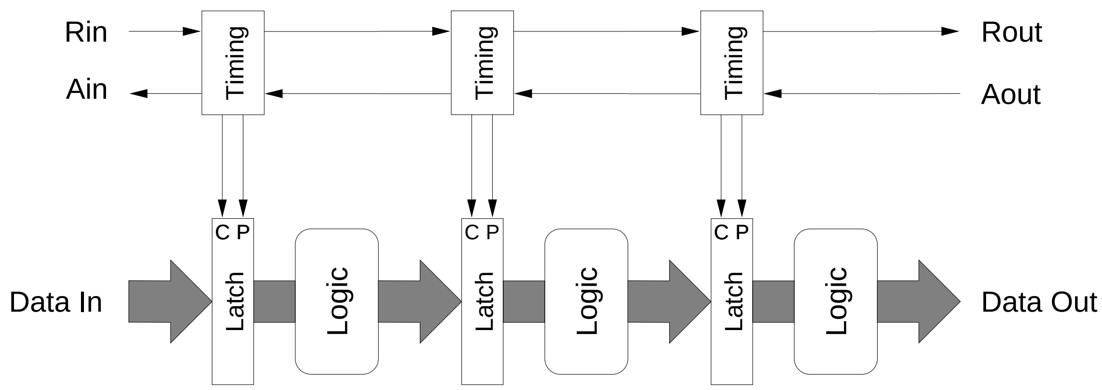


Figure 3.7: A Self-timed Pipeline

the output acknowledge, since once the next stage has latched the data, the current stage's latches can go to the pass state.

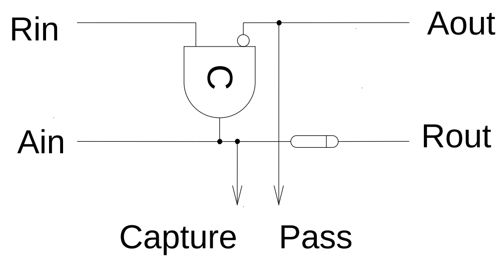


Figure 3.8: Micropipeline Timing Block

Two additional elements are added to the basic C-Muller gate. First, the output acknowledge signal, `Aout`, is inverted. The effect of this is that when the first request event is received, there is no need to wait for an acknowledge, since `Aout` is already at the required level. This is the required behaviour, since when the first request is received, there is no data further along the pipeline that needs to be acknowledged.

The second addition is a delay element, represented by the oval on the output request wire. This delay is required so that the output request `Rout` is asserted after the data is valid, so that the bundling constraint of the bundled-data protocol is met. In Micropipelines [111], Sutherland uses the routing delay on distributing the capture and pass signals to delay stages, with the addition of delay elements, if required, to meet the bundling constraint.

Typically, data and the associated request acknowledge events are regarded as moving forward in the Micropipeline. However, when the pipeline is nearly full, it is useful to consider, events propagating back from the output, as a *bubble* or empty register stage passes back in the pipeline, and all the data stored passes forward by one stage. Greenstreet et al [52] argue that the num-

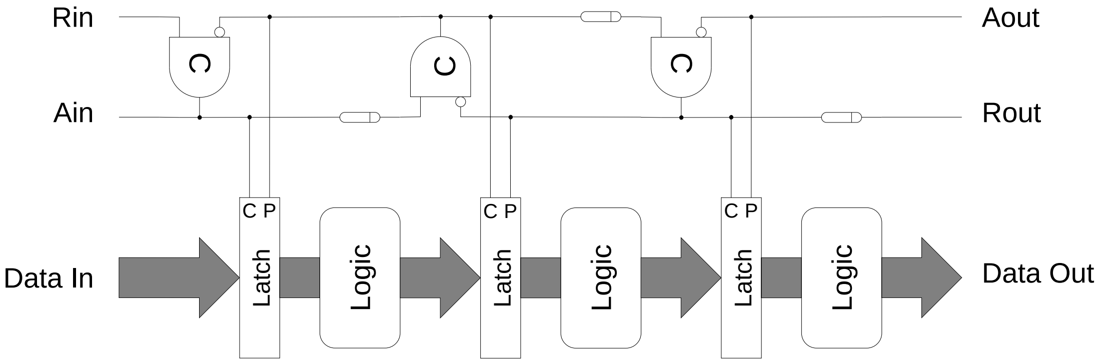


Figure 3.9: A Micropipeline

ber of bubbles is critical to the behaviour of self-timed pipelines when they are nearly full.

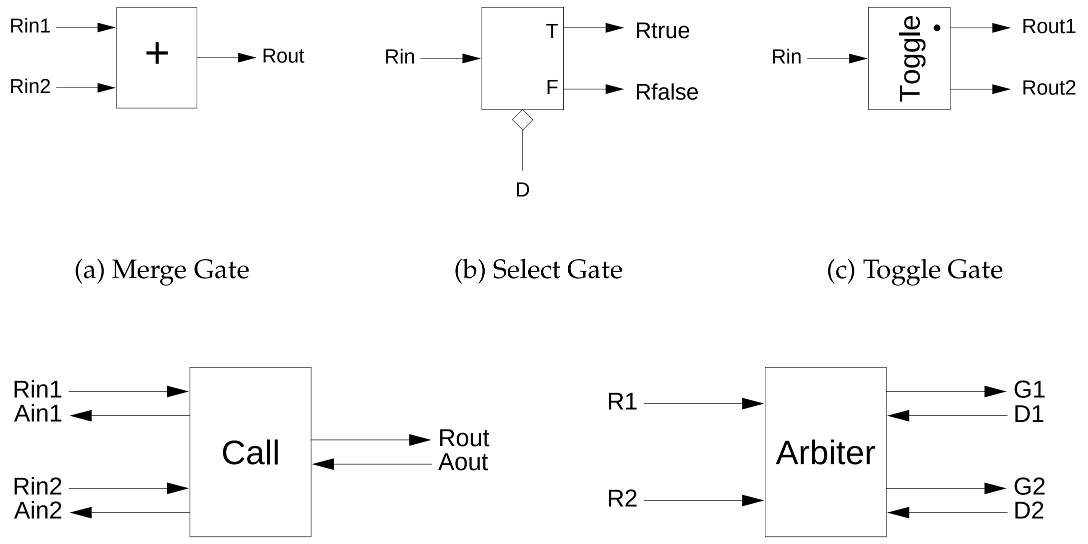
An observation that Sutherland is keen to point out is the symmetry in Micropipelines. First, there is a clear symmetry between the request and acknowledge signals which are identical apart from the inversion of the acknowledge. Second, there is the symmetry of viewing the pipeline as data propagating forward through the pipeline or as bubbles propagating backwards. Sutherland is also keen to point out the analogous behaviour of such pipelines with other phenomena such as wave propagation, and to compare bubbles with hole propagation in semiconductors.

3.5.3 Micropipeline Control Blocks

Sutherland uses a variety of control blocks for Micropipelines, which provide similar functions to control blocks proposed for other self-timed design methodologies. The functions provided by these blocks allow the branching and merging of control flow in self-timed circuits. The Micropipeline control blocks are illustrated in Figure 3.10 and are described below.

Merge Gate

Figure 3.10(a) illustrates the *Merge* gate. As the name suggests, its behaviour is to merge event flows together. Events on either `Rin1` or `Rin2` cause an output event on `Rout`. In two-phase protocols, the Merge gate is generally implemented using an XOR gate. However, this can causes problems, when the events `Rin1` and `Rin2` are not mutually exclusive, since when both input events occur, the output will return to its original level. If mutual exclusion cannot be guaranteed then the Merge gate's behaviour has to be extended to arbitrate between its inputs.



(a) Merge Gate (b) Select Gate (c) Toggle Gate

(d) Call Module (e) Arbiter Module

Figure 3.10: Micropipeline Control Blocks

Select Gate

The *Select* gate causes a branch in the control flow. The input event `Rin` is directed to one of two outputs depending on the value of the select input, `D`. The value of `D` must be established before `Rin` occurs for correct operation of the Select gate.

Toggle Gate

The *Toggle* gate, is similar to the select gate in that it causes a branching in the event flow, but it lacks a select signal. Instead input events are passed alternately to the outputs. The dotted output on a Toggle gate indicates that this is the output to which the first input event is directed.

Call Module

The *Call* module performs an analogous function to a software procedure call: it allows a common section of data path to be accessed from several different points. Unlike the previous control blocks, the Call module operates on pairs of request/acknowledge handshaking signals. Incoming requests are directed to the output request `Rout`. When the output acknowledge `Aout` is received, this is directed back to the appropriate input acknowledge. If the input request events are not mutually exclusive then some form of arbitration is required within the Call module.

Arbiter

Figure 3.10(e) shows Sutherland's *Arbiter* module. The Arbiter module arbitrates between two possibly concurrent requests `R1` and `R2` and will only pass one through at a time to the grant outputs `G1` and `G2`. The `D1` and `D2` signals are used to release the resource. Either the grant signals or the release signals can be used to generate acknowledge signals for the input request, depending on the behaviour required of the Arbiter.

3.6 Current Research

Much of the research into asynchronous systems has concentrated on the synthesis of delay-insensitive circuits. Such circuits are suited to automated synthesis, since only the ordering of events is important and not their actual timing. Hence, reasoning about actual delays is not required, so a good mapping of the problems can be made to abstract formalisms, such as process algebras. Martin [80] developed a synthesis method based on CSP, whilst Weber et al [119] synthesised circuits using CCS. Others researchers have used specialised representational forms such as trace theory [29]. Petri Net based State Transition Graph (STG) representations [70] are another common representational form for the synthesis of asynchronous circuits. However most current tools such as Assassin [125] involve creating a reachability graph of the Petri Net that suffers from the state explosion problem. Semenov et al [105] are looking at compact representations using partial orders to solve this problem.

The most mature of the synthesis tools that have been developed is the TANGRAM compiler [67] developed at Phillips Research. TANGRAM has been used in the synthesis of a Digital Compact Cassette (DCC) decoder system. An interesting feature of the TANGRAM tool is that it is flexible enough to produce both delay-insensitive dual-rail encoded circuits and bundled-data circuits. The bundled-data version [116] of the DCC chip out performs the dual-rail version in area by 40% and in speed by a factor of three. In comparison to a synchronous design, the bundled-data design had an area overhead of less than 20% and used five times less power. When a novel voltage switching scheme was introduced, the power saving increased to a factor of twenty.

The most complex systems that self-timed design has been applied to are microprocessors. Martin, using CSP synthesis methods implemented a small processor [82] that displayed increased resilience to temperature and voltage effects [83]. The AMULET project at the University of Manchester have been involved in the implementation of self-timed versions of the ARM microprocessor. The AMULET1 chip [39] is a functionally equivalent version of the ARM microprocessor built using Micropipelines. Though AMULET1 demonstrated the viability of producing large scale self-timed parts, AMULET1 used more silicon and power for less performance than the comparable ARM6 processor. The successor AMULET2e [36, 38] includes additional features such as on-board RAM for embedded systems applications. The main difference in the underlying implementation is a switch from two-phase to four-phase bundled-data protocol. AMULET2e gives performance at 3.3V which is better than the ARM7 chip, but behind that of the ARM8. However, the power/performance ratio is comparable with that of the ARM8 chip.

Other researchers have examined alternative processor architectures specifically designed for asynchronous control. The Counterflow processor architecture [109] has two pipelines, one for results and one for instructions and operands, which flow in opposite directions. Instructions and results interact as they move along the pipelines. The MAP architecture [8, 100] introduces a model for decentralising control in a microprocessor to the functional units called 'Micronets'. Current work [7] is looking at the design of a super-scalar asynchronous processor using Micronets.

Other researchers have examined systems that combine elements of both asynchronous and synchronous design styles in what are often termed *Globally Asynchronous Locally Synchronous (GALS)* systems. The aim of these approaches is to allow the incorporation of synchronous designs into larger asynchronous systems with minimal modification, so that synchronous design experience and tools can be used within asynchronous systems. Arguably, bundled-data systems in general can be classed as a GALS approach, since the only modification to a synchronous data path that is required is to source the register control signals from a local timing control block rather than from a global clock, and possibly change the style of memory elements (as in Micropipelines [111]). Gopalakrishnan and Josephson [48] give a good overview of various GALS methodologies, and their terminology is adopted here. Most approaches can be classified as using either a *stoppable clock*, that is restarted by the arrival of data, for example Asynchronous Wrappers [14, 15] and Chapiro's original work on GALS [22]. Others approaches use a *stretchable clock* (eg. Q-Flops [103], Gopalakrishnan and Josephson's work [48] and STRiP [27]), which delay the next clock cycle until meta-stability has been resolved and/or a data-path completion signal is received.

3.7 Summary

The key benefit of self-timed protocols is their speed-independence, which allows different parts of a system to run at their own rate, rather than being forced to operate at a rate determined by a global clock. This leads to increased modularity and robustness, in comparison to synchronous systems, and allows data dependent delays to be exploited. The latter part of the chapter discussed various self-timed protocols, and in particular the bundled-data protocols that are used in the self-timed FPGA architectures developed in this thesis.

Chapter 4

Self-Timed FPGAs

4.1 Introduction

The key idea of this thesis is that the speed-independence of self-timed protocols make them ideally suited to the reconfigurability of FPGAs. In particular, current dynamic hardware systems are limited by synchronous protocols, since changes to the structure or environment of a circuit, change its delay characteristics and hence may increase the circuit's delay beyond that of the global clock period. This chapter takes this initial idea and refines it by addressing two questions: what benefits can be expected from self-timed FPGA systems and what is the best way to obtain these benefits ?

The first question is addressed in Section 4.2, which considers the benefits of self-timed FPGA systems in general, and Section 4.3 which considers the specific benefits of self-timed dynamic hardware systems. The second question of how to obtain these benefits is considered in the latter half of the chapter. Section 4.4 discusses implementation of self-timed circuits on current FPGA architectures. The problems encountered with implementing self-timed circuits on current FPGAs has led to the proposal of dedicated asynchronous FPGA architectures; these are discussed in Section 4.5. The solution proposed in this thesis, STACC, is described in the following chapter.

4.2 Motivation for Self-Timed FPGA Systems

In the previous chapter, the benefits and drawbacks of self-timed systems relative to synchronous systems in general were discussed. This section examines how these properties can be gainfully utilised or avoided in the specific context of FPGAs. Many of the potential benefits are specific to dynamic hardware systems, and are left for discussion in the following section on the motivation for self-timed dynamic hardware systems.

Several assumptions are made below when discussing the advantages of self-timed circuits on FPGAs. First, it is assumed that the mapping algorithms (i.e. partitioning, placement and routing) are working with self-timed elements. If the underlying function blocks of the FPGA architecture are not self-timed elements then self-timed elements need to be produced from the function blocks by performing some local placement and routing. Additionally, for bundled-data systems, it is assumed that the routing of the FPGA architecture is such that the bundling constraint will be maintained. The validity of these assumptions is considered later in Section 4.5

4.2.1 Benefits of Self-Timed Circuits on FPGAs

Partitioning and Extensibility

Compared to other ASIC implementation styles, FPGAs can implement less logic per device, so the partitioning of designs is more common. Problems arise in partitioned systems from the difference between off-chip and on-chip delays; in synchronous designs, the off-chip delays can drastically reduce the clock speed of a design. Partitioning algorithms can help with the problem, but often re-design of the system is needed to explicitly cope with the partitioning.

In contrast, self-timed protocols allow designs to extend naturally across several chips. An array of FPGAs can be treated as a uniform array of cells, since the self-timed protocols accommodate the additional off-chip delays. A further advantage of self-timed FPGAs is that the performance degradation from off-chip links will only affect the systems performance when the off-chip links are in use. In a synchronous systems, unless the system is re-designed to wait multiple clock periods for the off-chip links, the performance penalty of a longer clock period is incurred whether or not the off-chip link is used during a particular clock cycle.

Saturated Routing

In FPGA designs with high logic utilisation, the interconnect can become saturated, causing many signals to be placed on long snaking paths through the FPGA. This can severely limit the clock rate of synchronous designs. In self-timed designs, these paths can be allocated to infrequently used signals, which will only limit performance when the signal path is used.

Faster Mapping and Design Turn Around

In synchronous systems, detailed timing analysis of all signal paths is needed to ensure that the result of routing, placement and partitioning of a design meets the global clock constraint. Because of the speed-independence of self-timed designs, any route, place and partition of self-timed modules that implements the required connectivity will produce a correctly functioning system. Detailed timing analysis is only required to improve the performance of the mapping rather than to ensure a working system. Thus, initial mappings may be done quickly, enabling faster design turn-around. Detailed timing analysis for improved performance can be reserved for the mapping of the final design.

Migration to Different FPGA Architectures

Currently, there are a large number of commercial FPGA architectures produced with a wide range of speed grades and array sizes. The robustness of self-timed designs to delays eases the process of migrating designs between different FPGA architectures.

4.2.2 Disadvantages of Self-Timed Circuits on FPGAs

Area Overhead

This drawback has already been mentioned when discussing the disadvantages of self-timed circuits in general. However, the problem with FPGAs is more acute, since FPGAs use of the order of ten times the area of a dedicated silicon circuit to implement the same system. Thus, the extra area required for a self-timed circuit is consuming a scarce resource.

Interfacing with Synchronous Systems

FPGAs are commonly used for implementing sub-systems within larger systems. If the rest of the system is synchronous, it makes little sense to use an asynchronous FPGA.

4.3 Self-Timed Dynamic Hardware Systems

The previous section outlined the potential benefits of self-timed FPGAs with regard to static designs. However, the speed-independence of self-timed circuits makes them ideal for use in dynamic hardware systems. The ability to alter the shape or layout of a circuit, and hence the delay without it causing the circuit to fail seems to be the key to exploiting dynamic hardware.

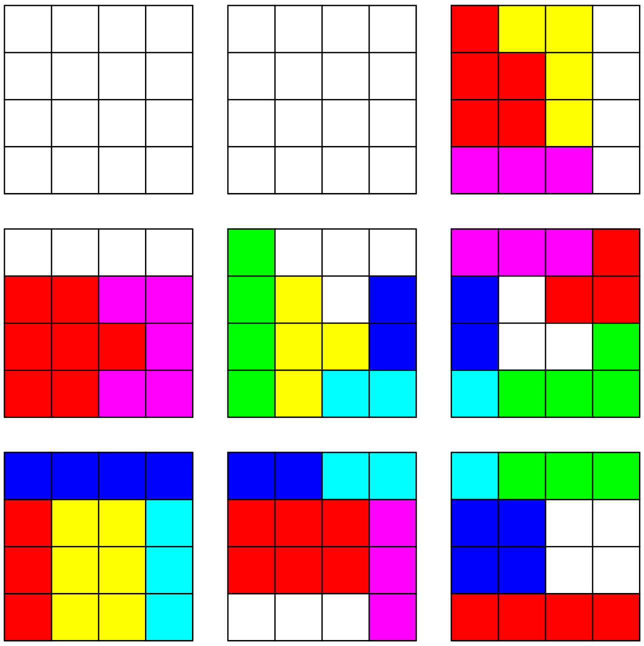


Figure 4.1: Idealised Model of Virtual Hardware

The benefits are illustrated by the idealised model of a virtual hardware system shown in Figure 4.1. The system is composed of a 3×3 array of FPGA chips, each containing a 4×4 array of hardware pages. A number of different circuits are swapped in; these are represented by the different coloured blocks in the figure. The management of the virtual hardware for such a system is essentially a two-dimensional version of the problem in a segment based virtual memory. The virtual hardware manager must allocate a contiguous two-dimensional area of reconfigurable hardware which is of the correct shape for the circuit that is to be swapped in (the problem is in many ways similar to that encountered in the computer game Tetris).

An attempt to build such a system using synchronous technology is going to run into several problems. One problem is determining the clocking period for the reconfigurable hardware. This is determined by the slowest circuit swapped in, but changes every time a new circuit is swapped in. Also, the virtual hardware manager must ensure that no circuit is split across several FPGAs, otherwise the off-chip delays will cause the circuit to fail to meet the global clock constraint. Other problems occur with the interfacing of software to hardware. The speeds at which hardware and software are running are different, so some kind of flow control is required between the two. Indeed, flow control is also needed between hardware communicating with hardware, since it is possible that one circuit has not yet been swapped into the virtual hardware.

The problems with synchronous dynamic hardware systems are expanded on below, and it is shown how self-timed circuits overcome these problems.

Modules Operate Independently

In a synchronous dynamic hardware system, all the circuits would be forced to run at the speed of the slowest circuit loaded. The alternative would be to have multiple clocks, but this would require additional routing and circuitry, and would cause interfacing problems between circuits with clocks running at different rates.

In practice, the reconfigurable hardware would probably be clocked at a fixed rate and circuits would have to be designed with this in mind. To accommodate most reasonable designs, the clock would have to be set to go fairly slowly, leading to poor performance. In contrast, the circuits within a self-timed dynamic hardware system are independent of each other, so the speed at which one circuit runs does not limit the speed at which the others run.

Large Arrays

Dynamic hardware systems will often require a large amount of reconfigurable hardware resource, which requires circuits to be split over several FPGAs. Synchronous designs cannot easily be partitioned over several chips unless originally designed as such, since the off-chip delays require a recalculation of the delays within the circuit. In contrast, self-timed circuits are naturally extensible to large multiple chip arrays since the self-timed protocols accommodate the extra off-chip delays. Different self-timed modules of a circuits can easily be split over several different chips.

Natural Interface to Software

FPGA and host microprocessor in a dynamic hardware system compute at different rates, so some form of flow control is required to regulate the flow of data between software and reconfigurable hardware. In self-timed designs, the flow control comes as part of the protocol. Synchronous dynamic hardware would require the use of additional circuitry to regulate the flow of data between host and FPGA.

Flexible Mapping

The ability to map circuits quickly and flexibly is essential for virtual hardware so that efficient use of the reconfigurable hardware resource can be made. Synchronous designs can not be mapped quickly, since this alters the layout of the circuit, so the clock speed would have to be recalculated. Alternatively, the clock speed could be set low enough so that correct operation was guaranteed for all possible mappings, but this leads to poor performance.

Self-timed FPGAs allow circuits to be separated to make efficient use of the reconfigurable hardware resource. For example, there may be sufficient free reconfigurable hardware in a synchronous virtual hardware system to swap a circuit in, but if this space is different from the shape of the circuit to be swapped in then this forces a circuit to be swapped out. In a self-timed virtual hardware system, the circuit's shape could be altered so it fits into the available space.

Circuits can Block

In virtual hardware, if two circuits are communicating directly in the hardware then potentially a communication between the circuits would have to block whilst waiting for one of the circuits to be swapped in. In general, synchronous circuits could not be used as they could only be blocked by stopping the clock, and hence, all the circuits in the reconfigurable hardware. However, all self-timed circuits would block naturally, since they could only continue when the other circuit had been swapped in and had generated an acknowledge signal.

4.4 Self-Timed Systems on Current FPGAs

Most of the current research on self-timed circuits using FPGAs has concentrated on using commercially available synchronous FPGAs. The main focus of these researchers has been on whether asynchronous circuits can be prototyped using current FPGA architectures.

Several researchers have built Micropipeline [111] circuits. Oldfield and Kappler [92] implemented Micropipeline FIFO circuits using the Algotronix CAL1024 [3]. Both Maheswaran and Akella [78], and Gamble et al [41] have implemented Micropipeline circuits using Xilinx [124] FPGAs. Brunvand [20], also built a library of Micropipeline elements, but this time using the Actel FPGA [2]. Subsequently he used the library to implement a small self-timed processor [19].

To date, the only dynamic hardware system that has used self-timed circuits is Shaw and Milne's SPACE (Scalable Parallel Architecture for Concurrency Experiments) machine [106]. SPACE was used to implement delay-insensitive asynchronous circuits for road traffic simulation [88]. However, the focus of Shaw and Milne's research was on the formal synthesis of delay-insensitive circuits rather than the potential for self-timed dynamic hardware systems.

The above works have illustrated the feasibility of using current FPGA architectures to build self-timed circuits, and that large self-timed circuits such as processors can be built using them. Furthermore, the use of current FPGAs for implementing asynchronous systems is advantageous, since the chips are readily available standard parts, and can implement synchronous systems as well. However, the above works have shown that self-timed circuits on current FPGAs required careful design to overcome the deficiencies of current architectures for implementing self-timed circuits. These problems are discussed below.

Hazards

Within synchronous systems, signals are only sampled when the global clock ticks. A signal may go through several different logic values before reaching its final value (a *hazard* condition), as long as it reaches its final value before the next clock tick. However, in delay-insensitive circuits, and the control path of bundled-data circuits, signals are being continuously monitored, and so must be free from hazards.

Current synchronous FPGAs are not designed to produce hazard free signals. Furthermore, hazards may be introduced by the circuit decomposition performed by technology mappers for synchronous FPGA architectures. Careful design is required to avoid introducing hazards in synchronous designs; Maheswaran and Akella [78] discuss methods to avoid hazards in the Xilinx XC4000 FPGA.

Ordering and Delaying Signals

Self-timed circuits rely on the ordering of signals for their correct operation. In delay-insensitive circuits, this manifests itself as a need for isochronic forks, whilst in bundled-data systems, it requires that the bundling constraint is met. Current FPGA routing architectures can easily re-order signals and make such ordering constraints difficult to meet. In addition, for bundled-data systems, the delay of the request signal relative to the data-bundle should be as small as possible for performance reasons. Again, this can be difficult to achieve in current FPGA architectures.

Arbitration

Arbitration is a common function within asynchronous circuits. Current FPGA architectures provide no support for building the special circuitry needed in arbiters for providing clean output signals from the meta-stable states that such circuits can enter. Arbiters can be built in synchronous FPGAs, but require careful design and have a finite chance of failure. Brunvand [20] gives a comprehensive account of building an arbiter using Actel FPGAs including a mean time between failure analysis for his design.

4.5 Current Asynchronous FPGA Architectures

At the start of this chapter, the benefits of self-timed circuits on FPGAs were discussed. However, these benefits were based on the assumption that self-timed elements can be created on the FPGA architecture. Furthermore, for bundled-data systems it was also assumed that the routing architecture would preserve the bundling constraint. The previous section has discussed the problems encountered with current synchronous architectures in fulfilling these conditions. Hazards, signal re-ordering and the lack of arbitration elements make design of self-timed elements and their routing on current architectures problematic.

Thus, the potential benefits of self-timed circuits on FPGAs are difficult to obtain, due to the constraints placed on self-timed designs using current architectures. In particular, the benefits for self-timed dynamic hardware systems are hard to realise, since the swapping algorithms will need to solve these problems on the fly rather than during the circuit design. To overcome the problems with current synchronous-oriented FPGAs, asynchronous FPGA architectures have been proposed. Asynchronous FPGA architectures shift the burden of solving these problems from the circuit designer to the architecture designer. The aim of dedicated asynchronous architectures is to provide a clean architecture for circuit designers and tools to work on, so that the advantages of self-timed design can be exploited to the full.

Currently, two self-timed FPGA architectures have been proposed: MONTAGE [57] and PGA-STC [77]. These two architectures are described below, with particular focus on how the architectures overcome the problems of hazards, signal re-ordering and arbitration encountered in current FPGA architectures. GALSA [42], an architecture designed for massively parallel processing (MPP) is also discussed, as such architectures are closely related to FPGA architectures. The discussion of GALSA focuses on how it converts a synchronous processing element to use asynchronous communication.

4.5.1 MONTAGE

MONTAGE [57], designed at the University of Washington, was the first asynchronous FPGA, though it includes two global clock signals for implementing synchronous circuits as well. It is closely based on the TRIPTYCH [56] architecture. MONTAGE extends TRIPTYCH by adding special arbitration cells, and modifying the function block to allow the creation of state retaining elements. The design of MONTAGE with respect to its function block design, arbitration, and signal delaying is discussed below.

Function Block Design

The MONTAGE function block (Figure 4.2), in common with many synchronous FPGAs, uses a LUT (Look Up Table) based function block. The inputs `A`, `B` and `C` are used to select a value from the LUT's configuration memory. A LUT-based implementation was chosen as it is free from hazards on single input changes. Though free from hazards on single input changes, a LUT based function block may still create output hazards on multiple input changes. For the example configuration, the transition of `ABC` from `010` to `100` can cause a momentary `1` to occur on the output of the LUT. MONTAGE leaves the problem of multiple input changes as a problem for mapping tools to avoid.

A feature of delay-insensitive circuits is their use of feedback to create state holding elements. Problems may occur if the next state has not been established before the next input change. Hence, MONTAGE includes a feedback path within the function block, to allow new states to be established quickly; any of the inputs to the LUT may be replaced with a feedback signal from the output. In the example configuration of Figure 4.2, the `C` input is used as an output feedback, whilst `A` and `B` are used as the inputs of the C-Muller gate. MONTAGE uses the smallest LUT that is feasible for an asynchronous architecture since, for a basic two input state retaining function, two inputs and one internal feedback are required.

A surprising feature of the MONTAGE function block is the inclusion of a D-latch. The D-latch is included since MONTAGE was designed as a hy-

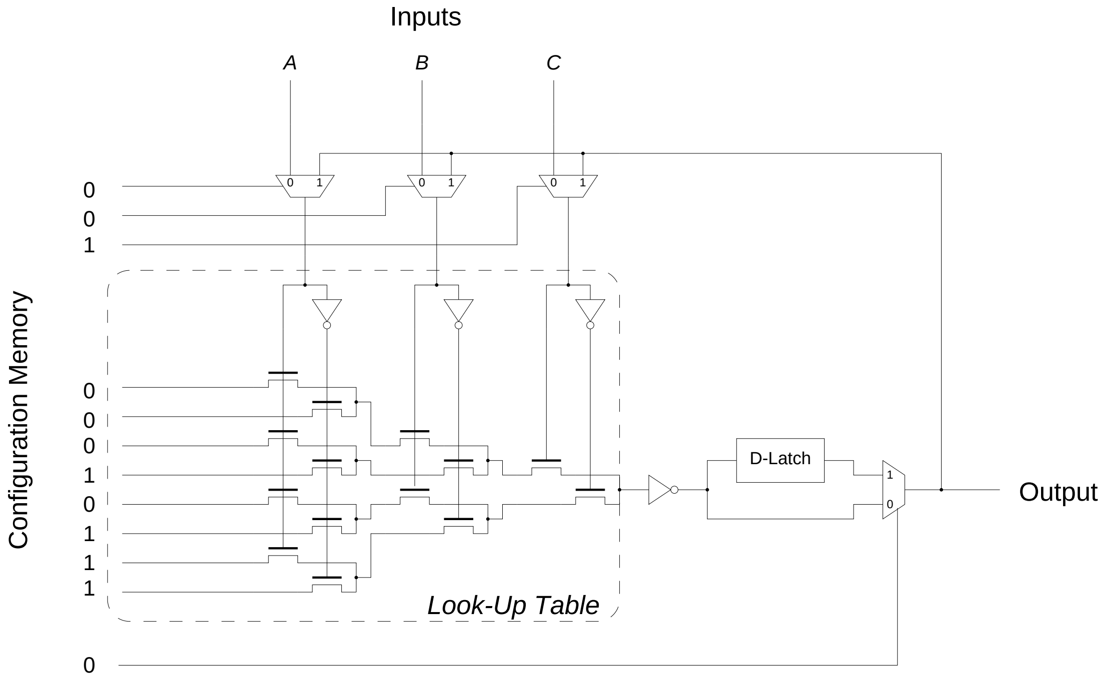


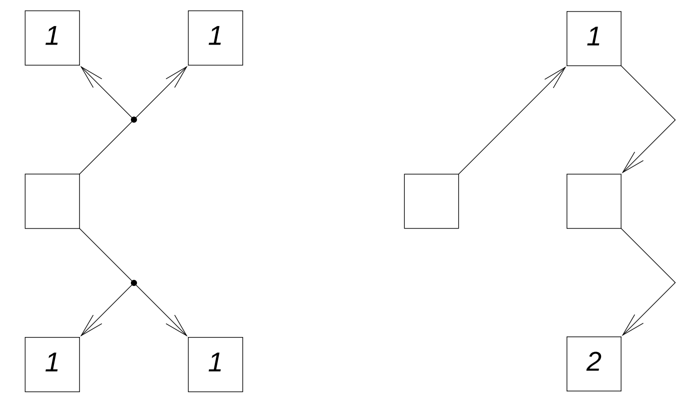
Figure 4.2: The MONTAGE Function Block (Configured as a C-Muller Gate)

brid FPGA suitable for asynchronous and synchronous circuits. The D-latch is used when building synchronous circuits and is normally bypassed in asynchronous designs. However, the MONTAGE designers utilise the D-latch in asynchronous mode to allow initialisation of state retaining functions. Since the state of any output feedback path can be indeterminate at initialisation, the output is taken from the D-latch. The D-latch can be preset or cleared to the correct value for the initial circuit feedback. Once the initial state is established, the D-latch is bypassed in asynchronous operation.

Ordering Signals and Delay Elements

MONTAGE does not have dedicated delay elements, instead it relies upon a tight regular routing structure. Figure 4.3(a) shows how isochronic forks are created by placing each branch of a fork on similar routing paths. Also, since MONTAGE has integrated routing and function blocks, asymmetric forks can be generated by routing the signal for the longer fork from the destination cell of the short fork. This is illustrated in Figure 4.3(b).

Bundled-data systems have a two-sided delay bound. The request signal has to be asserted after the data is valid, but for performance it should match the delay of the data as closely as possible. MONTAGE does not have special delay elements, so for bundled-data systems, it has to use routing and function



(a) Isochronic (b) Asymmetric

Figure 4.3: Isochronic and Asymmetric Forks in MONTAGE

blocks to build a chain of buffer elements with the appropriate delay.

Arbitration

Arbitration in MONTAGE is provided by special arbitration function blocks that are distributed through out the architecture to replace standard function blocks. The ratio of standard to arbitration function blocks is 15:1. The MONTAGE arbitration block is shown in Figure 4.4; the block is centred around a *mutual exclusion* element. The mutual exclusion element ensures that only one of the request signals `R1` or `R2` is granted by `G1` or `G2` at any one time.

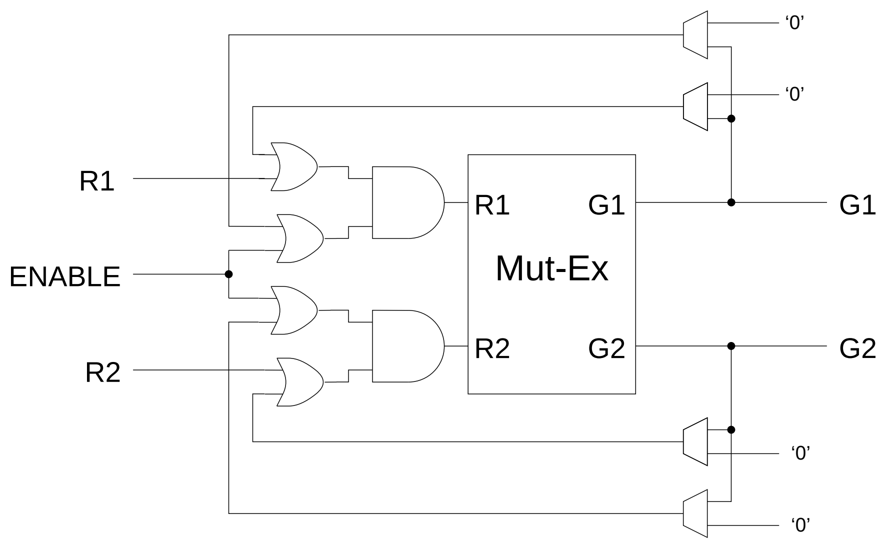


Figure 4.4: MONTAGE Arbitration Function Block

The MONTAGE arbitration block has been designed to implement a range of common four-phase arbitration blocks. By setting the `ENABLE` signal to logic one, and connecting the feedback multiplexors to logic zero, the block can be used simply as a mutual exclusion element. Connecting a signal to the `ENABLE` input allows the block to be used as an *enabled-arbiter*: arbitration will not begin until the `ENABLE` signal goes high. A *synchroniser* circuit, that samples an input signal when triggered by a clock signal, can be built by connecting `R1` to the signal to be sampled, `R2` to its inverse, and `ENABLE` to the sampling clock.

The feedback paths via the OR gates allow the grant signals, `G1` and `G2`, to remain valid after some of the outputs are de-asserted. For example, choosing to feed the grant signals back through the OR gates connected to the request lines, means the result of arbitration will remain valid until the `ENABLE` signal is de-asserted. Two-phase arbitration elements can be produced by using additional logic from standard function blocks.

4.5.2 PGA-STC

Concurrent with the work in this thesis, the PGA-STC architecture has been proposed by Maheswaran [77]. It is targeted at the implementation of two-phase bundled-data systems such as Micropipelines [111]. The architecture is loosely based on that of the Xilinx XC4000 series [124], with modifications to the function block, and the addition of arbitration cells and programmable delay elements.

Function Block Design

Figure 4.5 shows the PGA-STC function block. It has a similar structure to the MONTAGE function block, using a LUT with an output feedback. Also, logic is provided for initialising state-holding elements. Rather than a D-latch, PGA-STC uses a multiplexor that chooses between the LUT output and constant zero and one inputs.

The principal difference from the MONTAGE function block is the inclusion of the Programmable Delay Element (PDE). The PDE is included since PGA-STC is targeted at implementing bundled-data systems, where request signals have to be delayed to match the delay in the data path. The PDE is considered in more detail, below.

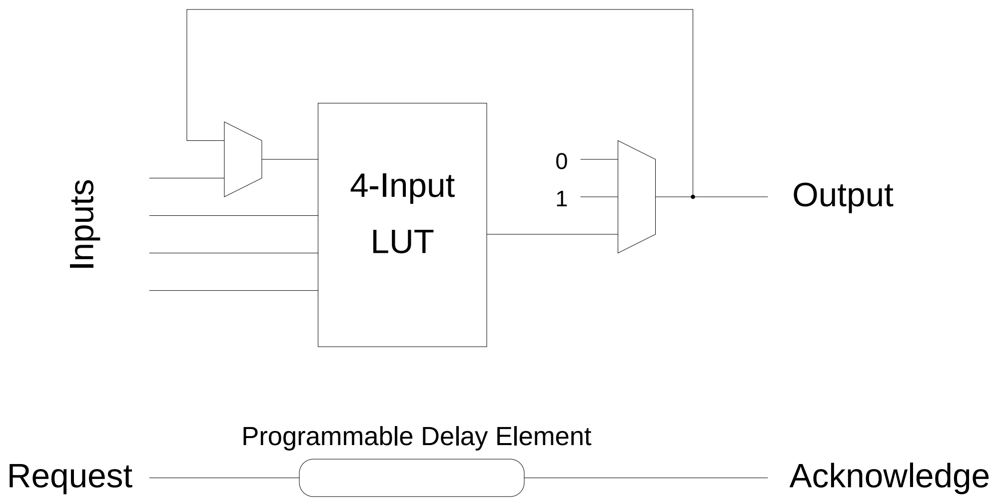


Figure 4.5: The PGA-STC Function-Block

Ordering Signals and Delay Elements

Providing accurate delays for bundled-data without a timing signal such as a clock is a difficult task. Without a clock, gate delays have to be utilised for timing. PGA-STC uses a programmable delay element that produces a delay by taking taps off a chain of inverters. However, in the PGA-STC architecture, the designers were concerned that the delay should match the delay of the function block as closely as possible. This is a natural concern in the PGA-STC architecture, since the delays are matched to each function block and a delay chain that is too coarse will introduce quite a large error for the delay over a series of function blocks. Hence, the PGA-STC design includes a fine delay generator as well as using an inverter chain as a coarse delay generator.

To produce a delay finer than one gate delay, the PGA-STC design utilises a novel structure called a *coupled ring oscillator*. The basic oscillator structure is shown in Figure 4.6. It uses a set of inverter ring oscillators (the horizontal connections). The inverter ring oscillators are coupled to the oscillators below using special two-input inverter elements (the vertical connections). The construction of the two-input inverter elements consists of two inverters driving the same output. Typically, having two elements drive the same output can cause problems with driver conflict, but the inputs to them are coupled by the oscillator structure.

The coupling of the inverter rings causes the oscillation of an inverter ring to be a delayed copy of the oscillation in the ring above. By coupling the bottom oscillator to the top oscillator (effectively forming a ring of ring oscillators), the phase shift around the whole loop is forced to be two inverter delays.

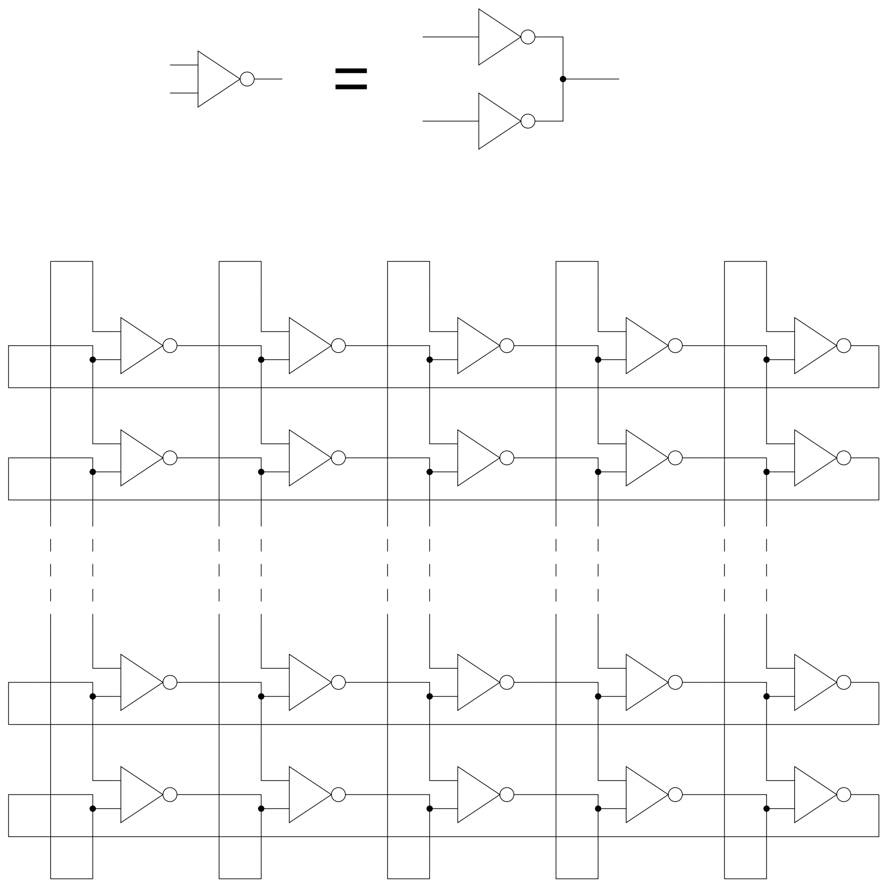


Figure 4.6: PGA-STC Coupled Ring Oscillator

So the phase difference between neighbouring oscillators is two inverter delays divided by the number of inverter ring oscillators. With addition of special control logic, delays of a fraction of a buffer delay can be generated.

There are two major problems with the coupled ring oscillator in PGA-STC. First, the oscillator is a big power drain, annulling the low power advantages of an asynchronous design. Second, the coupled ring oscillator takes up a large amount of silicon area that could be used for extra function blocks. To overcome these problems, a possible adaptation of the PGA-STC architecture would be to only have programmable delay elements in some of the function blocks, or to only use the coarse delay chain.

Arbitration

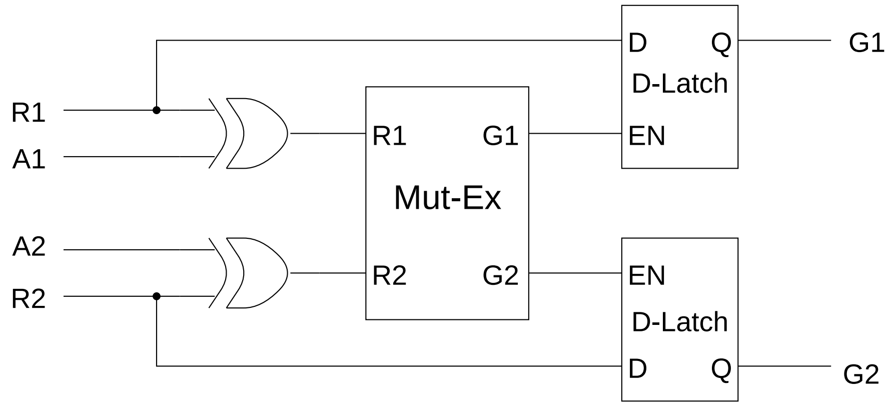


Figure 4.7: PGA-STC Arbitration Function

In common with MONTAGE, PGA-STC replaces some of the standard function blocks in the architecture with arbitration blocks based on the mutual exclusion circuit. Since the mutual exclusion element is a four-phase element, and PGA-STC is targeted at two-phase circuits, extra logic is added that allows the mutual exclusion element to be configured as a two-phase arbiter as well. The PGA-STC arbitration cell is shown in Figure 4.7. The grant signals from the Mutual Exclusion element are used to enable the D-latches, so that the request signals can only pass once they have been granted. The acknowledge signals, `A1` and `A2`, are used to acknowledge when the resource has been used, so that the grant signal can be disabled.

4.5.3 GALSA

GALSA (Globally Asynchronous Locally Synchronous Array) is an architecture developed at the University of Edinburgh by Gao. The architecture was developed for massively parallel computing architectures. Massively parallel computers that use single-bit processing elements, such as the Connection Machine [58] and DAP [59], have many architectural similarities to FPGAs, so GALSA can be considered as a form of asynchronous FPGA architecture.

Gao's GALS approach is similar to other GALS approaches such as Asynchronous Wrappers [14, 15] in that it surrounds traditional synchronous logic with additional circuitry for asynchronous data transfer (See Section 3.6 for discussion of other GALS approaches). However Gao's work differs from others GALS methodologies in that it neither uses a stoppable clock that is restarted by the arrival of data, or a stretchable clock that waits for meta-stability resolution and/or a data-completion signal. In both stoppable and stretchable clock approaches, the clock is generated locally. However in Gao's scheme the local clock is generated from a global clock signal which cannot be stopped or stretched. Each local clock is a gated version of the global clock. In GALSA, when no data has arrived for processing via the data transfer interface, the local clock is disenabled.

When gating the clock, meta-stability can arise between the global clock and the clock enable signal. Gao carefully designs a synchroniser to minimise the chances of meta-stability, but meta-stability still has a finite chance of occuring since the global clock is not stopped or stretched in any way to account for time required for meta-stability resolution.

Figure 4.8 illustrates the basic structure of a GALSA module (adapted from Figures 5.17 and 5.9 in [42]; the original signal names are marked in italics). The core of the module is a synchronous processing element. The architecture simulated in [42] is similar to other massively parallel processing elements, consisting of a single bit adder and various single bit registers. The synchronous processing element is surrounded by circuitry which implements the asynchronous data transfer using a two-phase bundled-data protocol.

The data transfer interface (DTI) consists of an asynchronous control block and the 3-state register element. The asynchronous control block generates the two-phase handshaking signals for the input and output of data, whilst the 3-state register stores the input data. The 3-state register is so called as each bit in the register has three states: logic zero, logic one and an empty state. The register clear signal is used to reset the register to the empty state, and the write enable signal is used to allow the input data to write to the contents of the register. When valid data has been written to the register, the register completion output signal is set.

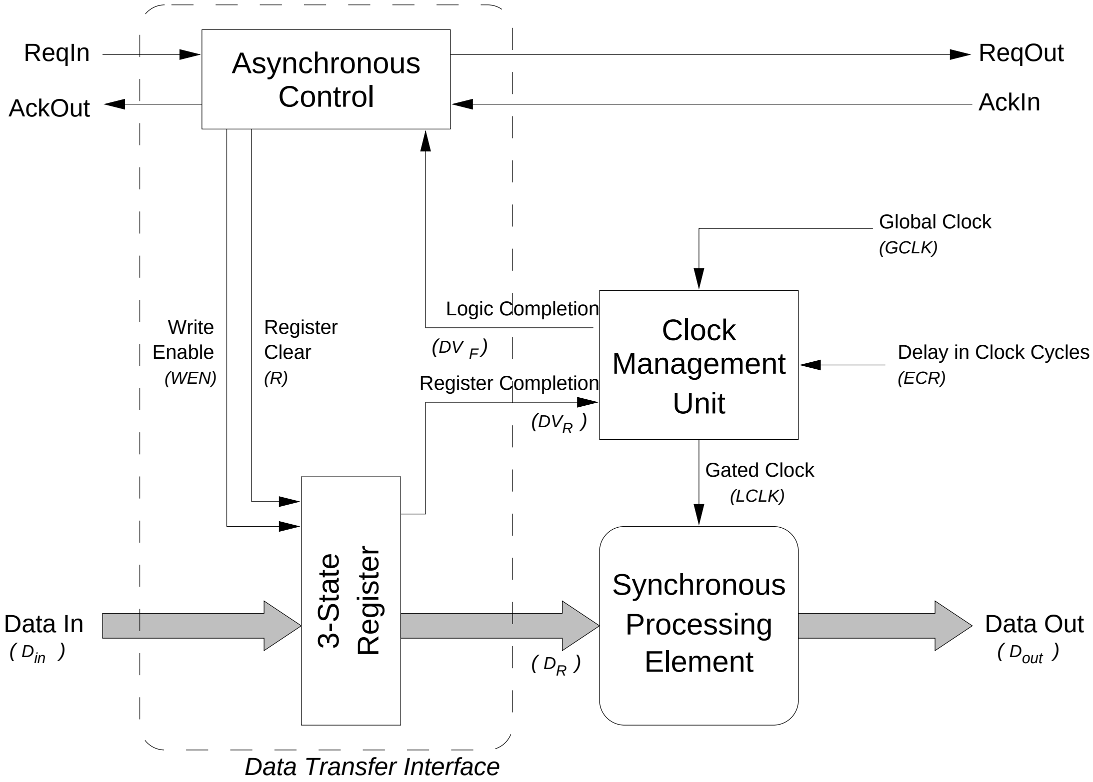


Figure 4.8: GALSA Module Structure

In operation, input data arrival is indicated by a transition on `ReqIn`. If the previous output data has been received (indicated by a transition on `AckIn`), the asynchronous control block write enables the 3-state register. Once valid data has been established in the register it is indicated by the setting of the register completion signal which feeds to the clock management unit.

The clock management unit provides a gated clock from a global clock source (`GCLK`). The local gated clock is enabled when valid data has been received on the data inputs, as indicated by the register completion signal. Since the register completion signal is asynchronous to the global clock, a synchroniser is required to overcome the meta-stability that can arise. Since, the global clock cannot be stopped of stretched in anyway, there is no way to wait for the potential meta-stable state to resolve itself, thus there is always a small chance of meta-stability occuring. A central aspect of Gao's design is minimising the chance of this occuring.

In addition to generating the local clock, the clock management unit generates the logic completion signal to indicate that the processing element has completed processing. The number of clock cycles that this takes is determined by an input to the clock management unit from a field in the ECR (Execution

Code Register) in the processing element. The ECR contains the instruction that the processing element is executing together with the number of clock cycles that it takes to produce a result. The logic completion signal is passed back to the asynchronous control block, which initiates the output handshake and resets the 3-state register to the empty state.

A feature of Gao's interface described in [42], and elaborated in [43] is that it contains no reconfigurable elements, except for the programmable delay in the ECR. All configuration is via the routing network. Since the elements are not reconfigurable, the approach for the processing element can be considered as a different style of 'asynchronous wrapper' that is placed around a synchronous core. The principal difference from the Asynchronous Wrappers work at Imperial [15, 14] is that the global clock is gated rather than generated locally.

4.6 Summary

This chapter has set the agenda for the rest of the thesis. The motivating factors for self-timed circuits on FPGAs have been discussed, in particular, the benefits for self-timed dynamic hardware systems. The idea central to the thesis is that the speed-independence of self-timed circuits supports dynamic reconfiguration, since the shape of the circuit and its operating environment can be changed and the self-timed circuit will adapt, unlike in synchronous systems where the clocking period may have to be changed.

Furthermore, it was argued that current FPGA architectures design for synchronous circuits do not allow the full advantages of self-timed circuits to be exploited, due to problems with hazards, signal re-ordering and arbitration. Asynchronous FPGA architectures attempt to overcome these problems; the two currently proposed architectures, MONTAGE and PGA-STC were discussed. The GALSA architecture for massively parallel processing was also discussed. The GALSA architecture illustrates an alternative approach to converting a synchronous architecture for asynchronous operation.

Chapter 5

STACC

5.1 Introduction

This chapter introduces a new model for self-timed FPGA architectures called STACC (Self-Timed Array of Configurable Cells). An objective of the model is that it is suited towards the implementation of self-timed dynamic hardware systems. This is an important difference from MONTAGE and PGA-STC, which are primarily intended for prototyping applications.

Before presenting the model, a number of the design decisions for STACC are discussed in Section 5.2. In this section, it is argued that dynamic hardware applications, in contrast to prototyping applications, favour an architecture dedicated towards a particular protocol. Subsequently, the reasons for choosing a bundled-data protocol in STACC are considered. Finally, before introducing the STACC model, the various ways in which synchronous architectures can be adapted to self-timed operation are considered.

The STACC model is introduced in Section 5.3. STACC involves replacing the global clock of a synchronous FPGA with an array of timing cells that provide local timing control. To illustrate the use of the timing array, example configurations using a simple STACC architecture are provided in Section 5.4. This chapter does not discuss the implementation of the timing array. This is the subject of the following three chapters.

5.2 STACC Design Decisions

5.2.1 Architectures and Protocols

Given the wide range of self-timed protocols, a major decision in the design of a self-timed FPGA architecture is whether to support the implementation of a wide variety of protocols, or whether to specialise the architecture for a particular self-timed protocol.

Of the architectures discussed in the previous chapter, MONTAGE is the least specialised. It is designed for the implementation of both delay-insensitive and bundled-data circuits, and also includes two global clocks for the implementation of synchronous circuits as well. However, the lack of dedicated delay elements makes MONTAGE more suited towards delay-insensitive circuits. PGA-STC is a more specialised architecture; it is designed for implementing two-phase bundled-data systems. Both the choice of programmable delay element and arbiter block are designed for two-phase operation. Though the elements are chosen for two-phase operation, the routing architecture does not force the elements to be used in this way, so four-phase and delay-insensitive circuits can be constructed in the PGA-STC architecture.

The decision whether to have a general purpose or specialised architecture is largely motivated by the intended application for the FPGA. Prototyping applications suit an architecture that can implement a large variety of protocols, so that only one type of FPGA is required to implement all the different self-timed protocols. Specialisation to a particular protocol limits the style of circuits that can be implemented, but it does not limit the functionality that can be implemented, since equivalent circuits can be implemented using different protocols. Also, it allows the architecture to be optimised for a particular protocol; very much as today's commercial FPGAs are optimised for the implementation of synchronous circuits.

In particular, specialisation to a specific protocol is unimportant, when the application for the self-timed FPGA is dynamic hardware rather than prototyping. As long as the architecture facilitates the construction of self-timed circuits that can easily be manipulated by the dynamic hardware management software, then the actual protocol used is not critical. Since, the focus of this thesis is on the benefits of self-timing for dynamic hardware, a specialised architecture is considered.

5.2.2 Protocol Choice

Having decided to examine a self-timed architecture dedicated to the implementation of a specific protocol, this section considers the choice between specialising the architecture for delay-insensitive circuits or bundled-data circuits.

Delay-insensitive protocols have favourable properties for dynamic hardware systems; once self-timed elements have been built that contain isochronic forks, the circuits are resilient to arbitrary delays that may be introduced by the routing. However, delay-insensitive circuits commonly use dual-rail encoding, which requires two wires to encode one bit of data. This overhead is highlighted by McAuley [84], who designed circuits using just one type of cell: a delay-insensitive two-input multiplexor. He reports an area overhead of between two to six times depending on the design.

The area overhead of bundled-data protocols is determined by the width of the data bundle. For the worst case of a data bundle of one signal, the overhead is the same as for dual-rail encoding. Kean considered a self-timed cell for the CAL architecture (Section 2.4.1 in [65]), but rejected it because initial designs used three times the area of a synchronous design. This large overhead results from trying to self-time single bits in the architecture. By increasing the number of bits in the data bundle, the area overhead reduces rapidly, since the overhead is inversely proportional to the number of bits in the data bundle. However, the architecture cannot force the data bundle to be too large, since many signals will be left unused in large bundles.

An additional benefit of bundled-data protocols is their similarity to synchronous designs, since the data path of a bundled-data circuit is the same as that for an equivalent synchronous circuit. This allows designers to move readily from synchronous design to bundled-data design. Also, design tools for synchronous circuits can often be applied to the design of bundled-data circuits.

Due to the overheads of delay-insensitive designs, the decision was made in this work to concentrate on an architecture dedicated towards the implementation of bundled-data systems.

5.2.3 Self-Timing Synchronous Architectures

Both MONTAGE and PGA-STC are derived from synchronous FPGA architectures; MONTAGE from TRIPTYCH, and PGA-STC from the Xilinx XC4000. Self-timing a synchronous FPGA architecture is advantageous, since it allows many of the tools and much of the design experience of the synchronous architecture to be carried over to the self-timed architecture.

Although, both MONTAGE and PGA-STC derive their architectures from synchronous forebears, their designers do not explicitly consider the methodology in which the architectures have been made asynchronous. However, both architectures perform similar transformations on the synchronous architecture. Both change the design of the function block to aid the implementation of asynchronous circuits, and both also completely replace some function blocks with special arbitration blocks. The clock signal is either removed (PGA-STC) or not used (MONTAGE). The drawback to tinkering with the function blocks is that it makes the translation of designs and experience from the synchronous architecture more difficult.

5.3 The STACC Model

For bundled-data systems, an alternative approach from the one adopted by MONTAGE and PGA-STC is possible for deriving a self-timed architecture from a synchronous one. Bundled-data systems differ from their synchronous counterparts in how the register control signals are generated; synchronous systems use a global clock, whilst bundled-data systems use special circuitry to produce local register control signals. This suggests that, rather than scrapping the clock, a bundled-data FPGA architecture could replace it, with an array of *timing cells* that generate local register control signals. This model forms the basis of the self-timed FPGAs developed in this thesis, and is termed *STACC (Self-Timed Array of Configurable Cells)*.

The STACC model has several benefits. First, it does not alter the function blocks of the synchronous architecture as MONTAGE and PGA-STC do, so tools and experience from a synchronous FPGA architecture can be transferred to a self-timed version of the architecture. Second, all the self-timed control logic is contained within the timing array, hence the timing cells can be optimised for this task, rather than requiring cells capable of implementing both timing control and logic blocks. A final benefit is that the STACC approach to transforming a synchronous architecture has wide applicability. In fact, the original architecture need not be a FPGA, but could be any array of processing devices that are synchronised by a global clock.

Figure 5.1 illustrates the basic concept of the STACC approach. The logic blocks and routing of the synchronous architecture are retained and form the *data array*. Instead of a clock, timing control is implemented by cells in the *timing array*. Each timing cell provides register control (in other words, a local clock signal) to a region of cells in the data array. The shaded region in Figure 5.1 illustrates a group of data cells that are provided with a local clock from one of the cells in the timing array. The timing cell and the group of data cells that it controls form a *self-timed region*.

Within the timing array, routing is provided to connect the timing cells to-

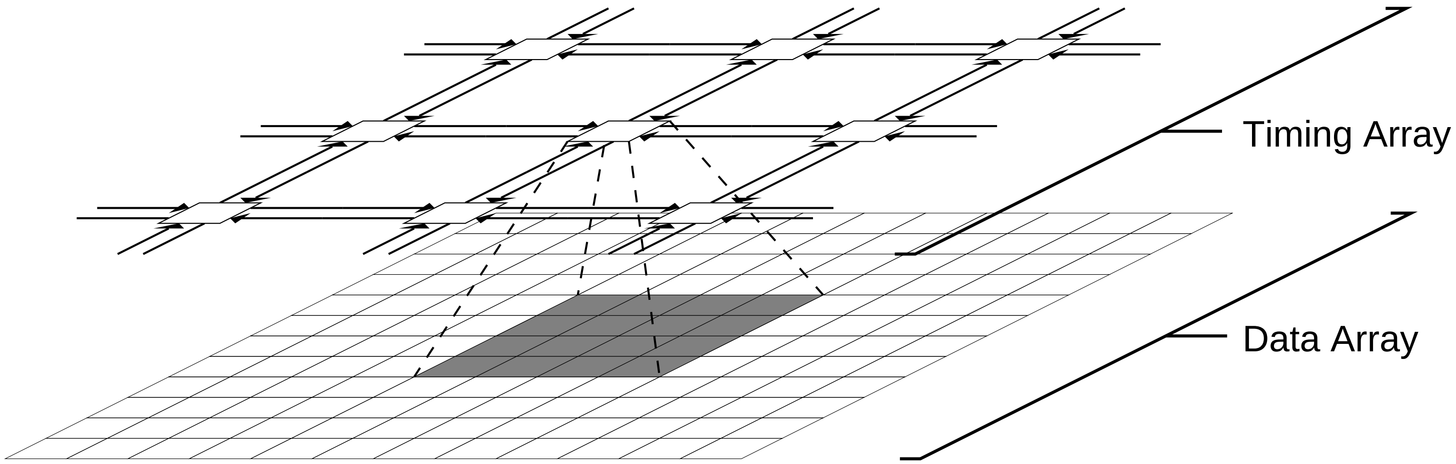


Figure 5.1: Basic Structure of the STACC Architecture

gether. In the example of Figure 5.1, a nearest neighbour mesh is used. Each timing cell is connected to its neighbours by two wires, one in each direction. These *handshaking links* are used to perform request/acknowledge handshakes with other timing cells. The timing array configuration determines whether timing cells joined by a handshaking link communicate, and the direction of the data flow between them. Thus, the configuration of the timing array reflects the pattern of data flow in the data array.

More complex patterns of data flow can be implemented by allowing the timing cells to sample values from the data array. These *select signals* allow conditional communication in the timing array, so that results from the data array can influence the flow of control. The arbitration function required by self-timed circuits is also integrated in the the timing cell. This allows the timing cell to decide which of several competing communications to service. In many cases, it is useful to provide the result of the arbitration to the data array, so that the data path can process data accordingly. To do this, *probe signals* are fed from the timing array to the data array.

A complete self-timed region is illustrated in Figure 5.2. The lower half of the figure illustrates the logic implemented by the data cells in the data array. The data array implements a Finite State Machine (FSM). The inputs to the FSM are data bundles from other self-timed regions, and probe signals from the timing cell. The outputs of the FSM are data bundles to other self-timed regions and select signals that control the pattern of communication in the timing array. The next state output of the FSM is fed back as an input to the logic via the register. Variations on this basic model of a self-timed region are possible depending on the relative position of the registers to the logic block.

The timing cell provides timing control for the FSM implemented by the data cells. The timing cell will clock the registers when all the input request signals and output acknowledges have been received on the handshaking links. After clocking the register, the timing cell acknowledges receipt of the input data bundles. The timing cell then waits a period of time dependent on the delay in the logic block. When the logic block has completed evaluating, the timing cell generates requests for the output data bundles. At this point the cycle of timing cell and FSM operation has completed, so the self-timed region can proceed to processing the next set of inputs.

The handshaking links from the timing cell in Figure 5.2 connect to the timing array routing. In the example of Figure 5.1, no timing array routing structures are present, the timing cells are simply connected directly to each

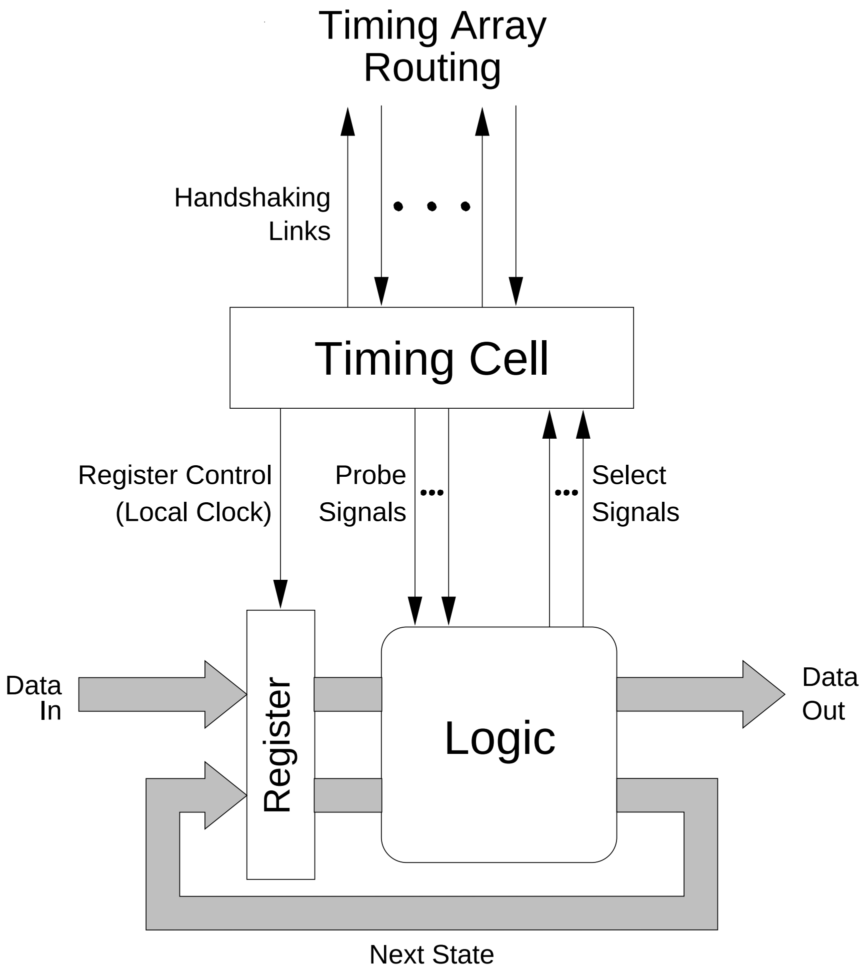


Figure 5.2: Self-Timed Region

other in a nearest neighbour mesh. However, in general, the timing cells would connect to *handshaking switchboxes*, which route the handshaking signals in the timing array to match the routing of the data bundles in the data array. The handshaking switchboxes allow one-to-one connection of handshaking links, but may also allow many-to-one or many-to-many connections of handshaking links, to match the fan-in and fan-out of data bundles in the data array routing.

The STACC model could loosely be classed as a GALS model, as it aims to preserve the synchronous data path structure but utilises asynchronous communication. However it differs from other models, such as GALSA [43, 42] and Asynchronous Wrappers [14, 15], in the more general way it provides interaction between the data path and control path.

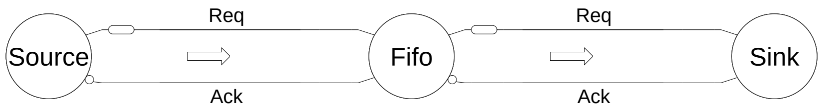
In GALSA, no provision is provided for interaction between the data path and control path; there is no way for the data path to influence the pattern of control flow. The Asynchronous Wrappers approach does include provision to allow the data path to influence the pattern of control flow using the Port Select Block (see Figure 1 in [15]). However, Asynchronous Wrappers currently have no provision for arbitrating between competing requests. The STACC model (first described in [96]) provides both mechanisms. The data path can influence the pattern of control flow via the select signals. Additionally, the STACC model allows competing requests to be arbitrated between and then for the result to be passed from the timing cell to the data cells via the probe signals.

Another key difference of STACC from Wrappers and GALSA is that the behaviour of the timing cell is configurable. Even though GALSA is a configurable array architecture, the data transfer interface in GALSA is fixed. All the reconfigurability arises from configuring the routing between processing elements and not in the GALSA wrapper itself. Another difference of STACC from GALSA is that in GALSA there is a one-to-one relationship between processing elements and timing control blocks. The STACC model uses one timing control block to control many processing elements.

5.4 Example Timing Array Configurations

This section illustrates how the timing cells can be used to control a wide range of data flow patterns in the data array. The issue of how the timing cell is designed to implement all these possible configurations is left for later chapters.

The configuration of the timing array determines how the timing cells com-



cell	name	link	DC	DIR
1	Source	East	0	1
2	Fifo	West	0	0
		East	0	1
3	Sink	West	0	0

All other links: DC=1; DIR=X;

Figure 5.3: Pipeline and Configuration Data

municate using the handshaking links. The timing array configuration has to match the pattern of data flow configured in the data array. Two key aspects of the data flow along a handshaking link have to be determined. First, whether the two timing cells do communicate with each other. This can implemented by a single configuration bit `DC` (Don't Care), which determines whether a timing cell communicates on a handshaking link or not. The second aspect of a link that has to be determined, is the direction of the data flow that it controls. This can be implemented with a second configuration bit `DIR` (Direction) which determines the direction of communication along a handshaking link. In other words, `DIR` determines which signal in a handshaking link is a request and which is an acknowledge.

These two configuration bits are sufficient to implement a large number of fan-in and fan-out data flows. Figure 5.3 shows how the timing cells can be configured to build a pipeline, using a simple nearest neighbour mesh of timing cells (as illustrated in Figure 5.1). The elements in the figure are designed to resemble the timing control circuitry for a two-phase bundled-data pipeline, as used in Micropipelines. The core of the timing cell is a C-Muller gate, represented by a circle in these diagrams. Instead of being marked with a 'C', the gate is marked with a name that describes the data path operation performed by the data cells in the self-timed region. Request signals from the C-Muller gate are marked with oval shapes representing delay elements. Acknowledge signals, which are inverted with respect to the request signals in Micropipelines, are marked by the bubbled inputs to the C-Muller gates. The direction of data flow is emphasised by the inclusion of the large arrows in the figure.

A feature of Figure 5.3 is a lack of connections to the environment external to the FPGA. Though such connections can easily be made, structures can be tested simply by configuring cells to model the environment. Timing cells that are configured with only output data flows are termed *source cells*, and cells configured with only input data flows are termed *sink cells*.

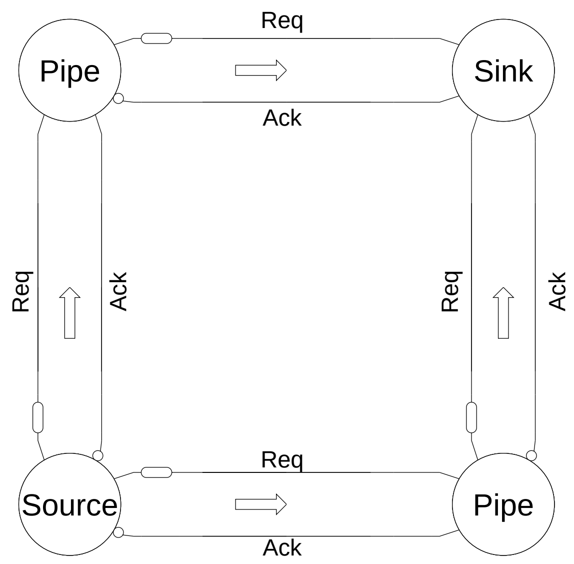


Figure 5.4: Forking and Joining Pipeline

The timing cell configuration bits described so far are flexible enough to implement a large range of forking and joining pipelines, including structures such as the 2D-Micropipelines described by Gopalakrishnan [46]. A simple example of forking and joining pipelines is illustrated in Figure 5.4. In this example, the source cell in the bottom left-hand corner sends data into two different pipelines which process the data concurrently. The sink cell in the top right-hand corner receives data from both pipelines. The choice of an example with a symmetrical fan-out and fan-in of data was made to illustrate the symmetry in how the timing cell deals with the fan-in and fan-out of handshaking links. A similar structure is used in the other examples in this section to show the symmetry between structures configured using fan-in and fan-out handshaking links.

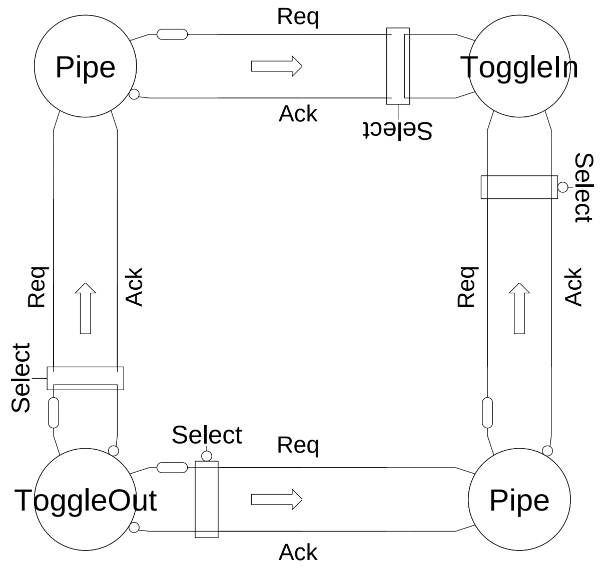


Figure 5.5: Selective Communication Example

The examples so far have only allowed a fixed pattern of data flow. To allow the branching and merging of data flows involves using the select signals from the data array to change the flow of control in the timing array. Figure 5.5 shows an example of selective communication. Selective links are indicated by the drawing of select boxes on the links. The input to the select box can be inverted; this is indicated by placing a bubble on the select input. When the handshaking link is selected, the handshaking link is connected normally to the neighbouring timing cell. When the link is not selected, the output handshaking signal is fed directly back to the timing cell, so the C-Muller gate acknowledges its own request. This select box performs a similar role to the Select gates in Micropipelines. For selective communication, the `DC` configuration bit can be replaced by `RDZ` (Rendezvous) configuration bits which determine whether the link is never used by a timing cell, always used, or selectively used.

The example illustrated in Figure 5.5 is similar to the forking and joining pipelines of Figure 5.4, but instead of sending data down both pipelines, the `ToggleOut` cell sends data down different pipelines on alternate cycles, since the select signal of one link is inverted with respect to the other. The `ToggleIn` cell performs the opposite process to the `ToggleOut` cell; it accepts data from each pipeline on alternate cycles. This example illustrates how work could be shared between two pipelines that perform the same function, so potentially doubling throughput.

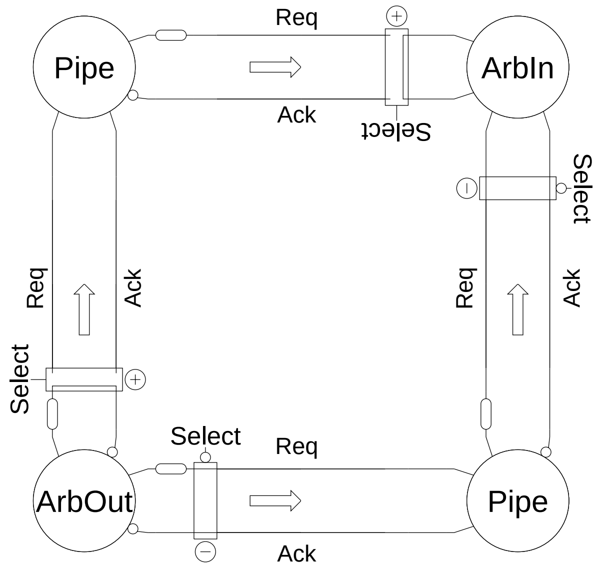


Figure 5.6: Non-Deterministic Communication Example

The branching and merging in the previous example was deterministic. Figure 5.6 illustrates a refinement of the work sharing pipelines of the previous example, where the timing cells arbitrate between competing communications. Instead of sending work down alternate pipelines as previously, the `ArbOut` cell probes the acknowledge signals of both output links to see which are ready to process data. The probing behaviour is shown by the plus or minus signs in circles, which indicates whether the acknowledge signal causes the arbitration function to resolve to '0' (minus) or '1' (plus). The result of the arbitration is used as an input to the select boxes. The result of arbitration is also passed to the data array via the probe signals, so it can act accordingly. The `ArbIn` cell implements the opposite function to the `ArbOut` cell. It probes the request signals of it incoming links and accepts data from whichever one has data ready. This is similar to the use of an arbitrated Merge gate in Micropipelines.

This example shows how work can be dynamically distributed between processes using the probing behaviour. In contrast to the deterministic behaviour of the last example, this arrangement can re-order data, depending on the delays encountered in each pipeline.

5.5 Summary

This chapter introduced the STACC model for self-timed FPGA architectures, which will be developed in the rest of the thesis. STACC differs from previous asynchronous FPGA architectures in two important respects. First, it is designed with dynamic hardware systems in mind, rather than exclusively for prototyping. Second, rather than scrapping the global clock, it replaces it with an array of timing cells. The generality of this approach allows it to be applied to a variety of synchronous FPGA architectures, and also to architectures other than FPGAs.

The basic unit in a STACC architecture is a self-timed region composed of a timing cell and the data cells that it provides timing control for. The timing cells and data cells interact via the select signals which allow the data array to influence the flow of control, and the probe signals which pass the results of arbitration to the data cells. A number of examples in the chapter illustrated how the timing array could be configured for a wide range of data flow patterns.

The actual implementation of the timing array has not been discussed. This is the subject of the following three chapters. The next chapter looks at the design of basic reconfigurable circuit elements. These reconfigurable elements are used in Chapters 7 and 8, which discuss the design of the timing cells and timing array routing respectively.

Chapter 6

Reconfigurable Elements

6.1 Introduction

Bundled-data control circuits, such as those used in Micropipelines, consist of three basic elements: a synchronisation element (normally the C-Muller gate), elements for the branching and merging of control, and delay elements to ensure that the bundling constraint is met. This chapter considers the implementation of these three types of element reconfigurably for use in a self-timed FPGA architecture. Though the chapter focusses on the design of elements for self-timed FPGA architectures, many of the elements introduced seem useful enough to find other applications in asynchronous circuit design. No attempt is made in this chapter to integrate these elements to construct a timing array for STACC. This is left to the following two chapters, which discuss the design of the timing cells (Chapter 7) and the timing array routing (Chapter 8).

6.2 Reconfigurable C-Muller Gates

The C-Muller gate forms the basic synchronisation structure in most self-timed circuits. This section introduces the *reconfigurable C-Muller gate*, which allows a reconfigurable synchronisation pattern to be defined between a set of inputs. In effect, an N input reconfigurable C-Muller gate allows all C-Muller gates of N inputs or less to be implemented between an arbitrary subset of its inputs.

6.2.1 Gate Level Implementation

Figure 6.1 shows a gate level implementation of a reconfigurable C-Muller gate. One reconfigurable input is shown, configured by the configuration bit `DC` (Don't Care). When `DC` is false, the multiplexor passes the input normally,

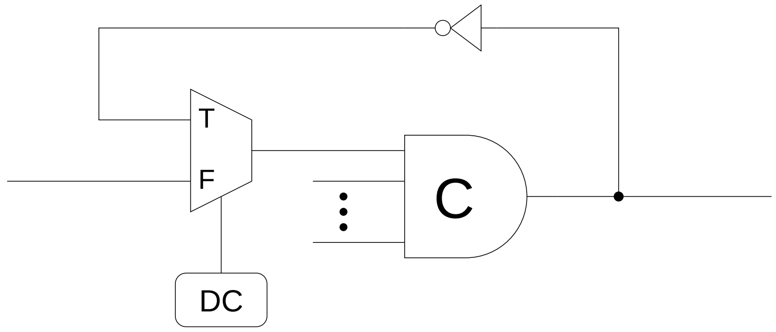


Figure 6.1: Reconfigurable C-Muller Gate: Gate Level Implementation

so the input is involved in the gate's synchronisation. However, when `DC` is true, the multiplexor passes the inverted output of the C-Muller gate back to itself. Since the inverted output of a C-Muller gate is always the next value that the C-Muller gate is waiting to synchronise on, the input becomes a don't care input. Hence, the `DC` configuration bit determines whether or not the input is involved in the gate's synchronisation.

The definition of the reconfigurable C-Muller gate leaves undefined its behaviour when no inputs are synchronised by it (i.e. when all the inputs are configured as don't cares). In this case, the behaviour of the gate is implementation dependent. For the gate level implementation in Figure 6.1, the gate's output oscillates.

The reconfigurable C-Muller gate requires far fewer configuration bits than using a general purpose function block. For example, a LUT based implementation of an N input C-Muller gate requires an $N + 1$ input LUT using 2^{N+1} configuration bits. The reconfigurable C-Muller gate only requires N configuration bits, i.e. one per input.

An alternative way of implementing the behaviour of the reconfigurable C-Muller gate could be achieved by routing. A C-Muller gate synchronising on less than N inputs could be created from an N-input C-Muller gate by routing duplicates of the input signals to make up the N inputs. However, this approach is less satisfactory as it places additional load on the input signals, and hence increases the transition times on the duplicated inputs. Another drawback to duplicating inputs is that it requires more configuration bits, since the same synchronisation pattern can be configured in several different ways.

6.2.2 Transistor Level Implementation

Figure 6.2 illustrates a transistor level implementation of a reconfigurable C-Muller gate. The circuit is based on a standard implementation of a C-Muller gate that uses a weak feedback inverter to maintain the gate's state [25]. Two multiplexors have been added which are controlled by the `DC` configuration bit. When `DC` is false the input is passed normally to the N-tree and the P-tree of the transistor structure. When `DC` is true, the inputs to the N-tree and P-tree transistor chains are connected to power and ground respectively, so that the transistors in both trees are always turned on. Hence the input has no effect on the gate's switching and becomes a don't care input.

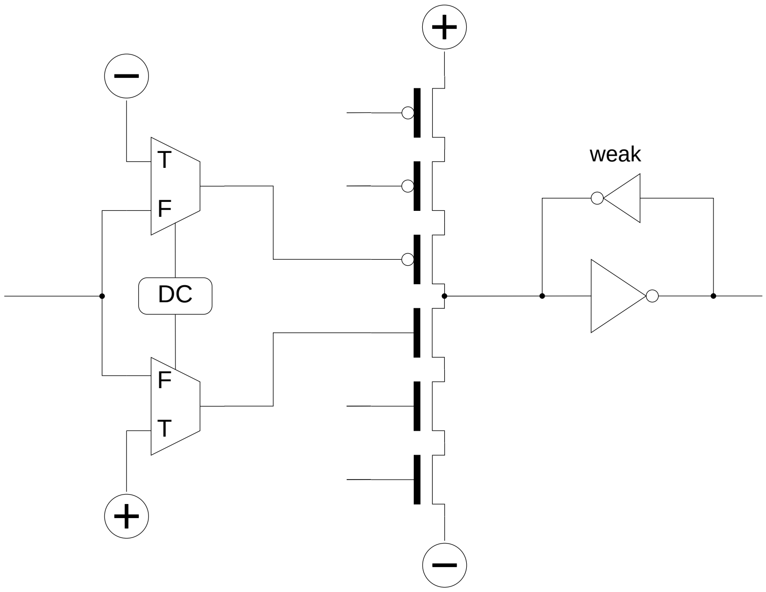


Figure 6.2: Reconfigurable C-Muller Gate: Transistor Level Implementation

The transistor level design is preferable to the gate level design as it does not have a feedback path from the C-Muller gate's output to its inputs. In the gate level design, this results in each input to the C-Muller gate going through a transition on each cycle, whether the input is involved in the synchronisation or not. In the transistor level design, these inputs are held at constant values so there are fewer transitions, and hence less power dissipation.

A problem with the transistor level implementation occurs when all the inputs are configured as don't care, which results in power being shorted to ground. To prevent this, one input should either be unreconfigurable or use the feedback implementation of the gate level design.

6.2.3 Reconfigurable Asymmetric C-Muller Gates

Figure 6.3 illustrates how the transistor level implementation of the reconfigurable C-Muller gate can be generalised to a *reconfigurable asymmetric C-Muller gate*. The `DC` configuration bit is separated into two configuration bits `DC0` and `DC1` that allow synchronisation on the input being logic zero and logic one respectively. Hence, each input can be configured to synchronise on only a logic one input, on only a logic zero input, on both, or on none (the don't care case).

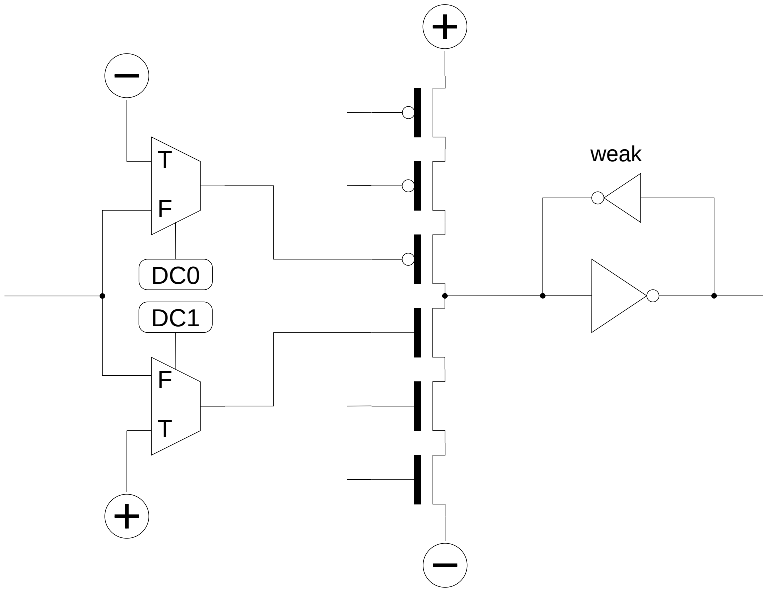


Figure 6.3: Reconfigurable Asymmetric C-Muller Gate

6.2.4 Distributed Reconfigurable C-Muller Gates

The process technology used for implementing transistor circuits limits the number of transistors that can be placed in series, usually to around four [121]. This in turn limits the fan-in of the basic C-Muller gate implementation. Wider fan-in C-Muller gates can be made by creating a tree of C-Muller gates, as in Figure 6.4. The root C-Muller gate synchronises on the outputs of the leaf C-Muller gates, which synchronise on the input signals.

Figure 6.5 illustrates another circuit for implementing wide input C-Muller gates. Figure 6.5(a) shows an implementation of a C-Muller gate using a SR flip-flop and two AND gates. The AND gates are used to detect when all the inputs are one, and when all the inputs are zero. These conditions are used to set and reset the flip-flop respectively. Wired logic can be used to implement the wide fan-in AND gates to create a wide fan-in C-Muller gate [103]. In wired

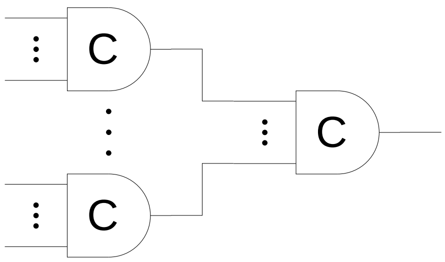
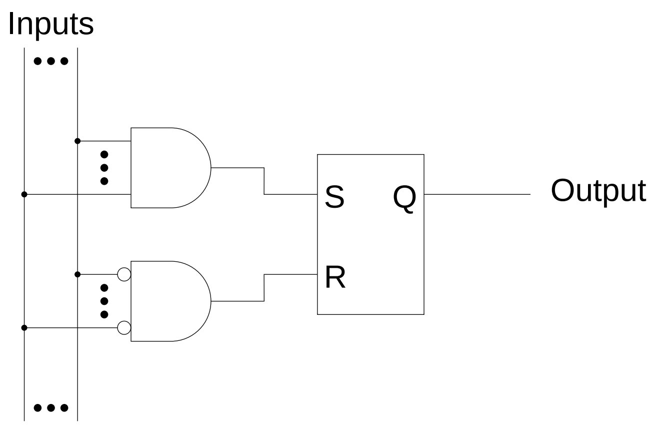
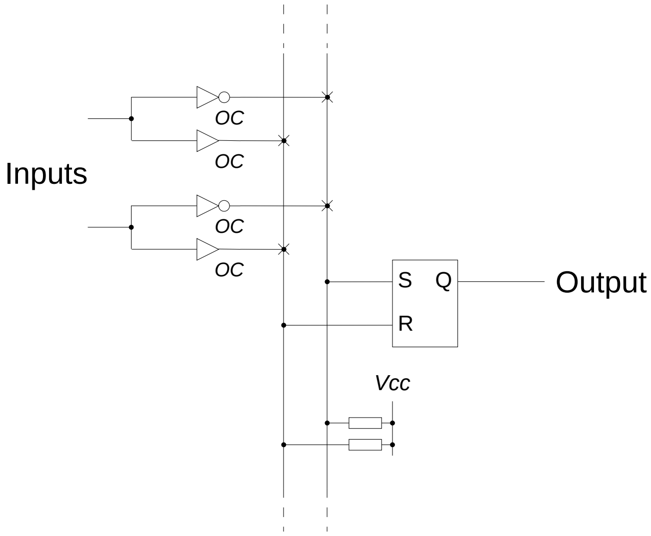


Figure 6.4: Wide Fan-In C-Muller Gate



(a) C-Muller Gate implemented using SR Flip-flop



(b) Reconfigurable Distributed C-Muller Gate

Figure 6.5: Distributed Reconfigurable C-Muller Gate Implementation

logic, open-collector (or open-drain) drivers are used to drive the inputs; the AND state is detected when all the open collector drivers stop driving the wire, which is then pulled high by a weak pull-up resistor.

Figure 6.5(b) illustrates a reconfigurable C-Muller gate implemented using wired logic. The two AND gates of Figure 6.5(a) are replaced by two synchronisation wires, which detect synchronisation to logic one and logic zero. In effect, the logic for the C-Muller gate is distributed over the two synchronisation wires, hence this implementation is termed a *distributed C-Muller gate*.

In contrast to previous reconfigurable C-Muller gate implementations, Figure 6.5(b) does not use an SRAM configuration memory. Instead, the `DC` configuration bits are replaced by fuses, marked by the crosses in the figure. When the fuses are intact, the inputs are involved in the synchronisation. When the fuses are blown, the outputs cannot drive either synchronisation wire so the input becomes a don't care input. Asymmetric gates can be made by only blowing one of the fuses, so only the appropriate synchronisation line is used. For SRAM based configuration memories, the distributed C-Muller gate can be implemented by using the configuration bits to act as enable signals for the open-collector drivers.

A fuse based implementation was illustrated, since it suits a wired logic implementation. The low resistance of fuses and their bidirectional transmission of signals, allows wire segments to be connected to form long low resistance wires. SRAM based configuration memories have to use pass transistors which have a higher resistance.

The advantage of the distributed design is that it allows wide fan-in C-Muller gates to be implemented using only two wires. Effectively, the logic is implemented in the routing. Furthermore, the distributed design can save routing resources, as instead of routing all the inputs to a single centralised gate, only the two synchronisation wires are routed to the inputs and the output SR flip-flop. However, a drawback to the distributed design is that wired logic has a slow transition time. Also, wired logic designs have static power dissipation, when the open collector drivers are switched on. Finally, careful design is required to avoid problems with transmission line effects.

6.3 Branch and Merge Elements

Current self-timed methodologies use a diverse range of control blocks [54]. Creating a reconfigurable control block capable of implementing all these variants is a difficult task. However, fundamentally these control blocks are only providing two functions: the branching and merging of control.

This section develops a reconfigurable control block for branching and merging. The approach taken is to restructure the control blocks introduced by Sutherland for Micropipelines (see Figure 3.10). Though Micropipelines use a two-phase protocol, Sutherland's control blocks can be implemented using a four-phase protocol as well. Hence, the development here can be applied to produce both two-phase and four-phase reconfigurable elements.

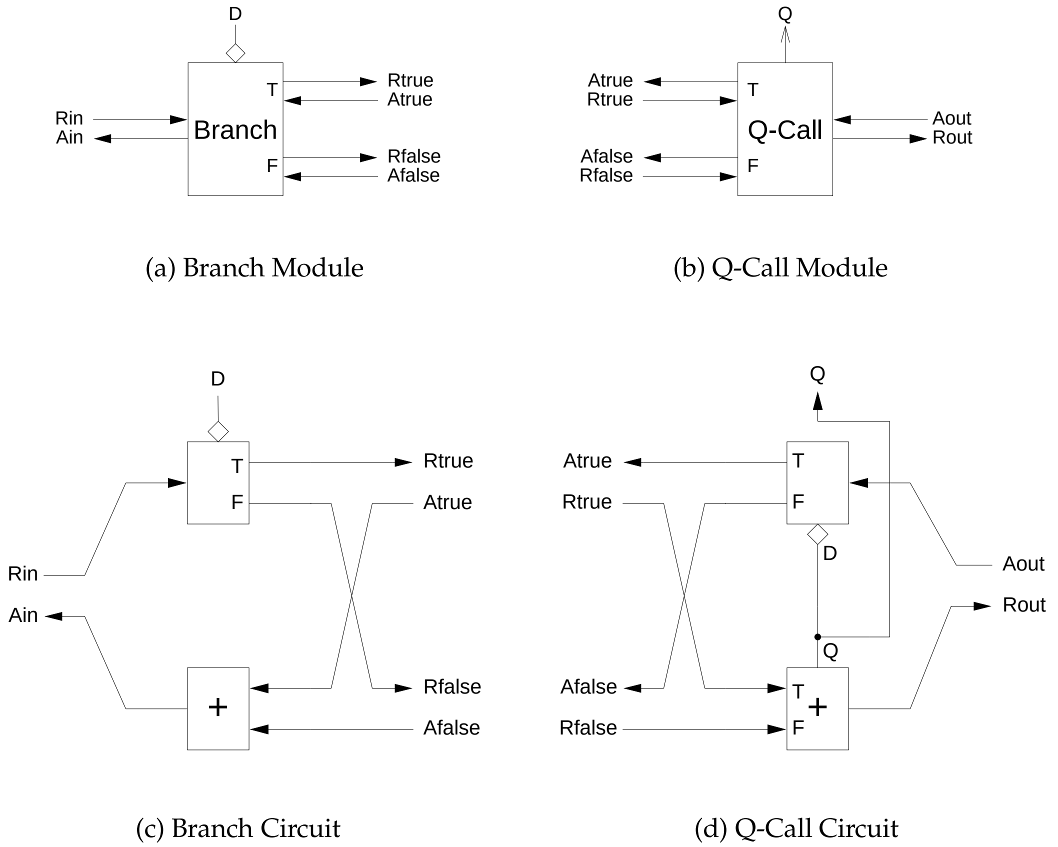


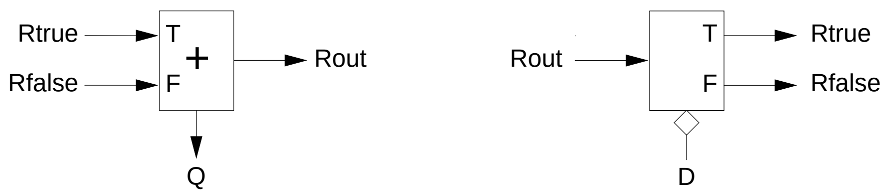
Figure 6.6: Branch and Q-Call Modules

Examining the control modules used by Sutherland [111], there is a lack of regularity and symmetry, which Sutherland is keen to point out in the rest of his Micropipeline paper. In particular, there is a difference between the Merge, Select and Toggle gates, which operate on individual handshaking signals, and the Call and Arbiter modules which operate on handshaking channels (i.e. request/acknowledge handshaking pairs).

This suggests that one way to standardise the control modules is to make all the modules operate on handshaking channels. To standardise these modules, the original Select, Merge or Toggle gate can be used to generate the request signal path and then additional behaviour is defined for the generation of the acknowledge signals.

Figure 6.6(a) illustrates the extension to the Select gate to operate on handshaking channels; this is termed a *Branch* module. Like the Select gate, the Branch module accepts an incoming request and directs it to one of its two output channels depending on the value of the Select input D. When the selected output channel generates an acknowledge, this is directed back to the input channel. The implementation of the Branch module is illustrated in Figure 6.6(c). As would be expected, a Select gate is used to generate the output request signals. The input acknowledge signal is generated by using a Merge gate.

Extending the Merge gate to operate on handshaking channels results in a module where incoming requests are merged into one channel and the output acknowledge is directed back to the calling channel. This behaviour is already defined by Sutherland as the Call module. The Call module is shown in Figure 6.6(b), and Figure 6.6(d) shows its implementation. The output request path is generated by a Merge gate. For the acknowledge path, a gate is required that steers the output acknowledge back to the calling input channel. This can be implemented using a Select gate if there is a signal which indicates which input channel called the module.



(a) Q-Merge (b) Select

Figure 6.7: Q-Merge and Select Gates

One way to generate this signal is to extend the behaviour of the Merge gate, so it generates an additional output Q, that indicates on which of its inputs the last event occured. The name *Q-Merge* is introduced for such an element. The Q-Merge gate is illustrated in Figure 6.7(a). The Call module of Figure 6.6(b) has also been extended to provide Q as an output from the module, and is termed a *Q-Call* module. Supplying Q as an output is useful as it allows the data path to know which calling channel is being serviced.

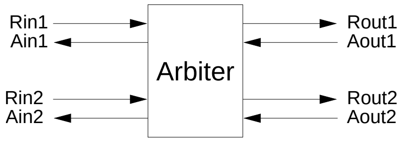
A reflective symmetry exists between the Select and Q-Merge gates, as can be seen in Figure 6.7. In fact, the two gates are inverses of each other. The Select gate can be considered as a converter from the bundled-data signals `Rin` and `D` to the dual rail encoded signals `Rtrue` and `Rfalse`. The Q-Merge element performs the inverse operation; it converts from the dual-rail encoded signals to the bundled data signals. The Branch and Q-Call modules are also inverses of each other. This is reflected in the symmetrical structure of the two modules, as shown in Figure 6.6.

Useful behaviour also arises when the direction of the data flow through the Q-Call and Branch modules is reversed, i.e. pull handshaking channels are used instead of push handshake channels. In the case of the Q-Call module, the module now arbitrates between competing requests for data, which is pulled from the output channel. This behaviour can be used to farm data between pipelines. When used in reverse, the Branch module allows an input channel to choose which of two output channels it wants to pull its data from.

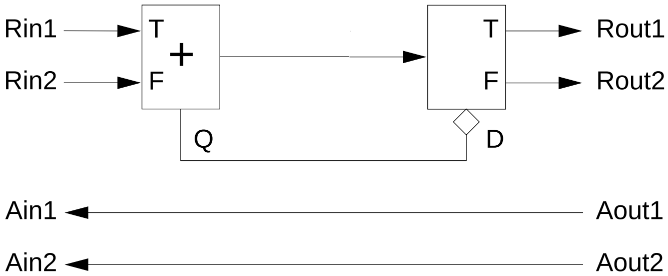
The fact that both Q-Call and Branch modules can be constructed using one Select and one Q-Merge gate suggests that this pair of gates would be a good candidate for a reconfigurable control block. It can be seen in Figure 6.6, that to configure from a Branch module to a Q-Call module requires only one configuration bit that determines whether the `Q` and `D` signals are connected or not.

The Select and Q-Merge gates can be used to build the other modules used by Sutherland. Figure 6.8 shows how a Select and Q-Merge pair can be used to construct an Arbiter module. Since events on the two inputs can occur simultaneously, the Q-Merge gate has to be capable of arbitrating between events on the two inputs. One difficulty in the circuit occurs when the arbitrating Q-Merge element determines the beginning of the next arbitration phase. A four-phase Q-Merge element can use the recovery phase on the serviced input request line to determine this. However, this is not an option in the two-phase protocol, so an additional signal needs to be supplied from the acknowledge signals to the Q-Merge element to specify when to accept the next input request.

The other module used by Sutherland is the Toggle gate. Figure 6.9 shows a *Toggled-Branch* module; an extension of the Toggle gate to operate on handshaking channels. The `Q` output of the Q-Merge gate is used to remember which output channel was selected, so that the other channel is selected on the next input request.

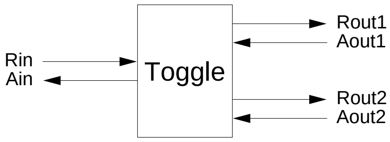


(a) Arbiter Module

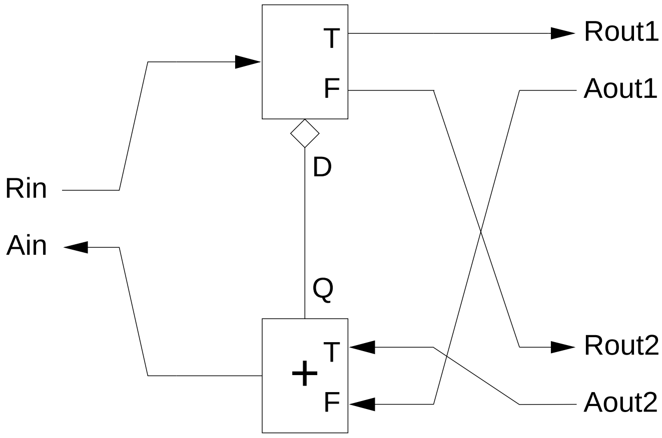


(b) Arbiter Circuit

Figure 6.8: Arbiter Module

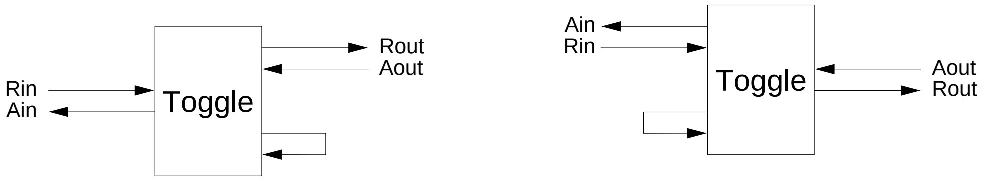


(a) Toggled-Branch Module



(b) Toggled-Branch Circuit

Figure 6.9: Toggled-Branch



(a) Divide-by-Two (b) Times-by-Two

Figure 6.10: Times-by-Two and Divide-by-Two Modules

Figure 6.10 shows how the Toggled-Branch module can be used to create Times-by-Two and Divide-by-Two modules. These are useful modules since they convert between two-phase signalling and four-phase signalling. The Divide-by-Two module operates by directing alternate transitions down a channel, which connects directly back to the Toggle module. As a result, only alternate transitions on `Rin` causes a transition on `Rout`. However the module is not the best converter from four-phase to two-phase since it has to wait for the two-phase acknowledge before the four-phase cycle can continue; whilst it is possible to overlap parts of the two-phase and four-phase cycles. Gopalakrishnan [47] considers circuits that do overlap parts of the two cycles. The Times-by-Two module is constructed in the same way as the Divide-by-Two module but is simply used in the reverse direction to convert from two-phase to four-phase.

6.4 Delay Elements

Delay elements are required in bundled-data circuits to ensure that the request signal is asserted after its associated data is valid. Also, for performance, the delay of the request relative to the data should be as small as possible. Delay elements can be classified according to how closely they match the request delay to the data delay. Fixed delay elements simply ensure that the bundling constraint is met, whilst variable delay elements match the delay to the current computation.

Fixed delay elements must be set at the worst case delay through the logic to ensure correct operation. For a simple pipeline, this restricts performance to that of the slowest stage in the pipeline; the same as for a synchronous pipeline. However, for more complex data flows, the difference in delays down different branches in the data flow can be utilised, which gives benefits over the synchronous case.

Variable delay elements allow the average case performance to be utilised by matching the request delay to each individual computation. Such elements work by generating a completion signal by some means from the data path. The generation of completion signals requires more complex circuitry than generating a fixed delay, and often requires substantial alteration to the structure of the data path.

6.4.1 Delay Generation

Below, various delay generation strategies for reconfigurable architectures are considered. Essentially these options for the delay element represent a trade-off between performance and complexity of the delay element.

Uniform Fixed Delay

The simplest delay strategy is to have an unreconfigurable fixed delay for all self-timed regions in a self-timed FPGA architecture. The worst case delay of any self-timed region would be calculated during the architecture design and then implemented in the delay element.

The easiest way to create such a fixed delay is to use an inverter, or a chain of inverters for a longer delay. The delay characteristics of an inverter can be modified by varying the dimensions of the transistors in the inverter, and varying the capacitive load it has to drive. Even though we have termed it as a fixed delay, the delay that an inverter gives is not constant since it can vary because of variations in temperature and voltage level. However, these variations also effect the logic delay accordingly.

Fixing the delay for all self-timed regions gives an extremely poor performance, as no attempt is made to utilise variable delay within individual data paths. The performance of a circuit is limited by the worst case delay of the architecture rather than the circuit.

Reconfigurable Fixed Delay

An alternative to a uniform delay for all self-timed regions in the architecture is to allow configuration data to set the delay of the delay element. One way to implement this is to allow configuration data to select between various taps off an inverter chain. If delays finer than one gate delay are required, then configuration data could choose between inverters with different delay characteristics. An alternative method of generating delays finer than one gate delay is the ring-coupled oscillator used by the PGA-STC architecture (see Section 4.5.2). However, it is questionable whether the additional circuitry required for the ring-coupled oscillator is worth the increased accuracy in the delay.

Fixed Delays determined by Layout

In Sutherland's Micropipelines [111], the delay between the stages is partly determined by layout. The request and acknowledge signals are routed to the latches for use as capture or pass signals before reaching their destination. This ensures that the latches have received the control signals before the C-Muller gates can change. It is not a complete delay strategy, since additional delay elements are required to account for the logic delay in the data path.

A problem arises when this method is applied to a self-timed FPGA architecture, since registers in a FPGA are typically spread across a two-dimensional area rather than arranged in a column as in Micropipelines. To route the request through the registers in a two-dimensional space requires a long snaking signal that generates a delay far greater than is required. Alternatively, the request signal could fan out to route more quickly to the registers, but this requires C-Muller gates to synchronise the various fan-out request signals once they have reached the registers.

Completion Signal generated by Data Path

This method of generating a variable delay places the onus on the data path to indicate when it has completed evaluating. The timing cell delays its outgoing request until a completion signal is received from the data path. The problem with this method is that it involves a major re-design of the data path to generate a completion signal. Furthermore, the completion signal must be made glitch free using data cells that are not designed to produce glitch free signals.

DCVSL (Differential Cascade Voltage Switch Logic)

DCVSL (Differential Cascade Voltage Switch Logic) [86, 87] is a dynamic logic family that generates its own completion signal using four-phase control. Lu [73] introduces a two-phase version for Micropipeline style implementations. Figure 6.11 illustrates a DCVSL gate. Two NMOS transistor trees generate dual rail outputs for the function and its inverse. When the request signal is logic zero, the two output nodes are pre-charged to `Vcc`. When the request changes to logic one, the NMOS trees are connected to `Gnd` and evaluate. When either output becomes logic zero, the gate has completed evaluation. The NAND gate is used to generate a completion signal when this occurs. DCVSL provides a variable delay since it completes as soon as either dual-rail output finishes evaluating.

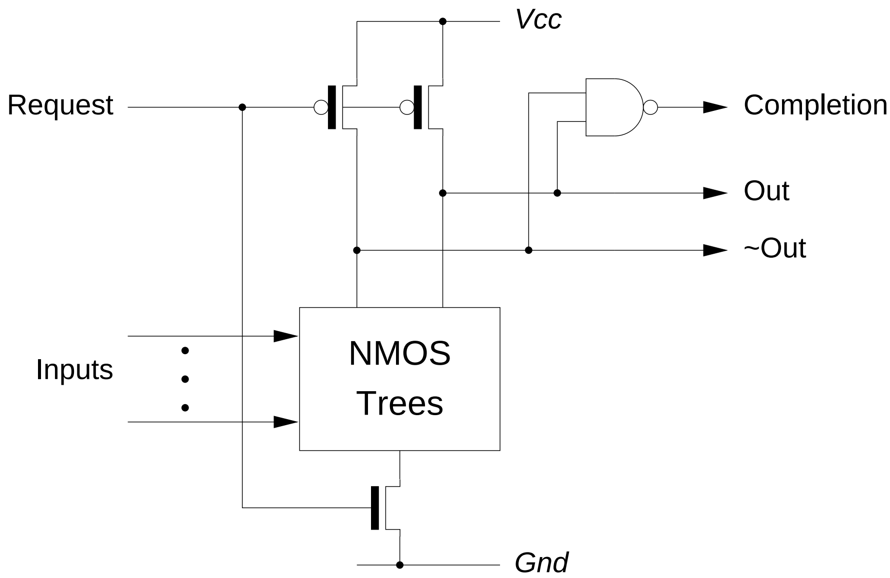


Figure 6.11: Differential Cascade Voltage Switch Logic

Although DCVSL requires altering the design of the function blocks, the same reconfigurable function can be implemented by a DCVSL function block as by a standard function block, so designs can be transferred from a synchronous FPGA to a self-timed FPGA using DCVSL. A disadvantage of DCVSL is that it requires twice the area for logic implementation since both the function and its inverse have to be evaluated. Also, the generation of a request for a data bundle needs wide fan-in C-Muller gates to collect the completion signals from each DCVSL gate.

CSCD (Current Sensing Completion Detection)

Figure 6.12 shows the concept behind CSCD [26]. CSCD utilises the fact that CMOS circuits draw close to zero current from the supply rails when the logic function has completed. By placing current sensing circuits between the supply rails and the logic function, a completion signal can be generated when the current drawn from the supply rails drops towards zero.

CSCD gives the optimum delay plus a delay for generating the completion signal for all possible combinations of inputs to the logic block. It can exploit

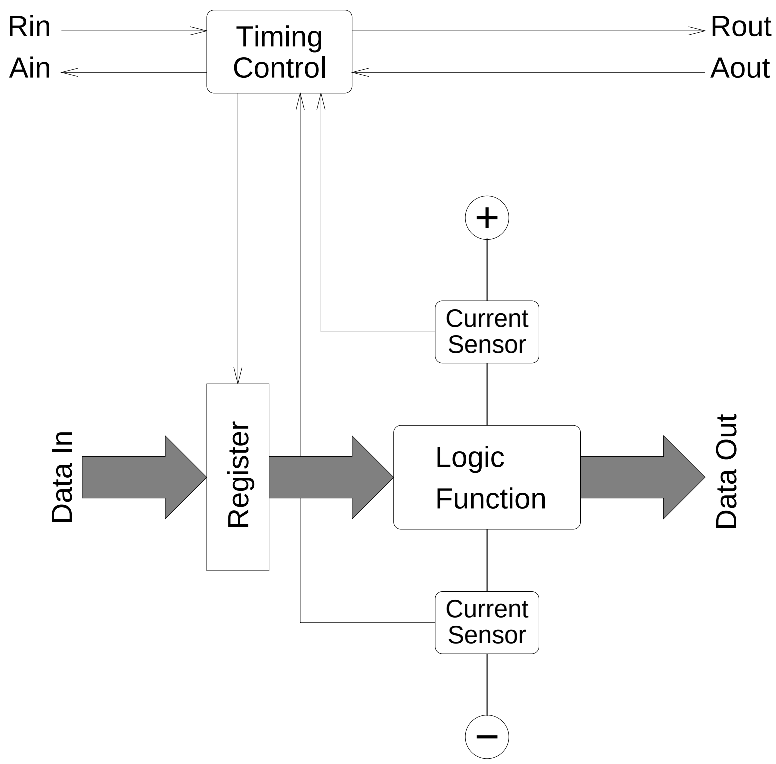


Figure 6.12: Current Sensing Completion Detection

similarity in the input data; for example, if the same values are presented to the logic block in succession then it will complete straight away. The delay incurred by generating the completion signal ensures that the bundling constraint is met. This extra delay only affects performance if the request signal reaches the next stage after its associated data has finished evaluating.

A disadvantage of CSCD is that analogue circuitry is required to implement the current sensors. Dean et al [26] favour bipolar transistors for the implementation of the current sensors, which requires a BiCMOS process technology. In addition, separate supply rails are preferable for the analogue and digital circuits. Also, the current sensors have a static power dissipation that negates some of the low power advantage of using asynchronous circuitry.

Another problem is the resolution of the current sensors; the sensor must be able to detect the current drawn by one changing signal. This becomes increasingly difficult as the circuit becomes larger. A solution proposed by Grass and Jones [50] is to use multiple localised current sensing circuits and produce the final result as the AND of the local completion signals. Careful design is required to ensure that transitions going from one localised area to another do not cause a false completion signal to be generated.

Recently Grass et al [51] proposed a related method of completion detection called Activity Monitoring Completion Detection (AMCD). AMCD uses a special activity monitor circuit attached to the outputs of gates in the circuit. However, attaching a activity monitor to every output in a self-timed architecture would require a substantial amount of silicon area devoted to activity monitors.

Despite the technical difficulties posed by CSCD and AMCD, they are an attractive technique as they generate a data dependent delay, without requiring alteration to the other parts of the data array design.

6.4.2 Choice of Delay Element

A number of delay strategies have been presented above. Of the fixed delay strategies considered, using a reconfigurable fixed delay formed by taking taps off an inverter chain gives the best delay matching. Though the delay of each stage is fixed, average case performance can arise by data taking different branches through the data path. Despite the implementation difficulties, CSCD is the most promising of the variable delay approaches, since it involves the least disruption to the data array, whilst generating an optimal data dependent delay. These two delay strategies are compared in Chapter 11, which investigates a self-timed version of the Xilinx XC6200 architecture using a reconfigurable fixed delay and CSCD.

6.5 Summary

This chapter has introduced reconfigurable elements for synchronisation, the branching and merging of control, and delay in bundled-data circuits. For synchronisation, the reconfigurable C-Muller gate was introduced. The reconfigurable C-Muller gate, and the circuits derived from it, allow a wide range of synchronisation patterns to be defined reconfigurably. For control, the Q-Merge and Select gate pair was developed, and it was shown how a wide range of self-timed control blocks could be implemented with them. Finally, delay generation strategies were reviewed in the context of self-timed FPGA architectures. The reconfigurable elements described in this chapter form the basis for the development of an integrated timing cell which is described in the next chapter, and the timing array routing structures developed in Chapter 8.

Chapter 7

Timing Cells

7.1 Introduction

The STACC model, introduced in Chapter 5, takes a synchronous FPGA architecture and replaces the global clock with an array of timing cells. Each timing cell provides self-timed control to a localised region of the data array. This chapter draws upon the reconfigurable elements introduced in the previous chapter, to develop a reconfigurable timing cell for the STACC architecture. This timing cell integrates all the basic bundled-data control functions of synchronisation, branching and merging, arbitration and delay in one reconfigurable unit.

The timing cell in this chapter is developed, as far as possible independently of the data array's structure. This is possible due to the clean split in the STACC model between the control path, which is implemented by the timing array, and the data path, which is implemented by the data array. However, the design of the timing array does depend on the type and positioning of the memory elements in the data array. Another important effect on the design of the timing array is the choice of protocol used on the handshaking links. These design decisions are discussed in Section 7.2

The timing cells developed in this chapter use a four-phase protocol with registers situated on the inputs to the function block. However, initially a two-phase timing cell without selective communication is introduced, as the design is simpler to describe. Various other design options are explored using the two-phase design. Subsequently, a series of four-phase timing cells are developed with increasing complexity to deal with selective communication and arbitration in the timing cell.

7.2 Timing Cell Design Decisions

The design of the STACC timing cell is affected by a number of different decisions, concerning the type and positioning of the memory elements in the data array, and the protocol chosen on the handshaking links. These decisions are discussed below.

7.2.1 Type of Memory Element

The choice of memory element determines the interface used by the timing cell to control the memory elements. Current FPGAs provide two basic types of memory element which have different control interfaces: *latches* and *registers*. Latches require two control events: a capture event that causes input data to be stored, and a pass event that causes data to be passed data directly to the latch's output. In contrast, registers only require one control event: the capture event which causes them to store the data on their inputs.

Different latch and register implementations are required for two-phase and four-phase protocols. Four-phase memory elements are commonly encountered in hardware design: the four-phase latch is the D-latch and the four-phase register is a D-type register. Both four-phase memory elements only require one control signal; in the case of the D-latch, the pass and capture events are combined into a single signal. This is possible since the control events alternate; transitions in one direction indicate capture events and transitions in the other direction indicate pass events.

Two-phase memory elements are less commonly encountered in hardware design. Two-phase latches, called *event latches*, were used by Sutherland[111] in micropipelines. Two-phase registers have been used by Yun et al [126], and were termed *double-edge D-types*. A problem with two-phase memory elements is that the circuitry is more complex than that for their four-phase equivalents. The event latch and double-edge D-type circuits, essentially involve the construction of two four-phase latches or registers that are used on alternate handshake cycles. For this reason, Sutherland suggests using four-phase latches and a two-phase to four-phase conversion circuit for wider data bundles in Micropipelines.

The choice between using latches and registers is a trade-off between complexity of the memory element and complexity of the control logic. Latch circuits are simpler to implement, and thus generally faster than registers, but require more complex control circuitry as two control events must be generated.

Registers are more complex circuits to build, normally requiring a master and a slave latch, but require simpler control, as only a capture signal is required. An additional benefit of register elements is that unlike latches they do not have a pass state. This means that they can easily be used to build state-retaining blocks such as Finite State Machines (FSMs).

In this chapter, the timing cells are designed to use registers, since most current FPGA architectures include registers in their function blocks. Latch based designs can be developed from these register based timing cells using additional logic to generate the pass events. The alterations required to produce latch based timing cells from register based designs are outlined in the text, though the circuits are not given.

7.2.2 Position of Memory Elements

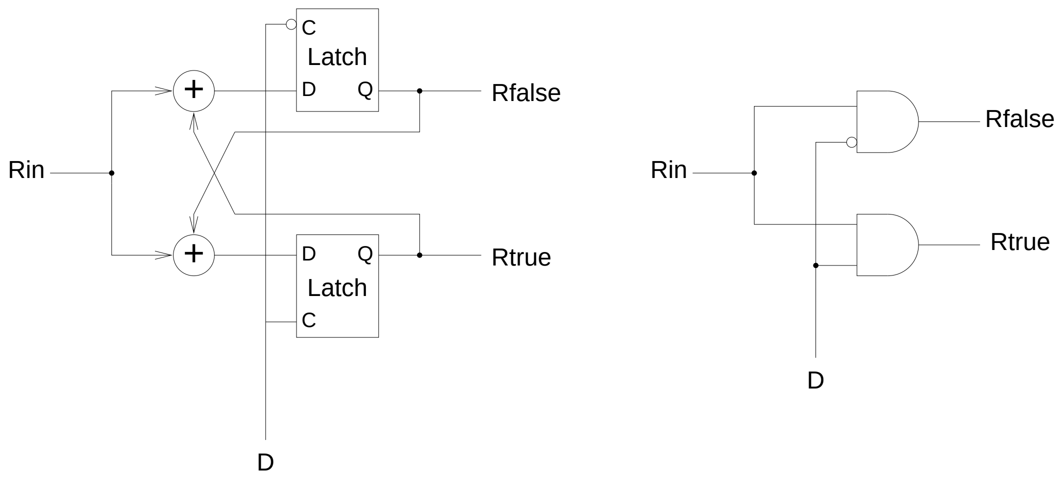
Another way in which the memory elements influence the design of the timing array is by their position within the function block. In most current FPGAs, the memory elements are placed on the output of the logic function implemented by the function block. This influences the design of the timing cell, as it requires that the delay element matched to the logic function comes before the synchronisation element that generates the control signals for the memory element. Similarly, placing the memory elements on the inputs to the logic function, requires that the delay element be placed after the synchronisation element.

When providing timing control for a region of data cells, the memory elements may occur between two logic functions. In this case, delay elements must be provided for the logic delay to the memory elements' inputs and for the logic delay from the memory elements' outputs. This requires delay elements placed before and after the synchronisation element for memory control.

In this chapter, timing cell designs are described for function blocks with memory elements on the input and outputs to the logic function. However, designs for function blocks with memory elements on the inputs are preferred, since this leads to simpler timing cell circuits, which are easier to describe.

7.2.3 Choice of Handshaking Link Protocol

An important design decision for the timing cell is the protocol to use on the handshaking links. The main decision is between using a two-phase or a four-phase protocol. Two-phase protocols are conceptually simpler, since every event on a signal is significant, whilst four-phase protocols requires an idle return-to-zero or recovery phase. Though conceptually simpler, the circuitry for two-phase control circuits is often more complex. For example, Figure 7.1 illustrates two-phase and four-phase Select gates. The two-phase gate is more complex, requiring two XOR gates and two D-latches, compared to two AND gates for the four-phase version. Two-phase designs also require more complex memory elements, such as Sutherland's event latches or double-edge flip-flops.



(a) Two-Phase

(b) Four-Phase

Figure 7.1: Comparison of Select Gate Implementations

Four-phase circuits are potentially slower than two-phase circuits, due to the extra recovery phase. However, the performance disadvantages of four-phase signalling can be hidden by performing the recovery phase concurrently with the computation. As a result, two-phase signalling only gives a significant performance advantage when communication times rather than computation times are critical.

Due to the simpler circuits for four-phase memory elements and control blocks, the four-phase protocol is preferred here for implementing the timing cell. A similar decision was made by Furber [36], for the second generation of the AMULET processor, and by Rebello [100] for the MAP processor. However, initially two-phase timing cells are developed, since the circuits are simpler to explain than the four-phase ones.

Another aspect of the handshaking link protocol that has to be chosen is whether to use a push, pull or two-way data passing protocol. Push protocols, where the sender of data initiates the handshake protocol are the most commonly used style of handshake protocol, so are developed in this chapter. Design of timing cells to implement pull and two-way data passing protocols are discussed in Section 7.3.3.

7.3 Two-Phase Timing Cells

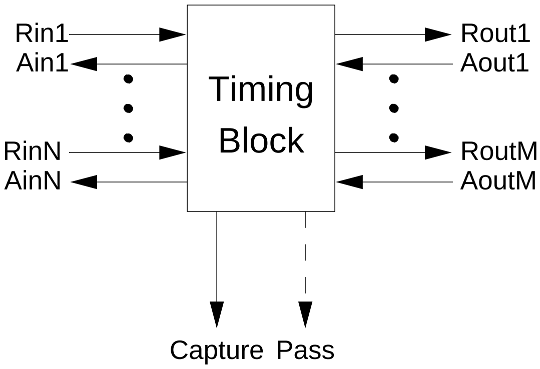
7.3.1 Input Registered Two-Phase Timing Cell

This section develops a simple input registered two-phase timing cell. The timing cell allows data to fan-in and fan-out on handshaking links from the timing cell, but does not support selective communication.

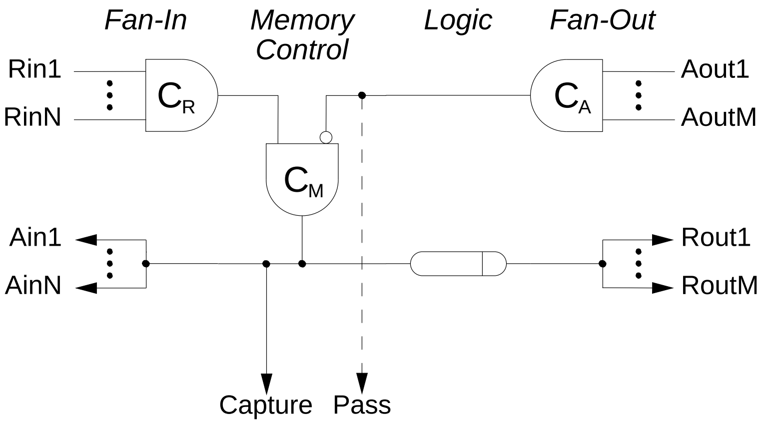
Figure 7.2(a) illustrates the family of timing blocks that the timing cell implements. The timing block can be used to control a self-timed region with N data bundles fanning in and M data bundles fanning out. N and M can be zero, so that a timing block with no input or output handshaking channels can be defined. The interface to the timing blocks consist of N fan-in handshaking links and M fan-out handshaking links. The other signal in the interface is the capture signal, which control the registers in the data array. A pass signal is also shown, to illustrate how the timing block could be adapted to control latches.

Figure 7.2(b) illustrates a family of circuits that implements the timing block logic. The various stages in the timing block match the flow of data in the data array. The stages correspond to the fan-in of the input data bundles, the capture of the inputs in the registers, the computation by the logic function and the fan-out of the output data bundles. For each stage, the circuitry maintains the bundling constraint between the data and the handshaking signals. The basis of the circuit is the memory control C-Muller gate C_M. This is the same control block which is used in Micropipelines (see Figure 3.9). The C_R and C_A C-Muller gates are used to synchronise the fan-in of the request and acknowledge signals.

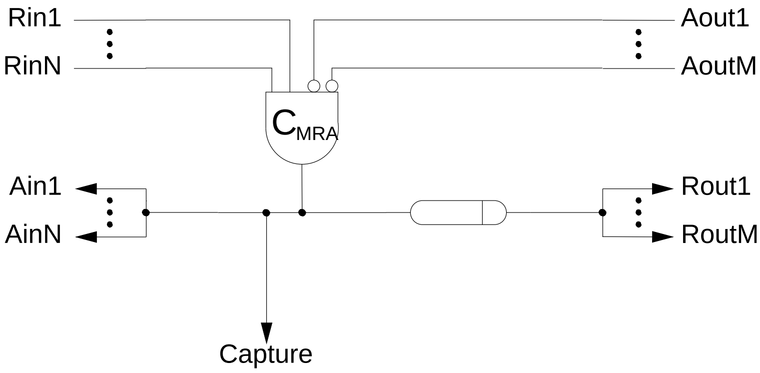
The circuit of Figure 7.2(b) can be simplified by combining the synchronisation in the fan-in C-Muller gate C_R and memory control C-Muller gate C_M into one C-Muller gate. For timing cells controlling registers, the pass signal is not required, so the circuit can be simplified further by combining the fan-out C-Muller C_A gate with the others to form one C-Muller gate C_{MRA}. This gives the simplified circuit of Figure 7.2(c).



(a) Interface



(b) Circuit



(c) Simplified Circuit

Figure 7.2: Family of Two-Phase Input Registered Timing Blocks

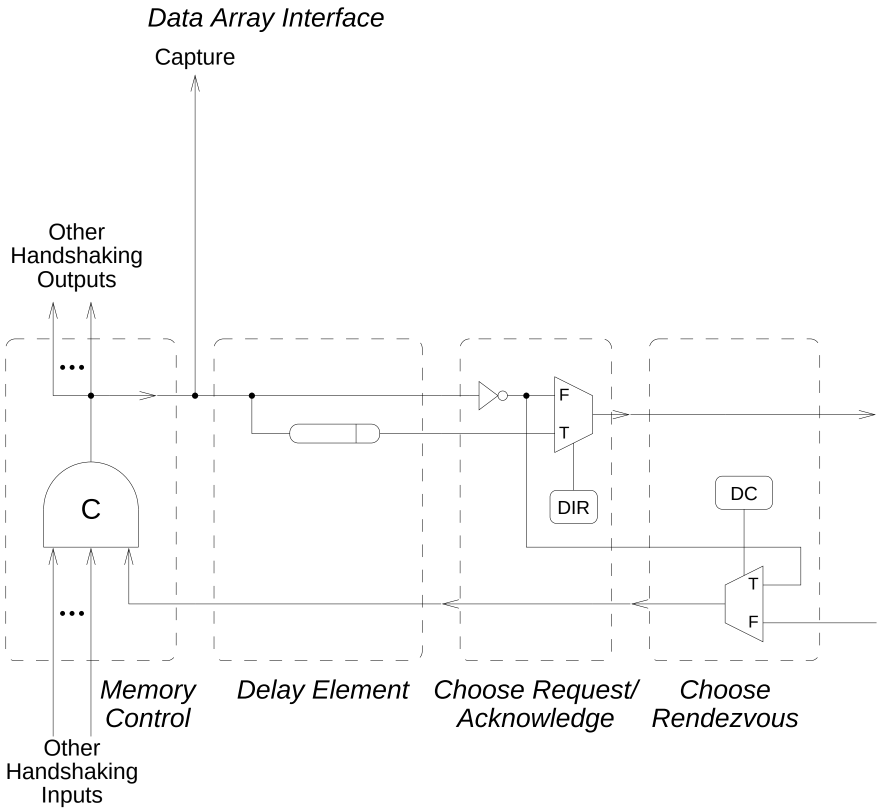


Figure 7.3: Input Registered Two-Phase Timing Cell

The family of timing blocks illustrated in Figure 7.2 represent the possible behaviours of a simple timing cell. The configuration bits required to define the behaviour of a timing cell were discussed in Section 5.4. Two configuration bits per handshaking link were used. The `DC` (Don't Care) bit determined whether the link was used or not. The `DIR` (Direction) configuration bit determined the direction of data flow synchronised by the link.

Figure 7.3 illustrates circuitry to implement a timing cell that can implement the timing blocks illustrated in Figure 7.2. The figure shows the circuitry required for one handshaking link. The circuitry controlled by the configuration bits is replicated for every other handshaking link into the timing cell. The C-Muller gate and delay element in the timing cell circuit corresponds to the C_{MRA} C-Muller gate and delay element in Figure 7.2(c).

The circuit can be thought of as a development of the gate level reconfigurable C-Muller gate. The `DC` configuration bit is common to both circuits, and is used in a similar way to create a don't care connection. When `DC` is false, the inputs and outputs to the neighbouring timing cell pass normally along the handshaking link. When `DC` is true, the timing C-Muller gate's output is fed back to itself to create a don't care connection.

The `DIR` configuration bit, determines whether the handshaking link is an input or output link. Since acknowledge signals in the timing block of Figure 7.2(c) are inverted, the `DIR` bit chooses between the inverted and non-inverted form of the output handshaking signal. Additionally in the timing block, request signals must be delayed to match the logic delay in the data array, so the non-inverted handshaking signal is taken from the output of the delay element.

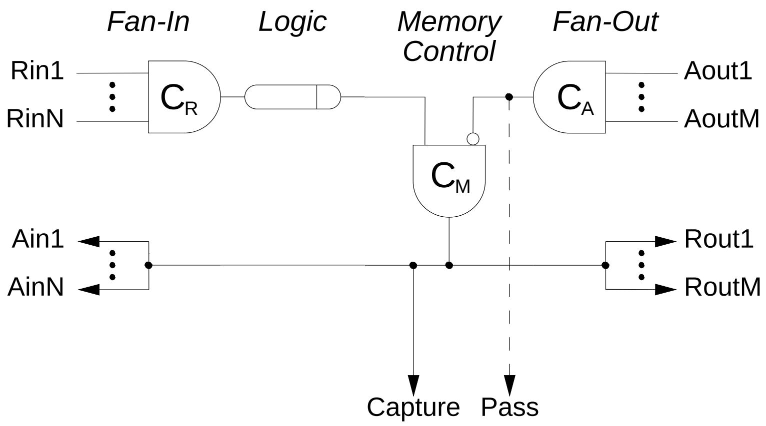
7.3.2 Output Registered Two-Phase Timing-Cell

Figure 7.4(a) shows a timing block designed for a data array with the register elements situated on the outputs of the logic block. Unlike the timing block for memory elements on the inputs, the fan-in C-Muller gate C_R and memory element control C-Muller gate C_M cannot be combined, due to the delay element in between them. However, for register based memory elements, no pass signal is required, so the fan-out C-Muller gate C_A and memory control C-Muller gate C_M can be combined, resulting in the family of timing block circuits illustrated in Figure 7.4(b).

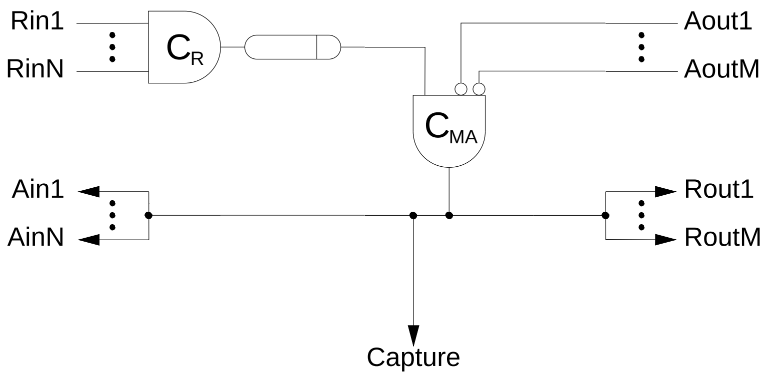
Since the timing block cannot be simplified as much as for the input registered design, a more complex reconfigurable timing cell results, as shown in Figure 7.5. As in Figure 7.3, the circuitry for one handshaking link is shown. The circuit has a similar structure to the input registered design. Both designs use the same circuitry controlled by `DC` for feeding back the outgoing handshaking signal to create a don't care link. The designs differ in the circuitry controlled by the `DIR` configuration bit. In the input registered design, the `DIR` bit only has to control the inversion of the handshaking signal. In the output registered design, it also has to control which C-Muller gate the handshaking signal synchronises with. To implement this, two reconfigurable C-Muller gate structures are used, controlled by the `DIR` configuration bit. These determine whether the handshaking signal is an input to the C_R or C_{MA} C-Muller gates. To account for the inversion of the acknowledge signals, the inputs to the fan-out C-Muller gate C_{MA} are inverted.

7.3.3 Pull Channels and Two-way Data Passing

The timing cells described so far use a push bundled-data protocol. In the push protocol, the request signal is bundled with a data transfer. The pull and two-way data passing protocols differ in which handshaking signals are bundled



(a) Circuit



(b) Simplified Circuit

Figure 7.4: Timing Blocks for Memory Elements on Outputs

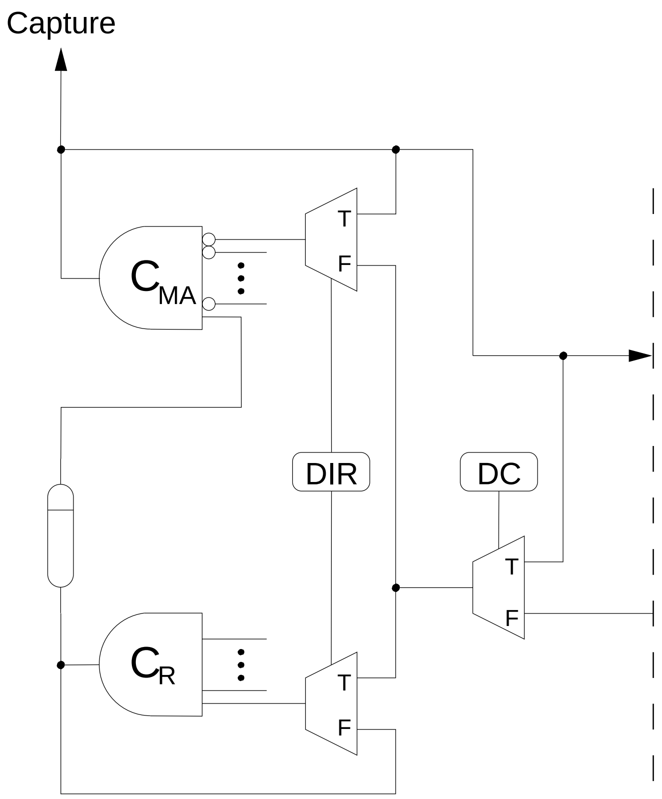


Figure 7.5: Output Registered Two-Phase Timing Cell

with a data transfer. In the pull protocol, the acknowledge signal is bundled with a data transfer, whilst in the two-way data passing protocol, both request and acknowledge are bundled with data transfers.

In all these protocols, if the handshaking signal is associated with a data transfer, then the handshaking signal must be delayed, so that the bundling constraint is met. If the handshaking signal is not associated with a data transfer then it is purely for synchronisation, so there is no need to delay it. Hence, to implement the pull protocol the acknowledge signal rather than the request signal should come from the delay element. For the two-way data passing protocol, both request and acknowledge signals should be sourced from the the delay element.

To allow a timing cell to implement all these protocols, a configuration bit can be added that controls which handshaking signals are delayed. Figure 7.6 shows the new delay control unit for the timing cell of Figure 7.3. An additional configuration bit `DLY` has been added that determines whether an outgoing handshaking signal is delayed. As each output handshaking signal has its own `DLY` configuration bit, push, pull, two-way data passing and pure synchronisation channels (neither request nor acknowledge are delayed) can be

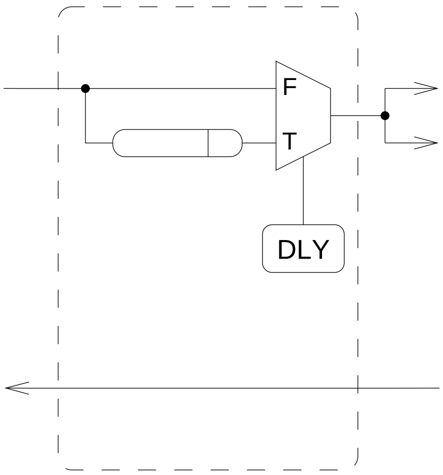


Figure 7.6: `DLY` Configuration Bit

mixed freely. However, it is questionable if the additional circuitry and configuration bits are worthwhile, since many circuits will typically only use one style of communication protocol.

Another design decision in the timing cell related to the use of the delay element is whether a reconfigurable delay element is provided individually for each link, or provided for the timing cell as a whole. Delay matching for each link provides some performance benefits in terms of improved latency, however throughput will not be markedly improved, since this is limited by the worst case delay of the timing cell. In general, the additional circuitry required for a reconfigurable delay element per link, rather than per cell, is not worth the small performance benefits.

7.3.4 Reconfiguration and Initialisation

One aspect of the timing array that has not been considered so far is initialisation. Initialisation is required after reconfiguration, so the method of reconfiguration is an important influence on the method of initialisation. For example, timing arrays that use global reconfiguration, would be initialised using a global reset signal. After reconfiguration, the global reset signal would feed into all the timing cells, and reset the C-Muller gate to its initial value.

Partial reconfiguration, as provided by addressable SRAM FPGAs, requires a more sophisticated initialisation strategy, since global initialisation is not suitable when only part of the array is being changed. A simple strategy for initialising the timing cell would be to reset it when its configuration is changed.

However, the timing cell is only part of a larger circuit, so a reconfigured timing cell could communicate with other timing cells in the circuit which have not been configured. Hence, some means of initialising a portion of the timing array is required.

A solution is to provide an extra configuration bit in each timing cell that determines whether the timing cell is being reconfigured. This bit is named the `RESET` bit. When the `RESET` bit is set, the timing cell is reset and held in the reset state. When the `RESET` configuration bit is cleared, the timing cell will be activated. Thus, to reconfigure a region of the timing array, first the `RESET` bits in all the timing cells to be reconfigured would be set. This can be done in a small number of write actions if a wild card addressing scheme is being used. Next, each self-timed region in the array is reconfigured. After reconfiguration of a self-timed region, the `RESET` bit is cleared. Since all the unconfigured timing cells are locked in the reset state until reconfigured, a reconfigured timing cell will only be able to communicate with other reconfigured timing cells. A benefit of this approach is that the timing cells in the circuit become active as soon as reconfigured. By configuring from the inputs of a circuit to the outputs, the circuit will begin processing whilst the rest of it is being configured.

The `RESET` bit is also useful for deactivating unused timing cells in the array. Unused timing cells within the timing array would generally be configured with all the links as don't care connections. However, this causes the C-Muller gate in the timing cell to oscillate, in a similar fashion to the gate level reconfigurable C-Muller gate, as discussed in Section 6.2.1. When selective communication is introduced, this behaviour is useful as it allows the timing cell to go through several internal states before communicating, but in an unused timing cell, this behaviour is undesirable, since it consumes power. The `RESET` bit can be used to overcome this problem by setting the `RESET` bit in all unused timing cells, which prevents them from oscillating.

Another issue relating to initialisation, is which state the timing cells should be initialised to. So far, it has been assumed that the timing cells are initialised waiting for events on their input handshaking links. In some situations, it can be useful to initialise the timing cells so they immediately generate events on the output handshaking links. For example, feedback loops in designs, require one of the timing cells to be activated to initialise the loop. This can be provided by allowing the initial state of the timing cell to be determined at initialisation. An alternative, is to use the selective communication mechanisms described later in this chapter, so the feedback value is not required for the first data item which enters the feedback loop.

7.4 Four-Phase Timing Cells

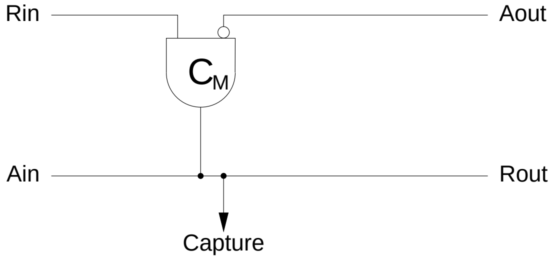
The two-phase protocols used so far are easier to describe than the equivalent four-phase protocols, since every transition is significant. However, four-phase protocols are preferable in general, since they require simpler memory elements and simpler branching and merging control blocks. The key issue in four-phase design is how to hide the redundant recovery-phase by performing it concurrently with the computation in the timing cells. Two-phase protocols become preferable when the recovery phase cannot be hidden, for example, when communication times dominate over computation times. Various ways to hide the recovery phase in four-phase protocols are considered below.

7.4.1 Implementation of Four-Phase Control

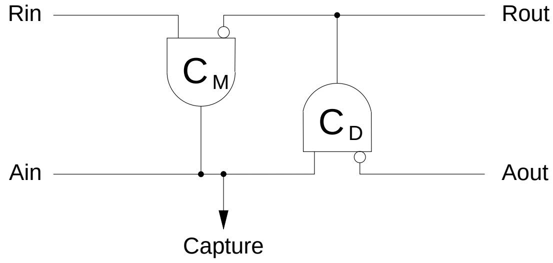
Figure 7.7(a) illustrates the simplest form of four-phase control that uses the same circuit as the two-phase protocols. Since only alternate events are significant, only alternate stages in the pipeline are active, so throughput is halved. This is a severe penalty over the two-phase design. Some improvement can be made by using asymmetric delay elements, that only delay signals on the rising or falling edge. These elements can be used to allow the recovery transitions to pass without delay. Though this gives some improvement in performance, alternate stages are still inactive.

By using additional circuitry, it is possible to allow the recovery to occur concurrently with the computation in the four-phase protocol. One method, used by Arvind and Rebello [8, 100], for allowing the recovery phase to occur concurrently is to use a second C-Muller gate, as in Figure 7.7(b). The second C-Muller gate C_D acts as a place-holder for return-to-zero events, so that all the C_M C-Muller gates in a pipeline can be in the active (logic one) state simultaneously.

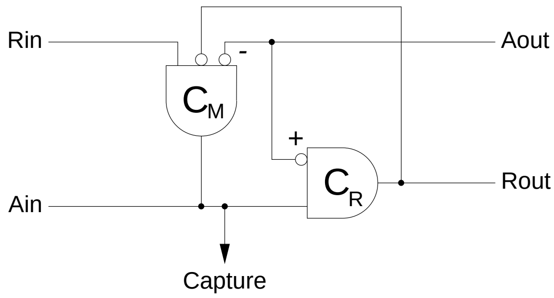
The circuit of Figure 7.7(b) is only suitable for controlling registers. To control four-phase latches a combined capture/pass signal is required, which keeps the latches in the capture state until the four-phase handshake is completed. Hence, the capture/pass signal must remain in the capture state until both C-Muller gates have returned to zero. This requires a two-input OR-gate that takes the OR of the outputs of C_M and C_D to produce the capture/pass signal.



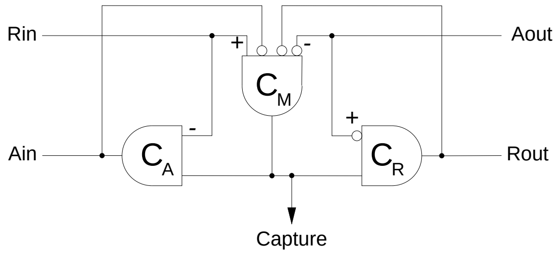
(a) Simple Control



(b) Symmetric C-Muller Implementation



(c) Asymmetric C-Muller Implementation



(d) Fully Decoupled

Figure 7.7: Four Phase Implementations

Figure 7.7(c) shows a four-phase memory control circuit, used by the AMULET Group [25, 37], which utilises asymmetric C-Muller gates. Unlike the symmetric C-Muller gate circuit, the asymmetric C-Muller gate circuit can be used to control latches without modification. This asymmetric C-Muller gate circuit can be considered as a development of the single symmetric C-Muller gate of Figure 7.7(a). In Figure 7.7(c), the symmetric C-Muller gate has been split into two asymmetric C-Muller gates; the `Aout` synchronisation being split into two separate synchronisations on rising and falling events. Since the two asymmetric C-Muller gates synchronise on the same events as the single symmetric gate, both circuits provide the same behaviour for the `Rout` signal.

The decoupling behaviour of the circuit is provided by taking the input acknowledge `Ain` from the output of the C_M C-Muller gate rather than from C_R. For C_M to change from logic one to logic zero, it must wait for `Aout` to reach logic one. This value of `Aout` indicates that the next stage has stored the output of this stage. However, C_M does not synchronise with `Aout` equal to logic zero, which is the recovery transition on `Aout`. Thus, events on the input channel are decoupled from the recovery transitions on the output channel.

Though the input channel is decoupled from the recovery transitions on the output channel, the circuit must ensure that the recovery transition has occurred on `Aout` before C_M synchronises on `Aout` equal to logic one again. For this reason, the feedback connection is included from C_R to C_M. Without this connection, if `Aout` remained at logic one, then a number of `Rin` and `Ain` events could occur on the input handshaking channel without any event being generated on the output channel. The feedback connection overcomes this problem by ensuring that every transition on C_M must generate a transition on C_D before C_M can change again.

The circuit of Figure 7.7(c) is known as a *semi-decoupled* handshaking circuit, since it decouples the input handshaking channel from synchronising with the recovery phase on the output channel, but does not decouple the output handshaking channel from synchronising on the recovery transitions on the input channel. A *fully-decoupled* control circuit can be produced by replicating the decoupling circuitry for the other handshaking channel as shown in Figure 7.7(d).

Both the decoupling symmetric and asymmetric decoupling methods described above, allow the recovery phases to occur on each channel independently. The asymmetric C-Muller gate circuit uses more gates, but is preferable when using latches, since no extra logic is required to generate a capture/pass signal. The symmetric C-Muller decoupling circuit requires an additional OR gate to control latches. In the rest of the chapter, the symmetric C-Muller gate implementation is used, since it leads to clearer circuits and is more suited to register based memory elements which are used in most FPGA architectures.

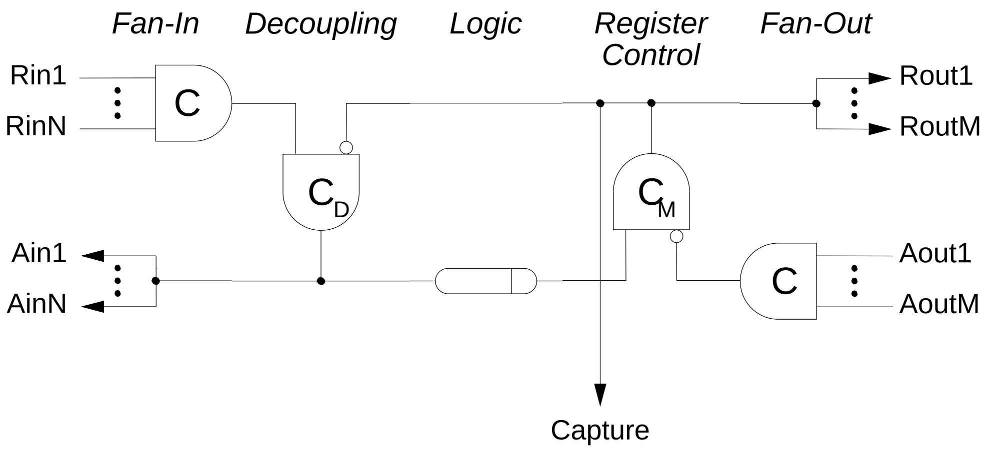
7.4.2 Position of Decoupling

In the symmetric C-Muller gate circuit of Figure 7.7(b), the decoupling C-Muller gate C_D can be separated from the memory control C-Muller gate C_M by inserting other stages, such as the delay stage or the fan-in and fan-out stages, between C_M and C_D. Figure 7.8 shows two positions for the decoupling C-Muller gate in an output registered timing block. Figure 7.8(a) has the decoupling after the fan-in of the links, so all the input links are decoupled as a group, whilst Figure 7.8(b) has each link decoupled individually, before the fan-in of the inputs.

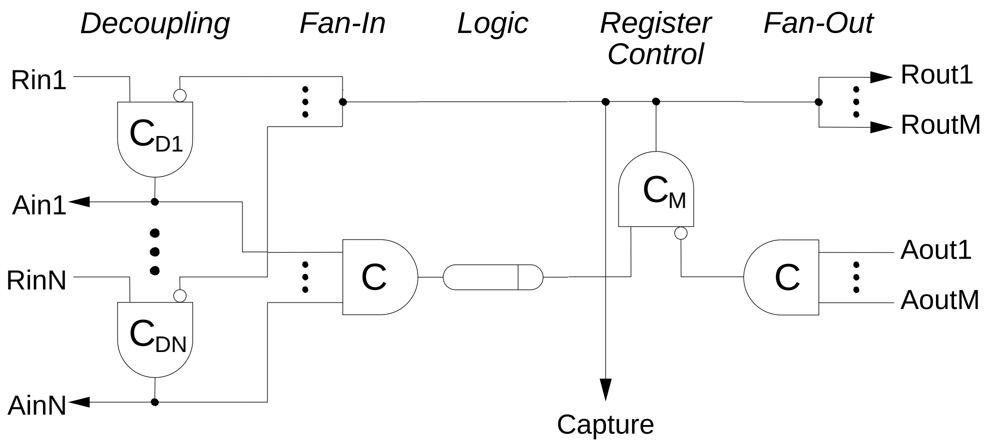
Decoupling the links as a group is advantageous as it uses only one decoupling C-Muller gate per timing block, whilst decoupling on the links requires one C-Muller gate per link. However, decoupling on the links has the potential for superior performance as the decoupling on each link is independent. With the central decoupling, all the links are decoupled as a group, so the delay of the slowest link determines the delay of the recovery phase on all the links.

An additional benefit of decoupling per link, is that it allows the decoupling C-Muller gate to be placed in the centre of the link, halfway between the two communicating timing blocks. This leads to a performance advantage, since the request and acknowledges only have to travel halfway along the link rather than travel the whole length of the link. However, a problem with having decoupling on each link is that links cannot be easily chained together, since multiple decoupling C-Muller gates between memory control C-Muller gates cause the four-phase protocol to fail.

For the asymmetric C-Muller decoupling of Figures 7.7(c) and 7.7(d), the decoupling gates cannot be separated, due to the feedback paths from the C_R and C_A C-Muller gates to C_M. As a result, the position of the decoupling cannot be moved, without the additional cost of routing the feedback wires. However, the decoupling can still be provided as a group or on individual links, depending on how the fan-in/fan-out stages are combined with the timing block.



(a) Decoupling Links as a Group



(b) Decoupling Links Individually

Figure 7.8: Position of Decoupling

7.4.3 Input Registered Four-Phase Timing Cell

This section describes a four-phase timing cell, which will be developed later in the chapter to include selective communication and arbitration. The timing cell is illustrated in Figure 7.9. It is designed to control input registered designs, and provides decoupling on each link using symmetric C-Muller gates.

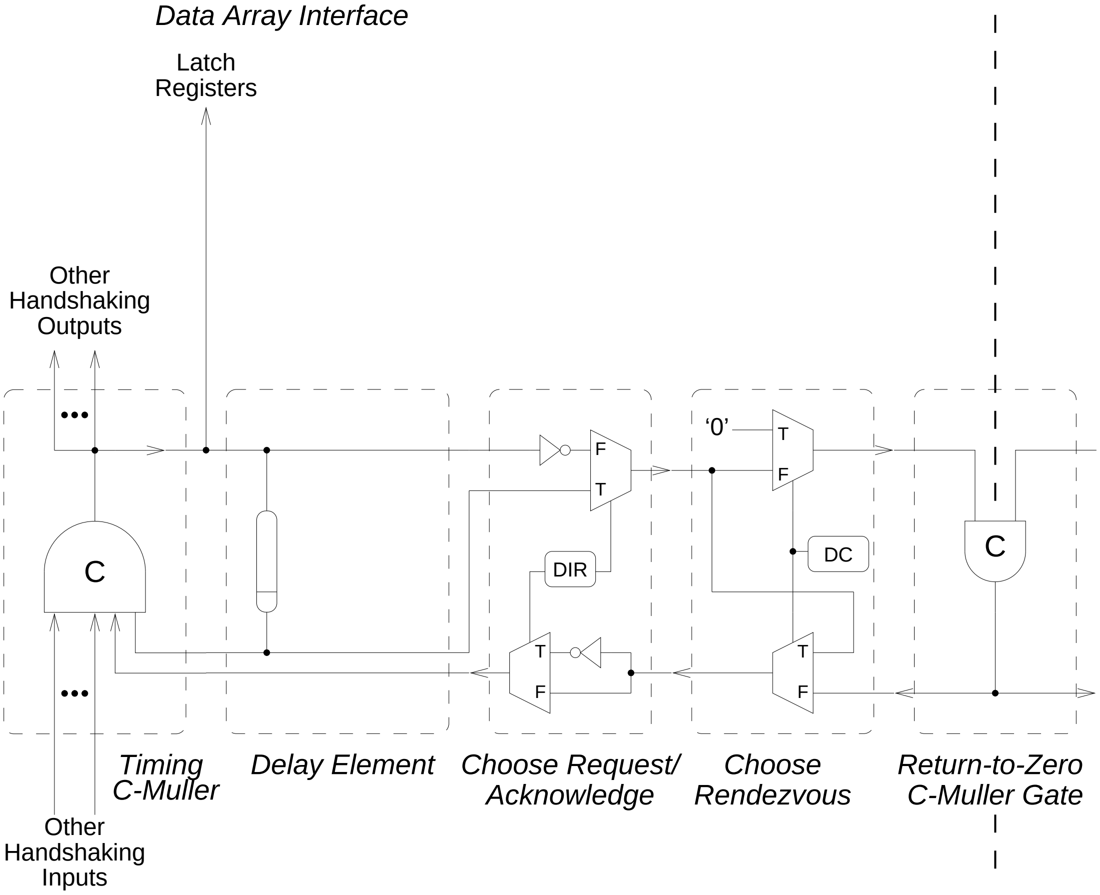


Figure 7.9: Input Registered Four-Phase Timing Cell

This choice of timing cell was chosen, since it produces simpler circuitry for explanation. As discussed previously, input registered designs allow more simplifications to be applied to the basic timing block, giving simpler timing cell circuitry. Also, using symmetric C-Muller gates for decoupling on each link was chosen as it allows the decoupling to be clearly separated from the rest of the timing cell.

The timing cell is similar to the two-phase input registered timing cell of Figure 7.3, however there are a number of difference in the four-phase design. Two additional multiplexors have been added: the first additional multiplexor is controlled by the `DIR` configuration bit and is used to control the inversion of the output of the decoupling C-Muller gate. The second new multiplexor is controlled by the `DC` configuration bit and is used to prevent transitions on the output handshaking signal when the link is configured as a Don't Care connection. This modification is not necessary when selective communication is not used, as the other side of the link will also be in a Don't Care state, so will ignore any transitions that occur on the handshaking signal output. However, when selective communication is introduced, this multiplexor is necessary, since the other timing cell on the link may be waiting for communication on the handshaking link.

Another difference in this timing cell is the feedback of the delay element directly to the C-Muller gate. The feedback connection ensures correct operation when the timing cell is disconnected from other timing cells. Again, this behaviour is useful when selective communication is introduced, since it allows the timing cell to loop through several internal states before communicating with any other timing cells.

7.5 Selective Communication Timing Cell

So far, a timing cell capable of implementing self-timed pipelines with fan-in and fan-out has been described. However, selective communication has not been supported. There has been no means for data values within the data array to influence the control flow in the timing array. This section introduces a timing cell capable of deterministic branching and merging in the control flow.

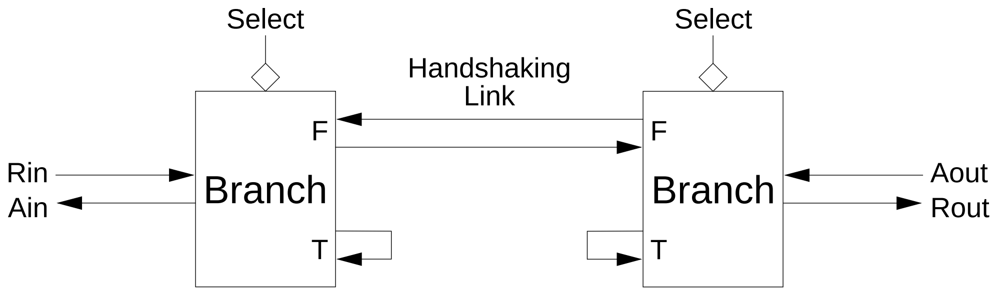


Figure 7.10: Link Between Two Timing Cells using Branch Modules

The behaviour of a handshaking link using selective communication is illustrated in Figure 7.10. The figure shows one link between two timing cells constructed using Branch modules. The request signals flow from left to right in the figure, whilst acknowledge signals flow from right to left. The Branch module on the left determines when to initiate a communication on the link. When the link is not selected, the request events on `Rin` are passed directly back to `Ain`. When the link is selected, the request event passes to the Branch module at the other end of the handshaking link, and the sender waits for an acknowledge event from the receiver. The Branch module at the receiver operates in a similar fashion. When not selected, events cycle around the `Aout` and `Rout` loop. When the link is selected, the receiver waits for a request event on the handshaking link before generating an acknowledgement.

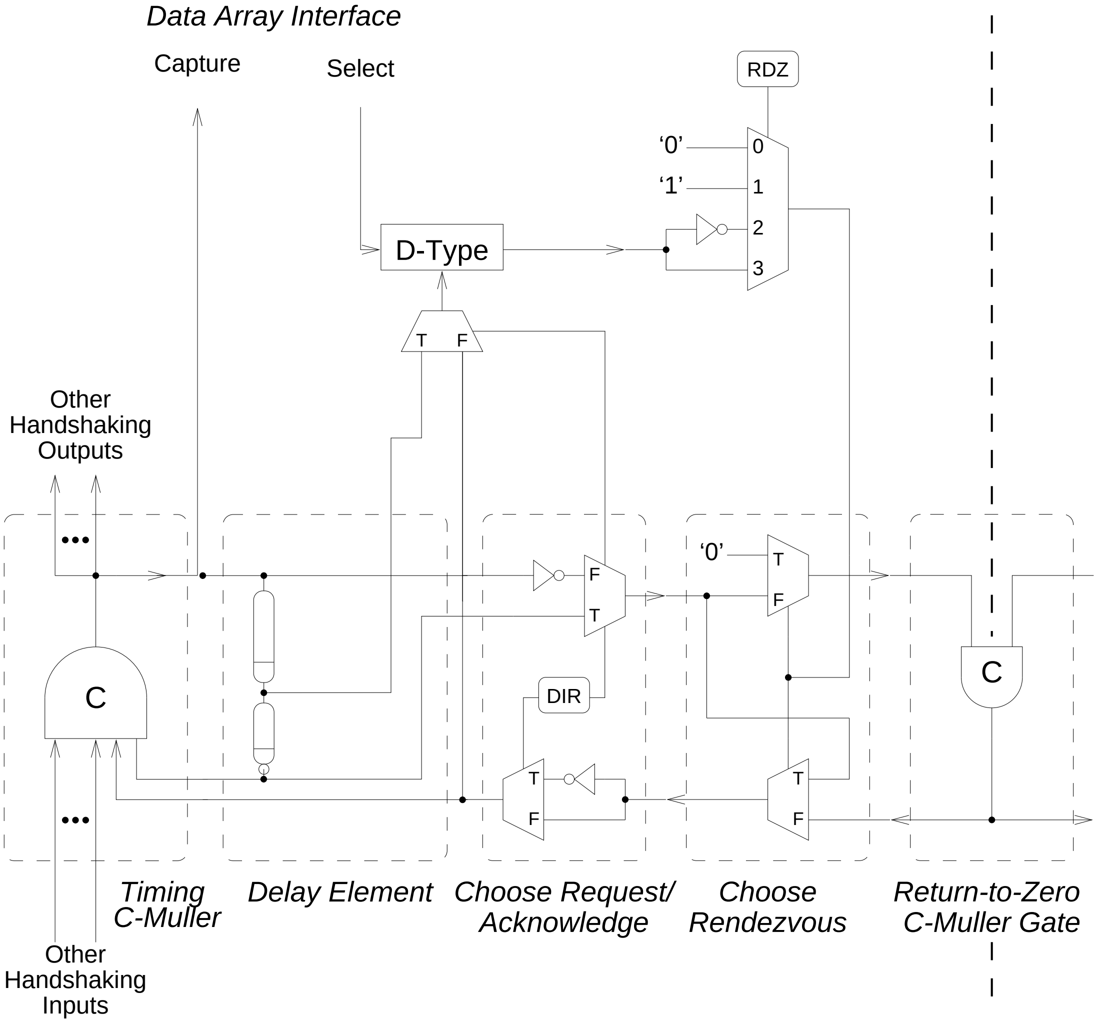


Figure 7.11: Timing Cell with Branching

Selective communication cannot be implemented in the timing cells described so far, however communication patterns can be altered by changing the configuration data. This suggests that a way to implement selective communication would be to pass control of the communication pattern from the configuration data to select signals generated by the data array. The idea is illustrated in Figure 7.11, which shows an adapted version of the timing cell in Figure 7.9, where the `DC` configuration bit has been replaced by a signal that can be driven from the data array.

This signal determines whether communication takes place on the handshaking link during the current cycle of the timing cell. The signal is generated by a new multiplexor controlled by the `RDZ` (Rendezvous) configuration bits. The `RDZ` configuration bits allow the choice of an inverted or non-inverted select signal from the data array. The choice of inversion allows the initial value of the select signal to be defined (assuming that the D-type is reset to a predefined value). The `RDZ` configuration bits can also choose the constants logic zero and logic one, so that the common functions of 'never communicate' and 'always communicate' can be implemented without using resources in the data array.

In addition to the `RDZ` multiplexor, extra circuitry is included to capture the select signal. An edge-triggered D-type flip-flop is supplied for capturing the select signal. The triggering of the D-type is different depending on whether the link is an input or output link, hence an additional multiplexor controlled by the `DIR` bit is included to choose the triggering signal. For output links, the select signal is captured before communication with the neighbouring cell. For input links, the select signal is not captured until the completion of the current communication, to prevent problems with the select value changing during a communication on the link.

An extra fixed delay element is included after the reconfigurable delay in the delay stage. This delay is included to ensure that the select signal is sampled, and its value is established through the `RDZ` multiplexors before the out going handshaking signal reaches the 'Choose Rendezvous' multiplexors. This fixed delay element sets a minimum delay for the timing cell; the timing cell cannot go faster than the minimum delay set by the time to sample the select values.

It can be seen from Figure 7.11 that much of the circuitry for selective communication is for sampling the select signal. The amount of circuitry can be decreased by reducing the number of select signals from one per link, though this reduces the possible behaviours that can be implemented by one timing cell. For example, only one select signal could be provided per timing cell. The select signal's inverse can be generated using the `RDZ` configuration bits. This allows basic 'if-else' style communication structures to be built. However, this is limiting in that more sophisticated branching structures such as multiple way branches, cannot be configured in a single timing cell.

7.6 Arbitrating Timing Cell

The timing cell developed so far allows selective communication, so the timing cell and associated region of the data array can make deterministic decisions based on their own internal states. However, the timing cell cannot choose which links to select on the basis of which neighbouring timing cells are waiting to communicate. This requires some method to sample the state of incoming handshaking links. Since, the signals on these links are not synchronised to the timing cell, some form of arbitration or synchroniser element is required to allow the state of the links to be sampled.

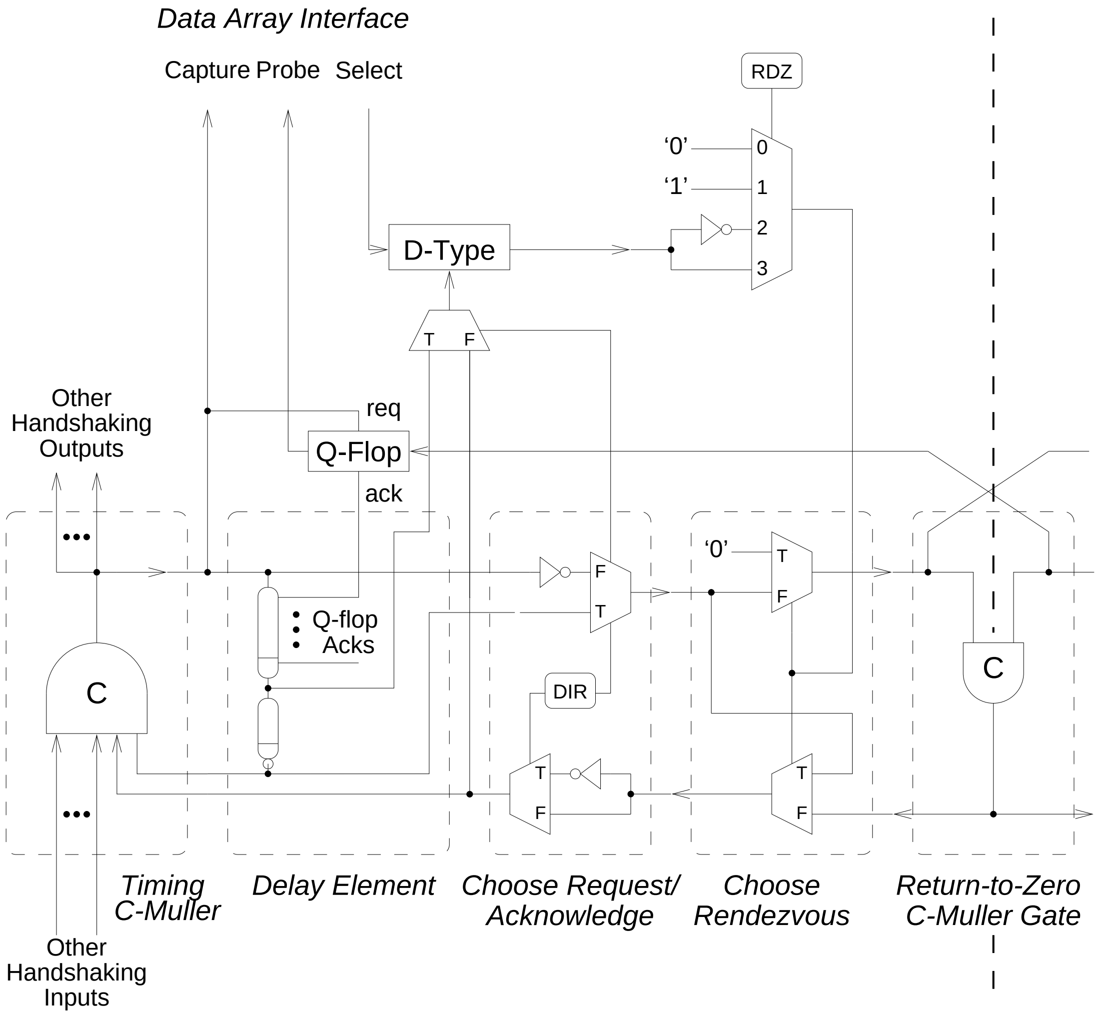


Figure 7.12: Q-Flop Arbitration Scheme

Below, two arbitration schemes are considered. Both use special arbiter elements in the timing array to resolve potential meta-stable states. The approaches differ in where the functions to be arbitrated between are generated. The first scheme evaluates the arbitration functions in the data array, the second evaluates them in the timing array.

7.6.1 Data Array Arbitration Function

One way to perform arbitration is to probe the state of the links to neighbouring timing cells, and provide these as inputs to the data array. The data array can make a decision based on these probe values and then use selective communication to choose which channels to communicate on. This arbitration scheme is flexible, since it does not fix the choice of arbitration function. Any arbitration function can be implemented, if sufficient data cells are used.

Figure 7.12 shows the development of Figure 7.11 that allows the state of the handshaking inputs from neighbouring timing cells to be probed. Since the probe signals are asynchronous to the cell, some form of synchroniser element must be used to sample the inputs. In this case, a Q-flop [103] is used. The Q-flop samples its data input after a transition on its request signal. After any meta-stable state has been resolved, the Q-flop generates an acknowledge. This is fed to the delay element. The delay phase will not begin until all the acknowledge signals from the Q-flops have been received.

There are several drawbacks to the Q-flop scheme. First, arbitrating elements such as the Q-flop are relatively complex circuit elements to implement, and the Q-flop scheme requires one Q-flop per link which adds substantially to the size of the timing cell. Furthermore, the Q-flop scheme requires the use of data cells in the data array to generate the arbitration function. However, as a result of using the data array to implement the arbitration function, no additional configuration bits are required in the timing array to implement the Q-flop scheme.

A further drawback to the Q-flop scheme is that the arbitration function is evaluated concurrently with the data path function implemented in the self-timed region. This means the result of the arbitration can only be used to select links at the end of the timing cell's cycle. Hence, input links cannot be selected in the same cycle as the arbitration function is evaluated, since the input links are selected at the start of the timing cell cycle rather than at the end. This leads to long latencies for arbitration on input links. For example, to implement an arbitrated Merge gate, two cycles of the timing cell would be required. In the first cycle the input links are probed and the arbitration function is evaluated, in the second cycle, the result of the arbitration is used to select the appropriate input link.

An alternative approach is to use Q-flops in place of the D-types to sample the select signals. In this scheme, the probe signals from the links would be passed unsynchronised into the data array. The arbitration function is evaluated in the data array, and then passed as a select signal to the timing array. The select signal is synchronised using the Q-flop. This scheme has the advantage that the output of the arbitration function is synchronised rather than its inputs, so the two cycle arbitrate/select loop can be avoided. However, this scheme means that the probe signals from the handshaking links cannot be used in the data array as inputs to the data path function, since they are not synchronised to the self-timed region. Hence, the synchronised select signals from the Q-flops also have to be fed back as probe signals to the data array, to allow the data path to respond to the result of the arbitration.

7.6.2 Timing Array Arbitration Function

The previous arbitration schemes are very flexible as they leave the choice of arbitration function to the circuit designer, but incur a high cost in terms of the number of arbitration elements required and the number of data cells required to implement the arbitration function. An alternative approach is to use a dedicated arbitration function implemented in the timing array, which saves on data cells and interfacing logic between the timing and data arrays.

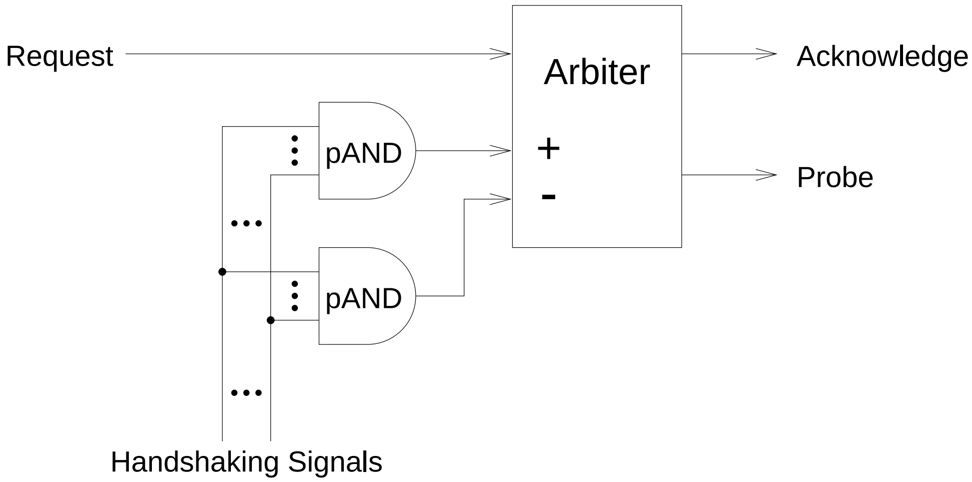


Figure 7.13: Dedicated Arbitration Block

A simple dedicated arbitration block that covers many common arbitration functions such as Merge gates is shown in Figure 7.13. The programmable-AND (pAND) gates form product terms from a subset of the handshaking signals. This subset is determined by configuration data. One pAND gate causes the output probe value to be high (the plus input), the other causes the probe output to be low (the minus input). More sophisticated arbitration functions can be used that include inputs from the data array as well as the handshaking links.

Another advantage of using a dedicated arbitration block over the Q-flop scheme is that the arbitration function is evaluated within the timing cell, so the result can be used directly to select the input link to the timing cell without using the data array and the problem of the two cycle arbitrate/select loop for input arbitration can be avoided. Furthermore, the pAND arbiter waits until either one of the AND gates goes high before sending an acknowledge to the timing cell. In the Q-flop arbitration scheme there is no such waiting state. For example, if the Q-flop scheme is used to implement an arbitrated Merge gate, then if neither input is active, the timing cell busy waits, continually sampling the input links until one becomes active.

7.7 Summary

This chapter has developed a reconfigurable timing cell for use in the STACC architecture. The final timing cell integrates the basic functions of self-timed control: synchronisation, selective communication, arbitration and delay. The chapter also illustrated how basic decisions concerning the memory elements and protocols affected the design of the timing cell.

Chapter 8

Timing Array Routing

8.1 Introduction

The previous chapter developed the STACC timing cell independently of the routing network used to interconnect the timing cells. In this chapter, structures are developed to allow handshaking signals to be routed in the timing array. Much of the discussion is applicable beyond FPGAs to any style of self-timed routing network, for example, the passing of data between self-timed processors. In particular, the chapter has relevance to the design of self-timed versions of FPIDs (Field Programmable Interconnect Devices), such as the Aptix [5] and I-Cube [61, 60] devices.

A benefit of self-timed routing is that the routing can be changed between modules and the circuit will still operate. As long as the bundling constraint is maintained the routing is transparent to the sender and receiver, so buffering and multiplexing can be included transparently. Section 8.2 discusses how these structures could be used in self-timed FPGA routing and I/O interfaces.

The routing of handshaking signal pairs in the STACC timing array requires the design of handshaking switchboxes that can match the routing of switchboxes in the data array. Section 8.3 develops handshaking switchboxes from the reconfigurable C-Muller gate, which can synchronise the fan-in and fan-out of data bundles in the timing array routing.

A different style of timing array routing can be developed using the distributed reconfigurable C-Muller gate; this is discussed in Section 8.4. The benefits of the distributed routing structures are best utilised with a different style of timing cell to that developed in the previous chapter. An outline development of such a distributed timing cell is given, which provides an interesting comparison with the centralised C-Muller gate timing cell developed previously.

8.2 Transparent Routing Structures

An important benefit of the self-timed approach is that it allows designs to operate independently of the routing delay between the various parts of the system. This allows routing structures that alter the delay on a route to be inserted 'transparently' into a self-timed FPGA architecture. The designer need not be aware of their presence, yet the design will still work. This cannot be done with synchronous systems, since changing the routing delay could cause the system to fail to meet the global clock constraint.

Below, three routing structures that can be inserted transparently into the routing are considered: buffering, route multiplexing and alternative routing schemes. These structure are of particular use over longer routes and for sharing input/output resources.

8.2.1 Buffering

Any number of buffering stages can be inserted transparently into the routing of a self-timed system. Including a buffer in a route increases the latency of the route, due to the additional delay for capturing the data in the buffer's memory elements, but throughput is increased, since the route can contain more than one value at a time.

Buffering is particularly useful over long routes, since the handshaking signal routing delay may be the critical source of delay, especially in four-phase protocols, where there are additional recovery transitions. In pipelines, it is the longest cycle time of all the stages in the pipeline that determines the speed of the pipeline. By splitting the route using a buffer, the latency of the route is marginally increased. However, if the route was on the critical path, the cycle times of the handshaking protocols to and from the buffer are less than the cycle time of the unbuffered route, so the performance of the pipeline is improved.

Buffering has also been shown to be important in self-timed systems with variable length delay schemes [66]. In such systems, when the worst case delay is encountered in a stage, it results in idle stages before and after the stage. With buffering between stages, stages in the pipeline can stay active by using data that is stored in the buffers.

In general, a buffer cannot be included transparently in a synchronous system, since it would delay the arrival of the data by a clock cycle. However, synchronous systems with flow control can include buffering, though the cost of inserting a buffer is greater, since the additional latency is one clock cycle, whilst the additional latency in the self-timed protocol is simply the time taken to capture the data.

8.2.2 Route Multiplexing

In the field of Communications, routes are commonly shared between a number of sources and destinations by using some form of multiplexing. The flow control properties of self-timed systems also allow such schemes to be incorporated transparently into self-timed routing for FPGAs.

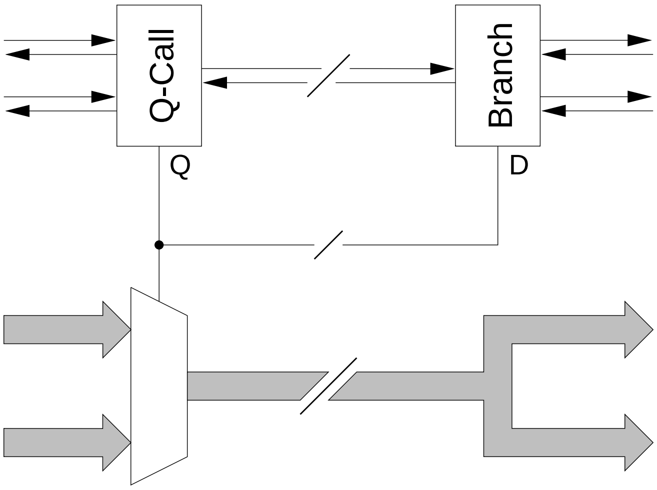


Figure 8.1: Route Multiplexing using Q-Call and Branch Modules

Figure 8.1 illustrates a multiplexing scheme using the Q-Call and Branch modules introduced in Chapter 6. The Q-Call module arbitrates between requests to use the route. The result of the arbitration `Q` is used to select the output data for the route via a bus multiplexor. The `Q` signal is also routed with the data bundle to indicate which route is using the shared channel. At the other end of the channel, the data is routed to its two destinations. The handshaking signals are separated by the Branch module that is controlled by the `Q` signal.

Route multiplexing has particular use in FPGAs, for multiplexing data on input/output pins. Most current FPGA architectures have far more signals on the edge of the array then can be routed via the I/O pins. This coupled with the additional off-chip delays prevents arrays being extended uniformly across multiple devices. Self-timed routing accommodates for the additional off-chip delays, and with the addition of route multiplexing, this allows self-timed FPGAs to be extended uniformly to multi-device arrays.

Pin sharing via multiplexing can be used in synchronous FPGAs. For example, the Virtual Wires System [11] used multiplexing to overcome the pin limitations in a multi-FPGA logic emulation system. However, these schemes need some form of flow control to be explicitly introduced into the design, and thus cannot be included transparently as in the self-timed case.

8.2.3 Alternative Routes

Typically, in FPGAs, each data bus is provided with a dedicated signal path, from sender to receiver. Thus, there is no need to provide alternative routes in the architecture to overcome bottlenecks. However, if route multiplexing is introduced, or if the timing array routing architecture is used in other self-timed routing applications, it may be be useful to provide alternative routes between sender and receiver, to overcome bottlenecks.

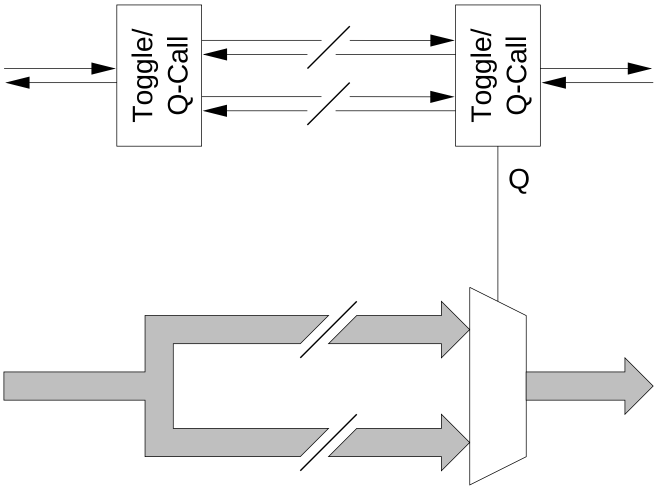


Figure 8.2: Alternative Routes

Figure 8.2 illustrates an alternative route scheme. It is similar to the work sharing pipeline examples from Section 5.4, but instead of sharing the processing load between two pipelines, communications are shared between two routing channels. For the routing channels to be used concurrently, the routes must contain some buffering, otherwise, the communication on one channel has to be completed before communication on the next channel can begin.

Either Toggled-Branch modules or Q-Call modules can be used as the control blocks. When Toggled-Branch modules are used, the left hand control block distributes the data alternately to each channel. The right-hand control block, a Toggled-Branch module connected in reverse, is used to alternately take data from each channel. Thus the potential bandwidth of the channel is doubled, by distributing the data between two channels.

When Q-Call modules are used as control blocks, a slightly different behaviour emerges. The control block on the left, a Q-Call module connected in reverse, sends data down whichever channel is free. The right-hand Q-Call block takes data from whichever channel has data available. The advantage of the Q-Call scheme is that it dynamically chooses which channel to use, and so can avoid blockages in the routing. However, it can re-order data, so either the order of data has to be unimportant, or the data has to be routed with a tag to identify its order.

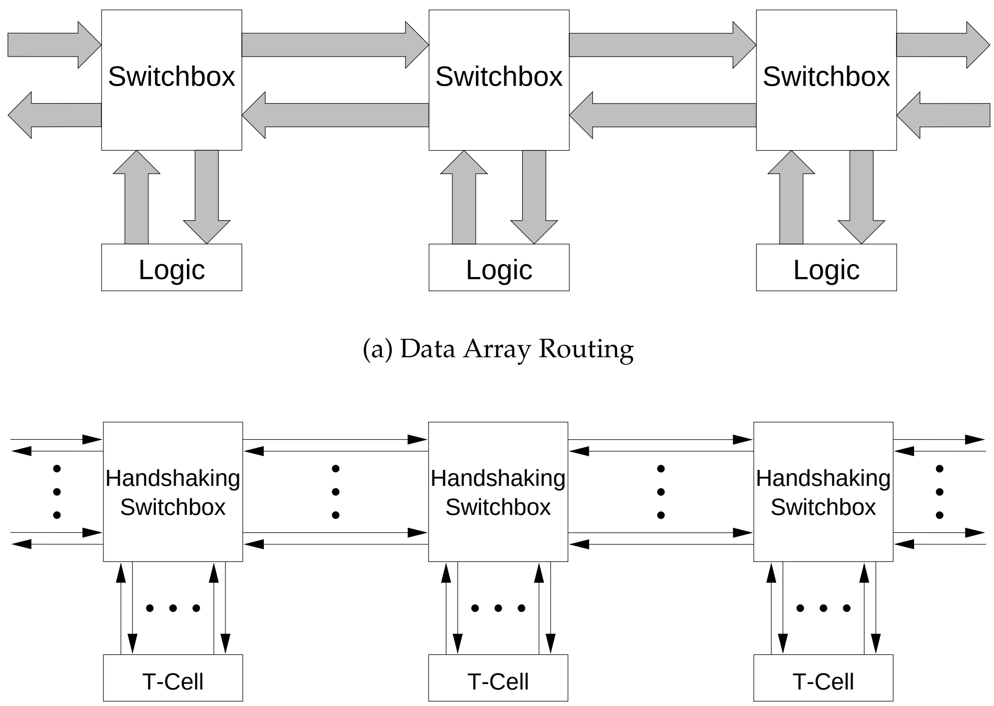
8.3 Routing Handshaking Signals

The purpose of the STACC timing array routing is to route handshaking signal pairs, in a way which matches the routing of their associated data bundles in the data array. Routing of handshaking signal pairs rather than individual handshaking signals is advantageous, since the request and acknowledge signals in a handshaking pair are routed in opposite directions, and so the same configuration data can be used to configure the routing of both signals. Also, the routing of handshaking signal pairs, rather than individual handshaking signals provides a clean interface between the routing architecture and the timing cells.

In routing handshaking signal pairs, it is important that the bundling constraint is maintained by the routing. For this to be achieved, the routing structure in the timing array should match as far as possible that in the data array. As a result, the interconnection network used in the data array has a strong influence on that used in the timing array. Depending on the routing structure of the data array, it may be necessary to include delay elements in the timing array routing to ensure that the bundling constraint is maintained. Delay matching is considered in Section 8.3.2.

The timing array routing limits the number of data bundles that may be routed through the data array. It is easy to construct routing patterns in the data array that would require an enormous number of handshaking pairs to be routed in the timing array. For example, designs with data bundles of single bits fanning out to many different destinations would require a massive quantity of handshaking pairs routed in the timing array. The overhead of providing timing array routing resources that could route such designs is massive, since two handshaking signals are being routed for each bit in the data array. Thus, unless a disproportionate amount of circuitry is given over to the timing array routing, there will always be data array routing patterns that could be implemented in a synchronous version of the architecture that cannot be implemented similarly in the self-timed architecture.

Although it is impractical to provide sufficient timing array routing resources that can cover every conceivable pattern of data flow in the data array, the timing array routing should be capable of implementing common patterns of data flow for a reasonable number of data bundles. Generally, the pattern of data flow is more structured than the scenario envisaged above of individual bits being routed to multiple different destinations. Even if timing cells only produce a data bundle of a single bit, these can generally be grouped together with other timing cells producing single bits to form a bundle of signals that are routed to a destination. Similarly, the destination for a bundle may not be a single self-timed region, but may be a group of self-timed regions, with each timing cell only requiring certain signals from the bundle. This suggests that the routing architecture should allow the fan-in and fan-out of data bundles within the architecture.



(a) Data Array Routing

(b) Timing Array Routing

Figure 8.3: Abstract Routing Architecture

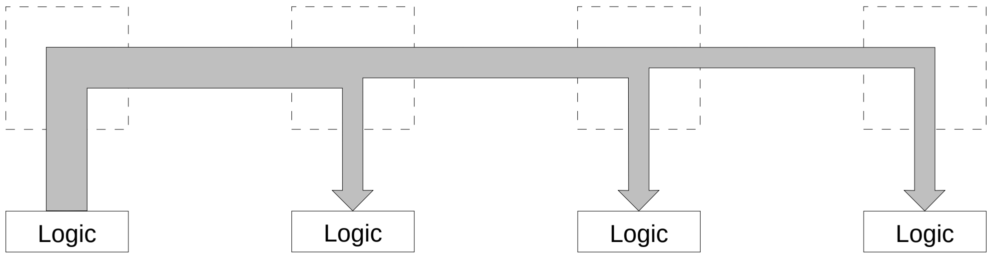
Since the routing of handshaking pairs has to match the routing of their associated data bundles, the topology of the data array routing determines the topology of the timing array routing. Rather than discuss the merits of various routing topologies, this section concentrates on the two basic routing operations of fan-in and fan-out of data bundles. To illustrate the discussion, the simple one-dimensional data array and timing array routing architectures of Figure 8.3 are used. The data array consists of logic blocks that are connected by a number of input and output signals to one routing switchbox. These routing switchboxes are connected to from a one-dimensional array. A similar timing array routing structure is used consisting of timing cells connected by a number of handshaking links to a one-dimensional array of handshaking switchboxes.

Figure 8.4 illustrates the fan-out and fan-in of data using the routing architecture introduced in Figure 8.3. In Figure 8.4(a), data signals fan out from one logic block to three destination blocks. Figure 8.4(b) illustrates the reverse data flow pattern; data fans in from three source logic blocks to a single destination.

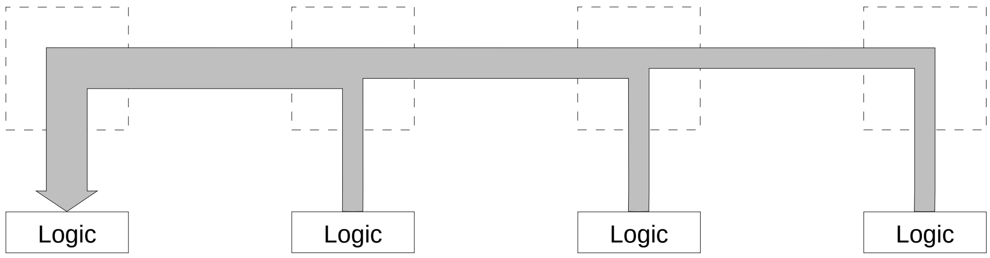
Both fan-in and fan-out of data flow can be synchronised using three timing cell to timing cell links as illustrated in Figure 8.4(c). The routing of the handshaking signals consists of two separate signal paths, one for the request signals and one for the acknowledge signals. One path consists of the handshaking signals fanning out from the timing cell on the left to the three timing cells on the right; the other path consists of three timing cells on the right fanning in to the timing cell on the left. The routing paths are inverses of each other; this allow the same routing to be used for both the fan-in and fan-out data flows by just swapping which signals are requests and which are acknowledges. This is configured using the `DIR` configuration bits in the timing cells rather than by the timing array routing configuration.

In Figure 8.4(c), the synchronisation of the fan-in handshaking signal occurs in the timing cells; each fanning in handshaking signal is routed to the destination timing cell on the left, which synchronises the three handshaking signals as a group. Likewise, the fan out of the handshaking signals also occurs in the timing cell; the three fanning out handshaking signals are routed separately after leaving the timing cell. This pattern of routing fan-out handshaking signals is wasteful, as the same signal is routed separately to three destinations.

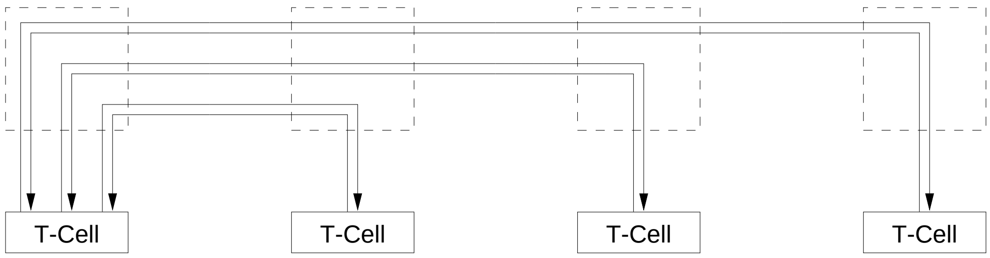
Rather than routing the same signal three times through the timing array, it could be routed once, fanning out when required in the routing. However, this would break the symmetry in routing of the handshaking pairs. This sym-



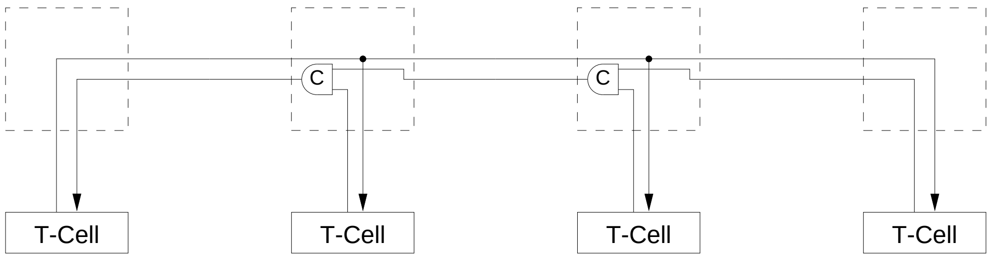
(a) Fan-Out Data Flow



(b) Fan-In Data Flow



(c) Synchronisation in Timing Cell



(d) Synchronisation in Routing

Figure 8.4: Fan-Out/Fan-in Routing

metry is important as it allows the same configuration data to be used to configure the flow of handshaking signals in both directions, since the signal paths are the reverse of each other. Also, the routing of handshaking pairs rather than handshaking signals provides a clean interface to the timing cells and other routing structures, such as those discussed in Section 8.2.

To allow handshaking signals to fan out in the routing, and to continue to route handshaking pairs requires synchronisation to be included in the timing array routing to synchronise the fan-in of handshaking signals. This is illustrated in Figure 8.4(d). The implementation of the timing array routing is more complex, since it requires that the routing be capable of implementing C-Muller gates rather than simply routing signals. However, allowing synchronisation saves on routing resources. Only one handshaking pair is routed through the timing array, fanning out and synchronising when required. Also, only one timing cell link is required into the timing array on the left, rather than the three used when synchronisation could only occur in the timing cell.

The above example has illustrated the complexity of routing handshaking pairs if synchronisation of handshaking signals is not implemented in the routing. For a data bundle, if synchronisation is not implemented in the routing, every source timing cell of the bundle must route a handshaking pair to every destination timing cell of the bundle. Hence, for N timing cells communicating to M other timing cells using a common data bundle, NM timing cell to timing cell links are required. In comparison, if synchronisation is included in the routing then only one path through the timing array routing is required, which synchronises all the timing cells that connect to the data bundle. Structures for synchronising handshaking signals in the routing are described below.

8.3.1 Handshaking Crossbars

For routing data, the most general routing switchbox is the crossbar switch, since it allows any input signal to be routed to any output signal. Other data routing switchboxes can be considered as a subset of the routing in the crossbar switch. Hence, a handshaking switchbox which can synchronise flows for a crossbar switch in the data array is the most general type of handshaking switchbox. Such a handshaking switchbox is termed a *handshaking crossbar*. Below, handshaking crossbars are developed using the reconfigurable C-Muller gate.

It is useful to consider in a crossbar switch how bundles of data signals are routed, rather than individual signals. For bundles of data signals being

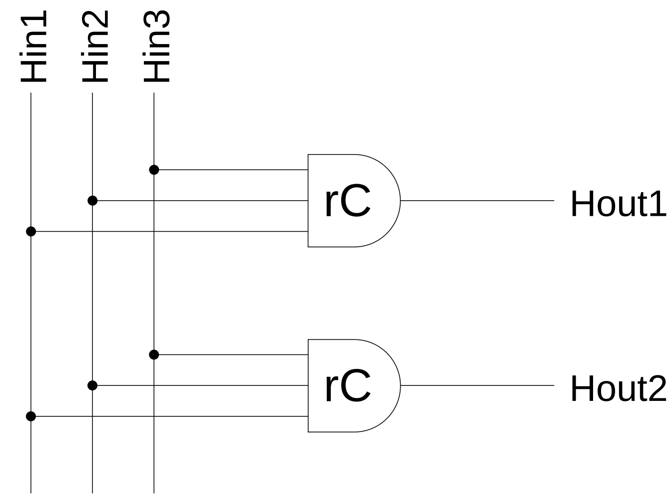


Figure 8.5: 3:2 Reconfigurable C-Muller Crossbar

routed to and from a crossbar switch in the data array, any output data bundle can take input signals from any subset of the input data bundles. Similarly, a signal from any input data bundle may route to any subset of the output data bundles. To synchronise such a data flow in a handshaking switchbox, each output handshaking signal must be able to synchronise on any subset of the input handshaking signals. This behaviour can be implemented by a reconfigurable C-Muller gate with connections to all the input handshaking signals. For multiple outputs, the reconfigurable C-Muller gate is duplicated. This structure is called a *reconfigurable C-Muller crossbar* (rC-crossbar). Figure 8.5 illustrates a 3:2 rC-crossbar; the two output handshaking signals can synchronise on any subset of three input handshaking signals.

The rC-crossbar only synchronises individual handshaking signals, not the handshaking signal pairs which are routed in the timing array. Two ways exist to create a structure that works on handshaking pairs. The first way is to use a rC-crossbar with the same number of input handshaking signals as outputs. As a result, each input handshaking signal can be paired with an output to create a handshaking signal pair. Such a structure is termed a *common input/output handshaking crossbar*. Figure 8.6 illustrates a 3:3 rC-crossbar used to create a common input/output handshaking crossbar.

The common input/output handshaking crossbar is termed as a crossbar since it allows any handshaking signal pair to synchronise with any other handshaking signal pair. It is termed as common input/output as it does not assign a direction of data flow to the handshaking pairs that it synchronises. Thus, it can be used to synchronise data flow across crossbar switches which have bidirectional data flows, i.e. every input/output connected to every other input/output.

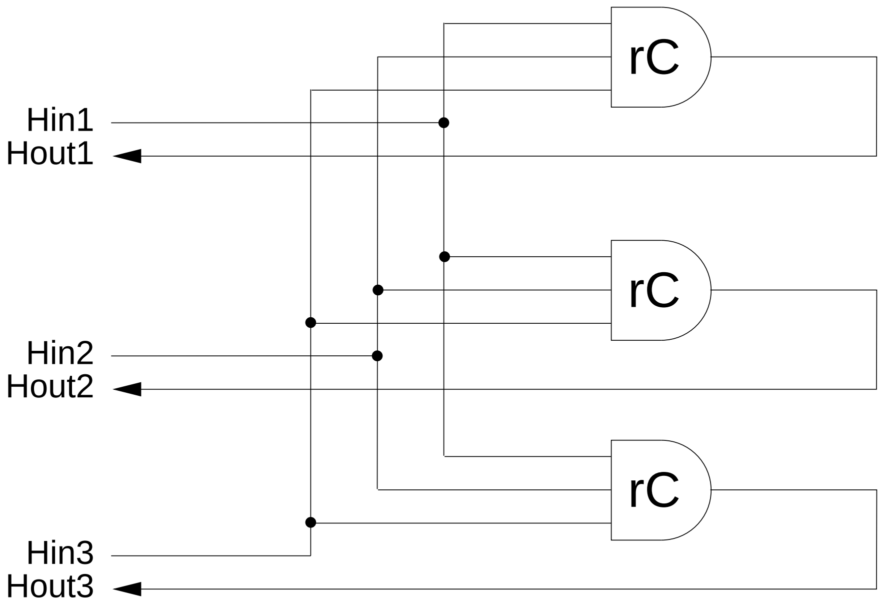


Figure 8.6: Common Input/Output Handshaking Crossbar

The second way of taking the rC-crossbar and creating a structure that operates on handshaking pairs is to take two rC-crossbars and use them to route handshaking signals in either direction. Figure 8.7 shows a 3:2 rC-crossbar coupled with a 2:3 rC-crossbar; it can synchronise any of the handshaking pairs on the left with any subset of the handshaking pairs on the the right. This structure is called a *disjoint handshaking crossbar*. It is termed disjoint as data flows can only be synchronised that flow from left to right or right to left. There is no way to synchronise data that exits on the same side that it enters. The disjoint handshaking crossbar can be used to synchronise data flows across crossbar switches with fixed input and output connections.

The disjoint handshaking crossbar can be considered as a subset of the possible synchronisations of a common input/output handshaking crossbar with the same total number of handshaking pairs. For instance, the 3:2 disjoint handshaking crossbar is a subset of the common input/output handshaking crossbar with five handshaking pairs.

The number of configuration bits needed for the reconfigurable C-Muller gates in a handshaking crossbar can be halved by sharing configuration bits. Configuration bits can be shared, since both the forward and reverse paths of the handshaking signals are synchronised in the handshaking crossbar. Thus, if the output of handshaking pair X is configured to synchronise on the input

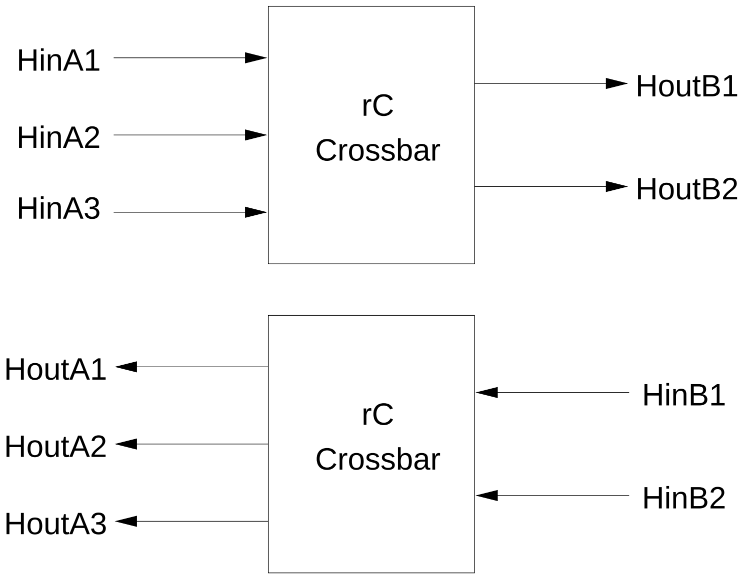


Figure 8.7: 3:2 Disjoint Input/Output Handshaking Crossbar

of handshaking pair Y, then the output of handshaking pair Y must synchronise on the input of handshaking pair X. This allows the same configuration bit to be used to configure both synchronisations.

8.3.2 Delay Matching

A critical property of the timing array routing is that it has to maintain the bundling constraint for handshaking signals that are associated with a data transfer. So far, this has been achieved by using the same topology of routing network to connect the switchboxes in the timing array and data array. The main difference between the two structures is that in the data array switchbox a multiplexor is typically used to implement the routing, whilst in the timing array a reconfigurable C-Muller gate is used to implement the routing.

To ensure the bundling constraint is met, it has been assumed that the reconfigurable C-Muller gate has a longer delay then the multiplexor. In some cases, this may not be so, for example, heavily loaded data signals may take longer to change than the time for the reconfigurable C-Muller gate to change. If this can occur, then additional delay elements need to be included with the C-Muller gates to ensure the bundling constraint is met. Generally, fixed delay elements would be added, as the cost of providing variable delay elements would be prohibitive.

A problem with placing additional delay on the C-Muller gate is that it adds additional delay to handshaking signals that are not bundled with a data transfer, as well as those which are. The handshaking crossbar can be adapted in several ways so that handshaking signals not bundled with data transfers are not delayed. One way is to add configuration bits, one per C-Muller gate, that determine whether the additional delay is required to ensure the bundling constraint is met. The disadvantage of this is that it requires extra circuitry, which also increases the delay. Another way to ensure that handshaking signals not bundled with a data transfer are not delayed would be to fix which handshaking signals in the routing are associated with a data transfer and which are not. This approach saves on configuration bits, but reduces the flexibility of the timing array routing, since it forces the handshaking pair to transfer data in a specified direction.

An alternative to trying to maintain the bundling constraint is to use a delay-insensitive protocol in the routing. As discussed in Chapter 3, delay-insensitive codes can be expensive to implement. Dual-rail encoding requires two wires per data bit, whilst delay-insensitive codes that require less wires, such as Sperner or Berger codes, require more complex decoding/encoding circuitry. In general, it is not worthwhile to implement these schemes internally to the FPGA, since the FPGA designer has a large degree of control over the internal chip delays. However, delay-insensitive protocols are useful for off-chip connections, since the FPGA designer has no control over the off-chip routing delays. Due to the packaging cost of input/output pins, the use of delay-insensitive codes that minimise the number of signal wires would be preferable for such routing. The circuitry required for encoding and decoding can be shared, along with I/O pins, by using the route multiplexing scheme described in Section 8.2.2.

8.4 Distributed C-Muller Gate Routing

In the handshaking crossbar structures developed so far, the implementation of the C-Muller gate has not been specified; either centralised or distributed reconfigurable C-Muller gates could be used. Unless large fan-in C-Muller gates are required, a centralised gate would be favoured, since all the handshaking signals are already routed to a central switchbox. However, the distributed C-Muller gate has the potential for a different style of timing array routing, where the synchronisation is distributed across the wires, and wires are joined using standard routing switchboxes rather than the handshaking crossbars developed earlier.

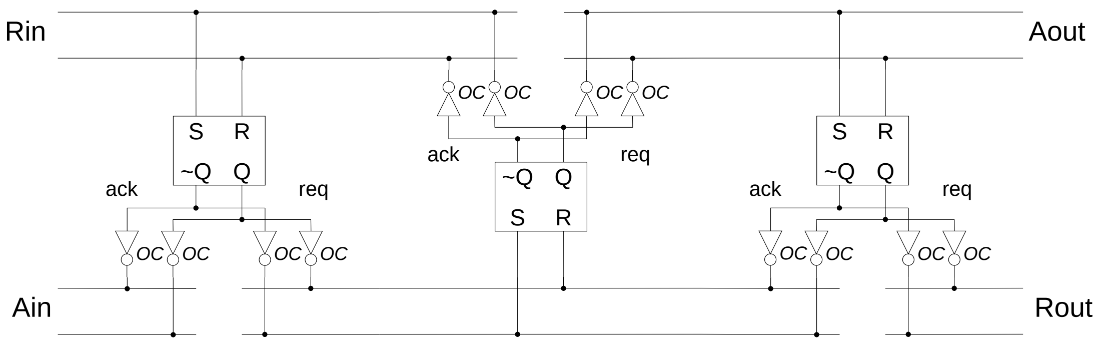


Figure 8.8: Pipeline using Distributed C-Muller Gates

Figure 8.8 illustrates the control circuitry for a two-phase pipeline implemented using distributed C-Muller gates. For reasons of clarity, the weak pull-up resistors and delay elements have been omitted from the figure. The outputs of the distributed C-Muller gates drive two synchronisation wire pairs. The right-hand set of open-collector drivers are request inputs to the next distributed C-Muller gate, whilst the left-hand set of open-collector drivers are acknowledge inputs to the previous distributed C-Muller gate. Since the acknowledge inputs to the C-Muller gate are inverted, the acknowledge drivers are inverted with respected to the request drivers. Data can be routed in the opposite direction through the pipeline using the same control structure, by swapping the request and acknowledge signals.

Labelling of the off-figure connections is more difficult with the distributed implementation, since rather than labelling individual wires, pairs of synchronisation wires must be labelled. Also, since the synchronisation wires are bidirectional, it is not possible to identify inputs and outputs to the circuit. However, Figure 8.8 is labelled in a similar way to the previous pipeline circuits. This labelling, rather than being based on the direction of signal flow, is based on the direction of event flow in the figure. Events are considered to travel from the open-collector drivers to the SR flip-flops. Once the direction of event flow is identified, then the off-figure connections can be labelled, in a similar way to previous pipeline control circuits.

In comparison to the implementation of Micropipelines using centralised C-Muller gates, the distributed C-Muller gate pipeline is more complex, since instead of one signal being routed for each handshaking signal, two wired logic signals are required. However, to create larger fan-in gates, no further signals are required, merely the extension of the synchronisation wire pair. This is illustrated in Figure 8.9, which shows a fan-out of two from the stage on the left to the two stages on the right. The fan-in of the acknowledge signals from the

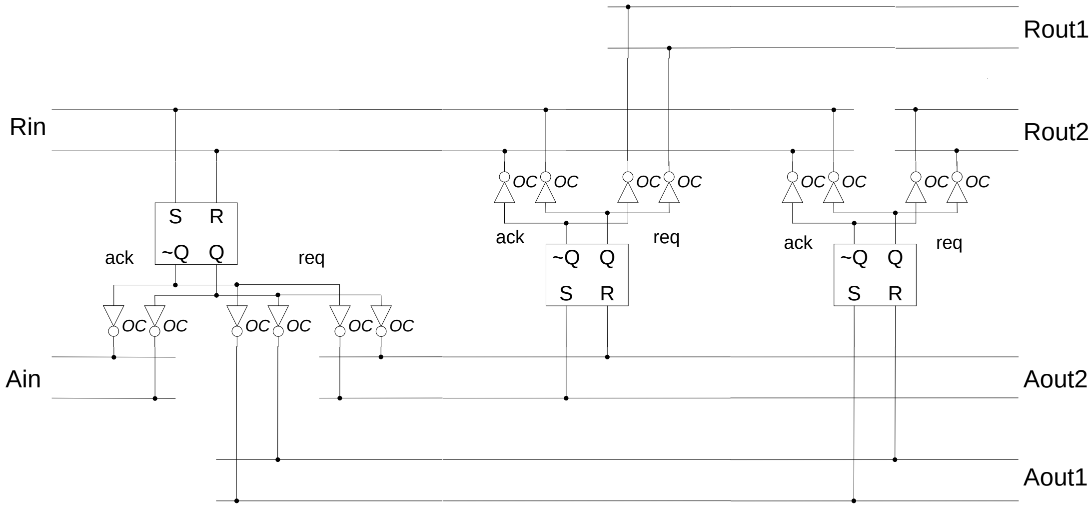


Figure 8.9: Two Way Fan-Out

two stages on the right can be implemented by connecting all the timing block acknowledge drivers to the same pair of synchronisation wires. In contrast to the centralised C-Muller gate, no additional signals are required.

However, the reverse process of fan-out of the request signals becomes more complex. In the centralised C-Muller gate architecture, the fan-out of these signals can be done by simply routing the same signal to all the destinations. In the distributed architecture, each synchronisation wire pair must be isolated; i.e. driven by separate drivers. In Figure 8.9, the left-hand stage has two separate sets of request drivers, to ensure that the fan-out request signals are isolated. If the synchronisation wire pairs are not isolated, then all the fan-out stages are synchronised together, as a group. Hence, although there is a gain in the ease of creating large fan-in C-Muller gates, the complexity of the fan-out routing is increased, which generally negates any advantage from the ease of creating large fan-in gates.

One exception to this is when the desired behaviour is to synchronise a group of timing blocks. Figure 8.10 illustrates a timing block on the left communicating to a group of two timing blocks on the right. The group of two timing blocks are duplicates; both are connected to the same input synchronisation wires and drive the same output synchronisation wires. This duplication seems wasteful, but the alternative scheme would be to use one timing block and distribute the memory control signals over the region covered by two timing blocks. This requires extra routing; the cost is not insignificant considering that the minimal circuitry for the timing block is only a SR flip-flop plus open collector drivers.

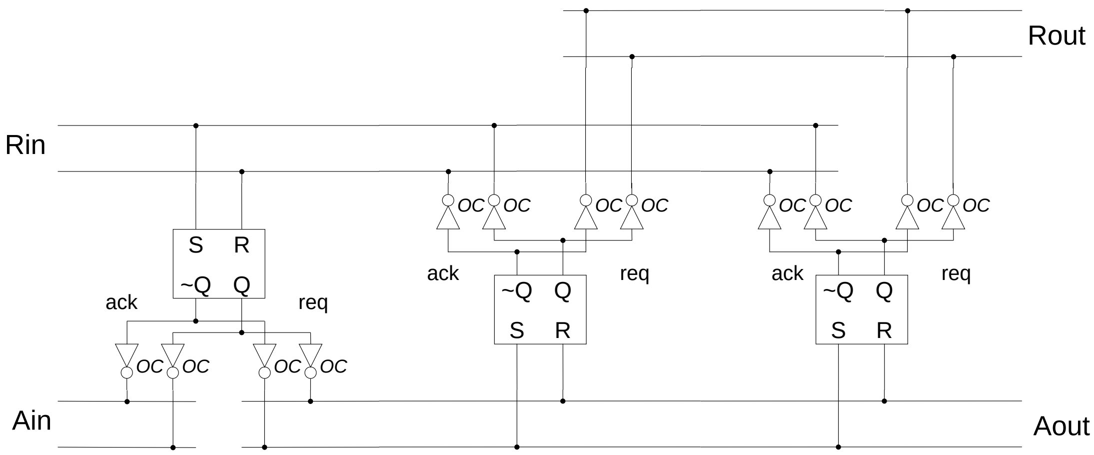


Figure 8.10: Grouping of Two Timing Blocks

Furthermore, synchronising groups of timing blocks is advantageous for its delay properties. The group of timing blocks is constrained to wait for the slowest timing block in the group before proceeding. This allows locally determined delay schemes to be used for the timing block group. The alternative scheme, of grouping data cells by using one timing block to control a larger region of the data array, requires a more sophisticated delay scheme, so that the single timing block could account for the wider range of delays. Thus, the ease of grouping timing blocks in the distributed architecture allows simpler local based delay schemes to be used.

8.4.1 Distributed Timing Cell

The routing structures based on the distributed C-Muller gate differ greatly from those developed for the centralised C-Muller gate. These differences, especially the minimal amount of circuitry required for a timing block, suggest that a different style of timing cell could be developed which is suited towards a distributed C-Muller gate routing architecture. As discussed in chapter 6, the distributed C-Muller gate suits fuse based configuration, since the wired logic requires low resistance bidirectional lines. Fuse based configuration precludes dynamic reconfiguration which is the focus of this thesis. However, the work is included as it illustrates both the wide applicability of the STACC architectural model, and also how a different configuration technology influences the design of the timing array.

The minimum circuitry required for the distributed timing cell is a SR flip-flop plus open-collector drivers. This circuitry is small enough that for many choices of data cell design, the overhead of providing one timing cell per data

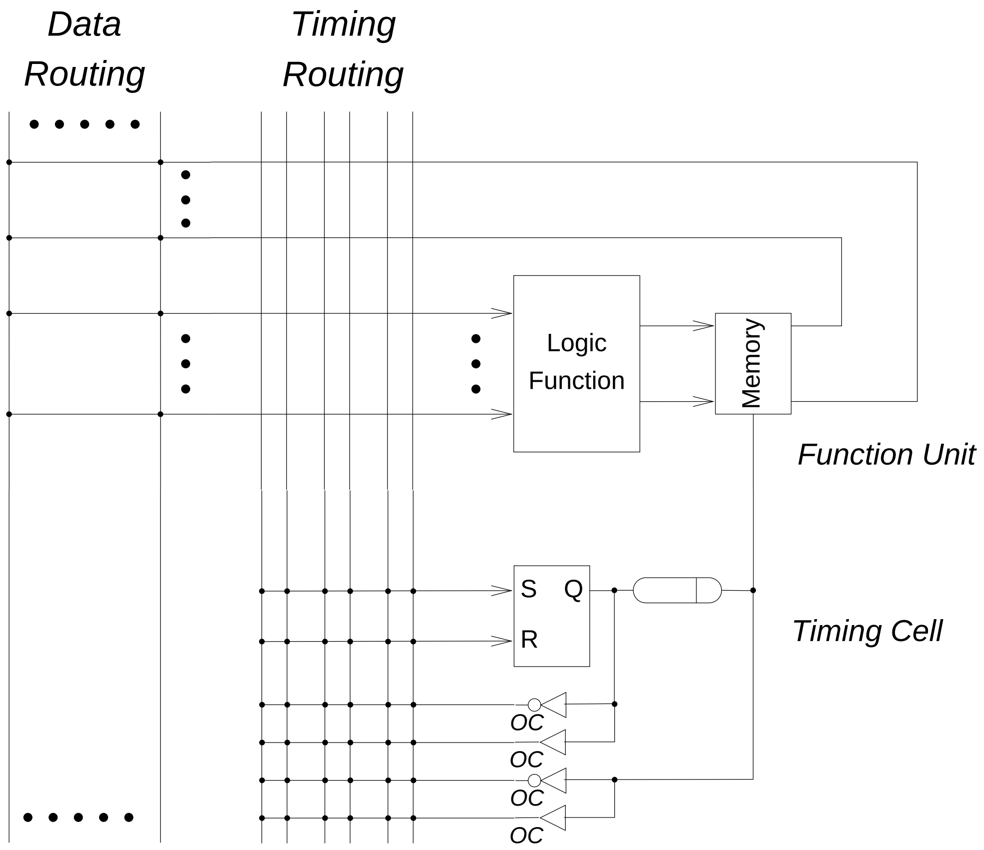


Figure 8.11: Distributed Timing Cell and Data Cell

cell would not be unreasonable. Furthermore, the previous section has illustrated that the main advantage of the distributed architecture is its ability to group timing blocks together. This suggests that a reconfigurable distributed timing cell could be provided at a fine level of granularity and then grouped together to from larger timing cell groups.

Figure 8.11 illustrates a distributed timing cell providing memory control to one data cell. The timing cell is similar to that used in the previous pipeline example with the addition of the delay element to the cell. The figure is laid out to highlight similarities with fuse based architectures such as the Cypress pASIC380 [24] series of FPGAs. The architecture maintains the clear separation between timing cell and data cell that was a design aim of the STACC architectural model. This differentiation is also maintained by using separate routing for the data signals and timing control synchronisation wires.

The only additional element to the distributed C-Muller gate in the timing cell is the delay element. Since the timing cell is going to be provided at a fine level of granularity, a simple delay element, such as a fixed unreconfigurable delay element, is preferable. More complex delay elements will significantly increase the size of the basic timing cell. Even though no variation is introduced by the individual fixed delay elements, variation in the delays of groups of cells occur, as larger groups will take longer to distribute the synchronisation wire values to all cells.

In contrast to the centralised timing cell design, the two-phase protocol is preferable in the distributed design. In the centralised timing cell, the four-phase protocol was preferred, since it made the basic branching structure simpler and allowed simpler memory elements to be used. This was at the expense of more complex synchronisation circuitry to deal with the recovery phase of the handshaking protocol. However, in the distributed design, a fine granularity of timing cell is being used, so making the timing cell more complicated incurs a greater area overhead. Also, communication time is an issue in the distributed architecture; the rise times on the synchronisation wires are slow, since the wires are only driven high by weak pull-up resistors. Thus, the additional recovery phase in a four-phase protocol may cause communication times to dominate over computation time. Hence, two-phase signalling is the preferred protocol for a distributed implementation.

8.4.2 Distributed Control Blocks

In the centralised timing cell, developed previously, branching and merging were done on a per link basis. Each link could be individually selected on each cycle of the timing cell. This is hard to achieve in the distributed timing cell, since individuals links are not routed, only the synchronisation wires. Also, the previous section has argued for providing a fine level of granularity of timing cell to data cells. Adding the complexity of branching and merging could significantly increase the size of the timing cell. To maintain a small size of timing cell, it is better to provide branching and merging as separate cells, though this does not integrate the two functions, which was an advantage of the centralised design.

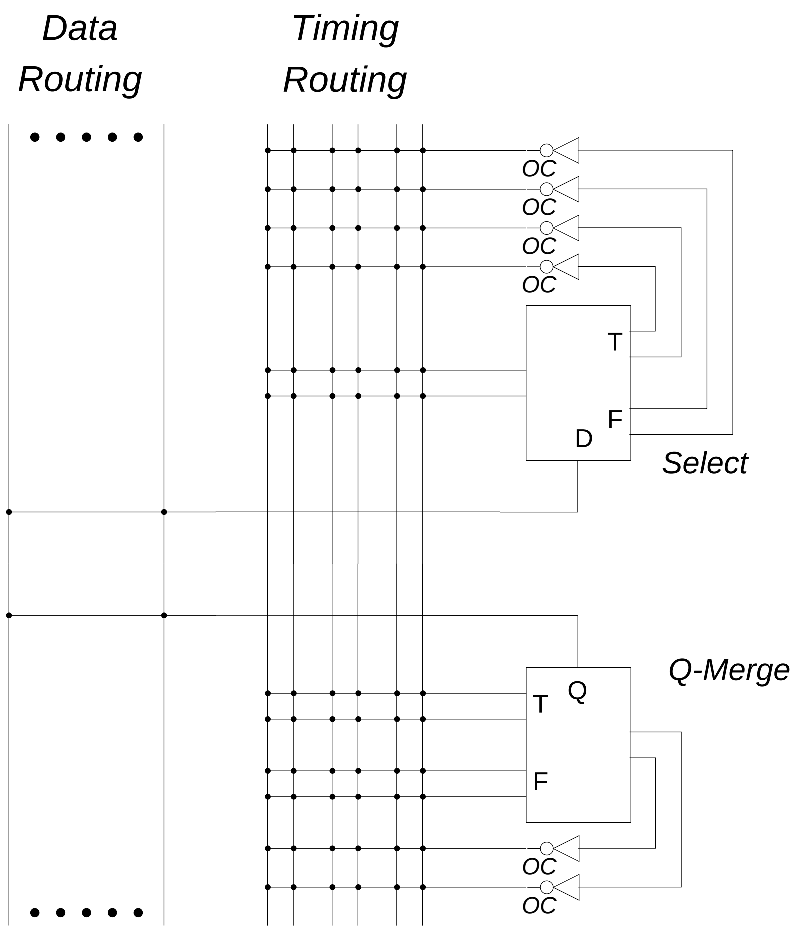


Figure 8.12: Distributed Control Structures

The Select and Q-Merge pair introduced in chapter 6 are good candidates for such branching elements, since they provide a wide range of control function with a minimum of circuitry. More complex branching and merging structures can be made by connecting several of the Select and Q-Merge gates to the same input or output synchronisation wire pairs.

Figure 8.12 illustrates the use of the Select and Q-Merge gates to allow branching and merging in the distributed architecture. These gates operate directly on synchronisation wires; this allows fan-in or fan-out synchronisation of the signals going into and out of the control blocks. Q-Merge and Select gates that operate on synchronisation wire pairs, rather than single wires can be implemented by placing SR flip-flops on the input synchronisation wires, to convert to a single wire, and then using the standard implementations of the Q-Merge and Select gates. The output of the gates and their inverses are used to drive open collector drivers for the output synchronisation wires.

8.4.3 Comparison with Centralised Timing Cells

This section has developed a distributed timing cell for STACC, which is very different from the centralised one. Many of the design decisions made for the centralised timing cell are made differently in the distributed architecture to exploit the different routing architecture.

Compared to the centralised timing cell, the simplicity of the basic distributed timing cell favours a fine granularity architecture where a timing cell can be provided per data cell. This large number of fine grain cells are grouped together using the distributed routing architecture. The simplicity of the basic timing cell also leads to the decision to separate branching and merging elements from the timing cell. This differs from the centralised timing cell where an integrated approach is favoured. Finally, the slow transitions of the wired logic signals favours a two-phase implementation rather than the four-phase one that was adopted for the centralised timing cell.

8.5 Summary

This chapter has illustrated the benefits of self-timed routing. Routing structures can be placed transparently in the routing, whilst maintaining the operation of the system. These structures have important use, in improving throughput of routes, and in sharing I/O resources in self-timed FPGAs. The chapter also developed structures for routing handshaking signal pairs in the STACC timing array based on the reconfigurable C-Muller gate. Handshaking switchboxes based on the handshaking crossbar allowed the fan-in and fan-out of data bundles to be synchronised in the timing array.

The final part of the chapter considered routing structures based on the distributed C-Muller gate. The distributed synchronisation structures were flexible enough to enable all the synchronisation to be performed in the timing array routing, rather than in the timing cell. This led to a very basic, stripped down form of timing cell, which could be provided in large numbers and synchronised as a group using the distributed routing architecture. The distributed timing cell is not developed further, since it suits fuse based configuration, and hence is not suitable for the dynamic hardware applications on which this thesis is concentrating.

Chapter 9

Self-Timing the Xilinx XC6200

9.1 Introduction

This chapter is the first of three concerned with applying the STACC model to create a new self-timed version of the Xilinx XC6200 architecture. In this chapter, the synchronous XC6200 architecture is introduced and a self-timed version created using the STACC model. The following chapters demonstrate the use of the self-timed XC6200 for dynamic hardware and evaluate the architecture with respect to the original synchronous XC6200. Chapter 10 presents a case study concerning the design of run-time parameterised circuits for finite field operations using the self-timed XC6200 architecture. Chapter 11 compares the synchronous and self-timed XC6200 architectures, and proposes improvements to the self-timed architecture.

For the study, it was decided to design a STACC architecture using a current synchronous FPGA architecture for the data array. Using a current architecture for the data array, allowed design experience and design tools to be transfered to the new self-timed version of the architecture. Also, choosing a current architecture allowed comparison to be made between the synchronous and self-timed versions of architectures with the same data array.

The XC6200 in particular was chosen for several reasons. It is a contemporary FPGA architecture (first silicon 1995), and includes features that make it suitable for dynamic hardware systems, an area where self-timing would be expected to provide most benefits. Another factor in choosing the XC6200 was that it's predecessor, the Algotronix CAL was also developed at Edinburgh.

9.2 XC6200 Architecture

9.2.1 Background

The Xilinx XC6200 FPGA [123] is the successor to the Algotronix CAL1024 FPGA [65, 3]. The CAL1024 is a fine grain random access SRAM FPGA composed of a nearest neighbour array of integrated routing and logic cells. The XC6200 architecture extends this structure by including non-local routing signals called *flyovers* (or *fast lanes* in later versions of the data sheet), based on a hierarchy of 4×4 blocks of cells.

The other main difference of the XC6200 from CAL is that the XC6200 has been designed specifically for use in dynamic hardware systems, especially the 'co-processor' type (see Section 2.7.1) with a close coupling between FPGA and microprocessor. To support this, the XC6200 has a sophisticated control interface, to minimise the read/write cycles required for configuration and data transfer between FPGA and microprocessor.

9.2.2 Function Block

Figure 9.1 illustrates the multiplexor based function block used in the XC6200. The inputs to the function block are chosen by three $8 : 1$ multiplexors: `X1`, `X2` and `X3`. The inputs to these multiplexors are sourced from the four local inputs to the cell (`N`, `E`, `S` and `W`) and the four flyover signals (`N4`, `E4`, `S4` and `W4`) that cross the cell.

The cell's logic function is built around a $2 : 1$ multiplexor, which generates the combinatorial output function `C`. The select input to the multiplexor comes from the `X1` multiplexor. The data inputs to the `C` multiplexor come from the `Y2` and `Y3` multiplexors; these multiplexors can choose between the inverted and non-inverted forms of the inputs from the `X2` and `X3` multiplexors. Thus, the logic function can be used to create a $2 : 1$ multiplexor with optional input inversions. Such a reconfigurable gate is capable of creating any boolean function of two inputs. The `Y2` and `Y3` multiplexors also allow the output of the memory element to be fed back and used as an input to the logic function without using external routing resources.

The memory element used in the function block is a single bit D-type register, which can be read and written via the configuration interface. The memory element takes its input from the output of the `RP` (Register Protect) multiplexor. When `RP` is clear, the register takes its input from the output of combinatorial logic function `C`. However, when `RP` is set, the output of the re-

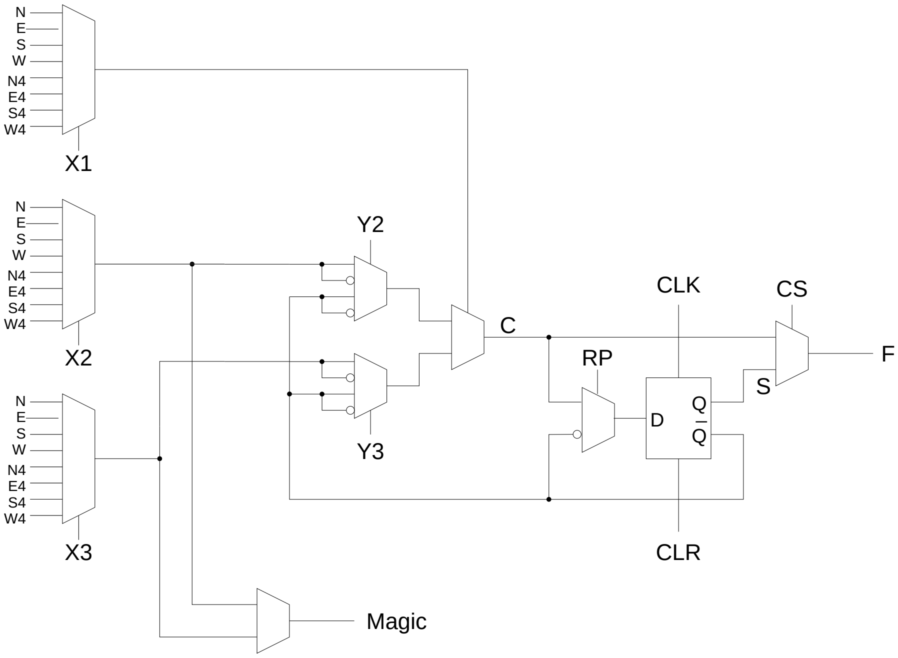


Figure 9.1: XC6200 Function Block

gister is fed back to its input; this protects the register from being updated. The protected register state is useful, as it allows a constant zero or one input to be fed into the function block without using external routing resources. The output of the register feeds into the `CS` (Combinatorial/Sequential) multiplexor. The `CS` multiplexor determines whether the function block output `F` is sourced from the register or from the output of the combinatorial logic `C`.

In addition to the function block output `F`, the function block also provides an output called `Magic`. The `Magic` signal is a special routing resource that routes to the edge of a 4×4 cell block. The `Magic` signal is driven from either the `X2` or `X3` multiplexors; this reduces the number of configuration bits required in the function block, since the `X2` or `X3` multiplexors are used to make the decision of which input to route. However, this means that the `Magic` output can only be used if the configuration of the logic function uses the required signals on `X2` or `X3`.

9.2.3 Interconnect

Figure 9.2 illustrates the basic routing/logic cell used in the XC6200; these cells are arranged in a nearest neighbour grid to form the basic routing structure for the XC6200. The routing function implemented by each cell is a variation on a crossbar switch. Each of the local outputs (`N`, `E`, `S`, `W`) may be sourced from any local input, except from the direction that the output goes to. Instead, this input to the multiplexor is replaced by the output of the function block F.

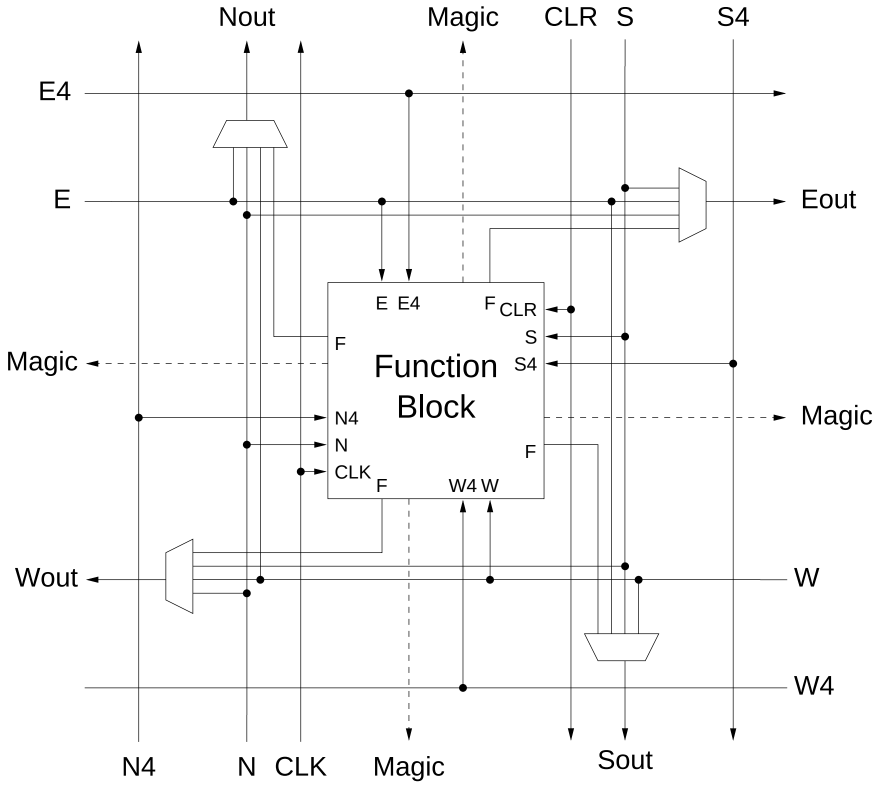


Figure 9.2: XC6200 Cell Routing

The flyover wires (N4, E4, S4, W4) that cross the cell are sourced directly into the function block. For these signals to be sourced onto the local routing, rather than being used as an input to the function block, the function block has to be configured as a buffer. The CLK and CLR inputs connect directly into the function block; CLK signals cross the cell in a North-bound direction, CLR cross the cell in a South-bound direction.

On top of the nearest neighbour grid, the XC6200 implements a hierarchical routing structure. Cells are grouped into 4×4 blocks as illustrated in Figure 9.3. In turn, the 4×4 blocks are grouped into 16×16 blocks and 64×64 blocks. Each level of the hierarchy has its own routing resources known as *flyovers*. Signals can only be routed on to flyovers via boundary multiplexors on the edge of the blocks so, for example, level-16 flyovers can only be driven from boundary multiplexors at the edge of a 16×16 block. Signals can be routed from any level of flyover onto the local and level-4 flyovers by multiplexors on the 4×4 boundaries.

The 4×4 blocks in the architecture differ from the higher levels of the hierarchy in two ways. First, the level-4 flyovers can be used directly as inputs

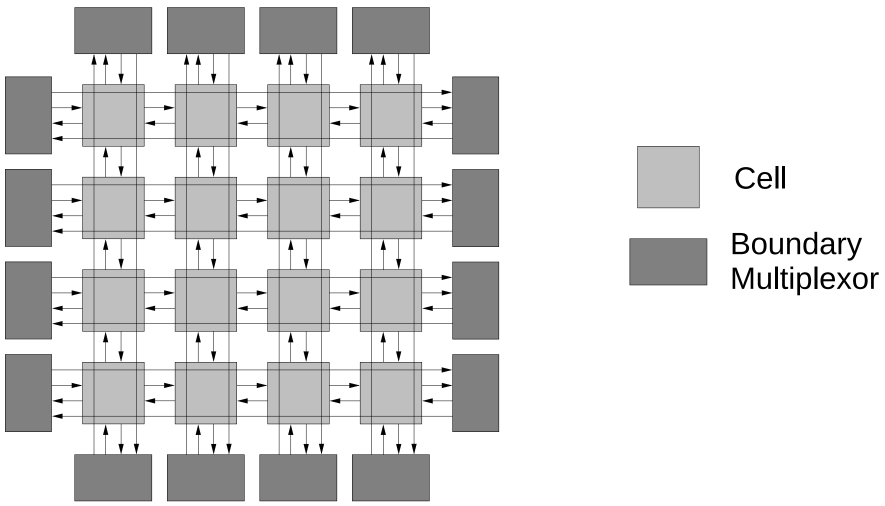


Figure 9.3: 4×4 Block

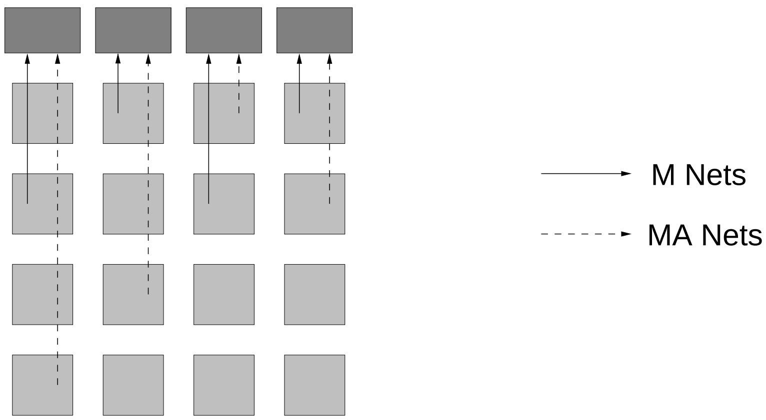


Figure 9.4: XC6200 `Magic` Routing

to the cells. Other flyover signals have to route via the local routing or the level-4 flyovers to reach the cells. The second difference of the 4×4 blocks from the higher levels is that they have additional routing resources provided by the `Magic` signals. `Magic` signals pass from each cell to the edge of the 4×4 blocks. Figure 9.4 shows the routing of the `Magic` signals to one edge of a 4×4 block. Two `Magic` signals route to each boundary multiplexor, known as the `M` and `MA` signals. The signals are designed to allow efficient corner turning in designs. The routing exhibits rotational symmetry, so the pattern of `Magic` routing is duplicated for every other direction. Though, the `Magic` signals exhibit rotational symmetry, the signals do not exhibit reflective symmetry, which prevents designs using the `Magic` signals from being 'flipped' by design software.

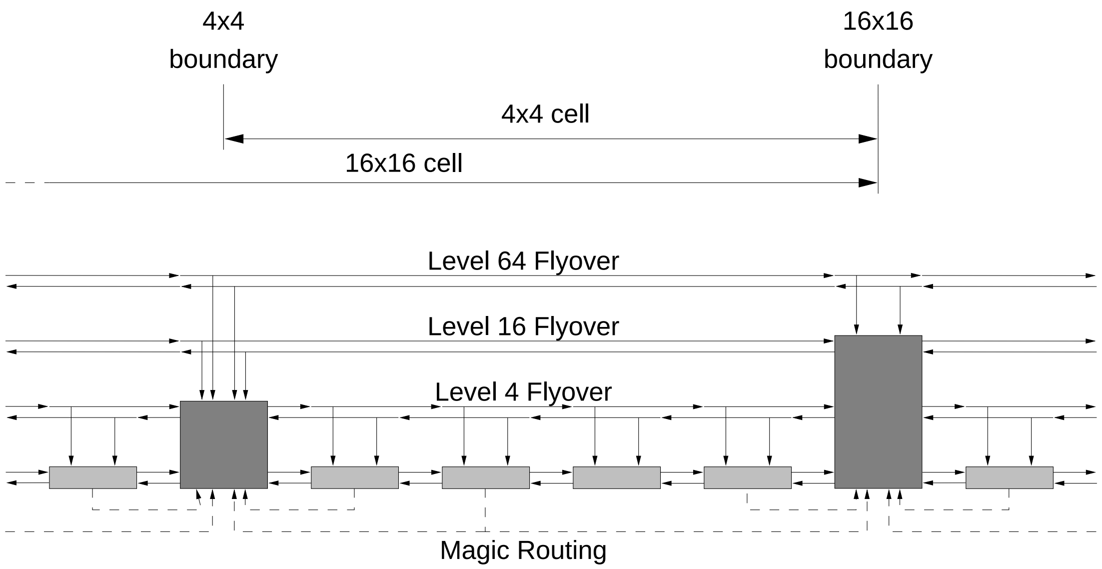


Figure 9.5: Cross-section of XC6200 Routing

The overall routing structure of the XC6200 series is illustrated in Figure 9.5, which shows a cross-section of the routing. Figure 9.5 is valid in both the North-South and East-West directions, since the basic routing architecture exhibits rotational symmetry.

The `CLK` and `CLR` routing are not included in Figure 9.5, since they do not exhibit rotational symmetry. The `CLK` signals are driven from 4×4 block boundary multiplexors and cross the chip in a northerly direction. The `CLR` signals are driven from 16×16 boundary multiplexors and cross the chip in a southerly direction. To ease the distribution of clock and clear signals, four global signals `G1`, `G2`, `GCLK` and `GCLR` are supplied, and can be used as inputs to the boundary multiplexors that drive the `CLR`, `CLK` and `N4` signals.

9.2.4 Configuration Memory and Interface

The configuration memory in the XC6200 uses an SRAM with a random access interface. In addition, the XC6200 also supports a number of serial configuration modes. The XC6200 is designed to be closely coupled to a microprocessor for dynamic hardware applications, so includes two features to facilitate rapid reconfiguration by a microprocessor: the *mask register* and the *wild card address register*.

The mask register allows individual bits in a word of configuration memory to be addressed. Effectively, the mask register gives read/write access permission to certain bits of a word. On writing to the XC6200, masked bits are left unchanged, whilst on reading from the XC6200, masked bits become zero. Without the mask register, to change certain bits in the configuration memory would require a read operation from the SRAM, followed by a series of bitwise logical operations by the microprocessor, followed by a write operation. With the mask register, only one write operation is required, plus the initial set up of the mask register.

To allow a microprocessor to reconfigure large regular designs quickly, the XC6200 includes a wild card address register. The wild card register allows the same value to be written to a group of addresses in one write operation. All configuration memory locations with addresses that match the write address but ignoring those bits set in the wild card register, are written to in one operation. The wild card addressing logic is bypassed on read operations.

Together, the two registers provide a powerful reconfiguration interface. Configuration bits relating to certain functions can be set over a large region of the array in one operation. Regular designs can be configured in very few operations via this interface. Semi-regular designs can be configured by a two-phase configuration scheme of configuring the regular design, and then configuring the exceptional cases. In effect, the XC6200 allows the compression of regular and semi-regular design configurations. This compression is important for dynamic hardware designs as it allows fast reconfiguration. However, it does not compress irregular designs; compression of irregular configurations would require a more general compression scheme.

9.2.5 Input/Output Interface

The XC6200 provides two I/O interfaces: I/O blocks (IOBs) that link the edge of the array to device pins, and register based I/O that utilises the configuration interface to access the memory elements of individual cells.

The XC6200 IOBs provides output to and from the device pins, for signals at the edge of the array. Input data from the IOB pins are routed via modified boundary multiplexors onto the XC6200 routing. The output drivers of the IOBs consists of a data input and an enable input; the data input is driven from the local boundary multiplexor, whilst the enable signal is driven from the level-4 boundary multiplexor. Many of the IOB pins are shared with device control pins. This minimises the pin count of the device, though does mean that not all IOBs may be used in all designs. The control signals interface through the same data/enable arrangement as the data signals. Pins that are not shared with device control pins, may instead be shared with a 'padless' IOB, which interfaces via the unused control signal interface.

A novel feature of the XC6200 is that it allows the FPGA to directly connect to its own control signals via the IOBs. This could be useful, for example, to allow the XC6200 to perform some control functions for itself. For example, it could be configured to control some of the bus control signals, or to perform its own address decoding. An intriguing possibility of this scheme, is self-reconfiguration of the FPGA. For example in a neural net application, the FPGA could change the connections of the neurons depending on an evaluation function. However, self-modifying hardware is likely to be as difficult to implement, use and control as self-modifying code in software.

The other input/output interface to the XC6200 is the register I/O interface, which allows the values of the registers within each cell to be read and written via the configuration interface. A similar mechanism was provided in the CAL1024, but access was only allowed to one cell at a time, since the value was stored with the configuration data for the cell. This limited reading and writing of values to one bit at a time. To overcome this, the XC6200 has separately addressed memories for configuration data and register values.

The register I/O interface is addressed by column; each column of cells in the XC6200 can be selected, and read or written in one read/write cycle. Since the height of the array (64 bits for the XC6020) is larger than the configuration bus width, the XC6200 includes a *map register*, to map the bus values to memory elements in the selected column. The map register signifies which of the cells in the selected column is to be read or written. For read operations, the mapping circuitry takes the register output of the column in the array, and removes unselected bits from the result, and shifts the other bits down. For write operations, the reverse process occurs. The map register is advantageous as it saves the microprocessor from performing a whole series of shift and logical operations to format the data that is transferred between it and the FPGA.

The signals that select the column of registers to be accessed in the XC6200 are named `RegWord`. The `RegWord` signals may be routed into designs from the drivers for the level-16 and level-64 flyovers in the North and South IOBs. By routing the `RegWord` signal to designs, it is possible for designs to detect with the addition of special circuitry when data has been written to or read from a register. This can be used to implement a restricted form of flow control between microprocessor and FPGA. This is discussed below.

Implementing Flow Control in the Synchronous XC6200

An important property of self-timed systems is their flow control behaviour. The routing of the `RegWord` signals to circuits in the XC6200 allows a simple form of synchronous flow control to be implemented in the I/O interface. Circuits can detect when data has been read or written from them using the `RegWord` signals. These signals effectively form a request or acknowledge signal from the microprocessor.

An assumption in this form of flow control is that the FPGA is ready to accept the data or that the data is ready to be read. In simple designs, it can generally be assumed that the FPGA processes data faster than the microprocessor, so that it will always be ready to accept data, and that the FPGA will always produce a result before the microprocessor comes to read it. However, this may not hold when more sophisticated functions are implemented in the FPGA.

For true flow control, where both FPGA and microprocessor can go at their own speed, the microprocessor must check that the FPGA has requested or acknowledged a data transfer. For writing to registers, in the XC6200 this requires an extra read cycle to read back a validity bit to indicate that the FPGA is ready to accept data. For reading from registers, only one cycle would be required if the data validity bit is combined with the data, though extra instructions in the microprocessor would be required to extract this status bit. Both these schemes essentially require busy waiting by the microprocessor until the request or acknowledge signal becomes valid from the FPGA.

An alternative to busy waiting would be to use interrupts. The request or acknowledge signal from the FPGA could be routed to the microprocessor interrupt lines via the IOBs. Given the overheads of interrupt processing in microprocessors, this will only be a good idea when the FPGA takes a long time to produce a result. In virtual hardware systems, interrupts could also be used to indicate virtual hardware faults.

9.3 STACC Architecture Design Process

Before applying the STACC model to create a self-timed version of the XC6200, this section discusses in general terms the overall design process of creating a self-timed FPGA architecture using the STACC model.

9.3.1 Granularity

The key decision in the design process is choosing the granularity of the self-timed architecture. In this context, the grain size being referred to is the size of the self-timed region, i.e. timing cell and the data cells controlled by it, rather than the granularity of individual data cells which is normally referred to in terms such as 'fine grain FPGA architecture'.

The limiting factor in this choice is the overhead involved in implementing and using the timing array for a particular choice of granularity. The overhead has two components, the fixed overhead of implementing the timing array and a configuration dependent overhead, that arises from fitting circuits to the granularity imposed by the architecture.

Architectural Overhead: The architectural overhead arises from the extra circuitry required to implement the timing array, which is not required in a synchronous FPGA architecture. Using a larger size of self-timed region, reduces the overhead since fewer timing cells are required. However, larger self-timed regions require timing cells with more handshaking links and more complex arbitration functions to match the complexities of the data flow in a larger region of the data array. Smaller self-timed regions will generally have simpler patterns of data flow, so can use timing cells with less handshaking links and simpler arbitration functions.

Configuration Dependent Overhead: The second component of the overhead arises from data cells that cannot be used in the data array due to lack of timing cells. Unlike the architectural overhead, this overhead is configuration dependent, since some circuits will fit to the imposed granularity better than others. Smaller self-timed regions have less potential for wasting cells, whilst larger self-timed regions have the potential for wasting large numbers of data cells through only a small number of cells in a self-timed region being used. Since the overhead is configuration dependent, designers will tend to modify the design of circuits to fit the granularity imposed by the architecture.

The above discussion has emphasised two conflicting trends in the choice of self-timed granularity. Large self-timed regions minimise the architectural overhead, but increase the potential waste of data cells through insufficient timing cells being available. The granularity chosen for a STACC based architecture needs to strike a balance between these two trends.

9.3.2 Variable Granularity

Architectures with a fixed granularity force circuits to be fitted to the granularity imposed on them by the architecture, even if this is at odds with the natural granularity of the circuit. Variable granularity architectures allow the size of self-timed region to be adapted to match the natural granularity of the circuit.

Even with the architectures discussed so far, some variability in the size of self-timed regions can be provided. The fixed pattern of local clock routing only bounds the locations of the memory elements within the self-timed region. Other data cells implementing a purely combinatorial function can be located outside this area. However, the larger the self-timed grain becomes, the more difficult it becomes to route signals to and from the registers located in the area covered by the local clock. Thus, this method is only suited to small variations in granularity.

Greater variability in granularity requires specific support by the architecture. Two ways of achieving flexible granularity are possible: a more flexible distribution of local clock signals from timing cells, or the grouping of timing cells together. These two methods are discussed below:

Flexible Local Clock Distribution: Variability in the size of self-timed region can be achieved by allowing data cells to source local clock signals from a number of different timing cells, rather than just one. Several problems exist with this approach. Timing cells can distribute their local clock signal over a wide area, so the required size of the local clock drivers and local clock skew become problems. Flexible local clock distribution also requires more configuration bits to determine the pattern of local clock routing. However, potentially the most severe problem is that the complexity of the timing cell may not match the complexity of the data flow that it controls, especially if one timing cell is used to synchronise a large region of the data array. Thus, this technique is more suited to allowing small variations in granularity.

Grouping Timing Cells: The other approach to providing variable granularity is for a group of timing cells to synchronise before generating their local clock signals. Potentially, grouped timing cells can provide the facilities of one large timing cell, for example, two timing cells with four links could be grouped to provide one with potentially eight links. This is advantageous, since the complexity of the timing cell scales with the complexity of the associated data array.

Of the two methods described above, flexibility in the local clock routing is easier to implement, since the forms of clock distribution used in synchronous FPGA architectures can be adapted for it. However, this approach does not scale the complexity of the timing cell with the complexity of the associated data array region. Hence, the second approach of grouping timing cells is better, however this requires new self-timed structures to implement the grouping. Possible structures for grouping timing cells in the self-timed XC6200 are discussed in Chapter 11.

9.3.3 Other Design Decisions

The choice of granularity is the most important design decision in the creation of a self-timed FPGA architecture using the STACC model. Once the choice of granularity has been made, the other design decisions in the architecture can be made. Essentially, most of these decisions involve defining the interfaces of the timing cell to other parts of the architecture: the timing cell to timing cell interface (timing array routing); timing cell to data array interface (the probe and select routing); timing cell to environment interface (timing cell I/O interface) and the timing cell to microprocessor interface (configuration interface). Finally some design decisions need to made concerning the behaviour and implementation of the timing cell, in particular, the choice of delay element.

These decisions are discussed below:

Timing Array Routing: Chapter 8 discussed timing array routing structures. The most important aspect of the timing array routing is that it should be able to match the routing patterns and delays of data bundles in the data array.

Select Routing: The routing of select signals can be made with minimal disruption to the data array by including additional outputs for the select signals from routing switchboxes that are already present in the data array. Signals are routed to the select output via the standard data array routing resources. To minimise the use of routing resources to them, switchboxes with select outputs should be well connected to the other parts of the self-timed region.

Arbitration and Probe Routing: An aspect of the timing cell behaviour that has to be defined is the type of arbitration block it uses. Arbitration schemes were discussed in Section 7.6. The choice of arbitration function determines how many probe signals have to be routed to the data array. The routing of probe signals to the data array can be done by providing additional inputs to routing switchboxes already present in the data array. It is preferable that undefined routing configuration values in the routing switchbox are used, otherwise additional configuration bits are required and the format of the configuration memory would be disrupted. Once the probe values have entered the data array, they are routed using the data array routing resources to their destination.

Timing Array I/O Interface: I/O interfaces for the timing array signals were considered in the previous chapter. The discussion showed how self-timing can allow self-timed FPGAs to be extended transparently to multi-chip arrays.

Configuration Interface: For fuse based configuration memories, no change is required to the configuration interface in a self-timed FPGA. For serial SRAM configuration interfaces, self-timed FIFOs are used instead of shift registers for the configuration memory elements. Finally, random access SRAM based interfaces require an acknowledge signal to be generated when the read or write from the SRAM has been completed. This needs the additional of a bundled delay element matched to the SRAM read/write delay.

Another issue in the design of the configuration interface is the mapping of configuration bits into the configuration memory space. Placing the data array and timing array configurations in separate memory spaces ensures a regular layout of both memory spaces. This regular layout is useful when wild card addressing schemes are used.

Delay Methodology: The choice between a fixed delay and variable delay methodology has important implementations for the performance of the self-timed architecture, and the complexity of its implementation. Delay elements were discussed in Section 6.4.

Timing Cell Implementation: A number of implementation decisions are left which determine the internal behaviour of the timing cell. These implementation decisions, such as the choice of protocol and decoupling, can have important implications for the performance of the timing cell. These issues were discussed in Chapter 7.

9.4 Self-Timing the XC6200

The previous section has outlined the design decisions required in creating a self-timed architecture using the STACC model. This section describes the decisions taken in self-timing the Xilinx XC6200.

9.4.1 Granularity

The hierarchical structure of the XC6200 provides a series of natural sizes for the self-timed region. The smallest of these is the basic cell in the architecture. Providing self-timed control for individual cells results in a massive overhead for implementing the timing array. The maximum size of a data bundle is one bit, so the overhead of providing separate request and acknowledge signals is enormous. Additionally, the timing cell is liable to be far slower than the data cell so leading to poor performance. However, providing self-timed control for single data cells has the benefit that data cells cannot be wasted through insufficient timing cells being available.

The next block size up is the 4×4 block. In one direction, this gives a potential data bundle width of eight bits, if all the local and level-4 flyover connections are used. The overhead of providing handshaking signals for this size of data bundle is far more reasonable than for individual cells. Another benefit of using the 4×4 block is that it encloses other asymmetric aspects of the XC6200 architecture, such as the `Magic` routing, and the special status of level-4 flyovers, which unlike other flyovers can be directly used as inputs into the cells.

The larger groupings of 16×16 and 64×64 cells give very large data bundles. For these sizes of block the overhead of providing the handshaking signals is minimal, but the potential waste of data cells when only a small part of the self-timed region is used is massive. Thus, the 4×4 block provides the most suitable granularity for self-timing the architecture. The choice of the 4×4 block as a unit of minimum granularity was also made by Brebner and Kwok [16, 69] in their work on virtual hardware operating systems for the XC6200.

In the XC6200 data sheet [123], the repeating unit of the 4×4 cell block is defined to include the boundary multiplexors that drive the outputs from the 4×4 block of cells. Instead, here the 4×4 block is defined as including boundary multiplexors that drive the inputs to the 4×4 block of cells. The reason for this choice is that it allows routing between cells in a 4×4 block that uses the level-4 flyover signals to be included logically in one 4×4 block. This is a common routing structure in the XC6200. For example, to route between cells in the same row or column but on opposite sides of a 4×4 block, the signal delay is smaller if the signal is routed via the level-4 flyover rather than via the local routing.

9.4.2 Variable Granularity

Section 9.3.2 presented two schemes for implementing variable granularity. The first, based on a local clock distribution network, could be implemented to a limited extent using structures already present in the synchronous XC6200. The XC6200 provides local clock routing for each column through clock drivers situated in the Northbound boundary multiplexors of each 4×4 block. In the self-timed XC6200, these clock multiplexors could source their inputs from a number of local clock signals generated by nearby timing cells. However, the problem with local clock distribution schemes is that the complexity of a timing cell does not scale with the size of its associated data array.

The second scheme discussed for implementing variable granularity involves grouping timing cells together. This is advantageous, since it can scale the complexity of the timing control with the size of data array controlled. However, structures for grouping timing cells have an additional implementation cost for the timing array.

For this work it was decided to use a fixed size of self-timed region consisting of a timing cell and a 4×4 block of data cells. This decision to use a fixed granularity was made, partly for reasons of simplicity in the design of the architecture, and partly as it allows the problems with using a fixed granularity to be assessed. Potential extensions to the self-timed XC6200 architecture to allow the grouping of timing cells are considered in Chapter 11.

9.4.3 Timing Array Routing

As discussed in Chapter 8, it is important that the timing array routing delays should match the delay of data bundles in the data array. The simplest way to achieve this is by using similar structures for the timing array routing and data array routing. This approach is adopted in Figure 9.6, which shows a cross-section of the routing structure used in the self-timed XC6200. The timing array routing is split into different levels, just as the data array routing has different levels of flyovers. The similarities in the timing array and data array routing can be seen by comparing the interconnect of the handshaking switchboxes in the timing array, and the interconnect of the boundary multiplexors in the data array.

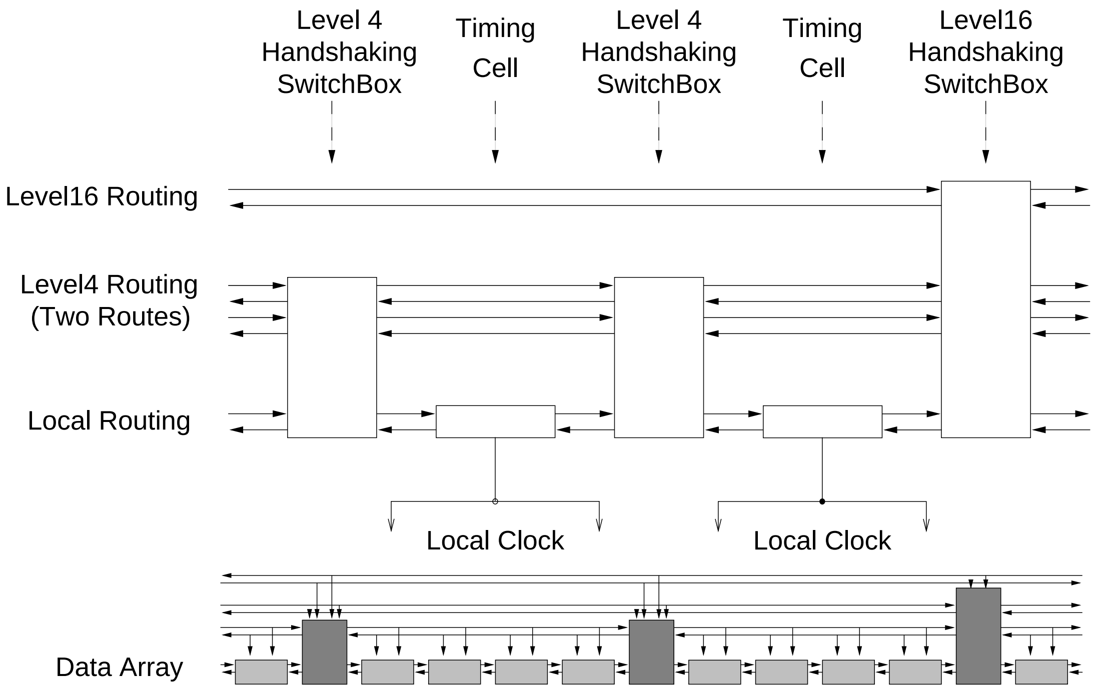


Figure 9.6: Cross-section of Timing Array Routing

An important decision in the design of the timing array routing is how many handshaking pairs to provide at each level in the routing. As discussed in Chapter 8, each handshaking pair can be used to synchronise the flow of one data bundle, and due to the symmetrical nature of the handshaking protocol, each handshaking pair can be used to synchronise data flowing in either direction. The more handshaking pairs provided in the timing array routing, the more data bundles can be synchronised in the data array, flowing in either direction. However, more timing array routing results in a larger overhead for implementing the self-timed architecture relative to the synchronous XC6200 architecture.

The choice made in Figure 9.6 represents a trade-off between flexibility in the timing array routing against the overhead of implementing the timing array. At the local level, the number of handshaking routes is naturally determined by the nearest neighbour arrangement of the timing cells. Thus, one handshaking pair route is provided from each side of a timing cell. At higher levels in the architecture the number of routes is not constrained by the nearest neighbour interconnect of the timing cells. More routing resources are required at the lower levels in the timing array routing, since the higher level timing array routes must connect to the timing cells, via the lower level timing array routing. This is reflected in Figure 9.6, where two handshaking pairs are provided at level-4 in the timing array routing against one handshaking pair for level-16. The choice of two handshaking routes at level-4 against one at level-16 was made on the basis of the needs of initial example circuits, set against the desire to minimise implementation costs.

No level-64 timing array routing is provided in Figure 9.6, since in the current XC6216 chip, the array consists of only one 64×64 block of cells. Thus, the level-64 flyover signals must be routed to cells within this one 64×64 block. These data bundles must be synchronised using lower levels in the timing array routing.

Although the timing array routing structure matches the structure in the data array routing, it does not follow that data using a certain level of flyover in the data array will use the same level of routing in the timing array. For example, consider a data bundle routed on a level-16 flyover that fans out to a number of 4×4 blocks within the 16×16 block it crosses. Handshaking pairs must be routed to the timing cells associated with the destination 4×4 blocks. Routing these handshaking pairs requires the use of local and level-4 timing array routing. The level-16 timing array routing is only used if the data bundle crosses into the neighbouring 16×16 block.

This example illustrates that more timing array routing resources are required at lower levels in the routing hierarchy, since the higher level timing array routing resources must connect to the timing cells via the lower level timing array routing resources.

The handshaking switchboxes only implement a subset of the routes in a full handshaking crossbar, which mirror the connections available in the boundary multiplexors. For example, the flyover routing in the data array does not allow signals routed on one flyover to be routed back in the opposite direction at the same level, so in the handshaking switchboxes, handshaking pairs cannot be connected to other handshaking pairs at the same level.

So far, the routing structures have been very one-dimensional, albeit duplicated in two dimensions. This reflects the structuring of the XC6200 fly-over routing, which is one-dimensional in nature. However, the XC6200 does provide `Magic` signals for routing corners. These flows can be implemented by handshaking pairs routed through a timing cell and associated 4×4 block. However, this prevents the timing cell from being used for anything else but routing. The problem could be solved by having corner turning handshaking pairs in the timing array routing. `Magic` routing was found to be rarely used in the designs, so the final self-timed XC6200 architecture used in the later chapter omits these corner routes.

9.4.4 Input/Output Interface

The XC6200 provides two I/O interfaces: IOBs at the edge of the array and register based I/O via the configuration interface. The IOB interface for the self-timed XC6200 can be simply extended by providing extra pins and IOBs for the handshaking signals. For simplicity, this is the approach adopted for the self-timed XC6200 architecture discussed in later chapters.

However, as discussed in Chapter 8, self-timing gives the opportunity to share routes transparently, which allows devices to be extended naturally to multi-device arrays without excessive numbers of pins. However, this approach is not compatible with the synchronous XC6200, and also makes it difficult to create dedicated input/output paths. A possible solution would be to implement both multiplexing and dedicated input/output schemes and have a flag in the configuration memory to choose which scheme was used.

The second I/O interface in the XC6200 is based on direct access to the registers. The same mechanism can be implemented in the self-timed XC6200. Since the cells in the self-timed XC6200 are not synchronised to a global clock, a register's state could be changing when sampled, leading to a meta-stable state. Hence, meta-stable resolving elements, such as Q-flops [103], have to be used in the output path of the register values.

As discussed previously, the synchronous XC6200 supports a partial form of flow control by routing the `RegWord` signals into the array. This signal acts as a request or acknowledge to the circuit in the array indicating that it has been read or written. However, it does not provide a mechanism for a circuit on the FPGA to indicate to the microprocessor that it has data ready, or is ready to accept data. Full flow control can be provided in the self-timed XC6200 by routing the `RegWord` signals on to the timing array routing instead of the data array routing. The `RegWord` signal acts as a request or acknowledge signal to the circuit on the FPGA. Handshaking pairs are routed in the timing array, so the `RegWord` signal will be paired with a handshaking signal, which is routed back from the circuit to the edge of the array. The signal paired to `RegWord` indicates to the microprocessor whether the circuit is ready to be read or written. Thus, routing `RegWord` using the timing array allows full flow control between FPGA and microprocessor to be implemented.

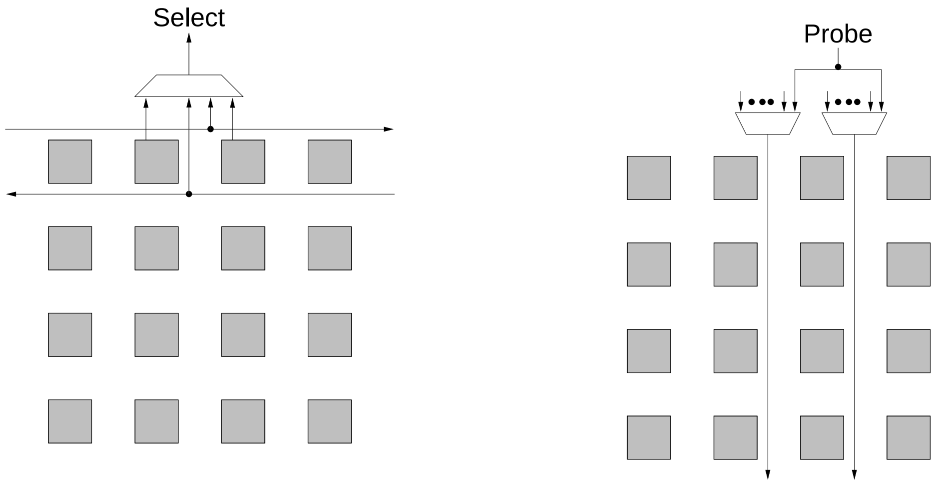
Simple request/acknowledge handshaking does not suit microprocessor interfaces, since if the circuit on the FPGA being accessed is not ready to be read or written, this locks the microprocessor bus waiting for the request or acknowledge signal to return. More sophisticated microprocessor interfaces can be used to overcome this. For example, if the circuit is not ready for the communication, then the condition could be returned to the microprocessor, instead of locking the bus. Alternatively, rather than wait for the microprocessor to access the circuit, the circuit could signal its readiness to the microprocessor using an interrupt signal. The microprocessor could determine which circuit on the FPGA had caused the interrupt by reading an additional control register containing the values of the handshaking signals paired to each `RegWord` input.

9.4.5 Configuration Interface

The timing array configuration memory is placed in a separate memory space for the self-timed XC6200. This allows the configuration bit layout of the synchronous XC6200 to be maintained. Also, separate memory spaces allow the wild card addressing to be used effectively to configure regular structures in the timing array or data array. To provide self-timed read and write to the SRAM, an additional output is required from the configuration interface to indicate that the read/write to the SRAM has been completed.

9.4.6 Select Routing

Figure 9.7(a) illustrates one of the multiplexors which choose the select input to the timing cell. The select inputs for the other three timing cell links are generated using the same routing pattern, but their orientation is rotated around the 4×4 block. An exhaustive study of which inputs to the select multiplexor to use was not performed, but a number of criteria were used in choosing the input. First, the input pattern should be symmetrical to allow designs to be flipped and rotated without problems. Second, the select multiplexor should be well connected to other parts of the self-timed region, using both flyovers and local routing, but the signals should naturally pass close to the select multiplexor. Finally, only four inputs to the select multiplexor were chosen as this only requires two configuration bits to be used.



(a) Select Routing (b) Probe Routing

Figure 9.7: Select and Probe Routing

9.4.7 Arbitration and Probe Routing

Arbitration blocks were discussed in Section 7.6. The arbitration block used in the self-timed XC6200 timing cell uses programmable AND gates, as this is less costly to implement than the Q-flop scheme. The pAND scheme only generates one probe signal per timing cell.

Figure 9.7(b) illustrates the probe routing pattern for one side of the 4×4 block; the pattern is repeated for the other sides of the 4×4 block. As with the select routing, an exhaustive study of the possible probe routing patterns was not performed. However, similar criteria were applied to its design; the pattern is symmetrical to allow designs to be flipped and rotated and allows the probe signal to be routed using a small number of multiplexors to all parts of the self-timed region. Finally, the number of configuration bits is minimised by using unused configuration values in the boundary multiplexors.

9.4.8 Delay Methodology

In the self-timed XC6200, since the memory elements of the 4×4 block are not situated on the boundaries of the self-timed region, two delay elements are required in the timing cell, one for the delay from the 4×4 block's inputs to the registers and one from the registers to the 4×4 block's outputs. Having two delay elements results in an increased implementation cost compared to a logic block where registers are either situated on inputs or outputs to the block.

In the simulations of run-time parameterised circuits discussed in the next chapter, two delay methodologies are used and compared for the self-timed XC6200: a fixed reconfigurable delay using taps off an inverter chain and a variable delay using Current Sensing Completion Detection (CSCD). The choice between fixed and variable delay elements has an important effect on the performance of the architecture. This is discussed in Chapter 11.

For the CSCD implementation, two delay elements are not required, since the completion detection of both stages can be provided using the same current monitoring circuitry. However, a problem occurs if the next set of input data enters the 4×4 block before the output data has left the block. The CSCD circuitry cannot tell the difference between output signals leaving the block or input signals entering the block. Thus, the output CSCD completion is extended until the new input data has finished evaluating. To overcome this problem, a fixed reconfigurable delay could be used for the output delay element.

9.4.9 Timing Cell Implementation

The various options for implementing the timing cell were discussed at length in Chapter 7. For the reasons discussed there, the implementation of the timing cell used in the self-timed XC6200 uses a four-phase protocol. The decoupling in the four-phase protocol is performed as a group for the handshaking links to minimise implementation cost (see Section 7.4.2).

9.5 Summary

This chapter has developed a self-timed version of the Xilinx XC6200 architecture based on the STACC architectural model. The chapter has also discussed in general terms, the decisions required in creating a self-timed FPGA architecture using STACC. Amongst these decisions, the most important is the choice of granularity of the self-timed region, which effects all the other design decisions made about the architecture. The consequences of these design decisions are the subject of the next two chapters, which describe the use of the self-timed XC6200 for dynamic hardware applications (Chapter 10), and evaluate it with respect to the synchronous XC6200 (Chapter 11).

Chapter 10

Circuit Design for the Self-Timed XC6200

10.1 Introduction

This chapter describes the design of run-time parameterised circuits for finite field operations on the self-timed XC6200. The examples are used to highlight the way that designs exploit the features of the self-timed XC6200 and overcome its limitations. Several of the design techniques used are specific to the XC6200 data array, so are equally applicable to the self-timed and synchronous versions of the XC6200. As well as illustrating the design of circuits for the self-timed XC6200, the examples provide insight into the design of run-time parameterised circuits in general.

The chapter is organised as follows. The design and simulation tools used for the example circuits are described in Section 10.2. The majority of the chapter is given over to Section 10.3 which contains detailed descriptions of the example circuits for finite field operations. Finite field arithmetic and its application are described in Appendix A. The design experience gained from implementing these circuits is summarised in Section 10.4.

10.2 Design Tools and Simulation

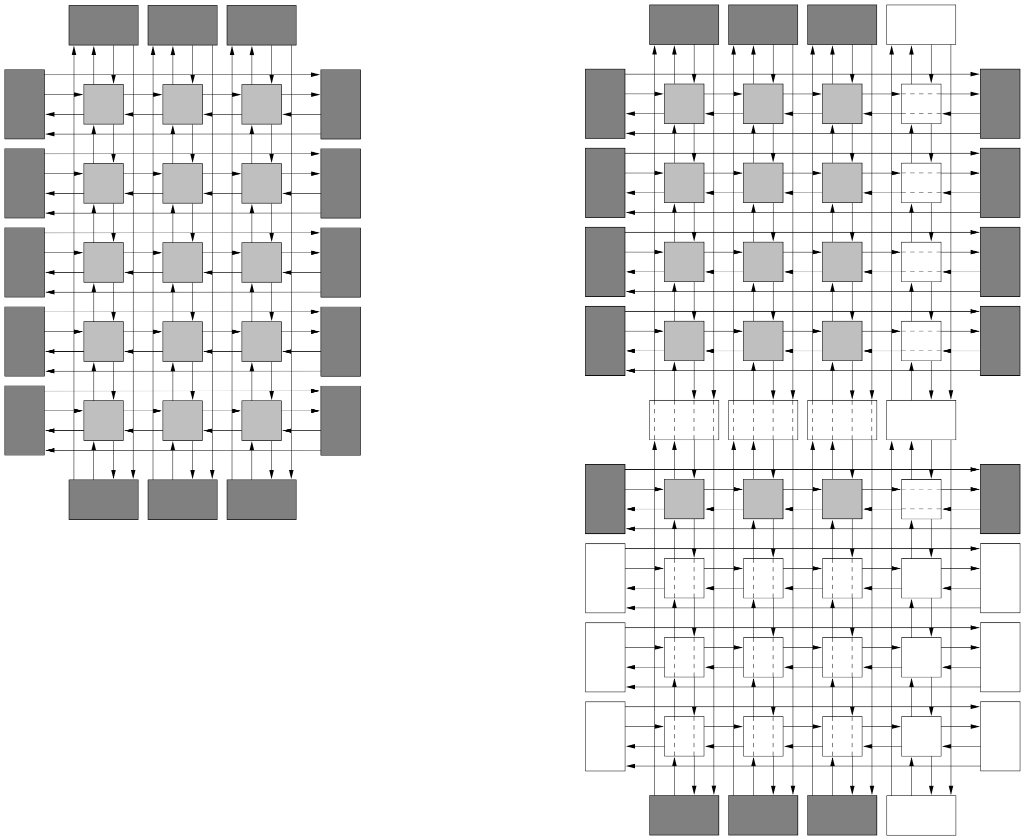
Current design tools support parameterisation, but these parameters are fixed at compile time. Once the parameters are fixed at compile-time, the circuit is passed through the standard place and route tools. Such tools do not support the run-time parameterised circuits, which were discussed in Section 2.7.6. The output of a tool for run-time parameterised circuits would not be an FPGA configuration, but would be a program that could generate the configuration at run time. To allow the circuits to be generated quickly at run time, traditional place and route tools cannot be used, as they take too long. Instead, run-time parameterised circuits require regular placement and routing to allow fast assembly of the FPGA configuration.

Given the requirements for run-time parameterisation, the circuits in this study did not use the standard Xilinx place and route tools. Instead, the configurations were generated by programs written in VHDL [62, 91]. Since the simulation models of the self-timed and synchronous XC6200 were also written in VHDL, this simplified the interfacing from design to simulator. For visual debugging of circuits, a VHDL library was designed to output configurations in the Xilinx XC6200 `.cfg` format. This allowed circuits to be viewed and the diagrams in the text to be generated. Fixed parts of the circuit were also designed manually without using the Xilinx place and route tools, since at the start of the study, these tools were still in the initial stages of development.

The drawback to designing circuits without place and route tools is the amount of effort involved; comparable in software terms to writing purely in assembly language. At this point in time, now that the Xilinx tools have matured, a promising approach would be to design the fixed parts of the run-time parameterised circuits using the synthesis tools. These parts could then be assembled on the fly, together with parameterised parts generated by software, to produce the final run-time parameterised circuit.

An issue in constructing run-time parameterised circuits for STACC architectures is the basic block used to build them. To gain the benefits of the self-timed implementation, the circuits must be defined in terms of the basic self-timed building blocks used in the architecture. For the self-timed XC6200, this is the 4×4 cell block. However, the definition of parameterised circuits is often easier at the level of individual cells in the XC6200 architecture, rather than at the level of 4×4 blocks. To allow the design of parameterised circuits at the cell level, the concept of an abstract $N \times M$ block of cells is introduced. The abstract cell blocks are then mapped on to the 4×4 blocks of the architecture.

Figure 10.1(a) illustrates an abstract 3×5 block. The abstract block has a similar structure to the 4×4 block. It consists of an array of cells surrounded by boundary multiplexors; the boundary multiplexors determine the routing for the flyover signals that cross the block. Figure 10.1(b) illustrates the conversion of the abstract block to the 4×4 blocks used in the architecture. The conversion works by mapping cells in the abstract block directly on to an array of 4×4 blocks large enough to accommodate the abstract block. Unused cells and



(a) Abstract 3×5 Block

(b) 4×4 Blocks

Figure 10.1: Converting from an abstract $N \times M$ block to 4×4 blocks

boundary multiplexors (unshaded in the figure) are configured so that signals pass through them unaffected and continue in the same direction.

The fixed granularity of the self-timed XC6200 architecture at the level of 4×4 blocks also presents other problems. Each 4×4 block forms a self-timed region, so must capture in registers the values of its outputs. This requires some cells that are not configured as registers in the abstract block to be configured as registers in the 4×4 blocks. The pattern of data flow between 4×4 blocks must also be reflected in the configuration of the timing array. Furthermore, in variants of the self-timed architecture that use a configurable fixed delay (i.e. not CSCD), the delay of each 4×4 block must be established. This may require delay analysis, though often some simple rule can be constructed for the delay of each 4×4 block from the structure of the parameterised circuit.

The problems with defining parameterised circuits at the level of individual cells, when the self-timed blocks of the architecture are defined in terms of 4×4 blocks, are discussed later in the chapter using specific design examples.

10.3 Example Circuits

This section details the implementation of circuits for finite field operations on the self-timed XC6200. The initial examples illustrate circuits with simple patterns of data flow and limited parameterisation. The later examples illustrate more complex patterns of data flow and a wider degree of parameterisation. The application area of finite field operations is described in Section 10.3.1. Section 10.3.2 describes the standard format used to describe the examples. Sections 10.3.3 to 10.3.9 describe the example circuits.

10.3.1 Finite Field Operations

The example circuits implement run-time parameterised circuits for finite field operations. Finite field operations have application to error detection and correction, such as Reed-Solomon codes [97] and erasure codes [102]. Finite field operations were chosen for the study, since they have a number of parameters which can be varied. Varying these parameters alters the error detection and correction power of codes based on them. Klindworth [68] has proposed using such circuits to build a communication system where the power of the error correction code is dynamically altered to match the current noise conditions on the transmission medium.

For self-timed implementation, finite field operations do not have any particular features that suit self-timed operation. However, the error correction applications based on them are well matched to a self-timed implementation, because of the large difference in work load between checking codewords and correcting them. Normally an error processor is only checking that the received message is a valid codeword, but an invalid codeword requires a complex error correction process. This difference in workload has been exploited by Kessels et al [67] in a self-timed Reed-Solomon error corrector to achieve superior power efficiency over a synchronous implementation.

The basics of finite field theory and the error correction applications based on them are reviewed in Appendix A. For more details on finite fields refer to Pretzel [97]. Details of how the data array implements the finite field operations is not required for the example circuits; only an appreciation of how the dataflow in the data array is reflected in the configuration of the timing array. More detailed description of the data array configuration is given for circuits with interesting parameterisation.

10.3.2 Example Format

The examples are described using a standard format, so that the reader may easily find relevant information on earlier examples that are used as components in later examples. In this section the format is introduced using a FIFO as a simple example. Each of the sections of the standard format are described below.

Description: Each example begins with a high level description of the circuit and its parameterisation. The FIFO is a simple circuit with only one parameter which specifies the number of stages in the FIFO.

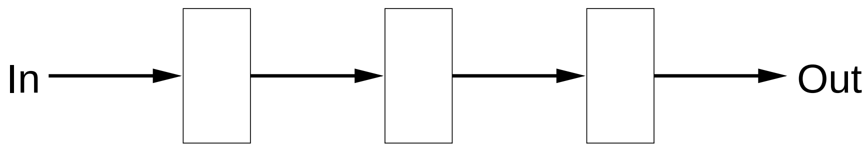


Figure 10.2: FIFO: Data Flow

Data Flow: An overview of the circuit operation is given using a diagram to illustrate the data flow in the circuit. Figure 10.2 illustrates the data flow for a three-stage FIFO. The symbols used in the data flow diagrams are shown in

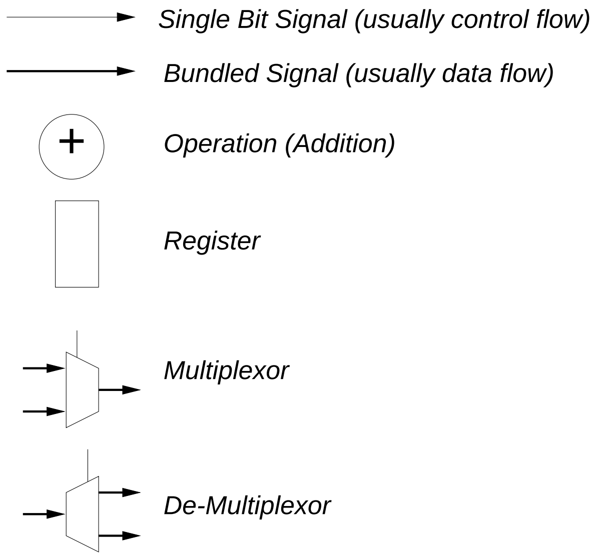
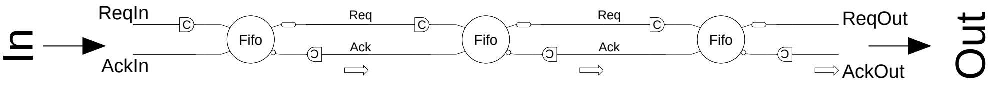


Figure 10.3: Data Flow Diagram Symbols

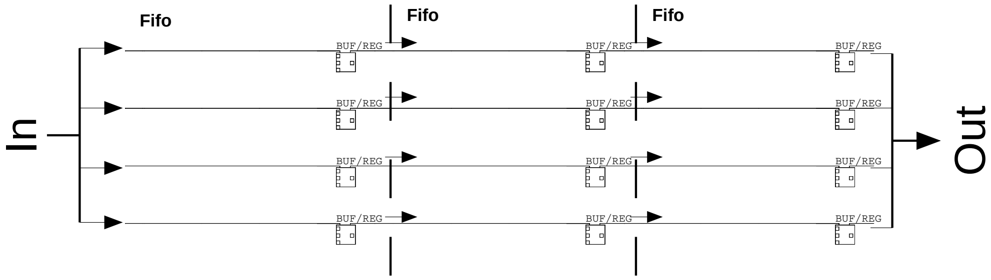
Figure 10.3. No processing is performed in the FIFO, so the only components in the data flow diagram of Figure 10.2 are registers and the flows connecting them. The placement of components in the data flow diagrams have been arranged so that they correspond with the layout used in the timing and data array configurations.

Timing Array Configuration: Figure 10.4(a) illustrates the configuration of the timing array for the FIFO. The timing array configuration is shown together with the data array configuration to illustrate how the pattern of timing array routing matches the pattern of routing in the data array. The timing array configuration is illustrated using the output of a tool developed specifically to allow the visualisation of the timing array simulations. The full tool allows simulation values to be animated on the diagram.

Figure 10.4(a) illustrates the configuration of the timing array for the FIFO. The representation used is the same as for the simple timing array examples given in Section 5.4. Circular elements marked with the name of the self-timed region represent the C-Muller gate controlling the registers in the self-timed region. These elements are connected by handshaking links. Request signals are marked by oval delay elements. Acknowledge signals are marked by bubbled inputs into the timing cell. Additional to the output of the visualisation tool, other symbols have been superimposed on the output of the visualisation tool to highlight the structure of the configuration. These additional symbols are shown in Figure 10.5.



(a) Timing Array



(b) Data Array

Figure 10.4: Three Stage FIFO

In Figure 10.4(a), each `Fifo` timing cell uses two handshaking links, the left link for the input data, and the right link for the output data. The C-Muller gates on the handshaking links represent the routing implemented by the handshaking switchboxes. In the current example of a FIFO, there is no fan-in or fan-out of data, so each C-Muller gate has only one input. In general, several handshaking signals may be inputs to these C-Muller gates, to synchronise the fan-in and fan-out of data bundles.

Data Array Configuration: Following the description of the timing array, the data array configuration is described. The Xilinx Development Tools [123] are used to produce these figures. Each cell is shown by a box, which is marked with the function it performs (e.g. `BUF` for buffer, `MUX` for multiplexor, etc.). Cells that use their D-type register are additionally marked with the letters `REG`. Local wiring is marked by solid lines joining cells. Level-4 flyovers and level-16 flyovers are marked by dotted connecting wires. Figure 10.4(b) illustrates the data array configuration for the `FIFO`. Input inversion is not displayed by the Xilinx tools, so the `BUF` cells in the FIFO could also be implementing an inverting buffer.

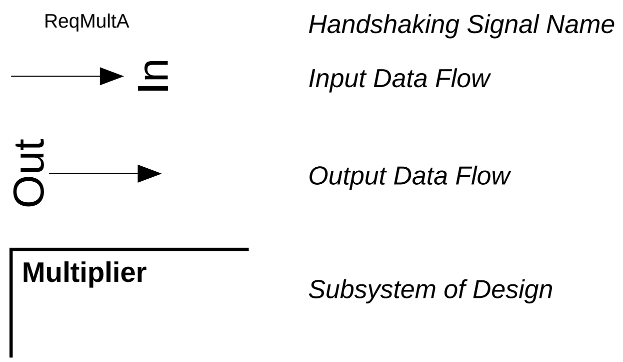


Figure 10.5: Timing Array Symbols

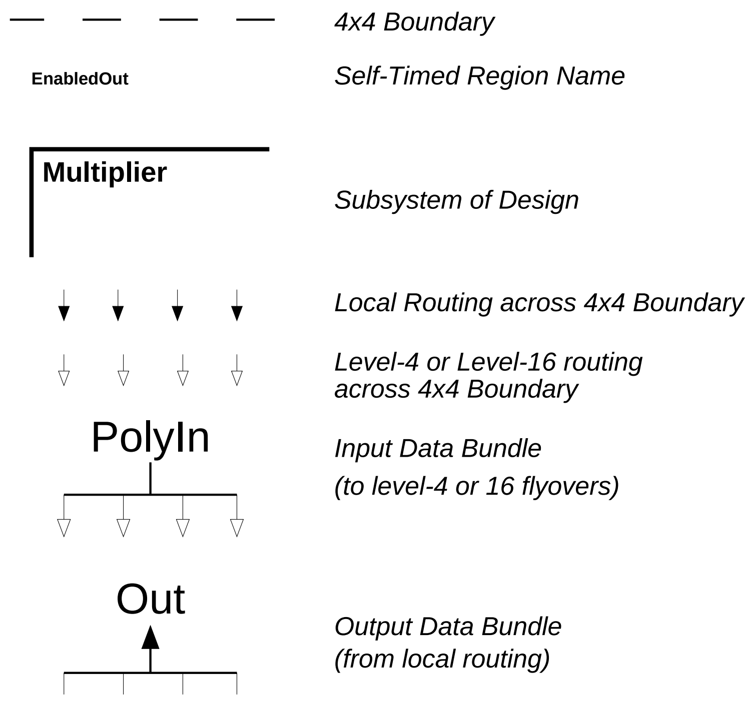


Figure 10.6: Data Array Symbols

The data array configuration is shown to illustrate how the timing array configuration matches the pattern of data flow in the data array. The output of the Xilinx tools does not clearly mark the direction of signal flow, so supplementary symbols have been superimposed to show the pattern of signal flow between 4×4 blocks. These symbols are shown in Figure 10.6.

Discussion: Finally, each example is completed with a discussion section which highlights the interesting aspects of the circuit's design and implementation. Analysis of the delay behaviour of the circuits is left to Chapter 11 which evaluates the performance of the self-timed XC6200 relative to the synchronous XC6200.

10.3.3 $GF(2^4)$ Multiplier

Description: The circuit multiplies two numbers over $GF(2^4)$. The example illustrates the use of the timing array routing to synchronise the fan-in and fan-out of data bundles in the data array.

Data Flow: As described in Appendix A, operations over $GF(2^k)$ can be defined in terms of operations over polynomials with terms in $GF(2)$ modulo an irreducible polynomial. Each bit of the result polynomial can be evaluated separately, so a bit-sliced implementation can be used. Figure 10.7 illustrates the data flow for a bit-sliced multiplier. Each bit slice takes both input values `Ain` and `BIn` and generates one bit of the result, which fans in to form the result data bundle `Out`.

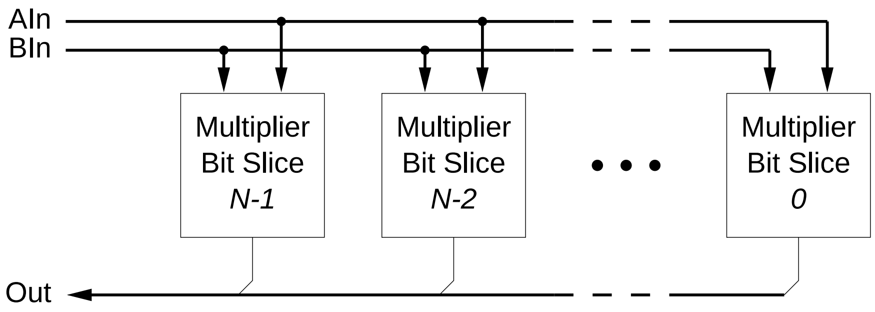
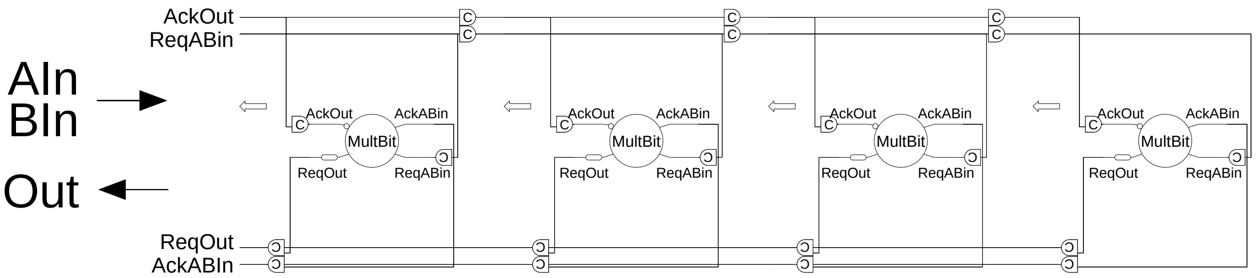
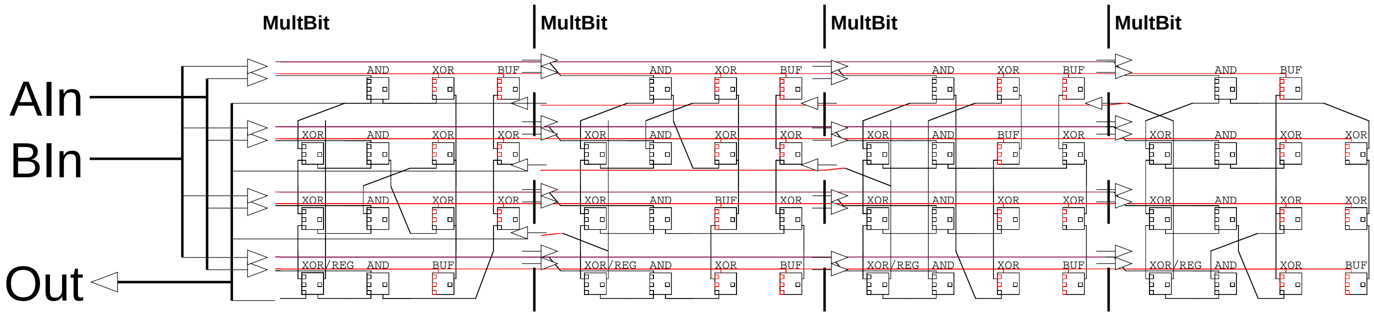


Figure 10.7: Data Flow for Multiplier over $GF(2^k)$

Timing Array Configuration: Figure 10.8(a) illustrates how the timing array routing is used to synchronise the fan-in and fan-out of data bundles in the data array. The input data `Ain` and `Bin` fans out to each bit slice. `Ain` and



(a) Timing Array



(b) Data Array

Figure 10.8: Multiplier over $GF(2^4)$

Bin are bundled with the ReqABin and AckABin handshaking pair which are routed on the level-4 handshaking routing. The handshaking pair is routed to the right-hand link of each MultBit timing cell. The ReqABin signals fans out from the level-4 handshaking routing to each timing cell. The AckABin follows the reverse path, with the acknowledge signals being combined using a chain of C-Muller gates so that the multiplier only acknowledges receipt of the data when all the stages have completed.

One bit of the output data Out is produced by each bit slice. The data is associated with the ReqOut and AckOut handshaking pair which comes from the left-hand link of each MultBit timing cell. The local ReqOut signals fan-in onto the level-4 handshaking routing, being synchronised by a chain of C-Muller gates, so that the output ReqOut is only produced when bits have been produced by all the bit slices. The AckOut signal fans out to all the bit slices to indicate that the Out been received.

Data Array Configuration: The correspondence of the data array routing and the timing array routing can be seen by the pattern of the dataflow in Figure 10.8(b). The input data Ain and Bin are routed to each bit slice using the E4 and E16 flyovers respectively. The data generated by each bit slice Out fans in on the W4 flyovers. It can be seen from the data flow across the 4×4 blocks that the data bundle starts as a single bit on the rightmost 4×4 block. Each bit-slice adds its output to the bus, until the full 4-bit wide output is reached on the left.

For those interested in the internal operation of each self-timed region, a description is provided below. It can be skipped by those only interested in how the timing array routing matches the pattern of data array routing.

Multiplication can be expressed using the following equation derived in Section A.2.2:

$$A.B = \sum_{h=0}^{k-1} x^h . \sum_{i=0}^{k-1} a_i . \sum_{j=0}^{k-1} b_j . \left. F_{hi} \right|_j \tag{10.1}$$

Each coefficient of x^h in the final sum can be evaluated separately, so a bit-sliced circuit implementation can be adopted. The evaluation within each bit-slice involves a bitwise AND of the terms b_j with the jth coefficients of F_{hi}. The result of the bitwise AND are them summed (using XOR gates) to produce intermediate partial sums which are then ANDed with the terms a_i. Finally, these are then summed to produce the output coefficient.

Figure 10.9 shows the internal dataflow for the calculation for each bit of the result. The two operators shown are the dot operator (bitwise AND) of

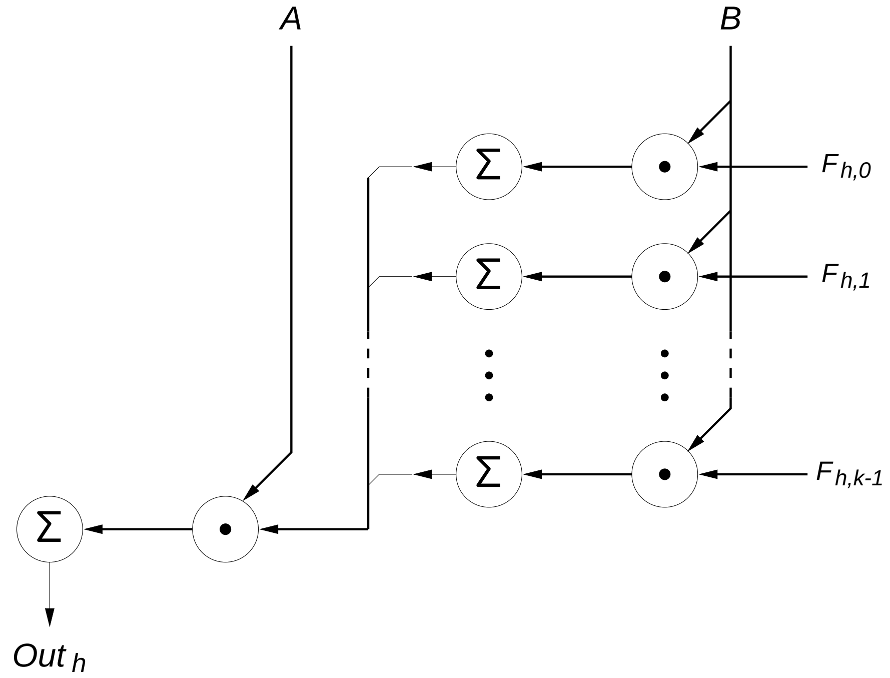


Figure 10.9: Internal Structure of Multiplier Bit Slice

the coefficients of two polynomials and the summing operator which sums the coefficients of a polynomial to produce a single bit output. Single $GF(2)$ coefficients are represented by thin lines, whilst polynomials over $GF(2)$ are shown by thick lines. In the actual circuit, the bitwise AND of B and F_{hi} is not required since F_{hi} is a constant. Hence, the first summing operator is a partial sum of terms b_j for which $F_{hi} = 1$

The structure of this evaluation can be seen in the structure of each bit slice in Figure 10.8(b). The right two columns of each `MultBit` 4×4 block are sets of XOR gates that generate the partial sums of the b_j coefficients. These partial sums are ANDed with the a_i terms in the second column from the left and summed using XOR gates in the leftmost column.

The multiplier uses the normal basis for the representation of the polynomials. A important property of the representation is that the same function can be used to produce each bit of the result, by just cyclically shifting the inputs (see Section A.2.1). However, cyclically shifting the inputs in the flyover routing is difficult. Instead, the routing in the circuitry for each bit slice has been modified to do the shifting. An alternative approach to this problem would be to 'cyclically shift the circuit'. Such an approach is possible in the XC6200, since by using the feedback flyover connections, a toroidal routing structure can be created within a 4×4 block. This would allows the interconnect pattern of the cells to be maintained whilst cyclically shifting the location of the cells.

Discussion: This example has illustrated the use of the timing array routing to synchronise the fan-in and fan-out of data bundles. It has shown how the timing array routing structures can be routed in a similar fashion to the routing for the data bundles in the data array.

10.3.4 Constant $GF(2^k)$ Multiplier

Description: The circuit multiplies a number in $GF(2^k)$ by a constant over $GF(2^k)$. The example illustrates the technique of mapping a circuit defined for a $N \times M$ block of cells onto the 4×4 blocks of the XC6200. The example also illustrates the benefits of run-time parameterisation over the general $GF(2^k)$ multiplier discussed in Section 10.3.3.

Data Flow: The constant multiplier is much smaller than the general multiplier described in Section 10.3.3. Figure 10.10 shows the simple data flow of the circuit; the input enters the circuit and is scaled by a constant C. The internal data flow and parameterisation of the circuit is described when the data array configuration is discussed.

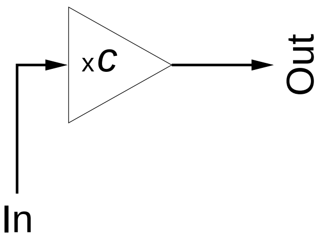
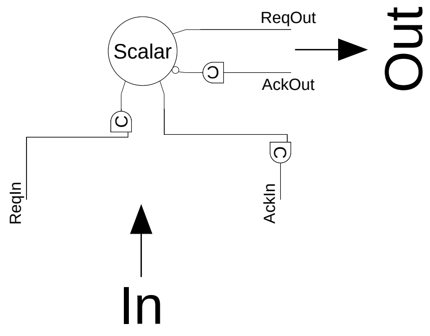


Figure 10.10: Constant Multiplier Data Flow

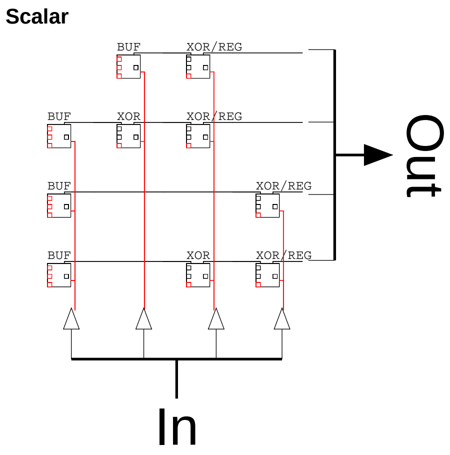
Timing Array Configuration: For small fields $GF(2^k)$ with $k \leq 4$, only one timing cell is required. Figure 10.11(a) shows the configuration of one such timing cell. The timing array configuration is a simple pipeline stage, consisting of one input link from the bottom and one output link to the right.

However, when the parameterised circuit is too large for a 4×4 block, the circuit has to be divided into separate 4×4 blocks, which are different self-timed regions. The timing array needs to be configured to reflect the flow of data between the individual self-timed regions of the multiplier This is illustrated in Figure 10.12(a) for $GF(2^5)$. This shows how the input data is treated as being composed of two data bundles. The bundles `In1` (4 bits) and `In2` (1 bit) are distributed to the 4×4 blocks using the level-4 handshaking signals.

The partial results from each block then flow from left to right in the data array, to become two separate output data bundles `Out1` and `Out2`.



(a) Timing Array

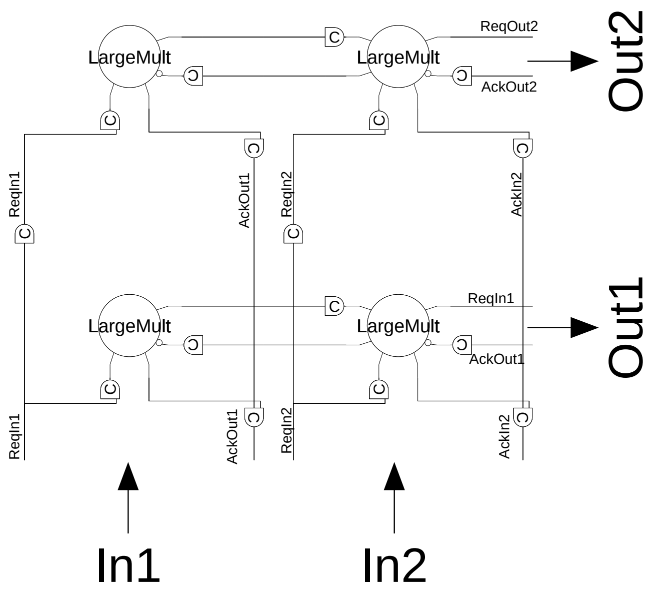


(b) Data Array

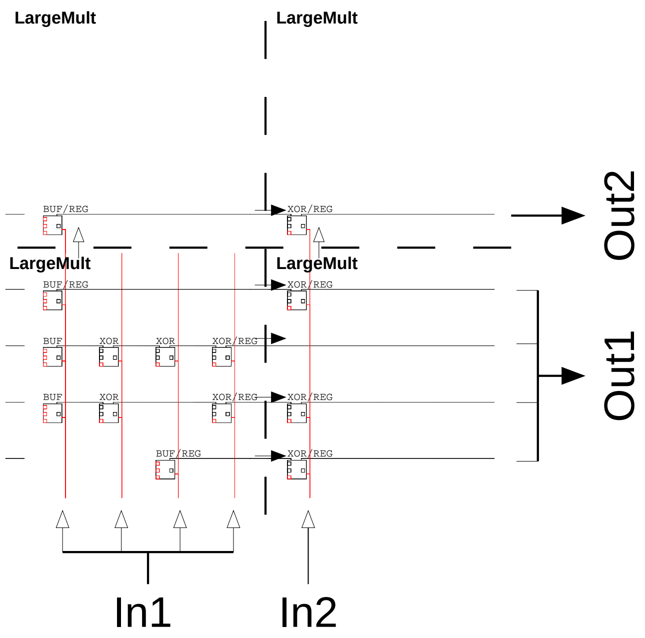
Figure 10.11: Constant Multiplier over $GF(2^4)$

Data Array Configuration: Figure 10.11(b) illustrates the data array configuration of a constant multiplier over $GF(2^4)$. The inputs to each circuit enter on the `N4` flyovers at the bottom of the block and the outputs exit on the local `E` signals on the right of the block. The internal data flow consists of partial sums being calculated by chains of XOR gates, with results flowing from left to right. The details of the finite field arithmetic are described below.

The circuit can be extended to deal with base fields $GF(2^k)$ with $k > 4$ by using the concept of an abstract $N \times M$ block that was introduced in Section



(a) Timing Array



(b) Data Array

Figure 10.12: Constant Multiplier over $GF(2^5)$

10.1. Figure 10.12(b) illustrates a constant multiplier for $GF(2^5)$. The choice of $GF(2^5)$ was made to show an example of a poor mapping to 4×4 blocks. In this case, most of the cells, in all but the bottom left 4×4 block, are unused.

As discussed in Section 10.1, the fixed granularity of the self-timed XC6200, forces designs to be pipelined internally for each 4×4 block. The extra registers that are required can be seen on the figure by examining the cells that generate the outputs of the left-hand 4×4 blocks, which have `REG` elements added.

For those interested in the internal finite field arithmetic of the multiplier the details are given here. Equation A.12 can be re-expressed as:

$$A.B = \sum_{h=0}^{k-1} x^h . \sum_{i=0}^{k-1} a_i . \left. C_h \right|_i \tag{10.2}$$

Where

$$\left. C_h \right|_i = \sum_{j=0}^{k-1} b_j . \left. F_{hi} \right|_j \tag{10.3}$$

C_h is constant as B and F_{hi} are constant. Thus each coefficient x^h can be calculated by a bitwise AND of C_h with A, and then summing the resultant bits. Figure 10.13 illustrates the internal data flow. It consists of several bitwise AND operators followed by summing operators, which evaluate each output bit Out_h. Since C_h is constant, the bitwise AND and the summing operators can be reduced to a partial sum of terms a_i for which $C_h = 1$. This is the structure used in the data array configuration of Figures 10.11(b) and 10.12(b).

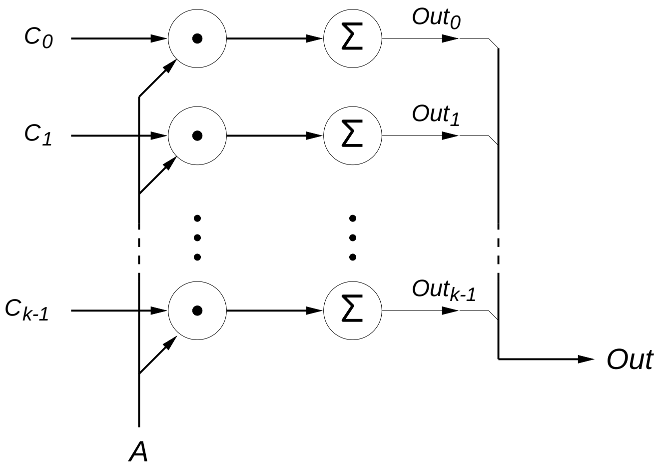


Figure 10.13: Constant Multiplier Internal Data Flow

Discussion: This example has illustrated the advantages of parameterised circuits. The circuits for multiplying by a constant are far more compact than a general multiplier circuit. A general multiplier requires k bit slices for a multiplier over $GF(2^k)$, where each bit slice is approximately the same size as one constant multiplier. This gives a saving of approximately k times by using a parameterised circuit.

The example has highlighted the problems with mapping a circuit which is defined in terms of an array of cells onto the fixed granularity of 4×4 blocks of cells. In the data array, the technique of mapping from an abstract $N \times M$ block was used. The $GF(2^5)$ multiplier illustrated the potential waste of cells when circuits are a poor match to the imposed granularity. Also, the timing array configuration has to reflect the internal pattern of data flow. This could be harder to implement if the pattern of data flow internally was more complex. In many cases it would be preferable if the timing cells could be grouped to form one large self-timed region.

10.3.5 Division by a Fixed Polynomial

Description: The circuit divides an input polynomial with terms in $GF(2^k)$ by a fixed polynomial with terms in $GF(2^k)$. Division by a fixed polynomial can be used for the generation of Reed-Solomon error correction codes (See Section A.3.1). The example illustrates the construction of a run-time parameterised circuits with a large number of parameters. The circuit is parameterised with respect to the length of divisor and dividend polynomials, and the terms A_i of fixed divisor polynomial $A_n x^n + A_{n-1} x^{n-1} + ... A_1 x + A_0$.

Data Flow: Figure 10.14 illustrates the data flow for the fixed polynomial division circuit. It is similar to the linear feedback shift registers used for division by a fixed polynomial over $GF(2)$ in the generation of CRCs (Cyclic Redundancy Codes). The terms of the dividend are shifted down the shift register on the right. On each shift, a multiple of the divisor is subtracted from the value in the shift register. Adders are shown in Figure 10.14, since addition and subtraction are identical over $GF(2^k)$. In CRC circuits, generating the multiples is simple, since the only multiples possible in $GF(2)$ are zero and one. For $GF(2^k)$, constant multiplier circuits must be used for each term in the divisor polynomial. The constant multiplier circuits developed in Section 10.3.4 can be used for this purpose. When the whole dividend has been shifted in, the `End` flag is set, which zeros the feedback path and allows the remainder to be shifted out.

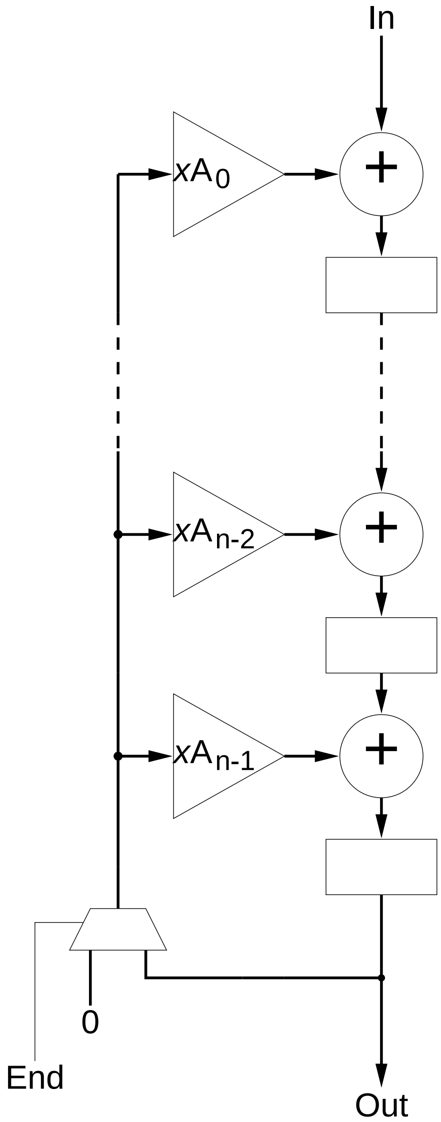
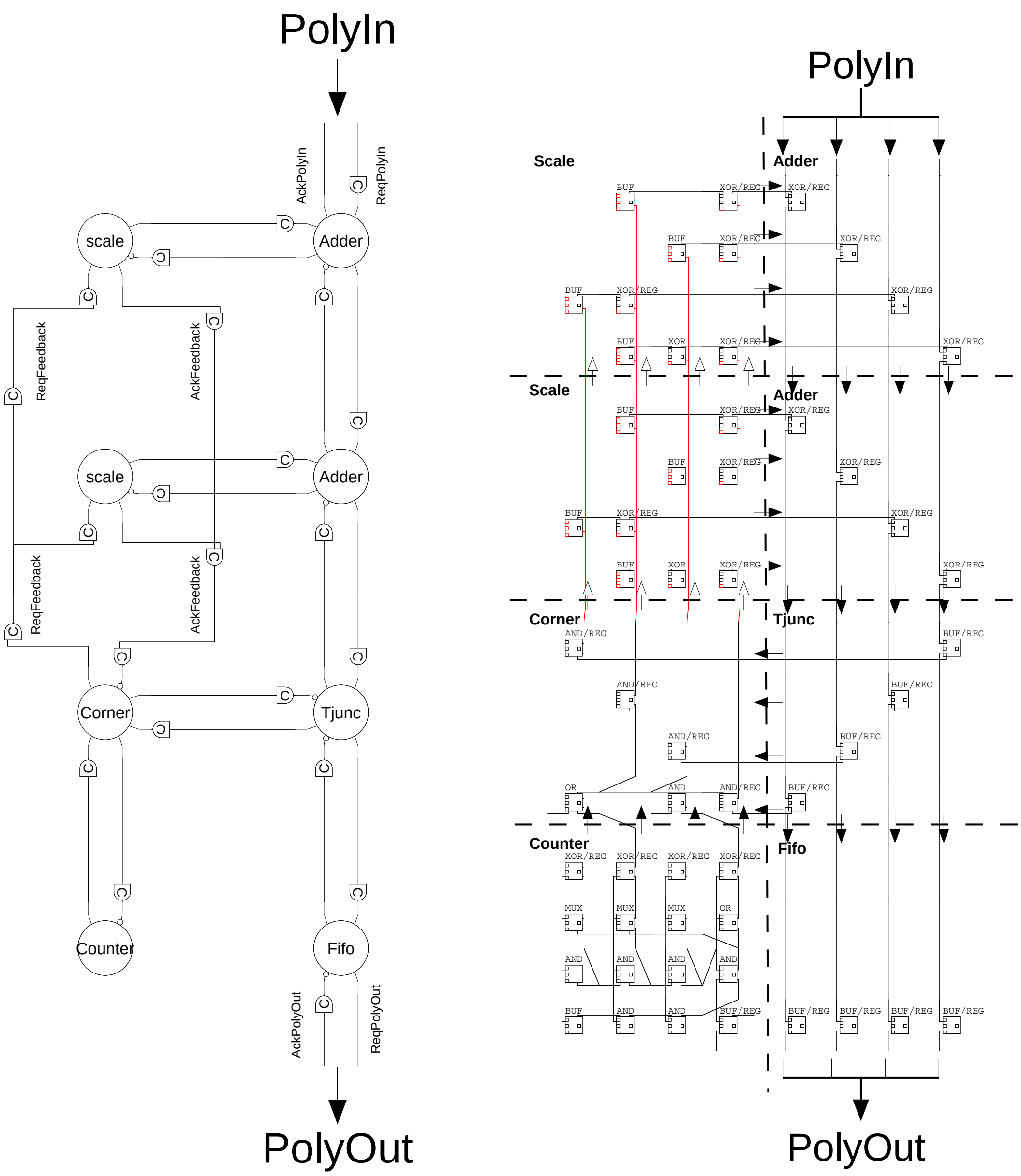


Figure 10.14: Data Flow for Fixed Polynomial Division



(a) Timing Array

(b) Data Array

Figure 10.15: Fixed Division by a Polynomial of Length Two

Timing Array Configuration: Figure 10.15(a) illustrates the timing array configuration for a small divisor polynomial of length two with terms in $GF(2^4)$. The layout is similar to the layout of the data flow shown in Figure 10.14. The `Adder` blocks integrate the addition and shifting stages shown in the data flow. The `Scale` blocks perform the constant multiplication, using the circuits discussed in the Section 10.3.4. The `Tjunc` block routes the output and feedback values. The `Corner` block zeros the feedback when the remainder is shifted out. The `Counter` block generates the `End` condition, which determines when to zero the feedback. The `Fifo` block routes the output value to the edge of the circuit.

The feedback value in the data flow diagram fans out to all the `Scale` blocks. This is implemented in the timing array using the level-4 handshaking routing; the `ReqFeedback` signal fans out from the `Corner` block to all the `Scale` blocks. The `AckFeedback` fans in the acknowledges from all the `Scale` blocks back to the `Corner` block. This fan-in synchronisation is done using a chain of two-input C-Muller gates; one input of each C-Muller gate comes from the local `Scale` block, and the other enters on the level-4 handshaking routing from above.

The fixed polynomial division circuit is the first example of a looping structure in a timing array configuration, rather than a straight pipeline. As discussed in Section 7.3.4, loops in the timing array need to be initialised. In this case, the loop is initialised by resetting the `Adder` blocks to an active state, so on initialisation, the shift register is effectively already filled with zero values.

Data Array Configuration: The data array configuration reflects the break down into self-timed regions discussed for the timing array. The `Counter` block is parameterised to count up to a value N before returning to zero. This allows the length of the input polynomial to be set. The length of the fixed divisor polynomial sets the overall size of the circuit. Finally, the individual terms A_i of the fixed divisor polynomial determines the constant values used in the `Scale` blocks.

Discussion: The example has illustrated run-time parameterisation of a circuit with a complex degree of parameterisation. The main parameter is the choice of the fixed polynomial divisor. The coefficients of the fixed polynomial are altered by changing the constant scalar multiplier (`Scale`) blocks. As these are themselves parameterised circuits, the circuit illustrates a hierarchy of parameterised circuits. The length of the fixed polynomial divisor affects the number of stages in the shift register. In the case of Figure 10.15, the polynomial has a length of two. Longer generator polynomials are used in Reed-Solomon codes to increase the error detection and correction ability of the code.

The change in size of the circuit for different fixed polynomial divisors is a good example of the advantages of self-timed parameterised circuits. Due to the self-timed behaviour, no delay calculations are required by the configuration software to account for the different lengths of the feedback path. The feedback path could include off-chip routing and the circuit would still operate correctly. This would not be the case for a synchronous version of this parameterised circuit, since the delay along the feedback loop would have to be recalculated.

10.3.6 Polynomial Evaluation at a Fixed Value

Description: The circuit evaluates an input polynomial $A(x)$ with terms in $GF(2^k)$ for $x = C$. The circuit can be used to evaluate the syndromes for Reed-Solomon error correction (see Section A.3.1). The example illustrates the use of selective links, and the problems encountered when a self-timed region requires more links than are available in the timing cell.

Data Flow: Polynomials can be evaluated as a series of additions and multiplications using the *Horner form*:

$$A(x) = a_0 + x(a_1 + x(a_2 + x(\ldots(a_n + 0)\ldots))) \tag{10.4}$$

Figure 10.16 shows the data flow for evaluating an input polynomial using the Horner form. At the start of the evaluation the `Start` flag is set, so that the first coefficient of `PolyIn` is stored in the accumulator register. The Horner form evaluation continues by multiplying the value stored in the accumulator register by C and then adding the next term from the polynomial, and storing the result back in the accumulator. The end of the evaluation is signified by the `End` flag which signals that the value of `Out` from the accumulator is the result of the evaluation.

Timing Array Configuration: Figure 10.17(a) shows the configuration of the timing array. The evaluation loop is composed from the `Multiplier` and `Feedback` blocks. In the example, two copies of the evaluation circuit are

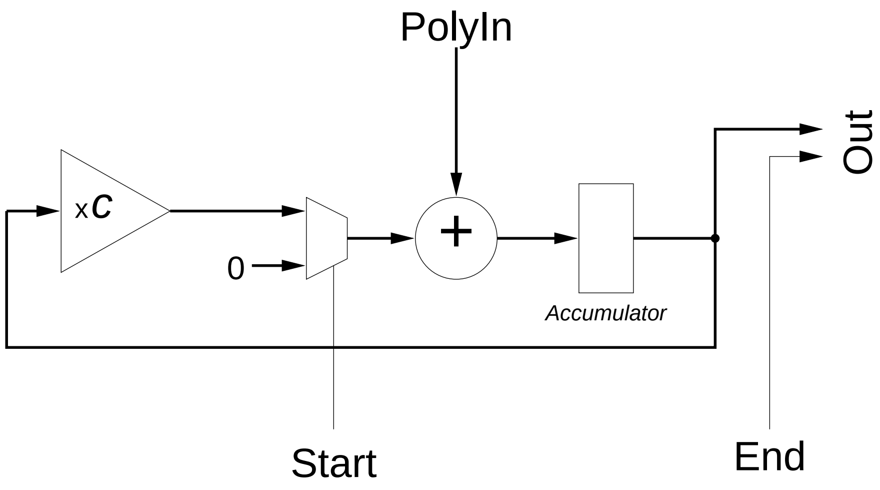


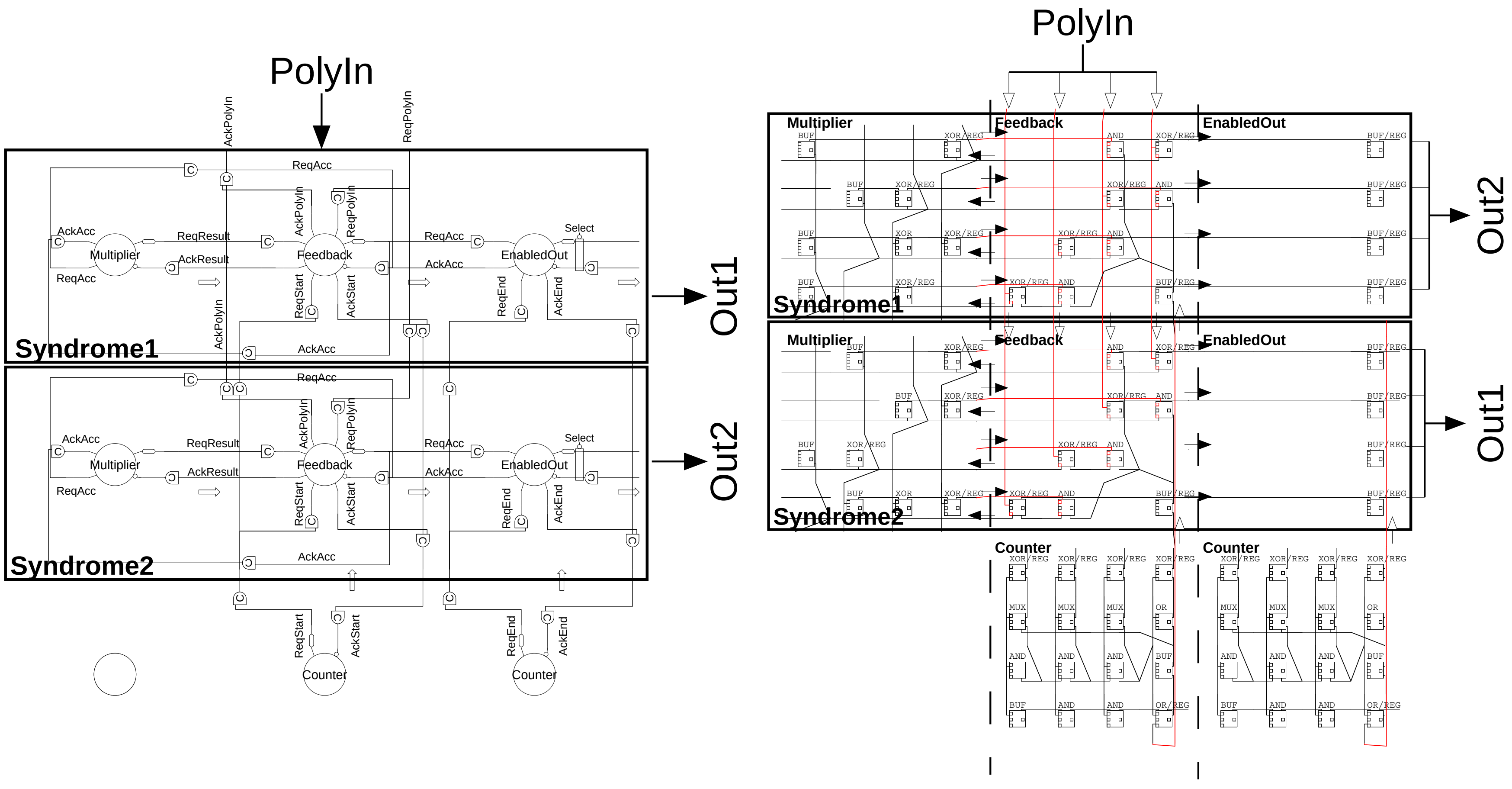
Figure 10.16: Data Flow for Polynomial Evaluation at a Fixed Value

shown for evaluating a polynomial at two different values. This is useful as it allows the syndromes of a Reed-Solomon code to be evaluated in parallel.

The `Multiplier` block performs the multiplication by the constant C. It is similar to the constant multiplier blocks introduced in Section 10.3.4, except that the input enters on the West link rather than the South link. This illustrates a problem with the layout of blocks used within several different circuits: the layout of a block for one circuit may not be suited for its use in another.

The `Feedback` blocks integrate the adder, accumulator and multiplexor from the data flow shown in Figure 10.16. The `Feedback` block is the most complex example of timing cell use so far. The block takes three inputs, the input polynomial coefficients `PolyIn` (using `ReqPolyIn` and `AckPolyIn` on the North link), the `Start` signals from the `Counter` block on the South link (using `ReqStartIn` and `AckStartIn`), and the input from the `Multiplier` block on its West Link (using `ReqResult` and `AckResult`). The block also generates one output, the value of the accumulator on its east link, which fans out using the timing cell routing to the `Multiplier` block and to the outputs using the `ReqAcc` and `AckAcc` handshaking signals.

A problem encountered in the circuit is that the final result should be implemented as a selective link, as it is only required when the last coefficient of the polynomial is read in, as signified by the `End` flag. However, the output to the constant multiplier is required on every cycle. To implement one output link selectively, and the other non-selectively, requires two separate links. However, insufficient links from the timing cell are available. To overcome this, the `EnabledOut` blocks are used. The unselective output link from the feedback block fans out to the `Multiplier` and the `EnabledOut` timing cells.



(a) Timing Array

(b) Data Array

Figure 10.17: Polynomial Evaluation at a Fixed Value

The `EnabledOut` timing cell is then used to generate the selective output link that is required. The `Counter` blocks generate the `Start` and `End` signals for the circuits. Each counter counts down from the length of the input polynomial, and generates the flag when reaching zero. Using two separate `Counter` blocks that count down from the same value could be considered a wasteful use of data array resources. However, implementing the counters separately allows the counters to proceed at different speeds, so `Feedback` blocks are not dependent on the `EnabledOut` blocks.

Data Array Configuration: The data array configuration reflects the breakdown into self-timed regions described for the timing array. Level-4 flyovers are used to route the input polynomial and the `Start` flag to the `Feedback` blocks. The `EnabledOut` blocks are implemented as FIFOs, but their output is only selected when the `End` flag is set. The select inputs for the `EnabledOut` blocks are driven directly from the level-4 flyover which carries the `End` flag.

Discussion: The example is a further illustration of creating a parameterised circuit from component blocks which themselves are parameterised. Changing the number of errors that a Reed-Solomon code can correct alters the number of syndromes required. The circuit can cope with this by adding more polynomial evaluation circuits to evaluate the extra syndromes. The circuit can also be parameterised to deal with longer polynomials by changing the value counted down from by the `Counter` blocks.

The example has also illustrated the limitations in terms of the number of links available to a single timing cell. If some form of variable granularity was available, then for example, the `Feedback` and `Multiplier` blocks could be combined to form one self-timed region, with more handshaking links. Thus, the `EnabledOut` blocks would not be required.

10.3.7 General Polynomial Evaluation

Description: The circuit described in Section 10.3.6 evaluated a polynomial over $GF(2^k)$ at a fixed value. This circuit evaluates a circuit at an arbitrary value, which is required during Reed-Solomon error correcting (see Section A.3.1).

Data Flow: Figure 10.18 illustrates the data flow for the circuit. The circuit again uses the Horner form (Equation 10.4) to evaluate the polynomial. The part of the dataflow which reads in the polynomial `PolyIn` and includes the adder, accumulator, and reset multiplexor is the the same as the data flow for the fixed value evaluation shown in Figure 10.16. The key difference from the fixed value evaluation circuit is the additional circuitry which reads in the value to be evaluated at, `Xin`. In the data flow of Figure 10.18, the value of `XIn` is read in to the evaluation value register when `Start` is set, and then fixed for the rest of the evaluation. The constant multiplier of the fixed evaluation circuit is replaced in this data flow with a general multiplier that takes one input from the accumulator and from the other from the evaluation value register.

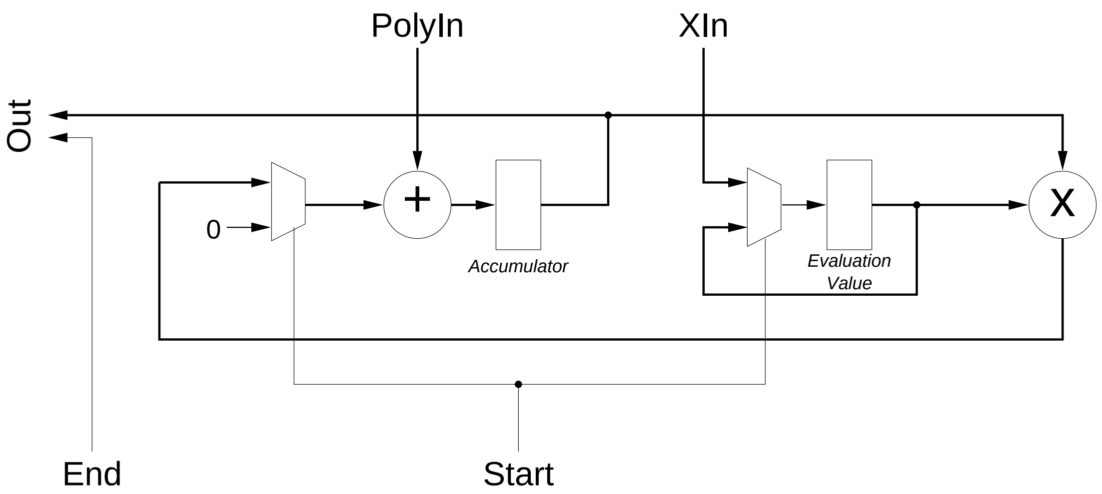
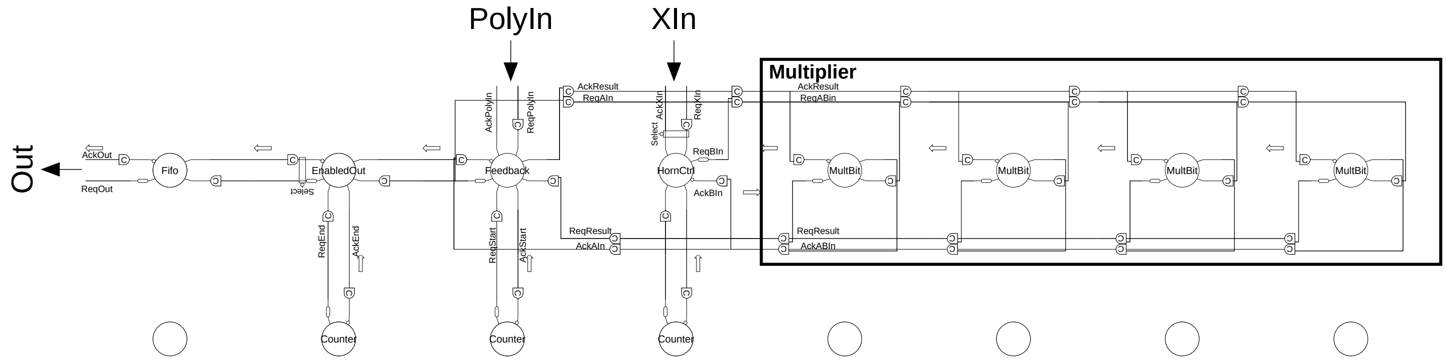


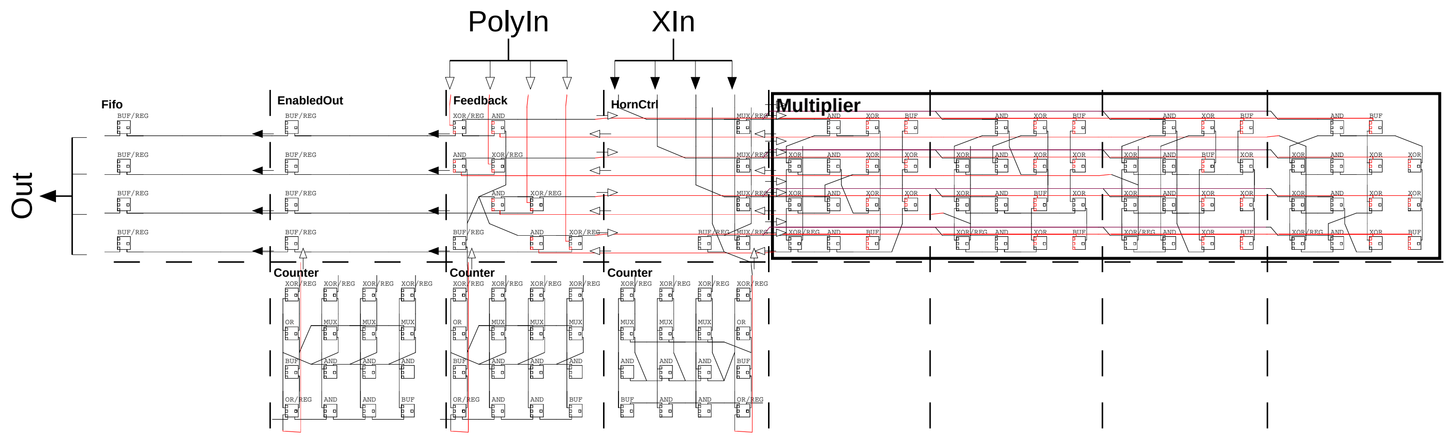
Figure 10.18: Data Flow for General Polynomial Evaluation

Timing/Data Array Configuration: The timing array and data array configuration of Figure 10.19 reflects the layout in Figure 10.18. The `MultBit` blocks on the left are an instance of the $GF(2^4)$ multiplier circuit from Section 10.3.3. The multiplier needs to be aligned on a 16×16 block as it uses level-16 flyovers. The `Feedback` and `EnabledOut` blocks were used in the fixed evaluation circuit, though their orientation is different in this example. The `Fifo` block is used to route to the edge of the 16×16 block.

The main new block in the example is the `HornCtrl` block, which is used to read in the value of `XIn` and output its value to the multiplier block on each cycle of the evaluation. Another `Counter` block is included which supplies the `Start` control signal to the `HornCtrl` block. Again, duplication of the counter is wasteful on data array cells, but allows the individual parts of the circuit to proceed independently of each other.



(a) Timing Array



(b) Data Array

Figure 10.19: General Polynomial Evaluation

Discussion: This example again illustrates the differences in parameterised circuits and general purpose circuits. The general evaluation circuit requires more complex control and a more complex multiplier circuit than the fixed value evaluation of Section 10.3.6.

10.3.8 $GF(2^k)$ Division

Description: The circuit divides two numbers in $GF(2^k)$.

Data Flow: Division can be expressed as the product of the dividend and a series of squares of the divisor, as discussed in Section A.2.3. Thus to divide y by x:

$$y/x = yx^{2^1}x^{2^2}\ldots x^{2^{k-1}} \tag{10.5}$$

a total of $k-1$ multiplications are required. Squaring is simplified by using the normalised polynomial representation, since squaring is simply implemented by a cyclic shift of the coefficients in the polynomial.

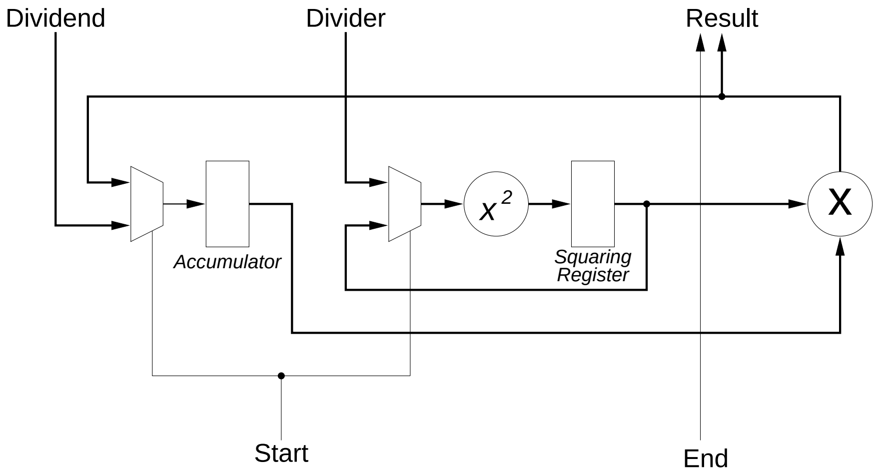
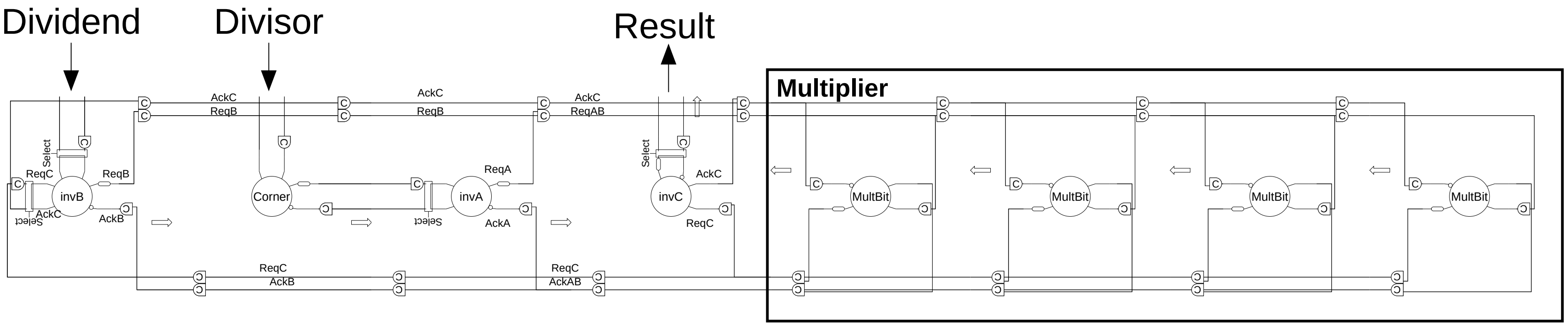
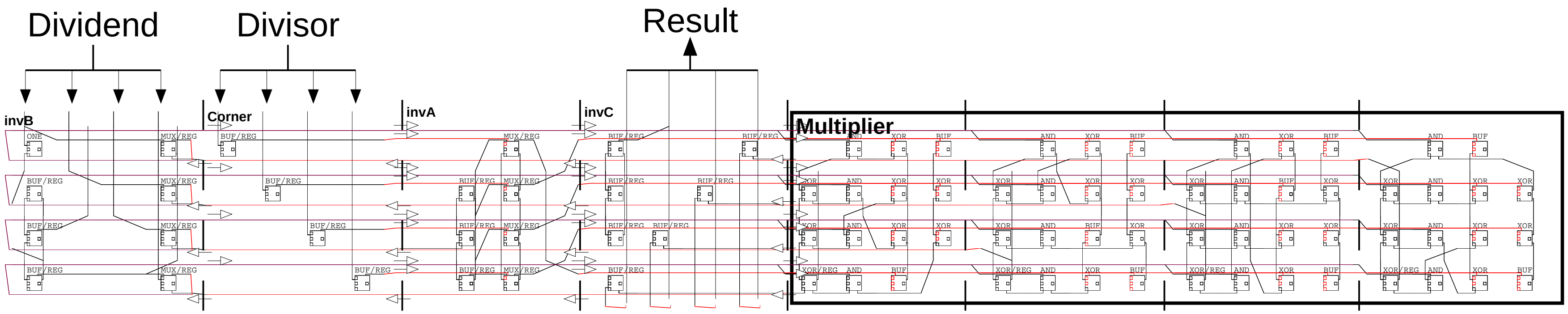


Figure 10.20: Data Flow for $GF(2^k)$ Division

Figure 10.20 illustrates the data flow for the division circuit. Initially, the `Start` flag is set and the dividend is read into an accumulator and the divisor is squared and stored in the squaring register. On each cycle of the evaluation, the contents of the accumulator and squaring register are multiplied, and the result placed in the accumulator, whilst the contents of the squaring register are squared. The `End` flag indicates the end of the evaluation, and signals the validity of the output.



(a) Timing Array



(b) Data Array

Figure 10.21: $GF(2^k)$ Division

Timing and Data Array Configuration: Figure 10.21 illustrates the configuration of the division circuit for the timing array and data array. The right hand four blocks are an instance of the $GF(2^k)$ multiplier circuit from Section 10.3.3. The `invB` block reads in the dividend and acts as accumulator for the calculation. The `Corner` block routes the divisor to the `invA` block which implements the squaring register. The normalised polynomial representation is used so squaring can be achieved by a cyclic shift (see Appendix A). The `invC` block acts in a similar way to the `EnabledOut` blocks used in previous examples; it reads out the result at the end of the calculation.

A difference of the `invA`, `invB` and `invC` control blocks from previous control blocks is that they do not require separate `Counter` blocks to generate the `End` signals. This is because only a small counter that counts to $k-1$ is required. This is implemented internally to the blocks using a single 'one' bit circulating in a shift register.

Discussion: The example is similar in layout to the general polynomial evaluation circuit discussed in Section 10.3.7. Both circuits use the $GF(2^k)$ multiplier circuit of Section 10.3.3 in their data paths. The main difference is in the generation of the `End` signals in the respective circuits. In the general polynomial evaluation circuit, the counter has to be implemented in a separate `Counter` block, whilst in the division circuit, the counters are small enough to fit within the self-timed regions that they control. Thus, the two examples illustrate how the breakdown into self-timed regions depends on the complexity of the different parts of the circuits.

10.3.9 Polynomial Remainder

Description: The circuit generates the remainder after dividing two polynomials $A(x)$ and $B(x)$ of the same length. The circuit is important as it forms the central loop in the Euclid's algorithm calculation that forms the basis of Reed-Solomon error processing [97]. The example illustrates some of the problems of constructing large circuits from parameterised building blocks.

Data Flow: The circuit generates $R(x)$ in the following equation:

$$A(x) = q.B(x) + R(x) \tag{10.6}$$

In the equation, q is the quotient, $A(x)$ is the dividend, $B(x)$ is the divisor and $R(x)$ is the remainder. The condition that $A(x)$ and $B(x)$ are polynomials of the same length ensures that q is not a polynomial but a scalar in $GF(2^k)$.

Calculation of the remainder involves dividing the first two (highest order) terms of the divisor and dividend polynomial to give q. Having calculated the quotient, $q.B(x)$ is subtracted from $A(x)$ to give the remainder $R(x)$. The generation of $q.B(x)$ could be done in one cycle given one $GF(2^k)$ multiplier for each term in $B(x)$; however this number of multipliers would be expensive to implement. Instead the multiplication is performed for each term of the polynomial in different cycles using only one multiplier.

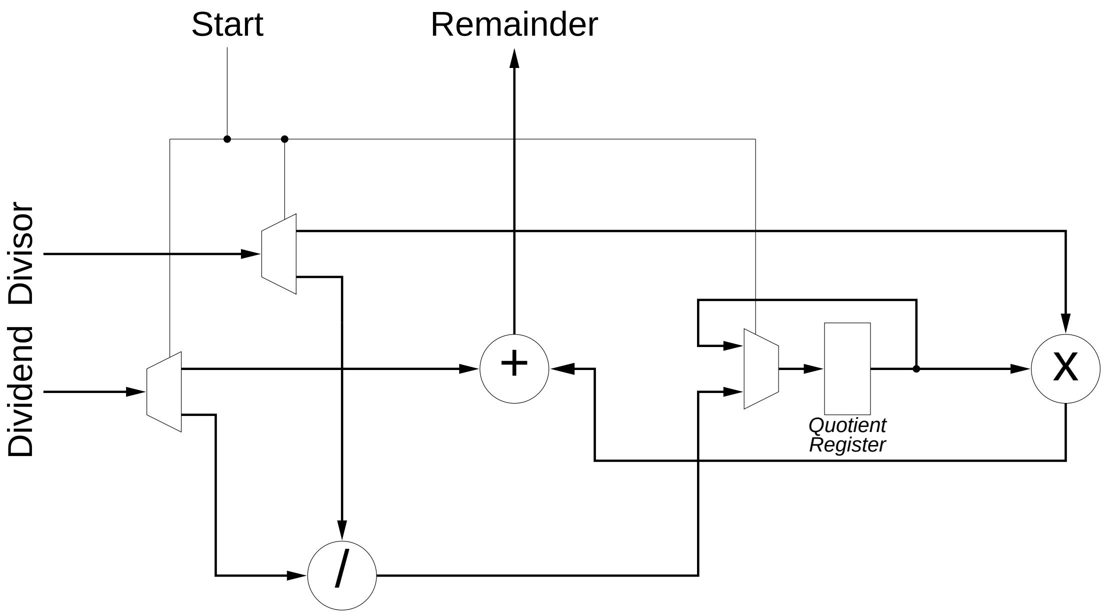


Figure 10.22: Data Flow for Polynomial Remainder over $GF(2^k)$

Figure 10.22 illustrates the data flow for the polynomial remainder circuit. The divisor and dividend polynomials enter on the left. The first term in each polynomial is sent to the division operator by the de-multiplexors. The result of the division is then read via a multiplexor into a register that supplies one of the inputs to the multiplier. The rest of the divisor polynomial $B(x)$ is fed to the multiplier, which multiplies it by q. This is then subtracted from the dividend polynomial $A(x)$ by the adder (addition and subtraction are identical over $GF(2^k)$), which generates the remainder $R(x)$.

Timing Array and Data Array Configuration: Figures 10.23 and 10.24 illustrate the timing array and data array configurations of the polynomial remainder circuit. It contains several circuits developed previously: the $GF(2^k)$ multiplier block was described in Section 10.3.3 and the $GF(2^k)$ division block was described in Section 10.3.8. The other principal data-path operator is the

Adder block.

The rest of the circuit is principally routing. The TJSelect (T-Junction Select) blocks implement the de-multiplexors that are used to route the first terms of the divisor and dividend polynomials to the divisor block rather than to the multiply/add data-path. The result of the division is fed into the HornCtrl block, which was used in the polynomial evaluation circuit of Section 10.3.7. As before, the HornCtrl block reads in a new value when the Start flag is set and retains this value for the rest of the calculation. The Counter blocks are used to generate the Start control signals that mark the start of a new polynomials. The Fifo and Corner blocks are used for routing.

Discussion: The circuit illustrates problems in the layout of parameterised circuits. The various parameterised circuits previously defined do not fit together well. This leads to a large number of blocks being unused in the rectangle which naturally bounds the circuit. Also, many of the blocks are being used purely for routing, such as the Fifo and Corner blocks. When parameterising circuits at run time, ease of layout is likely to be more important than optimal use of resources, so this wasteful style of layout is likely to be a common occurrence.

10.4 Design Techniques and Experience

The previous section detailed the implementation of run-time parameterised circuits for finite field operations. The example circuits illustrated a number of techniques for the design of circuits using the self-timed XC6200, as well as techniques for run-time parameterisation. These design techniques and the experience of the design process for the self-timed XC6200 are summarised below. Section 10.4.1 looks at the difference between the self-timed and synchronous design processes. Section 10.4.2 looks at the effect of granularity of the self-timed XC6200 architecture on designs. Sections 10.4.3, 10.4.4 and 10.4.5 discusses issues relating to the layout, utilisation and routing of the run-time parameterised designs.

10.4.1 Self-Timed Design Process

It is interesting to compare the difference between self-timed design and synchronous design processes. As shown in the examples, the design process for the self-timed system is similar to that for a synchronous circuit. The high level

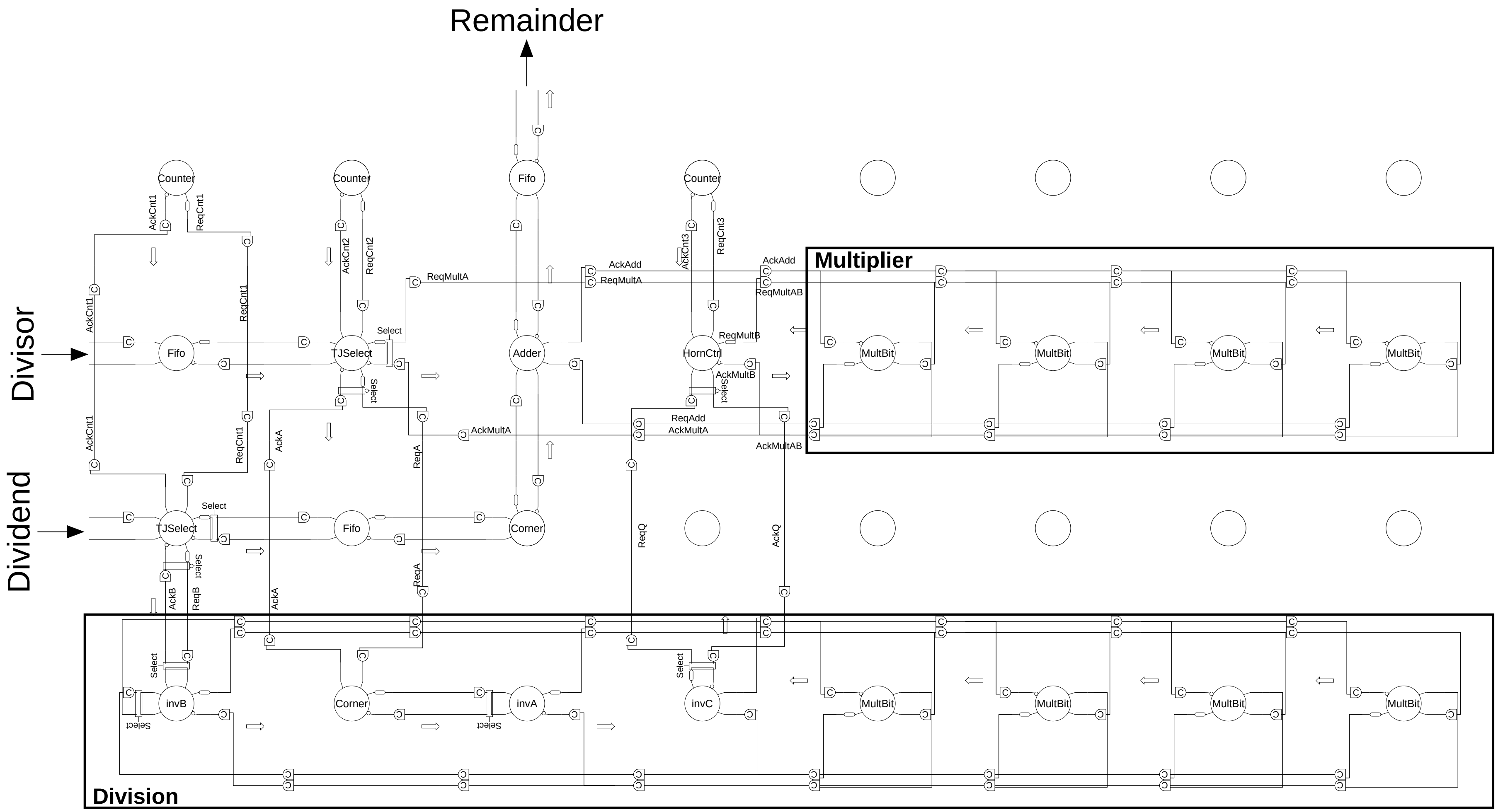


Figure 10.23: Polynomial Remainder Timing Array

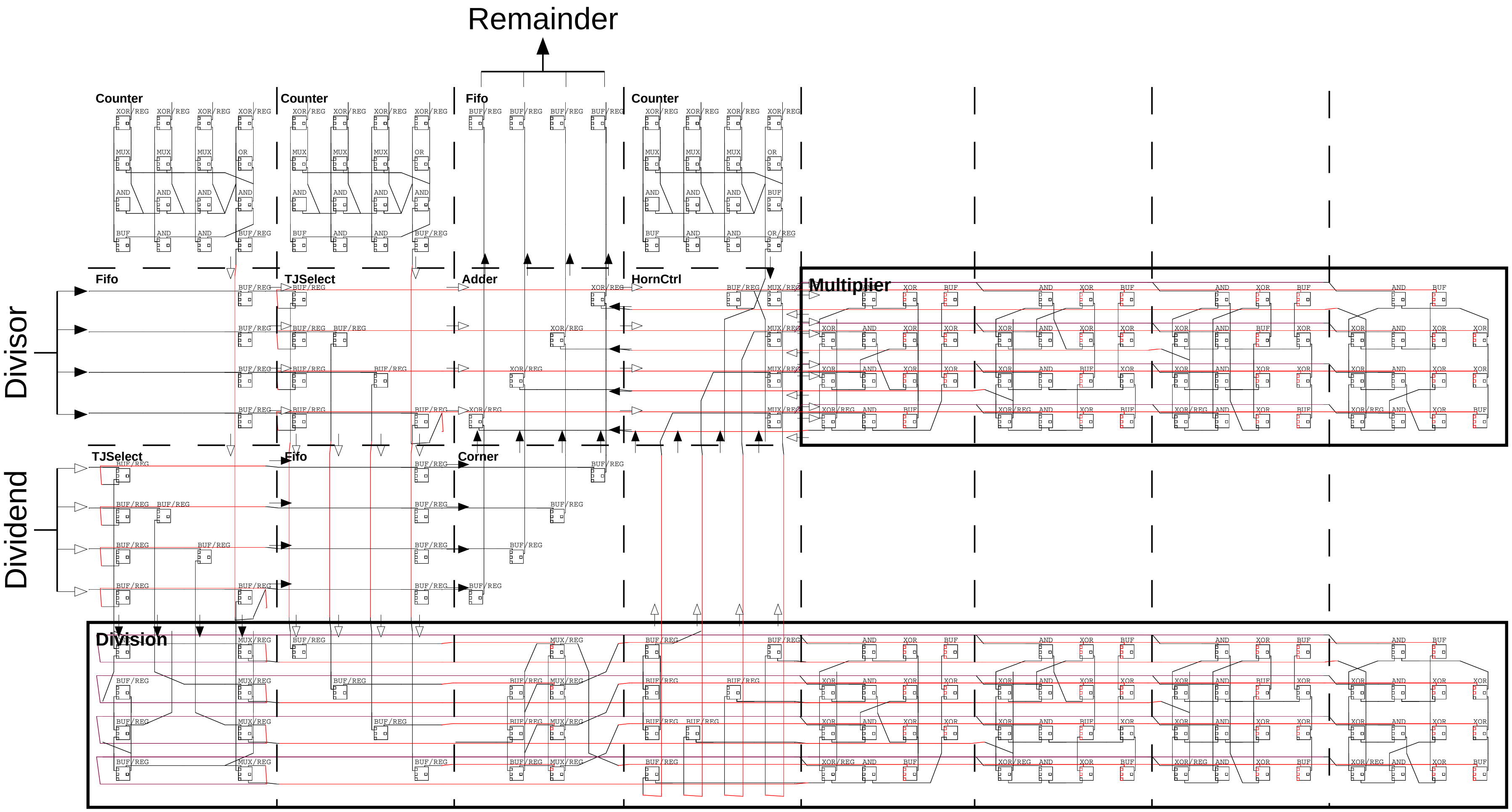


Figure 10.24: Polynomial Remainder Data Array

description of the circuit was expressed using a data flow diagram that gave a high-level view of the components required in the data path, and how these were interconnected. The data path elements were then refined into the basic gates to be implemented by the data cells in the data array.

The similarities of the data path between the synchronous and self-timed systems arise from using the bundled-data protocol. One of the principles behind the STACC architecture was to exploit this similarity to allow the transfer of design tools and experience from the synchronous architecture. This transfer was demonstrated in the examples by the use of the Xilinx tools to generate the pictures of the data array configuration.

Furthermore, despite the circuits being designed for the self-timed XC6200, the same data array configuration can be used directly in many cases to implement an equivalent synchronous design. In many other designs, the only alteration required is the removal of register elements. These elements are required in the self-timed XC6200, so each self-timed region can retain state. In synchronous designs, these registers delay the data by a clock cycle, which can change the values calculated by a circuit when there are loops in the data flow.

The key difference between the synchronous and self-timed design processes is the definition of the control path, which controls the flow of data between the data path elements. In synchronous circuits, additional circuitry has to be implemented in the data array to control the flow of data. In self-timed designs, the timing array configuration defines the pattern of data flow between data path elements, and imposes a style for implementing the control of the data flow between them.

This contrasts with synchronous designs where the methodology for controlling the flow of data between data path elements is left to the designer. For simple pipelines, no control is required: results are forwarded down the pipeline on each clock cycle. However, most designs involve more complex patterns of data flow that require explicit control. Synchronous circuits designers can use a distributed control methodology similar to the self-timed handshaking protocol to control the flow of data. Often, the control of a synchronous system is centralised, so that all the logic used to control the data path is grouped into one control block. Sometimes, a mixture of centralised and distributed control is used.

Synchronous versions of many of the example circuits would need additional circuitry to control the flow of data. For instance, the polynomial remainder circuit of Section 10.3.8 makes explicit use of flow control in the implementation of the circuit. The flow control involves sending the first term of each polynomial to the division circuit, whilst sending the rest of the polynomial terms to the multiply/subtract section of the data path. In the self-timed circuit, the calculation in the division block and the multiply/subtract blocks is triggered by the arrival of data. A synchronous version of the remainder circuit would require control circuitry to coordinate the flow of data between the division circuit and the multiply/subtract data path.

Apart from the difference in control implementation, the examples have illustrated the benefits of self-timed circuits for constructing run-time parameterised circuits. Many of the examples were composed from a number of blocks which themselves were parameterised, and had a wide range of different delays. When these blocks were assembled together to create a run-time parameterised circuit, the composition worked correctly, due to the modularity of the self-timed protocol. This is not possible with a synchronous run-time parameterised circuit, since delay analysis would be required to determine the speed at which the circuit could be clocked. Furthermore, parts of the self-timed run-time parameterised circuit could be separated, and even split across different FPGA chips, and the circuit would still operate, due to the speed-independence of the self-timed protocols.

10.4.2 Granularity

The fixed granularity of the self-timed XC6200 using 4×4 cell blocks as self-timed regions imposes structuring constraints on the data array and the timing array. It is often easier to define parameterised circuits at the level of individual cells rather than 4×4 blocks. For this reason, the technique of mapping from an abstract $N \times M$ block to the 4×4 blocks in the architecture was introduced.

Breaking down self-timed regions larger than a 4×4 block into self-timed regions of 4×4 blocks poses difficulties. Since each self-timed region must retain state, the granularity of the architecture forces designs to be pipelined at the level of 4×4 blocks. Though pipelining in self-timed designs can be done transparently, this has implications for performance of designs. This matter is discussed in the Chapter 11.

The breakdown into 4×4 blocks also requires the definition of the control flow between the component self-timed regions. In some cases, with linear flow of data from input to output, such as the constant multiplier of Section 10.3.4, this is simple. However, designs with less regular flow of data, such as large state machines, may be more difficult to partition between self-timed regions.

Another aspect of the granularity of timing array was illustrated with the `Feedback` block in the polynomial evaluation circuit of Section 10.3.6. In this circuit, insufficient timing cell links were available to implement a selective output link, so instead an extra timing cell had to be used to implement the selective output link. This resulted in a large number of wasted cells in the data array.

Many of the problems with the granularity can be solved by allowing more flexibility in the granularity of the self-timed regions. Possible extensions to the architecture to implement this are discussed in Section 11.5.

10.4.3 Layout

When parameterised circuits are composed into hierarchies, problems with joining the circuits together are encountered. For compile-time parameterised circuits, the problems of joining the parameterised parts together can be left for routing and placement algorithms to solve. However, run-time parameterised circuits require that the configuration can be constructed quickly, which requires a simple and regular pattern of routing and layout.

In the process of design, many of the smaller parameterised blocks had to be adapted to fit into larger designs. For example, the constant multiplier circuit of Section 10.3.4 was originally designed, with input and output data bundles at right angles, but a different version had to be designed for use in the syndrome evaluation circuit of Section 10.3.6 with inputs and outputs on the same side of the block. To give some flexibility in the use of blocks, the symmetry of the underlying XC6200 architecture was used extensively to allow designs to be flipped and rotated. To exploit this, asymmetrical features, such as the `Magic` routing were avoided in designs.

Despite using these techniques, fitting parameterised designs together still proved to be difficult. The general polynomial division circuit illustrates the problems; many of the blocks in the bounding box of the design are unused or simply used for routing. The design of such complex run-time parameterised circuits could benefit from the development of tools that allowed the circuit layout to be defined in terms of variable sized blocks. This would allow a more thorough exploration and design of 'virtual layouts' where the dimensions are not fixed until the run-time parameterised circuit is instantiated by configuration software.

10.4.4 Utilisation

Another noticeable aspect of many of the circuits is the poor utilisation of logic cells. For example, the polynomial remainder circuit of Figure 10.24 has a cell utilisation below 50%. It is possible to place and route the circuits using the Xilinx Tools, to give a superior place and route. The automatic place and route of the polynomial remainder circuit is shown in Figure 10.25. The utilisation of this circuit is superior, but is at the cost of losing the regular structure of the circuit. Circuits with such an irregular structure are difficult to construct rapidly at run-time. Thus, a penalty of the regular layout required for run-time parameterisation is that poor utilisation may often result.

Another aspect of the circuits leading to poor utilisation is that many cells are purely used for routing, and not for logic. This partially results from the regular layouts for different subcomponents not abutting well. Another cause is that the limited routing resources in the XC6200, forcing designs to be laid out in a more spacious fashion to ensure that they can be routed in a regular fashion.

A final cause of the poor utilisation is the fixed self-timed granularity of 4×4 blocks. Thus, cells are wasted as they do not fit within the 4×4 block. The constant multiplier of Figure 10.12(b) illustrated how this can lead to poor utilisation. A solution to this problem would be to allow greater flexibility in the granularity of the self-timed regions. This is discussed in Section 11.5.

An expected secondary effect of the poor utilisation and spacious layout of the run-time parameterised designs would be a degradation in performance due to longer routing paths being required. However, examining the automatic place and route in Figure 10.25, it can be seen that a lot of complex routing paths are found with signals taking long routes, sometimes with looping structures so that they can access the required routing resources. Thus, a cost of the more compact placement is that routing resource is more scarce, so more complex routing is required in many cases. This highlights the importance of balancing logic and routing resource in the design of FPGA architectures in general.

10.4.5 Routing

The design of run-time parameterised circuits with a regular routing structure illustrated a number of problems in the use of routing in the XC6200 data array. In particular, the use of the level-4 flyovers proved problematic. In general,

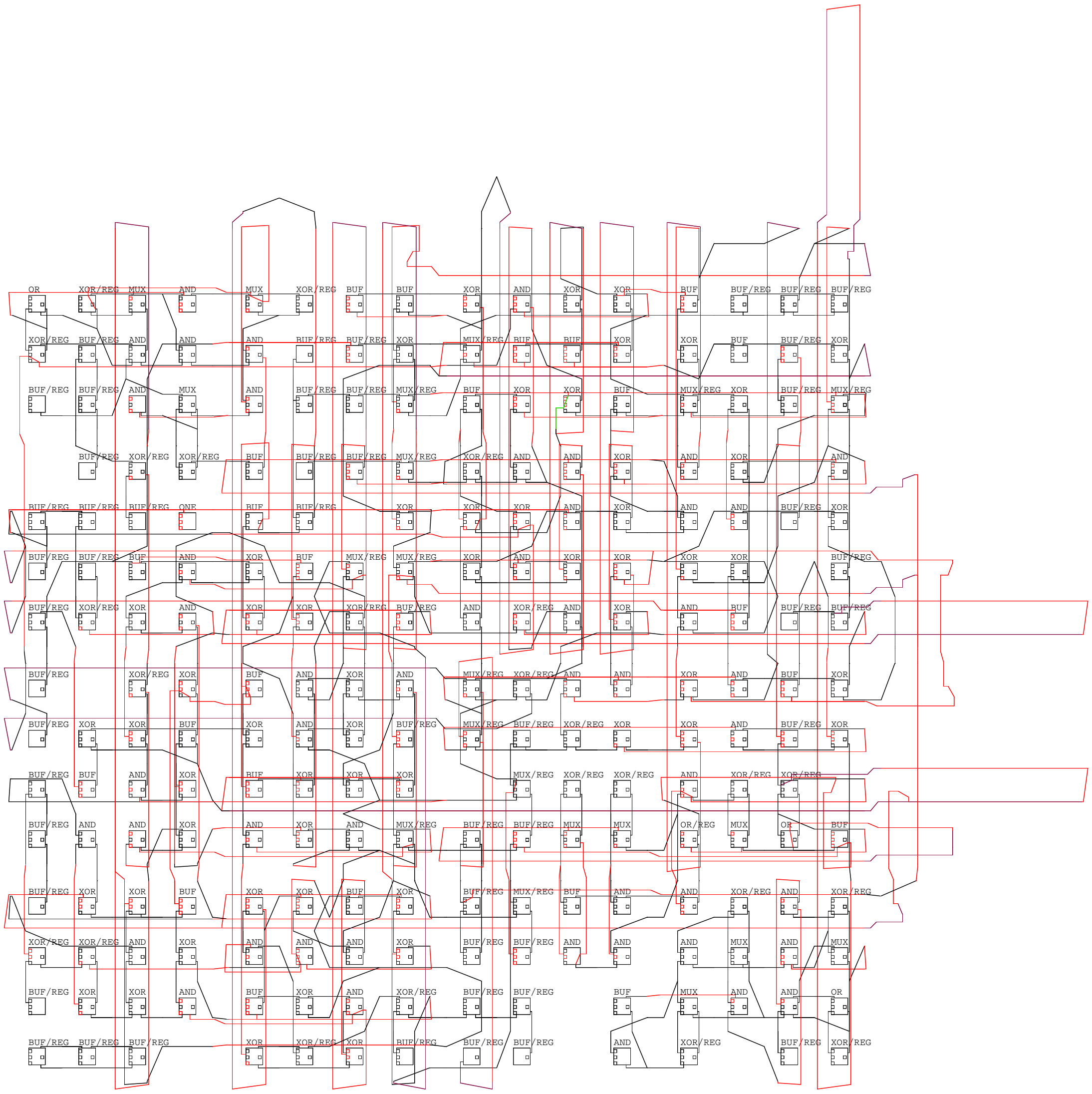


Figure 10.25: Polynomial Remainder laid out with no constraints

the level-4 flyovers were used for connecting the self-timed regions together. However, it is often useful to use the level-4 flyovers for routing within a 4×4 block. For example, the connection of two cells in the same row or column of a 4×4 block, but on different sides of the block, is quicker by using the flyover routing than by using the local routing.

This use of level-4 flyovers for routing internal to the self-timed region conflicts with its use for routing between self-timed regions. For instance, a single level-4 route used for routing internally to a self-timed region can prevent the regular routing of a bundle of data across the self-timed region. To avoid these conflicts the parameterised blocks were designed as far as possible to use the local routing.

Another conflict in the use of routing resources occurs with data buses turning corners. The XC6200 architecture provides the `Magic` signals to implement such corner turning. However, these signals can only be used if the `X2` or `X3` multiplexors in the cell at the corner can be suitably configured. To avoid such conflicts, buses that turned corners used blocks exclusively for routing, such as the `Corner` blocks that are used in the polynomial remainder circuit of Section 10.3.9.

The other aspect of the XC6200 routing architecture that caused problems in designs was the difficulty of routing from row to row or from column to column. The XC6200 routing architecture is good at routing signals within a single column or row, but changing rows or columns causes problems, especially when trying to construct regular designs suitable for parameterised circuits. For example, the general multiplier circuit of Section 10.3.3 was implemented as a number of bit slices, but with different cyclic shifts of the inputs. However, this cyclic rotation was difficult to implement in the flyover routing. Instead, the layout of the bit slices was modified to perform the cyclic shift internally.

10.5 Summary

This chapter has illustrated the construction of run-time parameterised circuits for the self-timed XC6200. The implementation of finite field operations is similar to the work by Klindworth [68]. However, the designs have been developed so that they can be constructed by a program on the fly, rather than synthesised by place and route tools as done by Klindworth. Configuration on the fly is made possible by the use of self-timed circuits; this allows the size of circuits to be altered, without having to perform delay analysis on the circuit. This allows the dynamic alteration of the power of an error correction code proposed by Klindworth to become possible over a wide range of codes, rather than being limited to the swapping in of pre-synthesised circuits for a small number of the different values of the parameters.

The chapter has also discussed a variety of design techniques for the self-timed XC6200. The designs have illustrated the way the timing array routing structure mirrors the routing patterns in the data array, and how the choice of a fixed granularity imposes constraints on the way designs are implemented. Also, the problems in construction of parameterised circuits have been shown: how the need for regular routing and layout structures causes problems in creating efficient designs. This chapter has not considered the performance aspects of the designs. This is discussed in the next chapter, which compares the self-timed XC6200 to the synchronous version.

Chapter 11

Self-Timed XC6200 Evaluation

11.1 Introduction

This chapter evaluates the self-timed XC6200 architecture relative to the synchronous XC6200. Detailed evaluation data is included in Appendix B. The self-timed XC6200 is evaluated in terms of delay performance and the implementation overhead of the timing array. The simulation of the self-timed XC6200 was not detailed enough to evaluate the power use of the architecture.

The timing array implementation overhead is considered in two ways. Section 11.2 evaluates how much additional circuitry is required to implement the timing array. Section 11.3 turns this question around, and considers how many data cells would be required to implement the timing cell behaviour using the synchronous XC6200. Section 11.4 examines the delay performance of the self-timed XC6200 architecture. Two delay methodologies are compared: reconfigurable fixed delays and Current Sensing Completion Detection (CSCD). Finally, in Section 11.5 the results of the evaluation are used to suggest improvements to the timing array and data array. In particular, the ability to group timing cells together to form larger self-timed regions is considered.

11.2 Implementation Overhead

This section examines the costs of implementing the timing array in the self-timed XC6200. The comparison does not try to assess whether the additional circuitry is justified by the additional functionality provided by the timing array. These benefits are considered in Section 11.3 which discusses the costs of implementing flow control in the synchronous XC6200 architecture.

There are several ways of evaluating the implementation costs of the architecture. A direct method of evaluation would be to compare the silicon area used by VLSI layouts of the synchronous and self-timed XC6200. However, creating a VLSI layout is a time consuming task, and gives a comparison that is highly dependent on the choice of process technology, and whether the circuits are designed for speed or compactness. Instead, three different metrics were used to evaluate the implementation cost: configuration bit count, wiring density and transistor count. Using all three metrics gives a range of assessments of the overhead of the architecture, largely free of concerns over the process technology used.

The comparison in this section is based on the fixed delay variant of the self-timed XC6200. A comparison based on the CSCD variant would give better figures for the self-timed architecture, since it uses fewer configuration bits and fewer transistors. However, the current sensing technology is best implemented using bipolar transistors, thus requiring a BiCMOS process technology [26]; this additional implementation cost is not easily evaluated by the metrics used.

11.2.1 Configuration Bit Count

The configuration bit count gives a measure of the extra information required to configure the timing array. Furthermore, the configuration memory forms a significant part of the circuitry to be implemented in the timing cell. The number of configuration bits can be used to give a rough guide to implementation cost of the circuitry, by assuming that the complexity of the circuitry is in proportion to the number of configuration bits. Though this assumption holds in general, there is a trade-off in the design of an FPGA between minimising the number of configuration bits and minimising the circuitry required to decode from the configuration bits to the control signals required by the FPGA.

Part	**Data**	**Timing**	**Overhead**
4×4 Block of Cells	256	34	13%
Boundary Routing	96	10	10%
Total	352	44	13%

Table 11.1: Configuration Bit Usage for 4×4 Block

A detailed breakdown of the configuration bits used in the timing array and data array in the XC6200 for a self-timed region is shown in Tables B.1 and B.2. The data is summarised in Table 11.1. Considering the configuration bits used for the timing cell (detailed breakdown in Table B.1(b)), most of the bits are used for functions that are configured for each link in the timing cell. The number of configuration bits for these functions could be reduced by either using a timing cell with less links, or centralising some functionality of the timing cell, for example, using one select input per timing cell rather than one per link. The disadvantage of this approach is that it reduces the potential behaviour of the timing cell.

Examining the figures for the boundary routing in Table 11.1, the timing array routing only uses an additional 10 configuration bits. However, this figure includes a deduction for the clock routing multiplexors, which are not required in the self-timed XC6200 architecture. Without this deduction, 26 configuration bits are required for the timing array routing relative to 96 for the data array. This gives an overhead of 27% for the timing array routing. This figure gives a fairer estimate of the complexity of the timing array routing relative to the data array routing.

Overall, the configuration bit overhead of the timing cell and timing array routing for a 4×4 block is 13%. The contribution from the timing array routing and the timing cells are similar in percentage overhead, but as the timing cell contributes a larger number of configuration bits to the total, the timing cell would be the best place to look for savings in the number of configuration bits.

11.2.2 Wiring Density

Wiring density gives an indication of the complexity of the routing, which is a significant implementation cost in VLSI systems. In the calculations, only signals connecting cells within the XC6200 are considered, rather than the implementation dependent routing internal to the cells. Also, the wiring required to implement the configuration SRAM is not included in the comparison, since it is again implementation dependent, but the costs could be expected to be in proportion to the number of configuration bits.

Part	**Number of Wires** *per 4×4 block*
Data Array	43
Timing Array	6
Overhead	14%

Table 11.2: Wiring Density in one dimension

Table B.3 has the breakdown of wires used in one dimension for the XC6200. The data is summarised in Table 11.2. Routing, such as the `CLK` and `CLR` signals, which only occur in one dimension, are averaged over the two dimensions in the data. The global signals GCLK, G1, G2, GCLR used for global clocking and reset in the synchronous XC6200 are not required in the self-timed XC6200. Global clock signals are not required, since timing is implemented locally, and global reset is not required, since reset is implemented in the self-timed XC6200 using the RESET configuration bit in the timing cell (See Section 7.3.4).

Comparing the wires required for the synchronous and self-timed versions, only 6 extra wires are required for the self-timed 4×4 block per dimension, giving an overhead of 14%. This measures the complexity of the circuit in terms of the additional inputs and outputs that need to be produced. However, if the routing area is the limiting factor in the silicon area used by an implementation then the one dimensional figure has to be squared, giving an overhead of 30%.

11.2.3 Transistor Count

The final metric used is transistor count; this metric should be used with caution, since the number of transistors is subject to variations depending on the implementation chosen for the circuitry. Also, transistors are not of constant size, since they require scaling according to the load that they drive.

Part	**Transistors**
Synchronisation	134
Select	224
Select Routing	96
To Register Delay	92
From Register Delay	54
Arbitration	108
RESET	6
Total	714

Table 11.3: Timing Cell Components Transistor Count

Part	**Data**	**Timing**	**Overhead**
4×4 block	4096	714	17%
Boundary Routing	1536	324	21%
Total	5632	1038	18%

Table 11.4: Transistor Count Summary

Table B.5(b) gives the breakdown of transistor usage in the timing cell; it is summarised in Table 11.3. The biggest component of the transistor count is the Select circuitry, largely due to the D-type registers required to sample the select state. The number of D-types could be reduced by using fewer select inputs.

Tables B.5 and B.6 detail the number of transistors used in the self-timed XC6200. The implementation used for the components is given in Table B.4. The data is summarised in Table 11.4. Overall the overhead for a 4×4 block of data cells, including routing is 18%. On these figures, the timing cell uses circuitry of equivalent complexity to three data cells.

11.2.4 Summary of Implementation Overheads

Table 11.5 summarises the overhead figures for the three different metrics used. The metrics cover a relatively small range from 13% to 18%. Thus, the timing array overhead is equivalent to between 2.5 and 3 data cells per 4×4 block.

Metric	**Value**
Configuration Bits	13%
Wiring Density	14%
Transistor Count	18%

Table 11.5: Summary of Implementation Overheads

In the table, the one dimensional wiring density is used. This gives a measure of the extra inputs and outputs required in the circuitry and thus is a measure of the circuit complexity. However, if the limiting factor in a design is fitting the wiring on to the chip, then the one-dimensional figure needs to be squared to give the additional chip area required. This gives a far higher overhead of 30%. The main cost if routing area is the limiting factor would probably be the need for extra routing layers in a VLSI implementation.

Overall, the figures suggest that the implementation overhead is less than 20%. Such an overhead is comparable with the overhead reported for other bundled-data systems [67, 39]. It is more difficult to extrapolate results from the self-timed XC6200 to STACC architectures in general. Other STACC architectures will have a different ratio of timing cells to data cells, which has a critical effect on the implementation overhead.

11.3 Flow Control without the Timing Array

One of the principal benefits of using a self-timed protocol is that flow control comes as part of the handshaking protocol. Hence, a useful way to evaluate the effectiveness of the timing cell is to consider how flow control can be implemented in the XC6200 without the timing array

Flow control can be implemented in the XC6200 in two ways: synchronously and asynchronously. Synchronous flow control uses a modified style of handshaking protocol within the constraints of the clocked protocol. Asynchronous flow control tries to directly implement the self-timed control structures that have been implemented in the timing array using cells in the data array. These two approaches are discussed below.

11.3.1 Synchronous Flow Control

Figure 11.1 shows the implementation of flow control for a FIFO using the synchronous XC6200. The request/acknowledge handshake of the self-timed protocol is not directly implemented, since this would require two clock cycles for a two-phase handshake and four clock cycles for a four-phase handshake. Instead, the synchronous flow control uses the `Valid` and `Ready` signals. The `Valid` signals are similar to request signals as they indicate that the data from a stage is valid, whilst the `Ready` signals are similar to acknowledge signals as they indicate that the stage is ready to accept data. In contrast to the local change of state that occurs in a self-timed pipeline, the next state of the whole synchronous pipeline is calculated on each clock cycle.

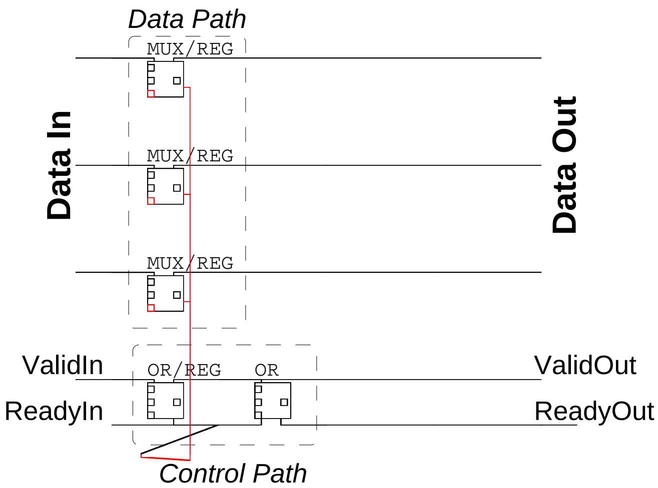


Figure 11.1: Synchronous Flow Control

The implementation shown in Figure 11.1, uses two cells for flow control: one cell generates the `ValidOut` output which indicates whether the stage is full or empty; the other cell generates the `ReadyIn` output. In a pipeline, the `ReadyIn` signals are generated by a chain of OR gates, one per stage, which takes the inverted `ValidOut` signals as inputs. Thus, if a stage is empty

(ValidOut for the stage is low), the ReadyIn outputs in the current stage and the stages before it go high, indicating that data can be shifted along the pipeline up to this point. The ValidOut output that indicates whether a stage is full is generated from the ReadyIn and ValidIn signals. A stage will remain full (i.e. ValidOut high), if all the following stages including the stage in question are full (indicated by ReadyIn being low), or will become full if the ValidIn input from the previous stage is high.

The ReadyIn output is also used as an enable signal for the registers controlled by the flow control circuit. Using ReadyIn as the enable signal means that spurious data can be captured into the registers when ValidIn is not set, but this data is ignored as ValidOut is only set when ValidIn is high. The capturing of spurious data does not overwrite useful data, since the ReadyIn signal indicates that the stage is ready to accept data. The implementation of registers with an enable signal in the XC6200 requires one cell per register element. Enabled registers are configured using multiplexors. When enabled, the register input comes from the circuit's inputs, when disabled, the output of the register is fed back to itself to retain the same state.

An alternative implementation of synchronous flow control, which does not require enabled registers would be to use a gated clock. This design would generate a local clock, by ANDing the ReadyIn signal with GCLK, and then distributing the result using the CLK routing to the cells. However, this introduces extra delays in the clocking, and also has problems with clock skew between CLK signals in different columns.

Considering the cell usage in Figure 11.1, to implement the flow control for a simple pipeline requires two cells for control plus one cell per register element to build an enabled register. This gives a minimum cell usage of three cells for a one bit wide FIFO. In comparison, the timing cell is equivalent to just under three data cells based on the transistor count metric. Implementing wider data paths in the synchronous flow control circuit requires more cells to be used to build enabled registers, whilst no penalty is incurred using the timing cell.

The FIFO represents the simplest flow control that could be required by a pipeline stage. In the self-timed XC6200, the timing cell can control more complex flows including the fan-in and fan-out of links and selective communication. Extending the synchronous flow control to deal with fan-in and fan-out signals requires extra data cells to AND the input ValidIn or ReadyOut signals together. Extending the flow control to provide selective communication would require additional data cells to be configured as AND gates to act as enables for the `ValidOut` and `ReadyIn` signals.

From this discussion it can be seen that implementing the functionality of a full timing cell in a synchronous XC6200, with many links, selective communication, arbitration plus a dedicated routing structure for handshaking signals requires a large number of data cells. The timing cell implements this for a cost of three data cell equivalents based on the transistor count metric.

11.3.2 Asynchronous Flow Control

The alternative to synchronous flow control described above, is to directly implement self-timed flow control in the data array. The basic element required for flow control in self-timed systems is the C-Muller gate. Figure 11.2 shows an implementation using the XC6200 logic cells of a two input C-Muller gate with inputs A and B, and output C. The circuit implements the C-Muller as the logic function $C = A(B+C)+\bar{A}(BC)$. The `OR` cell implements the $B+C$ term, the `AND` cell implements the BC term, and these are combined to produce C using the `MUX` with A as the select input. This C-Muller gate design uses one less cell than the design in the CAL1024 architecture by Oldfield and Kappler [92].

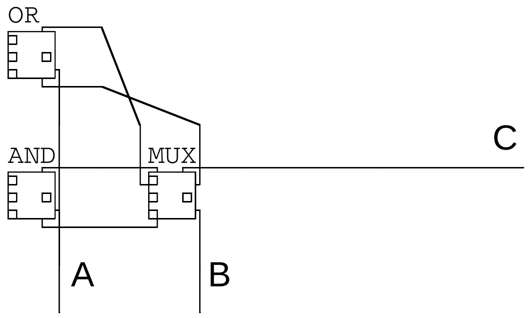


Figure 11.2: C-Muller Gate using XC6200 Logic Cells

A single C-Muller gate can control a two-phase pipeline, but two-phase memory elements are required. The two-phase memory element used by Oldfield and Kappler's designs [92] for the CAL1204 showed that the implementation of two-phase memory elements in a fine grained architecture is costly. An alternative would be to use four-phase handshaking and standard four-phase registers, but this option requires additional C-Muller gates for decoupling (see Section 7.4.1). To implement four-phase flow control using one C-Muller gate for controlling the registers and one C-Muller gate for decoupling would require six data cells. This is far greater than the overhead of implementing the timing cell (three data cell equivalents based on transistor count), even before considering more complex patterns of data flow that the timing cell can implement.

11.4 Delay Performance

This section compares the delay performance of the self-timed and synchronous versions of the XC6200. Two different delay strategies are compared for the self-timed XC6200, based on a fixed reconfigurable delay element and CSCD based delay element.

The discussion of delay performance is broken into three parts: in the first part, the method of calculating the delays is considered. The second part illustrates the different delays of the architectures using a number of example circuits. Finally, the delay performance of the circuits discussed in the previous chapter are tabulated and discussed.

11.4.1 Delay Calculation

The delays for the data array were calculated using the worst case delay values from the XC6200 data sheet [123]; these values are reproduced in Table B.7. The delay analysis was performed using specially written routines in the VHDL simulator. These routines give higher delay values than Xilinx's own tools, by about 20%. This difference occurs since the Xilinx tools take into account asymmetries in the delays for each direction of signal flow in the architecture, which are not published in the data sheet. The difference between the delays calculated within the simulator and by the Xilinx tools is illustrated in Figure B.1 which shows a profile of delays measured between logic blocks for a sample circuit by the simulator and the Xilinx tools.

The delay calculation routines were used to set the clock period in the simulation of the synchronous architecture, and to set the value of the delay elements in the self-timed version. In the simulations, it was assumed that both the global clock and the delay elements could take a continuous range of values. In practice, the reconfigurable delay element is restricted to a discrete number of delays by its implementation, and the clock is generally restricted to be a multiple of a master clock period generated by an off-the-shelf oscillator module. Thus, when implemented, both system clock period and delay element values would have to be be rounded up to the nearest discrete delay period.

The CSCD based architecture did not need delay analysis of the self-timed regions, since CSCD generates completion signals by monitoring the data array. In the simulator, the CSCD monitoring circuitry was attached to the power rails for the local routing multiplexors. The CSCD simulations had to use a value for the time taken by the CSCD circuit to determine that the circuit has completed; the value used in simulations was 5ns. The figure of 5ns was chosen as it is slightly longer than the longest possible un-monitored path that exists in a 4×4 block.

The simulation results for the CSCD architecture are based on the worst case delays given in Table B.7. The worst case values were used as no typical case data is given in the XC6200 data sheet. Thus, the CSCD values are conservative; actual CSCD implementations will have lower delays depending on how much better the actual operational delays are than the worst case.

In the timing array, the delay of the handshaking switchboxes was set to match the worst case delay of the level-16 boundary multiplexors (2.5ns). Signals routed via level-64 flyovers have longer delays (5ns) but are only routed from level-64 boundaries. Thus, level-64 signals can be accommodated by including additional delay into the the output of reconfigurable C-Muller gates on level-64 boundaries.

11.4.2 Synchronous and Self-Timed Circuit Implementations

In the following sections, self-timed and synchronous implementations of circuits are compared. To keep the comparison as fair as possible, the data array implementation has been altered as little as possible. However, the flow control that is part of the self-timed protocol does force some changes in synchronous versions of circuits, since some of the circuits rely on the flow control for correct operation. Synchronous implementations with the same behaviour would have to include flow control structures as outlined in Section 11.3.1.

Inclusion of flow control in synchronous designs can seriously disrupt the circuit structure; extra cells are required for the flow control logic and the enabled registers, and extra routing for the register enable signals. Rather than disrupt the data array, the synchronous circuits do not directly include the flow control. In most of the circuits, the flow control can be implemented separately, since the output is generated a fixed number of clock cycles after an input arrives. Instead, a counter can be used to generate a completion signal. However, for comparison purposes, the delay performance of synchronous flow control is discussed in Section 11.4.3.

The other difference of the synchronous circuits from the self-timed circuits is the nature of the pipelining. The fixed granularity of the current self-timed architecture forces pipelining (i.e. registers) to be included in each 4×4 block. In synchronous designs, there is more freedom on how designs are pipelined. However, in the case of loops in the design, the pipelining in the self-timed design must be removed for the synchronous design, otherwise the results being fed back will be delayed, changing the circuit's behaviour.

11.4.3 Case Studies

This section examines in detail the delays in four different circuits to illustrate the sources of delay in the different XC6200 architectures. The first example compares the delays for FIFOs; this compares the delays of the protocols, since no computation is done in the data array. The second case study examines the $GF(2^k)$ multiplier circuit to examine the delays in the timing array. The third case study looks at the delays in the fixed polynomial generator circuit; this example illustrates the delays arising from the pipelining imposed by the self-timed architecture. The final example looks at the data dependent delays for CSCD in a counter circuit.

FIFOs: Protocol Delays

FIFOs are pipelines without processing elements; as such they are a good comparison of the delays inherent in the synchronous and self-timed protocols, rather than the delays in the data array. The delays for various implementations of the FIFO are shown in Table 11.6. Two versions of the circuit are compared for the synchronous XC6200; one version includes no flow control circuitry and so effectively implements a shift register; the other includes the flow control circuitry described in Section 11.3.1.

Implementation		Delay /ns
Synchronous	Shift Register	13.5
Synchronous	Flow Control	7.5 + 11.5 N_{stages}
Self-timed	Fixed	14.4
Self-timed	CSCD	17.8

Table 11.6: FIFO delays

The synchronous shift-register implementation has a similar delay to the self-timed fixed delay version; both are close to the data path delay of 11.5 ns.

In the case of the synchronous shift register, there is the overhead of clock distribution (2 ns), whilst in the self-timed fixed delay version there is the overhead of returning the acknowledge signal to the previous stage (2.5ns), plus some small delays in the timing cell logic (0.4ns), which have not been accounted for in the delay element value. The main overhead in the self-timed architecture of returning the acknowledge signal could potentially be hidden by anticipating the completion of the logic block (a similar idea is used in [40]).

The synchronous implementation with flow control has a far larger delay than the shift-register implementation. The delay is proportional to the number of stages, N_{stages}, in the FIFO. This relationship with the number of stages arises from calculating the signals in the `Ready` chain; the longer the `Ready` chain, the longer the delay. Though in comparison with other FIFO implementations the delay is large, in general the flow control will only reduce the clock period of a pipeline when the processing between registers takes a shorter time than for the calculation of the flow control. For shorter pipelines, the processing will generally take longer, but for longer pipelines, the flow control may form the worst case path in the design. In this case, the `Ready` chain has to be broken by the introduction of registers into it. This lowers the performance of the pipeline, since a `Ready` signal propagating back from the end of the pipeline will take several clock cycles to reach the beginning of the pipeline.

The final value given in the table is the delay for a self-timed CSCD implementation. The period of 17.8ns is worse than the self-timed fixed delay implementation. The poor performance of the CSCD implementation occurs because there are few data dependent delays to be exploited in the FIFO. All the path lengths of signals in the FIFO are the same length, so the only data dependent delay to be exploited is when the same values are sent down the FIFO in succession. This did not occur in the test conditions used. With no data dependent delays to exploit, the same delays as for the self-timed fixed delay architecture are encountered by the CSCD architecture, plus an additional delay for generating the completion signal from the CSCD logic. Thus, this example represents the worst case example for CSCD; in later examples, data dependent delays can be exploited.

$GF(2^k)$ Multiplier Example: Timing Array Routing Delays

This section considers the delays in the timing array routing by examining the $GF(2^k)$ multiplier circuit described in Section 10.3.3. The circuit is composed of a number of 4×4 blocks that each generate one bit of the result. The inputs and outputs fan in and fan out using the level-4 and level-16 flyover routing. The timing array routing mirrors the data array routing with fanning in and fanning out routing to each timing cell using the level-4 handshaking routing.

Implementation		**Average Delay /ns**
Synchronous		50.5
Self-timed	Fixed	60.2
Self-timed	CSCD	42.9

Table 11.7: $GF(2^k)$ Multiplier delays

Table 11.7 summarises the delays for the different architectures. The synchronous implementation has a delay of 50.5ns. The critical path is composed of the delay from the 4×4 block furthest from inputs (42.5ns) plus the routing delay of the inputs on the level-4 flyovers through three boundary multiplexors ($3 \times$ 2.0ns), and clock distribution overheads (2.0ns).

For the self-timed fixed delay architecture, the average cycle time is 60.2ns. Again, the major component of the delay is the logic delay of the block furthest from the inputs (42.5ns). Additional to this, there is the delay of the request signal which passes through three reconfigurable C-Muller gates in the timing array routing (the delay of the reconfigurable C-Muller gate in the local routing is included in the fixed delay of the timing cell). The routing delay of the request signal in the timing array routing ($3 \times$ 2.5ns) is a close match for the delay of the input data routing through three boundary multiplexors ($3 \times$ 2.0ns), giving an overall request routing overhead of 1.5ns. A far larger overhead is the overhead of routing the acknowledge signal; this passes through four reconfigurable C-Muller gates giving a delay of 10ns. Thus, the example illustrates that the delay in the timing array routing can significantly delay the acknowledge signal, so leading to a large overhead relative to the synchronous architecture.

Despite the overhead in the acknowledge signal routing, the self-timed CSCD architecture has a shorter delay period than the synchronous architecture, showing that there are plenty of data dependent delays to be exploited in the circuit. Furthermore, the CSCD architecture overcomes the request signal routing overhead (1.5ns), since the late arrival of the request signal does not delay the data evaluation. The late arrival of the request signal will only delay the block if the request arrives after the block has finished evaluating. This property of CSCD can be used to give a greater margin between request and data to ensure the bundling constraint is met.

Fixed Polynomial Division: Pipelining Overheads

This section examines the delays in the fixed polynomial division circuit described in the previous chapter (Section 10.3.5). The circuit is an interesting example as it includes a feedback loop, rather than being a simple pipeline.

The self-timed and synchronous versions of the circuit differ in the pipelining (i.e. register usage) in the designs. In the self-timed designs, the architecture requires that each 4×4 block can retain state. In contrast, in the synchronous design, all the registers have to be removed, except in the `Adder` blocks, otherwise the feedback value would take more than one clock cycle to be fed back around the loop.

Implementation		**Delay /ns**
Synchronous		57.5 ns
Self-timed	Fixed Delay	73.3
Self-timed	CSCD	54.4

(a) Implementation Delays

Block	**To Reg Delay /ns**	**From Reg Delay /ns**	**Total Delay /ns**
`Corner`	16	5.5	21.5
`Scale`	17.5	4	21.5
`Adder`	14.5	1	15.5
`Tjunc`	10.5	1	11.5
Total			70.0

(b) Fixed Delay Breakdown

Table 11.8: Fixed Polynomial Division Delays

Table 11.8(a) summarises the delays for the fixed polynomial division circuit for a polynomial of length two. In the self-timed fixed delay architecture, the rate determining path is the feedback path through the bottom-most `Adder` block back to itself. This loop in the fixed delay self-timed architecture has a cycle time of 73.3ns. However, in the synchronous version the delay of the un-pipelined loop is only 53.5ns. Thus the unnecessary pipelining of the loop in the self-timed version has increased the delay by almost 20ns.

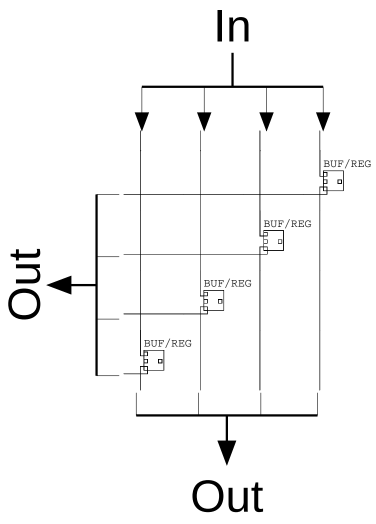
Table 11.8(b) shows the delay values used by the two delay elements in each 4×4 block that is part of the loop. The loop, which in the un-pipelined synchronous version is one continuous path, is broken into eight separate parts in the self-timed version. Each delay element represents the delay of the critical path through a small part of the loop. The only case when the worst case delay in the pipelined loop would equal the worst case delay in the un-pipelined loop would be when the critical path through the un-pipelined loop was composed from the critical paths of all the stages in the pipelined loop. This is very unlikely, especially when the loop is split into eight parts.

Thus, pipelining increases the delay around the loop, since the critical path through the pipelined stages of the loop does not correspond to the critical path through the un-pipelined loop. The problem can be overcome by moving the positions of registers in designs, so that new critical paths are not introduced. This is difficult to achieve in the the current self-timed architecture as there is not complete freedom on the positioning of registers, since registers must be placed in every 4×4 block. However, by moving the registers within the 4×4 block, the delay performance of the block can be improved.

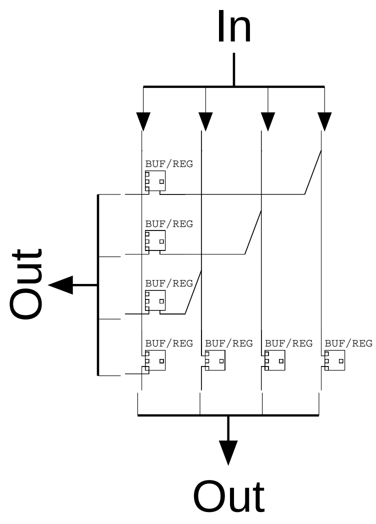
For example, Figure 11.3 illustrates the alteration of the `Tjunc` and `Adder` blocks to prevent the introduction of new critical paths. Without the registers, all the paths through the `Tjunc` block are critical as they are of equal length. However, the natural positioning of the registers in the original implementation of the `Tjunc` block changes the critical path (Figure 11.3(a)); the input critical path goes to the bottom left cell, whilst the output critical path comes from the top right cell. The delay performance is improved in the new version of the `Tjunc` block (Figure 11.3(b)) that has all the registers close to the outputs. This block does not disrupt the critical path through the block, so saving 4.5 ns relative to the old block (See Figure 11.3(e)).

Figure 11.3(c) illustrates the old version of the `Adder` block. In the un-pipelined `Adder` block, all the signal paths are critical, since they are of the same length, but in the pipelined version the critical path is changed. Again, this can be avoided by moving the registers to the edge of the block, as in Figure 11.3(d). However, to use the registers at the edge of the block, additional buffer (`BUF`) cells are introduced. These increase the length of the critical path, so that it is only 0.5ns faster than the original version of the `Adder` block (See Figure 11.3(e)).

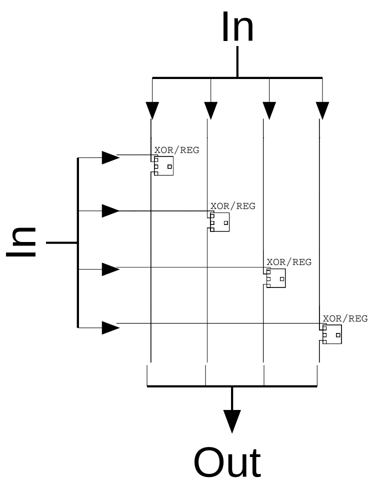
Returning to the comparison of delay performance shown in Table 11.8(a), the synchronous implementation has a different critical path to the self-timed version. The critical path in the synchronous version is not the delay around the bottom-most feedback loop as in the self-timed version, but is the delay of



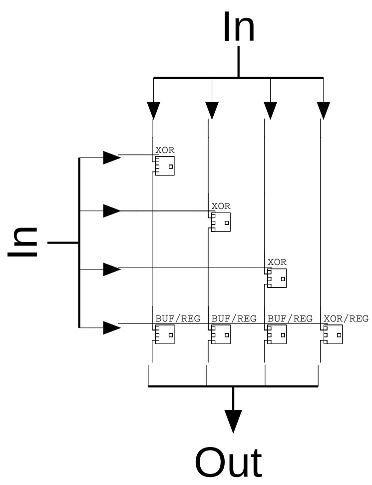
(a) Old Tjunc



(b) New Tjunc



(c) Old Adder



(d) New Adder

Block	**To Reg Delay /ns**	**From Reg Delay /ns**	**Total Delay /ns**
Old `Tjunc`	10.5	5.5	16.0
New `Tjunc`	10.5	1	11.5
Saving			4.5
Old `Adder`	10.5	5.5	16.0
New `Adder`	14.5	1	15.5
Saving			0.5

(e) Delay Table

Figure 11.3: Moving Register Locations

from the bottom-most `Adder` block, around the feedback loop to the topmost `Adder` block. This gives a cycle period for the synchronous architecture of 57.5ns. Despite the pipelining overheads of the self-timed architecture, the CSCD implementation manages to out-perform the synchronous version of the circuit, illustrating the large number of data dependent delays in the circuit.

Counter Example: CSCD Delays

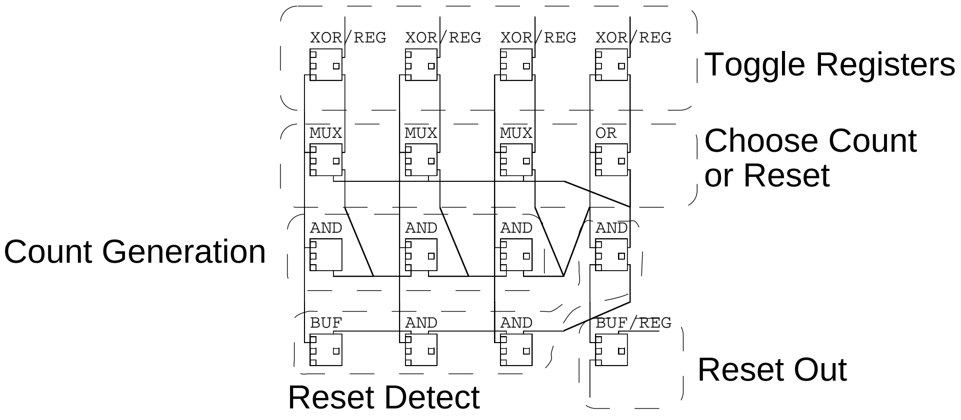
Figure 11.4(a) illustrates an example of a parameterised circuit that counts down from a set value to zero. The example is used to illustrate the exploitation of data dependent delays in the CSCD architecture implementation.

The functions of groups of cells in the circuit are marked on Figure 11.4(a). The state memory of the counter is implemented using the top row of cells which are configured as Toggle registers. The row of cells below are concerned with resetting the Toggle registers to a pre-defined value; these cells are parameterised according to the desired reset value. In the example, the counter is reset to the value 14 (this is not apparent as the Xilinx tools do not show the inversions in the circuit). The next row of cells generate the next state for the counter by determining which registers to toggle. Finally, the bottom row of cells detects the reset state, which is when all the toggle registers are zero. The reset signal is also available as an output from the circuit.

Table 11.4(b) lists the delays for each architecture. The delays for the synchronous and the self-timed fixed delay architectures are similar. This is to be expected, since there is no alteration in the circuit between the synchronous and self-timed architectures in this case. The cycle time for both architectures is determined by worst case delay through the circuit which occurs when the counter is reset from 0 to 14. The worst case delay runs from the output of the most significant Toggle register along the chain of AND gates used to detect the reset state, and then back along the line of multiplexor (`MUX`) cells used to reset the Toggle registers.

The CSCD architecture is far quicker, giving an average delay that is half the time of the other architectures. Figure 11.4(c) shows the distribution of delays. Seven of the 15 possible state changes only result in the lowest bit being changed, so the minimum delay of 16.5ns is encountered. As more bits are changed, the delays increase up to the worst case for the reset from 0 to 14, where the maximum delay of 25.5ns is encountered

Unexpectedly, the worst case delay for the CSCD circuit is not the same as the cycle time for the other architectures. This apparently anomalous beha-

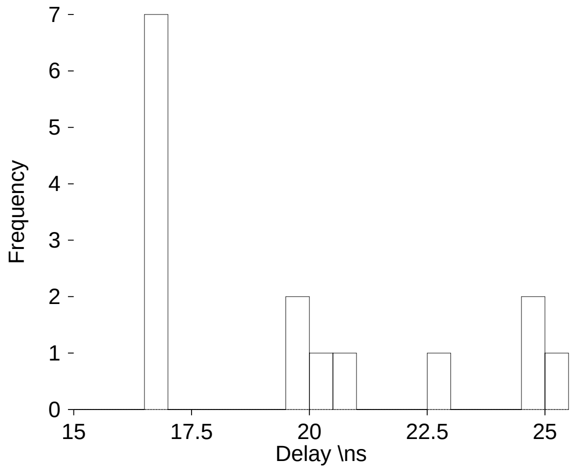


(a) Circuit

Architecture		Delay /ns
Synchronous		42.0
Self-Timed	Fixed Delay	43.7
Self-Timed	CSCD	19.5 [1]

[1] average cycle time

(b) Delay Table



(c) CSCD Delay Distribution

Figure 11.4: Counter Example

viour can be explained by examining the circuit in more detail. The critical path of the circuit in Figure 11.4(a) runs through a chain of AND gates that detect the reset state. The delay along the critical path through the logic is 42ns, but the CSCD architecture takes advantage of the actual sequence of states that the circuit goes through. When the actual reset state is reached, all except the least significant Toggle register are already zero, so the worst case delay only runs through the last AND gate in the reset detect chain.

Thus, the CSCD analysis shows that the counter can actually be run at a clocking period of 25.5ns for the fixed delay and synchronous architectures. However, there is no way for tools that simply calculate the critical path delay through the combinatorial logic of the circuit to calculate this value. Tools that analyse the worst case delay time for each possible state change are not in general use, due to the combinatorial explosion problem for large state spaces.

Despite the special circumstances of the counter, the average cycle time for the CSCD architecture is 7 ns (25%) better than the cycle time that the self-timed fixed delay and synchronous architectures could be run at.

11.4.4 Delay Performance Summary

This section has examined the sources of delay in the self-timed XC6200 architecture. The delays for the circuits discussed in this chapter and the previous chapter are summarised in Table 11.9. The calculation of the figures in each column of the table are summarised in Table 11.10.

The examples discussed above have highlighted several sources of the overhead for the self-timed XC6200. The FIFO example showed that the basic delays of the self-timed and synchronous protocols were comparable. In the synchronous protocol, the main overhead is clock distribution whilst in the self-timed protocol the main overhead is returning the acknowledge signal. The acknowledge overhead can be significant when it is routed through the timing array routing, as illustrated by the $GF(2^k)$ multiplier. The distribution of the request signal is a smaller overhead, since it matches the routing delay of the data signals.

The fixed polynomial division example illustrated the delay overheads encountered through the enforced pipelining of the self-timed architecture at the level of 4×4 blocks. The overhead arises from the pipelined circuit having different critical paths from the critical path in the un-pipelined circuit. These delays are particularly significant in loops, which form the rate determining step in many of the circuits listed in Table 11.9. In these cases, it would be

Circuit	Clock Period (self-timed pipelining) /ns	Clock Period (sync. pipelining) /ns	Cycles per Result	Synchronous Result Period /ns	Self-Timed (Fixed Delay) Result Period /ns	Self-Timed (Fixed Delay) : Sync Difference	Self-Timed (CSCD) Result Period /ns	Self-Timed (CSCD) : Sync Difference	CSCD: Fixed Delay Difference
FIFO	13.5	13.5	1	13.5	14.4	6.7%	17.8	31.9%	23.6%
3 Stage Synchronous FIFO	N.A.	42.0	1	42.0	N.A.	N.A.	N.A.	N.A.	N.A.
Count Down to 14	42.0	N.A.	1	42.0	40.2	-4.4%	19.5	-53.6%	-51.5%
$GF(2^k)$ Multiplier	50.5	50.5	1	50.5	60.2	19.2%	45.9	-9.1%	-23.8%
Fixed Polynomial Division	42.0	57.5	1	57.5	73.3	27.5%	54.4	-5.5%	-25.8%
Fixed Polynomial Evaluation	27.5	40.5	4	162.0	213.6	31.9%	137.0	-15.4%	-35.9%
General Polynomial Evaluation	53.0	78.5	4	314.0	387.6	23.4%	267.8	-14.5%	-30.9%
$GF(2^k)$ Division	59.0	82.5	4	330.0	323.7	-1.9%	272.1	-18.5%	-15.9%
Polynomial Division	59.0	98.5	1	98.5	94.7	-3.9%	85.1	-13.6%	-10.1%
Average						13.1%		-12.5%	-21.3%

Table 11.9: Performance Figures

Clock Period (self-timed pipelining): Clock period with pipelining at level of 4×4 blocks as used in self-timed circuit. Potential speed of synchronous circuit with flow control at this level of pipelining.
Clock Period (sync. pipelining): Clock period with pipelining removed to ensure correct operation in synchronous circuit without flow control; e.g. pipelining removed from around loops in circuit.
Cycles per Result: Number of clock cycles for synchronous circuit to produce result
Synchronous Result Period: Time between results being produced. Product of Clock Period (synchronous) and Cycles per result.
Self-Timed (Fixed Delay) : Sync Result Period: Average time between results output for self-timed XC6200 using fixed reconfigurable delay. Fixed delay architecture cannot exploit data-dependent delays within self-timed region.
Self-Timed (Fixed Delay) : Sync Difference: Percentage difference from synchronous result period.
Self-Timed (CSCD) Result Period: Average time between results output for self-timed XC6200 using CSCD delay. CSCD architecture can exploit data-dependent delays within self-timed region.
Self-Timed (CSCD) : Sync Difference: Percentage difference from synchronous result period.
CSCD: Fixed Delay Difference: Percentage difference of CSCD average result period from fixed delay result period. Indicates amount of data-dependent delays to be exploited within self-timed regions.

Table 11.10: Description of Performance Results Figures

preferable if the pipelining was not required; in other words that the loop was one self-timed region rather than several. This requires a self-timed architecture with variable granularity, which is discussed in the next section.

Despite the performance overheads of the self-timed XC6200 architecture, some of the larger examples listed in Table 11.9, such as the $GF(2^k)$ divider and polynomial remainder circuits, have better average delay times for the fixed delay self-timed implementation than the synchronous implementation. This arises because these examples include selective communication, so are not simple pipelines. Whilst the synchronous implementation is always limited to the worst case delay in the entire circuit, the self-timed circuit is exploiting the average case delays arising from the worst case delay parts of the circuit only being used selectively.

Examining the figures in Table 11.9 for the CSCD architecture, it can be seen that, except for the FIFO, the CSCD architecture out-performs the synchronous and self-timed architectures. These figures are conservative, because the CSCD delays are based on worst case delay data, rather than typical case delay data which is unavailable for the XC6200.

As well as exploiting data dependent delays, the CSCD implementation is beneficial as it masks the overhead of routing the request signals; a timing cell will only be delayed if the request signal arrives after the next pipeline stage has completed evaluation. Thus, the margin between the request and data signals can be increased to ensure the bundling constraint is met.

The final column in Table 11.9 highlights the difference between the self-timed architecture with fixed delay scheme and CSCD scheme. The fixed delay scheme represents the worst delay that the self-timed system could achieve, since their is no way to utilise data dependent delays within the self-timed regions; only data-dependent delays between self-timed regions can be exploited. In contrast, the CSCD architecture can exploit all data-dependent delays within the self-timed regions. Thus, the difference between these two figures give the range of performance that other self-timed delay methods (see Section 6.4) could expect to achieve.

11.5 Extensions to the Architecture

Following on from the delay performance analysis above, this section considers possible alterations and extensions to the self-timed XC6200 architecture to improve performance. First, alterations to the data array to improve performance for both the fixed reconfigurable delay and CSCD architectures are considered. These alterations were not included in the simulated architecture, since they conflicted with the aim of producing a self-timed XC6200 with the same data array as the synchronous XC6200.

The majority of this section is devoted to considering extensions to the timing array to allow variability in the size of self-timed regions. Many of the performance problems of the current architecture arise from pipelining being enforced at the level of 4×4 blocks.

11.5.1 Data Array Alterations for Fixed Reconfigurable Delays

A major source of performance overhead in the current self-timed XC6200 architecture is the requirement for each 4×4 block to be a pipelined stage. Thus, the critical path through the logic is split into smaller stages with their own critical paths, which make the overall critical path longer. This problem is compounded by the fact that registers occur internally to the 4×4 block in the XC6200. This means that two delay elements are required per 4×4 block; one for the delay from the edge of the 4×4 block to the register and one for delay from the output of the registers to the edge of the 4×4 block. Thus, the delay internal to a 4×4 block is split into two parts which may not correspond with the critical path of the block as a whole.

In Section 11.4.3, it was shown how the performance of 4×4 blocks could be improved by moving registers to the edge of the block, to balance the delays. Thus, the architecture could be improved by modifying the data array so that registers were located on the edge of 4×4 blocks rather than in the data cells. As a result only one delay element would be required per timing cell for the delay from the inputs of the 4×4 block to the outputs of the 4×4 block.

11.5.2 Data Array Alterations for CSCD

The simulations have demonstrated the performance benefits of exploiting data dependent delays using CSCD. However, the data dependent delays are limited by the use of registers for the memory elements in the architecture. Registers block any results from propagating further until they are clocked. In contrast, latches are transparent; results will propagate through them and be stored when the whole stage has completed. Hence, using latches as the memory elements in a CSCD based architecture would be advantageous, since partial results can propagate forward, and lead to early completion by CSCD detection mechanisms. Furthermore, using latches overcomes the extra delay encountered with pipelining in the current self-timed architecture as the flow of signals along the critical path is not blocked by registers.

Register memory elements are still required for storing the state in finite state machines. This can be implemented by adding a configuration bit to the memory element that determines whether it is configured as a latch or a register. As with the fixed delay architecture, placing registers on the edge of 4×4 blocks is advantageous, as it simplifies the timing cell logic because only one, rather than two, delay phases are required.

11.5.3 Variable Granularity

The performance analysis has shown that many of the overheads in the architecture arise from the fixed granularity imposed by the self-timed architecture, which results in unnecessary pipelining. This also leads to unnecessary complexity in the design of circuits, since designs have to be broken down into 4×4 blocks, regardless of their natural granularity. To overcome these problems, a mechanism to allow variability in the size of self-timed regions is required. Two main criteria have to be met by such a scheme:

Uniform Local Clock Distribution: The local clock should be distributed uniformly across the self-timed region, otherwise local clock skew can occur. This becomes increasingly problematic for larger self-timed regions.

Scalable Behaviour: Larger self-timed regions will often require more complex timing cell behaviour than smaller self-timed regions. Furthermore, larger self-timed regions will generally have longer delays, so the range of delays of the self-timed region should scale as well.

Section 9.3.2 discussed two methods of implementing variable granularity. The first method was based on a flexible local clock distribution. The local clock from a timing cell could be distributed over a range of different areas to allow flexibility in the granularity. However, this method does not give scalable behaviour, since only one timing cell is ever used to control a self-timed region. The second method discussed in Section 9.3.2 was to group timing cells to provide scalable behaviour. Potential mechanisms for grouping timing cells and extensions to the self-timed XC6200 architecture are discussed here.

The basic mechanism for grouping timing cells is shown in Figure 11.5. The behaviour of the timing cells is modified so that, before generating their local

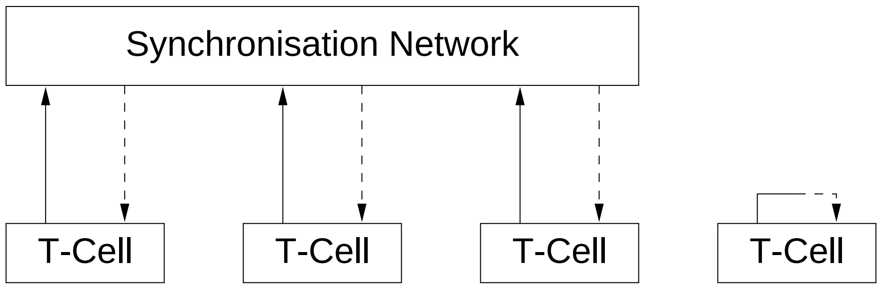


Figure 11.5: Grouping Model

clock signals, the timing cells are synchronised together, to produce a common local clock signal. In Figure 11.5, three timing cells are shown synchronising together to form a self-timed region with a common local clock (the solid line indicates synchronisation signals, the dotted lines represents the local clock signal). The timing cell on the right in the figure does not synchronise with any other timing cells; this is indicated by the synchronisation signal being fed directly back as the local clock signal.

The synchronisation network requires the implementation of a C-Muller gate with a potentially large fan-in. Constructing such C-Muller gates in a way that allows timing cells to be grouped flexibly into self-timed regions is difficult. Several schemes are discussed below.

Distributed C-Muller Gate Grouping

The distributed C-Muller gate seems a natural candidate for implementing the synchronisation network. Figure 11.6 illustrates a scheme that uses the distributed C-Muller's gate structure to synchronise a 3×2 group of timing cells. All the timing cells are connected to two synchronisation wires, which are used to construct the distributed C-Muller gate. The synchronisation wires are connected together using fuses. These fuses can be blown to isolate sections of synchronisation wires. Timing cells on connected pairs of synchronisation wires form a self-timed group.

The advantage of this scheme is the flexibility in forming self-timed regions. Any timing cells that can be connected by a pair of synchronisation wires can form a self-timed region. The problems with the scheme are similar to problems mentioned with other distributed C-Muller gate circuits. The wired-OR circuits require low resistance connections found in fuse based designs rather than SRAM FPGAs, and the wired-OR rise time is slow. Finally, the problems in where to place pull-up resistors and voltage gradient between these and the

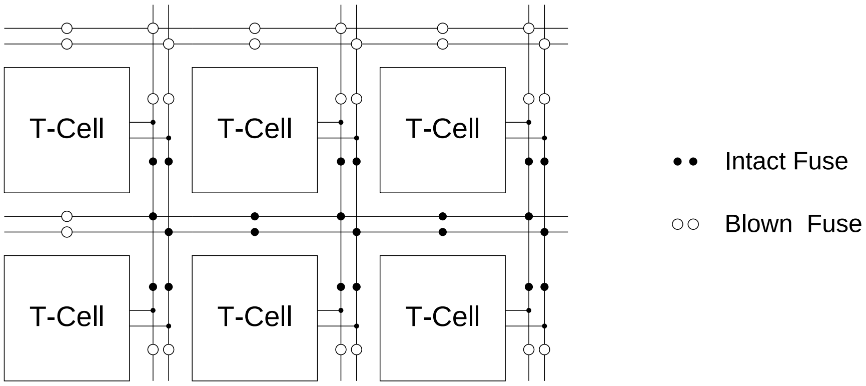


Figure 11.6: Wired-OR grouping

open-collector pull-down transistors, limit the size of self-timed region that can be formed using a distributed C-Muller gate implementation.

C-Muller Gate Tree using Local Handshaking Links

When timing cells are being used to form a self-timed region, the nearest neighbour links internal to the self-timed region are not used. Figure 11.7 illustrates a scheme that uses these links to create a synchronisation network.

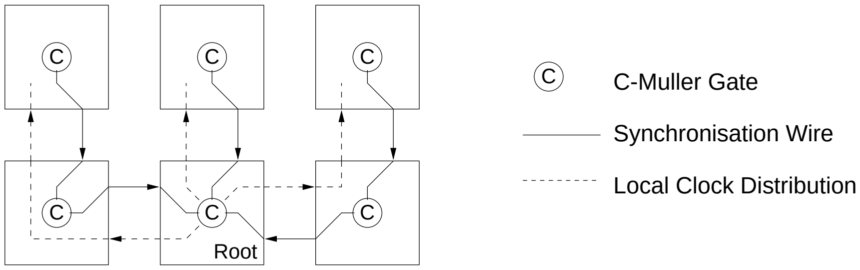


Figure 11.7: Grouping using Nearest Neighbour Links

In the scheme, each timing cell has an additional reconfigurable C-Muller gate associated with it. The synchronisation network is formed from a tree of these C-Muller gates. Each reconfigurable C-Muller gate takes the synchronisation input from the local timing cell as an input, plus optionally synchronisation inputs from other timing cells which are distributed on the nearest-neighbour links. If the reconfigurable C-Muller gate is the root of the synchronisation tree then the output of the C-Muller gate is the local clock signal for the self-timed region. The clock is distributed to all the timing cells using the nearest neighbour links, following the reverse path to the fan-in of the synchronisation signals. If the reconfigurable C-Muller gate is not the root of the C-Muller gate tree then the output is passed along a nearest-neighbour link to

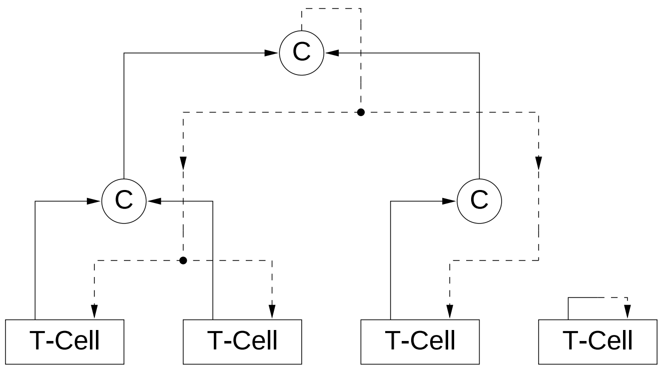


Figure 11.8: Hierarchical Grouping

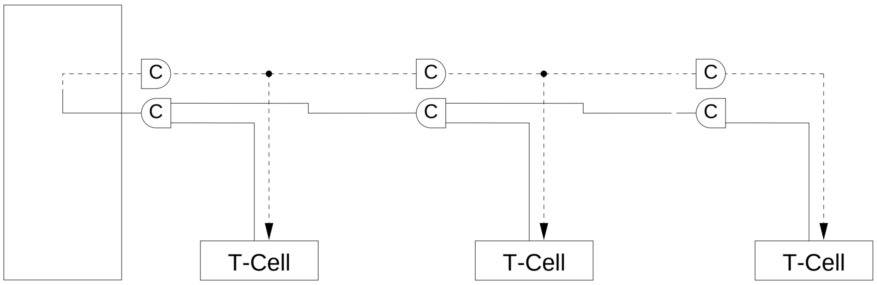
the next C-Muller gate up the synchronisation tree. The return signal on this link is the local clock signal, which is distributed to the timing cell and back down the synchronisation tree

This scheme has a relatively low cost to implement, since it only requires an extra reconfigurable C-Muller gate per cell and uses the nearest neighbour links. Also, it can implement any possible shape that can be formed using nearest neighbour connections. The disadvantage is that the distribution pattern for the local clock signal is unbalanced. In the example of Figure 11.7, the local clock can go through zero, one or two distribution nodes before reaching the timing cell and its associated data cells. Hence, the distribution of the local clocks is not uniform, potentially leading to local clock skew.

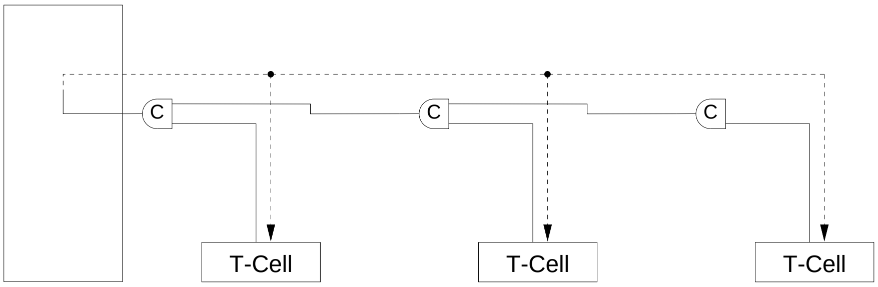
Hierarchy of Timing Cells

The problem with the previous scheme is that it forms an unbalanced tree for distributing the local clock distribution. One solution to ensure uniform clock distribution would be to always to ensure that a balanced clock distribution tree was formed. This can be achieved by using a hierarchy of reconfigurable C-Muller gates to form the synchronisation network. Figure 11.8 shows a simple hierarchy based on groups of two timing cells. Three timing cells are grouped through a tree of C-Muller gates. Each reconfigurable C-Muller gate in the tree can pass its output to a higher level in the tree. If the reconfigurable C-Muller gate is the root node in the tree, then its output is routed back down the tree as the local clock signal for the self-timed region.

A problem with this hierarchical grouping is that there is less flexibility in the grouping of timing cells to form self-timed regions. For example, the rightmost C-Muller gate in Figure 11.8 cannot form a self-timed region with cells to its right, since the C-Muller gate above it in the hierarchy is already



(a) Timing Array Routing



(b) Dedicated Local Clock Distribution Line

Figure 11.9: Grouping using Timing Array Routing

being used to form another group. The previous schemes have been more flexible in that any self-timed region could be formed regardless of the shape of other self-timed regions, as long as the timing cells were adjacent to each other. The problem can be solved by providing multiple synchronisation trees and allowing timing cells to connect to different ones. However, this incurs a greater implementation cost for the synchronisation network.

Synchronisation using Timing Array Routing Structures

A form of synchronisation structure that has already been discussed is the timing array routing structures; these structures are designed for synchronising the fan-in and fan-out of data bundles rather than synchronising timing cells that form a self-timed region. However, the timing array routing structures can potentially be adapted to synchronise groups of timing cells.

Figure 11.9(a) illustrates a timing array routing structure modified to synchronise a group of timing cells. The structure is the same as the fanning in of a data bundle, but rather than fan out to destination timing cells, the signal is returned directly as the local clock signal to the timing cells. In effect, the synchronisation can be thought of as a data bundle fanning in with a fan-out of zero, so is acknowledged immediately.

The problem in using the timing array routing is that the distribution of the local clock is not uniform. In Figure 11.9(a), the local clock signal goes through a number of intermediate reconfigurable C-Muller gates in handshaking switchboxes as it is distributed. To overcome this, a dedicated local clock distribution line can be used for a group of timing cells, so providing uniform clock distribution. This is illustrated in Figure 11.9(b).

Using the timing array routing is advantageous as it provides uniform clock distribution, and can be constructed by adapting routing structures already available in the timing array routing. The main disadvantage is that the shape of the self-timed region formed is limited by routing pattern of the local clock distribution signals. To provide larger self-timed regions requires a hierarchy similar to that discussed in the previous section. Furthermore, to provide flexibility in the shape and number of the self-timed regions formed, timing cells need to be connected to more than one local clock distribution signal, which incurs a higher implementation cost.

Variable Granularity for the Self-timed XC6200

The synchronisation network produced by modifying the timing array routing fits well into the hierarchical structure of the XC6200. Dedicated local clock distribution lines can be provided from the level-16 handshaking switchboxes to a row or column of timing cells. The synchronisation network can be implemented by either using the timing array routing already present, or by similar structures dedicated to forming self-timed regions.

The limitation of the synchronisation scheme described for the XC6200 so far is that only a maximum of four timing cells within a row or column of a level-16 block can be synchronised to form a self-timed region. To extend the row or column of blocks over the level-16 block boundaries, a hierarchy of synchronisation is required. Hence, level-64 blocks could have a local clock distribution lines running along columns and rows which would feed into the level-16 clock distribution network. The other limitation with the current synchronisation scheme is that it is limited to creating one dimensional rows or columns of timing cells. Two dimensional self-timed regions could be produced by adding another level to the hierarchy that synchronises rows or columns of timing cells together.

Delays

In the previous discussion, no mention has been made of scaling delay elements for larger self-timed regions. Larger self-timed regions can implement more complex functions, which have a longer delay than that which can be provided locally by the timing cells, even with the additional delays of synchronising the timing cells together included. Additional reconfigurable delay elements can be placed in the synchronisation network, to provide additional delay for the self-timed region. This leads to a two-level delay model: a central delay for the self-timed region and local delays provided by the timing cell. The two-level delay model can be used to provide some variability in delay to fixed reconfigurable delay architectures. Since some timing cells have a shorter local delay than others in the self-timed region, the delay of the self-timed region will vary depending on which input is last to arrive.

The discussion so far has considered fixed reconfigurable delay implementations. CSCD delay implementations pose more complex problems. In this case, the synchronisation network is acting as a completion detection network that detects when evaluation has completed. However, signals internal to the self-timed region may cross between the areas of data array monitored by different timing cells. As a result, the completion signal from a timing cell may be de-asserted if renewed activity is detected in the area of data array monitored by it. Thus, C-Muller gates with their change of state cannot be used for completion detection, instead AND gates have to be used to detect when all timing cells have completed.

A further problem with this scheme is a false completion signal being generated by a transition passing from one CSCD monitoring region to another. Potentially, the signal indicating inactivity in the original self-timed region may reach the top of the completion signal tree faster than the signal indicating activity in another region. To overcome this, the delays in passing the completion signal up the tree must be asymmetric. Signals indicating activity should be passed up the synchronisation tree quickly, whilst signals indicating inactivity, i.e. completion, should be delayed, to ensure that activity signals always travel faster up the completion tree then signals of inactivity.

Timing Cell Group Resources

The previous section considered the provision of delays specifically for groups of timing cells. This raises the question of whether other behaviour of the timing cell could be provided for groups of timing cells. For example, architectures can be envisaged that have routing resources dedicated to linking groups of timing cells together. This adds to the implementation cost of the architecture and these resources will be left unused if the groups of timing cells are not used. However, some features of the timing cell that are less frequently used but have a relatively high implementation cost, such as arbitration, could be provided only for groups of timing cells, so that the cost of implementation is reduced. If the grouping method allows a group of one timing cell to be produced, these resources can be used for individual timing cells.

11.6 Summary

This chapter has compared the self-timed XC6200 to its synchronous counterpart. The extra circuitry required was measured using a variety of different metrics, giving overheads in the range of 13% for configuration bits to 30% for two dimensional wiring overhead, with the most detailed calculation for transistor count giving an overhead of 18%. Based on the transistor overhead, this made a timing cell equivalent in complexity to about three data cells.

To evaluate the effectiveness of the timing cell implementation, flow control structures were built using data cells in the XC6200. The minimal flow control stage, one input link and one output link, implemented synchronously required three data cells. For the same cost, the timing cell allows a total of four links that could be conditionally selected, together with arbitration and a dedicated routing structure for the handshaking signals. Implementing flow control asynchronously using data cells is more expensive, requiring six data cells just for two back-to-back C-Muller gates.

The delay performance analysis showed that considerable overhead resulted from the enforced pipelining at the level of 4×4 blocks, which did not preserve the un-pipelined critical path. However, for larger examples, the fixed delay architecture showed improved performance as it can exploit the data dependent delays from selective communication. The CSCD version of the self-timed XC6200 consistently out-performed the synchronous XC6200, despite the pipelining overheads of the self-timed architecture. The simulation results were conservative in that they were based on the worst case delay figures, as typical case figures were unavailable, so an actual CSCD implementation would be expected to give further improvements in performance.

The final section of the chapter considered alterations and extensions to the self-timed XC6200 to improve performance. Moving the registers to the edge of the 4×4 block is beneficial, since only one delay element is required per timing cell. A further alteration for a CSCD architecture would be optionally to allow latches as the memory elements, to allow partial results to propagate forward. However, both these schemes involve altering the data array structure from the original synchronous XC6200 architecture.

Finally, possible ways to extend STACC architectures, including the self-timed XC6200, to allow variation in the granularity of self-timed regions were considered, to overcome the performance problems that a fixed granularity introduced. Several schemes were discussed; the scheme most suited to the XC6200 involved using a modified form of the timing array routing structures.

Chapter 12

Conclusions

12.1 Overview

This chapter summaries the conclusions of the thesis. In addition, it considers possible directions for the future development of STACC and the self-timed XC6200.

12.2 Conclusions

12.2.1 Self-Timed FPGAs and Dynamic Hardware

A key contribution of this thesis has been identifying the synergy between dynamic hardware and self-timed circuits. Dynamic hardware attempts to exploit the software like reconfigurability of FPGAs, whilst self-timed circuits free the dynamic hardware management system from the need to consider the delay properties of circuits. Thus, the routing and layout of self-timed circuits can be altered on the fly without having to reason about the effects on delays within the system. Furthermore, the flow control properties of self-timed circuits provide a natural way to regulate the flow of data between FPGA and host microprocessor in a dynamic hardware system.

In addition to the benefits for dynamically reconfigurable systems, self-timing provides benefits for FPGA based systems in general. Self-timing eases the partition of systems across several FPGAs, since the self-timed protocol can accommodate the additional off-chip delays. Also, self-timed FPGAs have greater freedom in the layout and routing of designs, since they to not have to meet a global clock constraint. Finally, self-timed circuits on FPGAs can exploit the low power consumption and average case delays of self-timed systems in general.

12.2.2 STACC: A Model for Self-Timed FPGAs

Current FPGAs are optimised for the implementation of synchronous circuits. Building self-timed circuits using these architectures is difficult; architectures can introduce hazards, cannot deal with arbitration, and can fail to meet the local delay constraints of the self-timed circuits. These problems detract from the benefits of using self-timed circuits for FPGA based systems.

To overcome these problems, the MONTAGE [57] and PGA-STC [77] architectures have been proposed. Both are derived from current synchronous FPGAs architectures by altering the logic blocks to suit the implementation of self-timed circuits. However, this disrupts the structure of the original synchronous FPGA architecture and hinders the mapping of design tools and design experience to the self-timed architectures.

In contrast, the STACC model for self-timed FPGAs does not tinker with the structure of the logic blocks. Instead, STACC replaces the global clock signal, with an array of timing cells that provide local timing control. This structure reflects the clear split between control and data path in bundled-data self-timed systems, and allows the two types of cell to be optimised for their particular function. Furthermore, the STACC model is general enough to be applied beyond FPGAs to any reconfigurable architecture, such as an array of processors with reconfigurable interconnect.

12.2.3 Self-Timed Reconfigurable Elements

This thesis has introduced several new self-timed elements for the construction of reconfigurable architectures. The Q-Merge and Select pair were shown to be suitable for building a wide range of bundled-data control structures. Furthermore, the inherent symmetry of the Q-Merge and Select pair complements the symmetry highlighted by Sutherland in his basic C-Muller gate pipelines.

However, the self-timed element central to the construction of the STACC architecture is the reconfigurable C-Muller gate (rC-Muller gate), which allows a reconfigurable synchronisation pattern to be defined between its inputs. The STACC timing cell was developed from the rC-Muller gate by allowing the pattern of synchronisation to be changed on each cycle of a self-timed region. The timing array routing was constructed by assembling multiple rC-Muller gates into handshaking crossbars. Together, these structures form the basis of the STACC timing array, which is configured to mirror the flow of data in the data array.

12.2.4 The Self-Timed XC6200

The self-timed XC6200 design has demonstrated the use of the STACC model to create a self-timed FPGA architecture with a data array compatible with the original synchronous FPGA. The self-timed XC6200 design process highlighted the importance of choosing an appropriate self-timed granularity in STACC architectures. The basic granularity in the self-timed XC6200 of one timing cell per 4×4 block of data cells largely determines the implementation overhead of the architecture. Also, the choice of a fixed granularity for the self-timed XC6200, as opposed to a variable granularity, often forces circuits to be mapped to a size of self-timed region that does not always suit their natural granularity. Furthermore, the choice of a fixed granularity influences the delay performance of circuits by requiring unnecessary pipelining.

The implementation cost for the timing array of the self-timed XC6200 was in the range of 13% to 18%, depending on the metric used, which is similar to the overhead reported for other bundled-data systems. This overhead compares favourably with the cost of implementing even the simplest of flow control circuits in the synchronous XC6200. This result demonstrated the benefits of using dedicated timing cells to implement the control path in a STACC architecture, rather than the general purpose logic blocks as used in previous architectures such as MONTAGE and PGA-STC.

The example circuits developed for run-time parameterisation on the self-timed XC6200 illustrated the benefits of self-timing for dynamic hardware systems. The examples showed how circuits of variable size could be constructed from self-timed blocks, and the resulting circuit would work without the need for delay analysis. These circuits could be split, even between different self-timed FPGAs, and still work.

The delay performance of the example circuits, when compared to equivalent synchronous circuits, showed that the fixed granularity of the current self-timed XC6200 architecture resulted in additional delays, due to unnecessary pipelining. However, the larger example circuits using the fixed reconfigurable delay elements had comparable delays to the synchronous versions, due to the exploitation of data dependent delays arising from selective communication. The self-timed XC6200 with Current Sensing Completion Detection (CSCD) delay elements consistently out-performed the synchronous architecture. Furthermore, the CSCD performance figures were conservative as they were based on worst case rather than typical case delay figures.

In conclusion, the design of a self-timed XC6200 architecture has illustrated the application of STACC to a contemporary FPGA architecture, and shown the benefits of self-timing for dynamic hardware. The implementation overhead of the self-timed XC6200 is comparatively modest, and the simulation results have shown the potential for superior performance through exploiting data dependent delays. The evaluation of the self-timed XC6200 has shown that the key limitation of the current architecture is the fixed granularity. Adopting a variable granularity architecture would give improved performance, and allow circuits to be mapped to self-timed regions that matched their natural granularity.

12.3 Self-Timed FPGA Architectures

12.3.1 Self-Timed Architectures and Granularity

A key issue highlighted in this thesis has been the size of the self-timed regions permitted within the FPGA architecture, i.e. the self-timed granularity. Many of the examples have highlighted that designs have a natural self-timed granularity, where the basic operations each form a self-timed region. Where the architecture does not support the natural granularity of the design, the basic operations have to be sub-divided resulting in unnecessary synchronisation between parts, and making designs more difficult to implement.

The various self-timed FPGA architectures proposed vary in the nature of the self-timed control provided and the variability in the granularity that is supported. MONTAGE and PGA-STC just provide the basic building blocks for building self-timed control elements, but force no higher level structure to the use of these building blocks which allows a variety of self-timed protocols to be used. Also, since no higher level self-timed control structures are provided, the granularity of self-timed region can be continuously varied. However, large self-timed regions, have to be carefully analysed to ensure correct operation.

Gao's GALSA architecture provides a fixed self-timing control structure which fixed the granularity of the self-timed region to be one-processing element. The fixed self-timed control elements and the fixed granularity is very limiting in building circuits. The STACC architecture has a reconfigurable timing array, which allows the mapping from one timing cell to control many data cells. The examples for the self-timed XC6200 case-study were limited as the mapping to 4×4 blocks is fixed. However, the reconfigurable timing array architecture allowed data flows to fan-in and fan-out in the routing which had the effect of loosely grouping self-timed regions together.

Overall the thesis has highlighted how self-timed FPGA with limited self-timed support such as MONTAGE and PGA-STC can adopt a far wider range of granularities. Self-timed architecture with more dedicated self-timed support are prone to be limit the potential granularities. However, Section 11.5 discussed how this could be extended to deal with combining timing cells to create larger self-timed regions.

12.3.2 Future Development of self-timed FPGAs

Of the asynchronous FPGA architectures currently proposed, none have been implemented as VLSI devices. The need for actual self-timed FPGA devices is linked with the wider adoption of self-timed design, as FPGAs typically are used as support devices. Self-timed systems are unlikely to replace synchronous design in the foreseeable future. However, the adoption of asynchronous design by industry in certain spheres such as low-power microprocessors for portable computing devices is a possibility. Several companies such as Intel, Sun and Phillips are currently engaged in asynchronous research. The emergence of self-timed systems in niche sectors would inevitable lead to demand for self-timed FPGAs for prototyping and as support devices.

A key issue in a future where self-timed and synchronous systems co-exist in different market sectors will be the need by industry to synthesise designs as both synchronous and asynchronous systems. An issue for future self-timed FPGA devices will be support for self-timed and synchronous design.

Of the current self-timed architectures, MONTAGE includes support for implementing synchronous systems by having two global clock signals as well as logic cells designed for building asynchronous logic. A STACC-based architecture with variable granularity could support this approach by allowing the granularity to be varied so that the whole chip was a self-timed region and thus making it effectively one synchronous region.

Regardless of the architecture finally adopted, the key issue to be addressed to allow systems to be synthesised as either a synchronous or asynchronous system will be design tools. Designs will have to be expressed in a way that is free from explicitly using a synchronous of asynchronous control paradigm, so allowing freedom of implementation in either form.

12.4 Future Work

12.4.1 Dynamic Hardware and Synchronous Flow Control

One of the main benefits of self-timed dynamic hardware is flow control, which regulates the flow of data between circuits, and between the FPGA and host system. Section 11.3.1 showed how flow control can be implemented synchronously, for lower implementation cost than attempting to directly implement self-timed circuits on current FPGAs. Thus, to gain the benefits of flow control for synchronous dynamic hardware systems, a library of synchronous flow control elements could be constructed. The library elements would perform functions similar to self-timed flow control blocks, such as those introduced by Sutherland [111].

A benefit of explicitly expressing the flow control for synchronous circuits is that it would make circuits portable between self-timed and synchronous FPGAs. In a STACC based architecture, the flow control elements would be implemented within the timing array, whilst in a synchronous FPGA, flow control would be implemented using a library of flow control blocks. However, this approach only gives the flow control benefits of self-timing; it does not give the speed-independence of self-timing, since all parts of the system must still meet the global clock constraint.

12.4.2 Other Self-Timed Reconfigurable Devices

The STACC architectural model is general enough to be applied beyond FPGAs, to other reconfigurable architectures. A good candidate for self-timed implementation would be Field Programmable Interconnect Devices (FPIDs), which are of increasing use in communication switching applications and interconnect for parallel processors. Essentially, these devices are just large crossbar switches, so timing control can be implemented using handshaking crossbars.

12.4.3 Tools

Run-time Parameterised Circuit Design Tools

The example circuits for the self-timed XC6200 were limited by the design tools available at the time. The Xilinx tools are now more mature and do offer support for creating compile-time parameterised circuits. However, run-time parameterised circuits require generation of the circuits on the fly. Thus, the output of a run-time parameterised circuit design tool should either be a program that assembles the required configuration on demand, or a parameterised configuration file that allows a standard support routine to parameterise the circuit rapidly at run-time.

Another issue for run-time parametrised circuit design tools is the layout of parameterised circuits. Run-time parameterised circuits require regular layout and routing to enable rapid assembly at run-time. As the complexity of these circuits increase, and especially as a hierarchy of parameterised circuits is formed, the fitting of the component circuits together becomes difficult. This requires tools that can assist in the regular layout of variable-sized components, together with variable-sized channels for regular routing structures to connect the component circuits.

Petri Net Tools

A C-Muller gate can be modelled by a single Petri net transition. Thus, Petri nets provide a useful formalism for modelling and reasoning about timing array configurations. Petri nets could be used to identify potential deadlock and livelock situation in designs. A significant problem for Petri net tools is the modelling of the data array behaviour, in particular, the generation of the select signals. Thus, a Petri net tool would either have to be supplied with a model for the behaviour of the data array, or would have to generate a set of constraints to met by data array, so that deadlock and livelock are avoided.

12.4.4 Architecture Development

Variable Granularity Architectures

The evaluation of the self-timed XC6200 has highlighted how fixed granularity architectures limit performance and disrupt the natural granularity of designs. Several proposals for variable granularity structures have been made but not simulated. To complete the investigation of STACC based architectures, these designs should be simulated to show the performance benefits of a variable granularity implementation, and also the additional implementation costs calculated.

CSCD Design

Simulations of the CSCD based architecture have shown the potential for performance gain over synchronous designs. An implementation using CSCD would need a large amount of low level simulations of the CSCD monitoring circuitry to ensure that it can detect a single 'on' transistor in the monitored region. Also, this thesis has proposed that CSCD has the potential to detect the current used by meta-stable states, and thus be used as an arbitration technique. Again, low level simulation is required to show that the monitoring circuitry can detect meta-stable states.

Formal Synthesis of Timing Cell

The current implementations for the timing cell were derived informally. There is no guarantee that the timing cell design will not deadlock under some conditions. Techniques for the formal synthesis of self-timed circuits are improving, and giving more efficient implementations [105]. A formally correct timing cell could be derived using these techniques, to provide stronger guarantees of the circuit's correctness. Alternatively, the timing cell implementation could be modelled using a process algebra and compared using bisimulation to a specification of the timing cell's behaviour.

Fuse based Architecture

The potential for implementing self-timed fuse based FPGAs has been highlighted in Section 8.4.1, but not developed, since the focus of the thesis was self-timing for dynamic hardware. Self-timed fuse based FPGAs using the distributed C-Muller gate could be developed further. There is also potential for improved implementations using a single synchronisation wire rather than two, by adopting an approach similar to single-track handshaking circuits [115].

VLSI Implementation

A final proof of the viability of a STACC based architecture would require an actual implementation of a chip. Though the time and resources available for this work have precluded the low level design of a VLSI chip, STACC and the self-timed XC6200 architecture have matured sufficiently that low level simulation and implementation would be a natural next step in the development of the architecture.

12.5 Summary

In conclusion, this work has shown the viability of creating self-timed FPGAs based on the STACC model. The self-timed XC6200 has demonstrated that the implementation overheads of self-timed architectures are reasonable and that the potential exists to out-perform current synchronous FPGAs through the exploitation of data dependent delays. The example run-time parameterised circuits for the self-timed XC6200 have illustrated the benefits that a self-timed FPGA brings to the implementation of dynamic hardware systems.

Appendix A

Finite Fields

A.1 Introduction

This appendix describes some results from finite field theory which are useful for a detailed understanding of the example circuits in Chapter 10. Section A.2 details the basics of finite field theory including operations, extension fields, the normal basis and the conventional basis. Section A.3 gives an overview of applications using finite field operations, in particular Reed-Solomon encoding. For a full review of finite field theory and Reed-Solomon encoding refer to Pretzel [97].

A.2 Finite Fields

A *field* is a mathematical structure with two defined operations: addition and multiplication. Addition is commutative and associative, and has a identity element, zero. Each member of a field has an associated negative such that their sum is zero. Multiplication in a field is commutative, associative, distributive across addition and has an identity element, one. Each member of a field except zero, has an inverse element, such that their product is one.

Several infinite sets of numbers, such as the rational, real and complex numbers, meet the field axioms described above. Finite sets of numbers can also form fields, if the addition and multiplication operations are suitably defined. One such construction is the *Galois Field*. A Galois field $GF(p)$ is formed by performing addition and multiplication modulo a prime number p.

$GF(2)$ is particularly useful for digital applications, since it has only two members in the field, corresponding to the binary digits zero and one. Addition in $GF(2)$ can be implemented using an XOR gate, whilst multiplication can be implemented using an AND gate.

Extension Galois fields $GF(p^k)$ can be created, where p is a prime number and k is a natural number. The operations in extension fields can be defined using operations on polynomials modulo an irreducible polynomial of degree k, where the coefficients of the polynomial are members of the base field $GF(p)$.

A.2.1 The Conventional Basis and Normal Basis

Since, operations in extension fields can be represented by operation on polynomials, members of extension fields can simply written using the coefficients of the polynomial. So for $GF(2^4)$, x^3+1 is represented as 1001, or encoding as a decimal, the element is 9. Thus, each element A of a field $GF(2^k)$ is represented by the coefficients a_i, in the following equation:

$$A(x) = \sum_{i=0}^{k-1} a_i x^i \tag{A.1}$$

The set of terms x^n used to the represent the polynomial is known as the *basis*, and the basis used above where n is in the range from $0...k-1$ is known as the *conventional basis*.

It is possible to represents elements of $GF(2^k)$ using a different basis, as long as the basis chosen is such that it each elements of $GF(2^k)$ has a different value. A useful basis is to represent elements of $GF(2^k)$ using terms x^n where n is taken from the set $\{2^0, 2^1, ..., 2^{k-1}\}$. This is known as the *normal basis*. Thus,

$$A(x) = \sum_{i=0}^{k-1} a_i x^{2^i} \tag{A.2}$$

represents a polynomial $A(x)$ using the normal basis. A particular advantage of the normal basis in $GF(2)$ is that squaring is simply a cyclic shift of the coefficients. This arises since,

$$A^2 = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} a_i a_j x^{2^i} x^{2^j} \tag{A.3}$$

However, when the sum is expanded, for each term $a_v a_w x^{2^v} x^{2^w}$ where $i = v$ and $j = w$ when $v \neq w$, there is the term with the same value for $i = w$ and $j = v$. These terms cancel since $a + a = 0$ over $GF(2)$, thus the only terms left are where $v = w$. Hence,

$$(A(x))^2 = \sum_{i=0}^{k-1} {a_i}^2 x^{2^{i+1}} \tag{A.4}$$

But ${a_i}^2 = a_i$ since $a^2 = a$ over $GF(2)$, so

$$(A(x))^2 = \sum_{i=0}^{k-1} a_i x^{2^{i+1}} \tag{A.5}$$

Thus shifting the index i,

$$(A(x))^2 = \sum_{i=1}^{k} a_{i-1} x^{2^i} \tag{A.6}$$

Which is a shift of the coefficients in $A(x)$. The shift is cyclic, since $x^{2^k} = x$ for $GF(2^k)$ due to Fermat's theorem (see [97]).

The fact that squaring is a cyclic shift in the normal basis can be used to simplify the design of multipliers over $GF(2^k)$ [118]. The multiplier can be built using bit slices, where each bit slice produces one bit of the result. Each bit slice is identical except for cyclic shifts of the inputs since:

$$A.B = \sqrt{A^2.B^2} =^4 \sqrt{A^4.B^4} = ... \tag{A.7}$$

In the following sections, the equations use the conventional basis for convenience in expressing equations. In general, the equations can be converted to use the normal basis by changing the terms x^n to x^{2^n}.

A.2.2 Multiplication

This section describes the definition of multiplication of two numbers A and B in $GF(2^k)$ in terms of operations on polynomials $A(x)$ and $B(x)$ over $GF(2)$ modulo a irreducible polynomial $P(x)$. This result is used in the design of the multipliers in Sections 10.3.3 and 10.3.4.

Multiplication of A and B can be expressed as:

$$A.B = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_i.b_j.x^{i+j} \mod P(x) \tag{A.8}$$

Where a_i and b_j are the coefficients of the polynomial $A(x)$ and $B(x)$ respectively. Let $D_{ij} = x^{i+j} \mod P(x)$, so Equation A.8 can be expressed as:

$$A.B = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} \sum_{h=0}^{k-1} a_i.b_j. \left. D_{ij} \right|_h .x^h \tag{A.9}$$

where $|_h$ gives the hth coefficient of a polynomial. Rearranging

$$A.B = \sum_{h=0}^{k-1} \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_i.b_j.\ D_{ij}|_h .x^h$$

$$A.B = \sum_{h=0}^{k-1} x^h.\sum_{i=0}^{k-1} a_i.\sum_{j=0}^{k-1} b_j.\ D_{ij}|_h \tag{A.10}$$

The above equation shows that each coefficient of x^h in the result polynomial can be evaluated independently, so a bit-sliced approach can be adopted. Finally, let F_{hi} be a polynomial such that:

$$F_{hi}|_j = D_{ij}|_h \tag{A.11}$$

Hence,

$$A.B = \sum_{h=0}^{k-1} x^h.\sum_{i=0}^{k-1} a_i.\sum_{j=0}^{k-1} b_j.\ F_{hi}|_j \tag{A.12}$$

which gives the form of the circuit used in Section 10.3.3

A.2.3 Division

This section presents some results used in Section 10.3.8. Division is a more complex operation to implement than multiplication in $GF(2^k)$. However, it can be defined in terms of multiplication by using Fermat's rule [97] that, for a field with n elements $x^{n-1} = 1$. Therefore for $GF(2^k)$, $x^{2^k-1} = 1$. Hence, to divide y by x:

$$y/x = yx^{-1} = yx^{2^k-2} = yx^{2^1}x^{2^2}\ldots x^{2^{k-1}} \tag{A.13}$$

Thus, division can be expressed as a series of $k-1$ products of the dividend and the squares of the divisor. This form is particularly suited to implementation using the normalised polynomial representation, where squaring is simply implemented by a cyclic shift of the terms in the polynomial.

A.3 Error Detection and Correction Applications

A number of error detection and correction codes are based on finite field operations. For example, Reed-Solomon codes represent strings of bits as symbols in $GF(2^k)$. Such codes are useful for burst error correction, since the correction of a single symbol in $GF(2^k)$ can correct an error burst of up to k bits in the bit stream. Reed-Solomon codewords are generally expressed as a multiple of a generator polynomial over $GF(2^k)$. The length of the generator polynomial determines how many errors can be corrected.

In error correction codes, the locations of errors are unknown. In erasure codes, the location of errors (erasures) are known. For example, erasure codes are used to deal with lost packets in computer networking and corrupted discs in RAID (Redundant Array of Inexpensive Discs). Given the additional information of the error locations, more erasures can be corrected than errors corrected by an equivalent error correction code. Rizzo [102] discusses erasure codes based on finite field operations over $GF(2^k)$ for use in networking applications.

A.3.1 Reed-Solomon Error Correction

Reed Solomon codes $RS(k,t)$ have two parameters, the base field representation k, and the number of error corrected t. Reed-Solomon codewords are represented as polynomials of degree $2^k - 1$ with coefficients in the field $GF(2^k)$. $RS(k,t)$ has a minimum distance between code words of $2t+1$, so is capable of detecting $2t$ errors, and correction t errors.

Since each coefficient corrected by a Reed-Solomon code is a number in $GF(2^k)$ and can be represented by k bits, correcting a single error in a Reed-Solomon code can correct a burst error of up to k bits. Combined with other techniques such as interleaving words of a message, Reed-Solomon coding provides very powerful burst error correction.

Encoding and Error Detection

Reed-Solomon codewords can be represented as multiples of a *generator polynomial*, $g(x)$ of degree $2t$. The generator polynomial is defined as

$$g(x) = (x-\alpha)(x-\alpha^2)\dots(x-\alpha^{2t}) \tag{A.14}$$

where α is a primitive element of $GF(2^k)$. The choice of a primitive element ensures that each factor $x - \alpha^n$ is distinct.

The standard encoding technique is to encode the data to be sent $m(x)$ by multiplying it by 2^{2t} and then finding the remainder when divided by $g(x)$. This is subtracted from $m(x).2^{2t}$ to give a codeword which is a multiple of $g(x)$. To detect errors in transmission, the received message can be divided by the generator polynomial. If the remainder is zero then a valid codeword has been received otherwise there has been an error.

Error Processing

A brief summary of the operations involved in Reed-Solomon error processing is given here (see Pretzel [97] for more details). The first stage of error processing is to generate the *syndromes* of $RS(k,t)$. The syndromes, S_i, are defined as:

$$S_i = d(\alpha^i) \quad \text{for } i = 1, \ldots, 2t \tag{A.15}$$

where α is a primitive element of $GF(2^k)$ and $d(x)$ is the received message polynomial. Since the generator polynomial is the product of the terms $(x - \alpha^i)$ then, for a valid codeword, all the syndromes will equal zero.

The next step of error processing involves finding the error locator polynomial, $l(x)$ in the following equation:

$$w(x) = l(x)s(x) + u(x)x^{2t} \tag{A.16}$$

where $w(x)$ is the error evaluator, $u(x)$ is the error co-evaluator, and $s(x)$ is the syndrome polynomial whose coefficients are the syndromes S_i. The error locator polynomial $l(x)$ is found by performing Euclid's algorithm on $s(x)$ and x^{2t}. The equation $l(x) = 0$ is then solved to find the positions of the errors. The error values are then found by evaluating the error evaluator polynomial, $w(x)$, for values generated by solving $l(x) = 0$.

All these steps in the error processing require operations to be performed on polynomials over $GF(2^k)$. These operations include the basic operations of addition, multiplication and division in $GF(2^k)$, and the operations to evaluate a polynomial in $GF(2^k)$ at a fixed value for syndrome calculation and non-fixed value for solving $l(x) = 0$, polynomial division by a constant divisor for generating the codes and polynomial division with a non-constant divisor for Euclid's algorithm. These operations are presented in Chapter 10.

Appendix B

Self-timed XC6200 Evaluation Data

Name	Bits
X1,X2,X3	9
Y2,Y3	4
RP	1
CS	1
Magic	1
Total	16

(a) Data Cell

Name	Bits	
	per link	*per cell*
RDZ	2	8
DIR	1	4
to Reg. Delay		3
from Reg. Delay		2
RESET		1
Arbitration	2	8
Select Routing	2	8
Total		34

(b) Timing Cell

Table B.1: Data and Timing Cell Configuration Bit Usage

Name		**Bits**		
		per direction	*per 4x4 block*	
Local		4	16	[1]
Flyover	level 4	16	64	
Clock			16	
Total			96	

[1] additional to local routing bits

(a) Boundary Multiplexors

Name		**Bits**	
		per switchbox	*per 4x4 block*
Handshaking	Local	5	10
	level 4	2×4	16
Clock			-16
Total			10

(b) Handshaking Switchbox

Table B.2: Routing Configuration Bit Usage

Signal Name		Wires	
Local	Neighbour	8	
	Magic	8	[2]
	`CLK`,`CLR`	1	[1]
Flyover	Level 4	8	
	Level 16	8	
	Level 64	8	
Global	`G1`,`G2`,`GCLK`,`GCLR`	2	[1]
Total		43	

[1] averaged over two dimensions
[2] average density in 4x4 block

(a) Synchronous XC6200

Signal Name		Wires	
Handshaking	local	2	
	level 4	4	
	level 16	2	
Global	`G1`,`G2`,`GCLK`,`GCLR`	-2	[1]
Total		6	

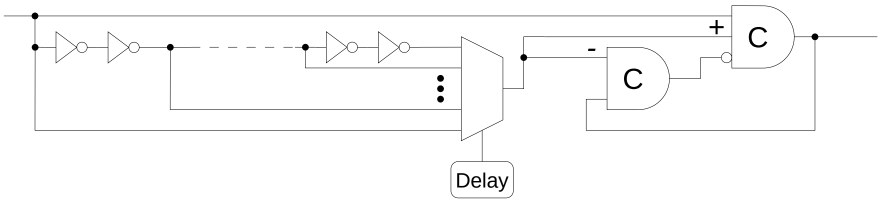
[1] averaged over two dimensions

(b) Self-Timed

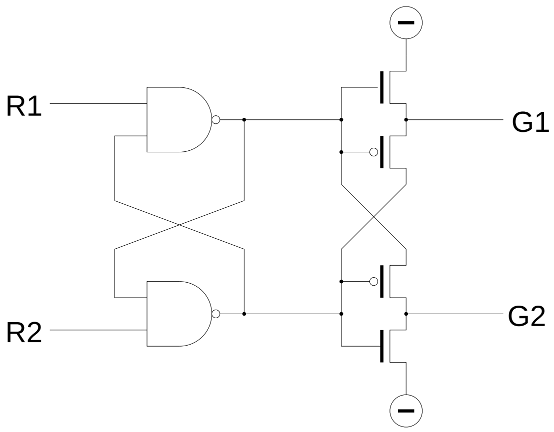
Table B.3: Wires per 4×4 Block

Part	Implementation
C-Muller Gate:	Weak feedback implementation, as shown in Figure 6.2. Includes reset input.
Multiplexors:	Tree of transmission gates.
D-type Registers:	based on the circuit of Figure 5.57(a) in Weste and Eshraghian [121].
SRAM Cell:	6-Transistor SRAM Cell is used in the XC6200 [123].
Mutual Exclusion:	See Table B.4(c) for implementation.
Delay Element:	See Table B.4(b) for implementation.

(a) Implementations used for Components



(b) Delay Element



(c) Mutual Exclusion Element

Table B.4: Implementations used for Transistor Counts

Part	Gate	Qty	Transistors	
			per gate	*total*
`X1,X2,X3`	8:1 Mux	3	28	84
`Y2,Y3`	4:1 Mux	2	12	24
`Y1,RP,CS,Magic`	2:1 Mux	4	4	16
	Inverters	2	2	4
	D-type Register	1	32	32
Configuration Bits	SRAM cell	16	6	96
Total				256

(a) Data Cell Transistor Count

Part	Gate	Qty	Transistors	
			per gate	*total*
Synchronisation	5-input C-Muller	2	15	30
	2:1 Mux	8	4	32
	SRAM Cells	12	6	72
	Subtotal			134
Select	D-type	4	32	128
	2:1 Mux	12	4	48
	4:1 Mux	4	12	48
	Subtotal			224
Select Routing	4:1 Mux	4	12	48
	SRAM Cell	8	6	48
	Subtotal			96
To Reg Delay	Inverters	14	2	28
	8:1 Mux	1	28	28
	Asym. C-Muller	1	8	8
		1	10	10
	SRAM Cell	3	6	18
	Subtotal			92
From Reg Delay	Inverters	6	2	12
	4:1 Mux	1	12	12
	Asym. C-Muller	1	8	8
		1	10	10
	SRAM Cell	2	6	12
	Subtotal			54
Arbitration	Mut-Ex	1	12	12
	4-input NAND	2	8	16
	2:1 Mux	8	4	32
	SRAM Cell	8	6	48
	Subtotal			108
`RESET`	SRAM Cell	1	6	6
Total				714

(b) Timing Cell Transistor Count

Table B.5: Data Cell and Timing Cell Transistor Count

Part	Gate	Qty	Transistors per gate	Transistors total	
Local	2:1 Mux	16	4	64	[1]
	4:1 Mux	16	12	192	[1]
`S4,W4,E4`	10:1 Mux	12	36	432	
`N4`	12:1 Mux	4	44	176	
`CLK`	4:1 Mux	8	12	96	
Configuration	SRAM Cell	96	6	576	
Total				1446	

[1] additional to local 4:1 Mux

(a) Data Array

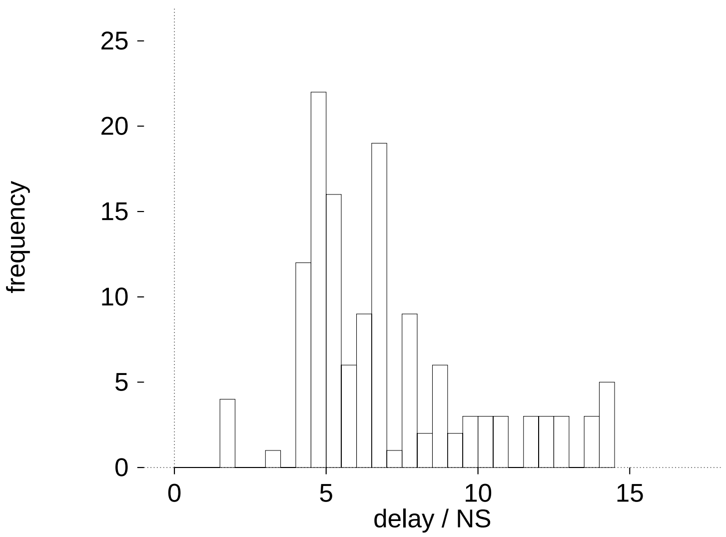
Part	Gate	Qty	Transistors per gate	Transistors total
Synchronisation	4-input rC-Muller	8	28	224
	5-input rC-Muller	4	34	136
Configuration	SRAM Cell	10	6	60
`CLK`				-96
Total				324

(b) Timing Array

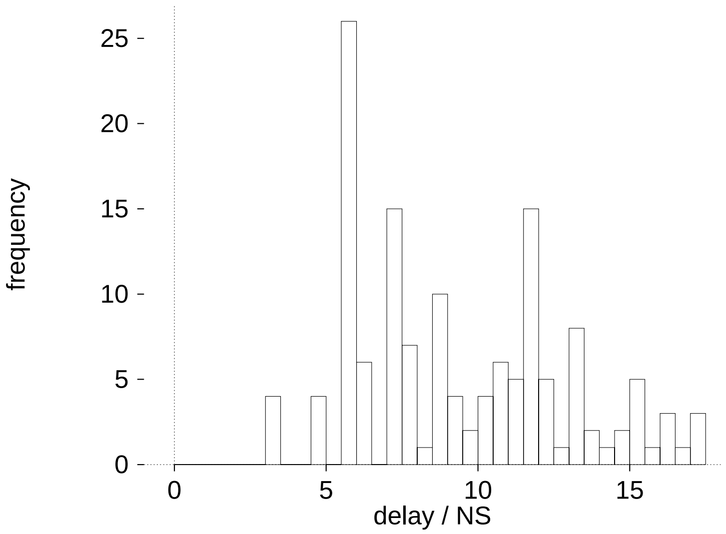
Table B.6: Boundary Routing Transistor Counts

Symbol	Description	Max Delay /ns
T_{ILO1}	X1 change to Function Out	2
T_{ILO23}	X2 change to Function Out	3
T_{FN}	Function Out to Neighbour	1
T_{NN}	Route Neighbour In to Neighbour Out	1.5
T_{Magic}	Route X2/X3 to Magic Out	2.5
T_{L4}	Level-4 Flyover	2
T_{L16}	Level-16 Flyover	2.5
T_{L64}	Level-64 Flyover	5

Table B.7: XC6200 Delays



(a) Xilinx Tools



(b) VHDL Simulator

Figure B.1: Example Delay Profiles for Fixed Polynomial Division Circuit

Bibliography

[1] Product Focus: Field Programmable Logic, Benchmarking FPGAs. *Electronic Engineering Times*, 65(799):53–77, July 1993.

[2] Actel Corp., Sunnyvale, California. *ACT Family FPGA Data Book*, 1991.

[3] Algotronix Ltd., The King's Buildings, TTC , Edinburgh EH9 3JL. *CAL1024 Datasheet*, 1991.

[4] Altera Corp., San Jose, California. *Data Book*, 1995.

[5] Aptix Inc., San Jose, California. *MP3 Data Sheet*, 1996.

[6] J. M. Arnold. The Splash 2 Software Enviroment. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[7] D. K. Arvind and R. D. Mullins. Instruction Compounding in MAP Architectures. In *Proceedings of the First U.K. Asynchronous Forum*, pages 26–33, 1996.

[8] D. K. Arvind, R. D. Mullins, and V. E. F. Rebello. Micronets: A Model for Decentralising Control in Asynchronous Processor Architectures. In M. B. Josephs, editor, *The 2nd Working Conference on Asynchronous Design Methodologies*, pages 190–199, London, UK, May 1995. IEEE Computer Society Press.

[9] AT & T. *AT & T Optimised Reconfigurable Cell Array (ORCA)*, 1995.

[10] Atmel. *AT6000 Field-Programmable Gate Array Data Sheet*, 1996.

[11] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[12] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to Programmable Active Memories. Technical report, DEC Paris Research Laboratory, June 1989.

[13] M. Boloski, A. DeHon, and T. F. Knight. Unifying FPGAs and SIMD Arrays. In *FPGA94: 2nd International ACM/SIGDA Workshop on FPGAs*, 1994.

[14] D. S. Bormann and P. Y. K. Cheung. Designing Globally Asynchronous Locally Synchronous Circuits Using VHDL. In *1st U.K. Asynchronous Forum*, pages 38–41, December 1996.

[15] D. S. Bormann and P. Y. K. Cheung. Asynchronous Wrapper for Hetrogeneous Systems. In *submitted to ICCD'97*, 1997.

[16] G. Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *6th International Workshop on Field Programmable Logic and Applications*, volume 1142 of *Lecture Notes in Computer Science*, pages 327–336, 1996.

[17] G. Brebner. The Swappable Logic Unit: a Paradigm for Virtual Hardware. In *FCCM97: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1997.

[18] G. Brebner and J. Gray. Use of Reconfigurability in Variable Length Code Detection at Video Rates. In *5th International Workshop on Field Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 429–438, 1995.

[19] E. Brunvand. Using FPGAs to Prototype a Self-Timed Computer. In *Workshop on Field Programmable Logic and Applications*, pages 192–198, 1992.

[20] E. Brunvand. Using FPGAs to Implement Self-Timed Systems. *Journal of VLSI Signal Processing*, 6(2):173–190, August 1993.

[21] S. Casselman, M. Thronburg, and J. Schewel. Creation of Hardware Objects in a Reconfgiurbale Computer. In *5th International Workshop on Field Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, 1995.

[22] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[23] S. Churcher, T. Kean, and B. Wilkie. The XC6200 FastMap Processor Interface. In *5th International Workshop on Field Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 36–43, 1995.

[24] Cypress Semiconductor Corp., San Jose, California. *Programmable Logic Data Book*, 1994/95.

[25] P. Day and J. V. Woods. Investigation into Micropipeline Latch Design Styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[26] M. E. Dean, D. L. Dill, and M. Horowitz. Self-Timed Logic Using Current-Sensing Completion Detection (CSCD). In *Proc. International Conf. Computer Design (ICCD)*, pages 187–191. IEEE Computer Society Press, October 1991.

[27] Mark E. Dean. *STRiP: A Self-Timed RISC Processor Architecture*. PhD thesis, Stanford University, 1992.

[28] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *FCCM94: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[29] D. L. Dill. Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 51–65. MIT Press, 1988.

[30] D. W. Dobberpuhl et al. A 200MHz 64-bit Dual Issue CMOS Processor. *IEEE Journal of Solid-State Circuits*, 11(27):1555–1567, 1992.

[31] J. G. Eldredge and B. L. Hutchings. RRANN: A Hardware Implementation of the Backpropagation Algorithm using Reconfigurable FPGAs. In *IEEE International Conference on Neural Networks*, 1994.

[32] J. G. Eldredge and B. L. Hutchings. RRANN: The Run-Time Reconfiguration Artificial Neural Network. In *IEEE Custom Integrated Circuits Conference*, pages 77–80, 1994.

[33] B. Felton and N. Hastie. Configuration Data Verification and the Integrity Checking of SRAM based FPGAs. In *FPGAs: International Workshop on Field Programmable Logic and Applications*, chapter 2.6. Abingdon EE&CS Books, 1991.

[34] P. C. French and R. Taylor. A Self-Reconfiguring Processor. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[35] S. B. Furber. Breaking Step: The Return of Asynchronous Logic. *IEE Review*, 39(4):159–162, July 1993.

[36] S. B. Furber. Lessons from AMULET1: Towards AMULET2. In I. E. Sutherland and S. B. Furber, editors, *Sun Annual Lecture in Computer Science at the University of Manchester*, September 1994.

[37] S. B. Furber and P. Day. Four-Phase Micropipeline Latch Control Circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[38] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and S. Temple. AMULET2e. In C. Muller-Schloer, F. Geerinckx, B. Stanford-Smith, and R. van Riet, editors, *Embedded Microprocessor Systems*, September 1996. Proceedings of EMSYS'96 - OMI Sixth Annual Conference.

[39] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A Micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

[40] S. B. Furber and J. Liu. Dynamic Logic in Four-Phase Micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[41] M. Gamble, B. Rahardjo, and R. D. McLeod. Reconfigurable FPGA Micropipelines. Technical report, U. of Manitoba, 1994.

[42] B. Gao. *A Globally Asynchronous Locally Synchronous Configurable Array Architecture for Algorithm Embeddings*. PhD thesis, University of Edinburgh, December 1996.

[43] B. Gao and D. J. Rees. Communicating synchronous logic modules. In *Proccedings of the 21st EuroMicro Conference*, pages 708–714, September 1995.

[44] J. D. Garside. A CMOS VLSI Implementation of an Asynchronous ALU. In *Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, 1993.

[45] GEC Plessey Semiconductors. *ERA 60100 Reconfigurable Array Data Sheet*, 1991.

[46] G. Gopalakrishnan. Some Unusual Micropipeline Circuits. Technical Report UUCS-93-015, Dept. of Computer Science, Univ. of Utah, July 1993.

[47] G. Gopalakrishnan and P. Jain. Some Recent Asynchronous System Design Methodologies. Technical report, U.of.Utah, 1990.

[48] G. Gopalakrishnan and L. Josephson. Towards Amalgamating the Synchronous and Asynchronous Styles. In *Tau'93: International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1993.

[49] P. Graham and B. Nelson. A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash 2. In *5th International Workshop on Field Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 352–361, 1995.

[50] E. Grass and S. Jones. Asynchronous Circuits Based On Multiple Localised Current-Sensing Completion Detection. In *Second Working Conference on Asynchronous Design Methodologies*, pages 170–177, May 1995.

[51] E. Grass, R. C. S. Morling, and I. Kale. Activity Monitoring Completion Detection (AMCD): A New Single Rail Approach to Achieve Self-Timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[52] M. R. Greenstreet and K. Steiglitz. Bubbles can Make Self-Timed Pipelines Fast. *Journal of VLSI Signal Processing*, 2(3):139–148, November 1990.

[53] S. Guccione and M. J. Gonzalez. Classification and Performance of Reconfigurable Architectures. In *5th International Workshop on Field Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 439–448, 1995.

[54] S. Hauck. Asynchronous Design Methodologies: An Overview. Technical Report TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.

[55] S. Hauck, G. Borriello, S. Burns, and C. Ebeling. MONTAGE: An FPGA for Synchronous and Asynchronous Circuits. In *Workshop on Field Programmable Logic and Applications*, 1992.

[56] S. Hauck, G. Borriello, C. Ebeling, D. Song, and E. A. Walkup. TRIPTYCH: A New FPGA Architecture. In *FPGAs: International Workshop on Field Programmable Logic and Applications*, pages 75–90. Abingdon EE&CS Books, 1991.

[57] S. Hauck, S. Burns, G. Borriello, and C. Ebeling. A FPGA for Implementing Asynchronous Circuits. *IEEE Design and Test of Computers*, 11 (3):60–69, 1994.

[58] W. D. Hillis. *The Connection Machine*. MIT Press, 1985.

[59] R. W. Hockney and C. R. Jesshope. *Parallel Computers - Architecture, Programming and Algorithms*. Adam Hilger Ltd., 1984.

[60] I-Cube Inc., Santa Clara, California. *IQ Family Data Sheet*, March 1996.

[61] I-Cube Inc., Santa Clara, California. *PSX Family Data Sheet*, February 1996.

[62] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1988.

[63] C. Iseli and E. Sanchez. Spyder: A Reconfigurable VLIW Processor using FPGAs. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[64] T. Isshiki and W. M. Dai. High-Performance Datapath Implementation on Field-Programmable Multi-Chip Module (FPMCM). In *4th International Workshop on Field Programmable Logic and Applications*, pages 373–384, 1994.

[65] T. A. Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation*. PhD thesis, University of Edinburgh, 1989.

[66] D. Kearney and N. W. Bergmann. Performance Evaluation of Asynchronous Logic Pipelines with Data Dependant Processing Delays. In *Asynchronous Design Methodologies*, pages 4–13. IEEE Computer Society Press, May 1995.

[67] J. Kessels. VLSI Programming of a Low-Power Asynchronous Reed-Solomon Decoder for the DCC Player. In *Asynchronous Design Methodologies*, pages 44–52. Philips Research Laboratories, 1995.

[68] A. Klindworth. FPLD-Implementation of Computations over Finite Fields $GF(2^m)$ with Applications to Error Control Coding. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 261–271. Springer-Verlag, 1995.

[69] D. K. Y. Kwok. An Investigation of Virtual Hardware using FPGA Technology. Technical report, University of Edinburgh, May 1996.

[70] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the State Assignment Problem for Signal Transition Graphs. In *Proc. ACM/IEEE Design Automation Conference*, pages 568–572. IEEE Computer Society Press, June 1992.

[71] D. Lewin. *Logical Design of Switching Circuits*. Thomas Nelson and Sons Ltd., 2 edition, 1974.

[72] X. Ling and H. Amano. WASMII: A Data Driven Computer on a Virtual Hardware. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[73] S-L Lu. Implementation of Micropipelines in Enable/Disable CMOS Differential Logic. *IEEE Transactions on VLSI Systems*, 3(2):338–341, June 1995.

[74] W. Luk, S. Guo, N.Shirazi, and N.Zhuang. A Framework for Developing Parameterised FPGA Libraries. In *6th International Workshop on Field Programmable Logic and Applications*, volume 1142 of *Lecture Notes in Computer Science*, pages 24–33, 1996.

[75] P. Lysaght, H. Dick, G. McGregor, D. McConnell, and J. Stockwood. Prototyping Enviroment for Dynamically Reconfgiurable Logic. In *5th International Workshop on Field Programmable Logic and Applications*, pages 409–418, 1995.

[76] P. Lysaght, J. Stockwood, J. Law, and D. Girma. Artificial Neural Network Implementation on a Fine-Grained FPGA. In *4th International Workshop on Field Programmable Logic and Applications*, 1994.

[77] K. Maheswaran. Implementing Self-Timed Circuits in Field Programmable Gate Arrays. Master's thesis, U.C.Davis, 1995.

[78] K. Maheswaran and V. Akella. Hazard-free Implementation of the Self-Timed Cell set for the Xilinx 4000 Series FPGA. Technical report, U.C.Davis, 1994.

[79] S. Mark and V. Jean. Fast Implementations of RSA Cryptography. In *11th IEEE Symposium on Computer Arithmetic*, 1993.

[80] A. J. Martin. Formal Program Transformations for VLSI Circuit Synthesis. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, UT Year of Programming Series, pages 59–80. Addison-Wesley, 1989.

[81] A. J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In W. J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[82] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The Design of an Asynchronous Microprocessor. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[83] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The First Asynchronous Microprocessor: The Test Results. *Computer Architecture News*, 17(4):95–110, June 1989.

[84] A. J. McAuley. Four State Asynchronous Architectures. *IEEE Transactions on Computers*, 41(2):129–142, February 1992.

[85] M.Edwards and J. Forrest. Hardware/Software Co-Design Project. Technical report, UMIST, 1995.

[86] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt. Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[87] Meng, T. H.-Y. and Brodersen, R. W. and Messerschmitt, D. G. A Clock-Free Chip Set for High-Sampling Rate Adaptive Filters. *Journal of VLSI Signal Processing*, 1(4):345–365, 1990.

[88] G. Milne, P. Cockshott, G. McCaskill, and P. Barrie. Realising Massively Concurrent Systems on the SPACE Machine. Technical Report HDV-29-93, U. of Strathclyde, 1993.

[89] G. J. Milne. Realising massively Concurrent Systems on the SPACE Machine. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[90] S. Monaghan. A Gate-Level Reconfigurable Monte Carlo Processor. *Journal of VLSI Signal Processing*, 6(2):139–154, August 1993.

[91] Z. Navabi. *VHDL Analysis and Modelling of Digital Systems*. McGraw-Hill, 1993.

[92] J. Oldfield and C. Kappler. Implementing Self-timed Systems: Comparision of Configurable Logic Arrays with Full Custom Circuits. In *FPGAs: International Workshop on Field Programmable Logic and Applications*, chapter 6.3. Abingdon EE&CS Books, 1991.

[93] I. Page. The HARP Reconfgiurable Computing System. Technical report, Oxford University Hardware Compilation Group, October 1994.

[94] I. Page. Reconfigurable Processor Architectures. *submitted for special issue of Microprocessors and Microsystems on hardware software co-design*, 1996.

[95] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, June 1994.

[96] R. E. Payne. Self-Timed FPGA Systems. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 1995.

[97] O. Pretzel. *Error-Correcting Codes and Finite Fields*. Oxford Applied Mathematics and Computing Science Series. Oxford University Press, 1992.

[98] D. Pryor, M. Thistle, and N. Shirazi. Text Searching on Splash 2. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[99] F. Raimbult et al. Fine Grain Parallelism on a MIMD machine using FPGAs. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[100] V. E. F. Rebello. *On the Distribution of Control in Asynchronous Processor Architectures*. PhD thesis, Department of Computer Science, University of Edinburgh, UK., 1996.

[101] C. F. Reese. The CM-2X a Hybrid CM-2/Xilinx Prototype. In *FCCM93: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[102] L. Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *Computer Communication Review*, April 1997.

[103] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T. Fang. Q-Modules: Internally Clocked Delay-Insensitive Modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, September 1988.

[104] C. L. Seitz. *System Timing*, chapter 7. Addison-Wesley, Mead and Conway Introduction to VLSI Systems edition, 1980.

[105] A. Semenov and A. Yakovlev. Partial Order Approach to Design, Verification and Synthesis of Asynchronous Circuits. In D. K. Arvind and S. B. Furber, editors, *1st U.K. Asynchronous Forum*, pages 47–50, 1996.

[106] P. Shaw and G. Milne. A Highly Parallel FPGA-Based Machine and its Formal Verification. Technical Report HDV-28-93, U. of Strathclyde, 1993.

[107] S. Singh, J. Hogg, and D. McAuley. Expressing Dynamic Reconfiguration by Partial Evaluation. In *FCCM96: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[108] G. Snider, P. Kuekes, W. B. Culbertson, R. J. Carter, A. S. Berger, and R. Amerson. The Teramac Configurable Compute Engine. In *5th International Workshop on Field Programmable Logic and Applications*, pages 44–53, 1995.

[109] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. Counterflow Pipeline Architecture. Technical report, Sun Microsystems Laboratories, April 1994.

[110] A. Stansfield and I. Page. The Design of a New FPGA Architecture. In *5th International Workshop on Field Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 1–14, 1995.

[111] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–38, 1989.

[112] J. A. Tierno, A. J. Martin, D. Borkovic, and T. K. Lee. An Asynchronous Microprocessor in Gallium Arsenide. Technical report, California Institute of Technology, 1993.

[113] R Per Torstein. A System for Asynchronous High-speed Chip to Chip Communication. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–10, 1996.

[114] S. H. Unger. *Asynchronous Sequential Switching Circuits*. John Wiley and Sons, Inc., 1969.

[115] K. van Berkel and A Bink. Single-Track Handshake Signalling with Application to Micropipelines and Handshake Circuits. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–133, 1996.

[116] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij, and R. van de Wiel. A Single-Rail Re-implementation of a DCC Error Detector Using a Generic Standard-Cell Library. In *Asynchronous Design Methodologies*, pages 72–79. IEEE Computer Society Press, May 1995.

[117] T. Verhoeff. Delay-Insensitive Codes—An Overview. *Distributed Computing*, 3(1):1–8, 1988.

[118] C. C. Wang, T. K. Truong, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed. VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$. *IEEE Transactions on Computers*, C-34(8):709–717, August 1985.

[119] S. Weber, B. Bloom, and G. Brown. Compiling Joy to Silicon. In T. Knight and J. Savage, editors, *Proceedings of Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, pages 79–98. MIT Press, March 1992.

[120] U. Weiser. Future Directions in Microprocessor Design. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996.

[121] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Desgin - a Systems Perspective*. Addison-Wesley, 2 edition, 1993.

[122] M. J. Writhlin and B. L. Hutchings. A Dynamic Instruction Set Computer. In *FCCM95: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[123] Xilinx. *XC6200 Datasheet*, 1996.

[124] Xilinx Inc., San Jose, California. *The Programmable Logic Data Book*, 1994.

[125] C. Ykman-Couvreur and B. Lin. Optimised State Assignment for Asynchronous Circuit Synthesis. In *Asynchronous Design Methodologies*, pages 118–127. IEEE Computer Society Press, May 1995.

[126] K. Y. Yun, P. A. Beerel, and J. Arceo. High-Performance Asynchronous Pipeline Circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.