CHASTE: a Hardware/Software Co-design Testbed for the Xilinx XC6200

Gordon Brebner
(gordon@dcs.ed.ac.uk)
Department of Computer Science
University of Edinburgh*

Abstract

This paper describes a testbed for exploration of the ways in which novel features of the Xilinx XC6200 FPGA technology can be harnessed as a basis for hardware/software co-design. Current exploration work is particularly concerned with how FPGA technology can be naturally presented to standard system software, such as an operating system. To support this work, the CHASTE toolkit allows low level use of XC6200 features, unobstructed by the efforts of high level circuit design and analysis tools. The two main components are a circuit specification tool and a simulation tool, both of which are used from within C programs. This program-based approach is intended to encourage contributions from researchers with a software background. However, the tools operate on standard XC6200 configuration data files, and so can be used in tandem with the Xilinx XACT*step* design software and, of course, with real XC6200 parts. The toolkit has proved invaluable to support the hardware/software experimental work and, for example, has allowed the feasibility of a practical 'virtual hardware' operating system to be demonstrated [2]. The simulation tool has also allowed preliminary investigation of some speculative extensions to the XC6200 architecture.

1 Introduction

Field Programmable Gate Array (FPGA) technology has the potential to play a vital rôle in bridging the traditional divide between hardware and software. In particular, it raises fundamental questions about how the major types of system software — compilers, run time systems and operating systems — might interact with FPGA hardware, which can variously exhibit characteristics of processors, memory and input/output devices. In [2], the author describes a working demonstration of how an operating system can make use of an FPGA in order to implement *virtual hardware* — a concept generalised from the familiar notion of virtual memory. This work was implemented using, and was inspired by, the recent Xilinx XC6200 FPGA family [7,13], which has two features that proved essential: selective and fast reconfiguration of the array, and direct bus access to registers configured on the array. Future work will explore further the relationship between system software and FPGA hardware, with the aim of facilitating a natural style of hardware/software co-design for computer systems.

The CHASTE[1] toolkit was built to facilitate the experimental work. It consists of a small collection of tools that allow experimentation using the XC6200 at a low level, close to the actual array of configurable logic. A single data structure for representing XC6200 circuitry is at the centre of the toolkit, as illustrated in Figure 1. The circuit representation used is described in Section 2 of the paper. The decoder and encoder are used to convert between the internal data structure representation of a circuit and the XC6200 standard configuration data file format. The

*James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom.

[1] **C**onfigurable **H**ardware **A**nd **S**oftware **T**estbed **E**nvironment — any implications of purity are entirely intentional.

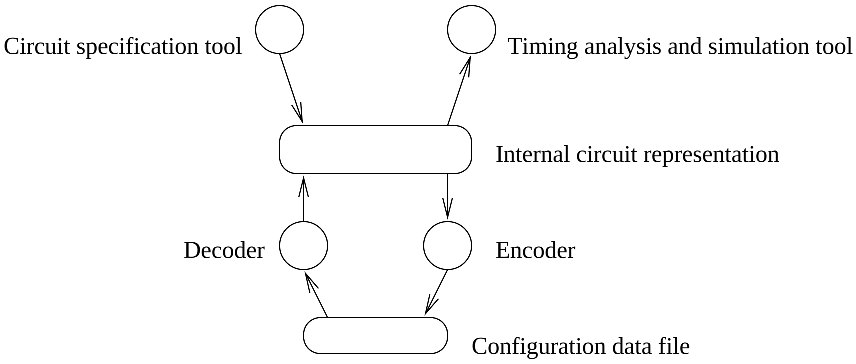

Figure 1: Components of the CHASTE toolkit

decoder allows configuration data generated using the Xilinx XACT*step* software to be handled by the toolkit, and the encoder allows valid configuration data for XC6200 parts to be generated.

The two main components of CHASTE are the circuit specification tool and the timing analysis and simulation tool, described in Sections 3 and 4 respectively. At first sight, there might seem to be little need for these, since the XACT*step* software (as a back end to Viewlogic or VHDL software) can be used for circuit specification, and the XC6200 part itself can be used to test circuitry. However, there were good reasons for following an alternative approach.

In the case of circuit specification, the nature of the experimental work made it desirable to be liberated from the concept of high level circuit design. That is, it was desirable to investigate the detail implications of XC6200 circuit configurations, and so it was not appropriate to carry out technology-independent circuit specification, where analysis and synthesis tools automatically deal with the lower level issues. This is analogous to operating system work in the software world, where machine-dependent features must be catered for at a level below the environment created by high-level programming languages and their compilers.

In the case of simulation, the issue was forced to an extent by the extremely recent appearance of the XC6200 part. Two XC6216 chips were available for real testing, but these were from the first batch of engineering samples, and did not allow the full range of XC6200 features to be used. In addition, the chips were used on a Xilinx development board that had only limited interfacing to a host computer. These practical matters pushed the case for a simulator, pending the arrival of production parts. However, the use of a simulator also allowed rather more creative experimental work than would be possible using real parts. First, a range of different interfaces to the XC6200 FPGA could be investigated, which is of vital importance in the search for good hardware/software interaction paradigms. Second, possible extended features for the XC6200 could be invented and tested. Just as the advent of operating systems inspired new features in processors, so the experiments carried out inspired possible new features for the FPGA.

Aside from the technical advantages of producing new circuit specification and simulation tools, there is one major *cultural* advantage. The specification tool is provided as a library of functions that can be called from a C program. The simulator is provided as another library of functions that can be called from a C program. Therefore, an experimentor from a software background can use the FPGA hardware concepts by working in a well-understood framework: programming. The author strongly believes that, in order to advance the state of hardware/software co-design using FPGAs, it is vital to tempt software people into a world still largely dominated by those who are used to working with hardware design tools. The toolkit enables the necessary spatial and temporal awareness associated with circuit design to be introduced gently.

This approach is not completely novel, of course. For example, normal programming languages, with additional libraries, have been used for VLSI design as 'silicon assemblers' [10]. More recently, this approach has also been used for configurable hardware. The PERLE programming environment for the Programmable Active Memories (PAM), developed at the Digital Paris Research Laboratory, involves the use of C++ programs to generate netlists for synchronous circuits [1].

These netlists are then transformed by normal placement and routing tools into circuit configuration data. The PERLE environment also includes a simulation tool, allowing combined simulation of both hardware and software components. The advantages and disadvantages of the overall PERLE approach are discussed very well in [1]. The major difference from the work reported here is that PERLE is intended to support the hardware/software co-design of complete subsystems. Here, the interest is in providing generic operating system support for user-level designs. Thus, the tools are hardware-specific, and are used at a lower level of abstraction. In fact, CHASTE would be a possible environment (augmented by harmonious place and route tools) for supporting an environment like that of PERLE on the XC6200 family, giving an overall approach completely based on a high-level programming language.

2 Internal circuit representation

An internal representation for XC6200 circuitry was needed to underpin the various tools within CHASTE. The level of circuit abstraction chosen was a compromise between the needs of different tools. For circuit specification, a reasonably high level of abstraction is desirable. For example, a representation based on logic gates and wiring nets would be appropriate, since it embodies the notion of circuitry but involves components that are easily comprehensible to a designer. In contrast, for timing analysis and simulation, a rather lower level of abstraction is desirable. For example, given that the XC6200 hardware consists of inter-connected configurable multiplexors, a representation based on multiplexors and wires would be appropriate.

After some preliminary experiments using alternative levels of abstraction, the final representation was chosen to have a strong multiplexor influence, but retaining sufficient abstraction that there is still a relationship back to specifications based on logic gates. In addition, the slightly higher level of abstraction allowed more efficient timing analysis and simulation, compared with a pure multiplexor representation.

2.1 XC6200 architecture, in brief

Familiarity with the XC6200 architecture [13] is helpful for grasping the details of the circuit representation, but the following brief summary is adequate for a reasonable understanding. An XC6200 part is composed of an array of simple configurable cells. Each cell contains a function unit capable of acting as a two-input logic gate or multiplexor, together with a D-type register. There is nearest-neighbour routing between adjacent cells, as well as fastlane routing over groups of 4, 16 and n cells, where n is the side length of the square array. In addition, there are four global signals that can reach all cells. Finally, there are input/output blocks with associated i/o pads located around the perimeter of the array, one per perimeter cell. All of the aforementioned features are configurable by writing to an SRAM control store.

2.2 Graph-based circuit representation

The internal circuit representation has the form of a directed graph, with vertices corresponding to **ports** and directed edges corresponding to connections from one port to another. The overall idea is that a port is capable of generating an output value, which can depend on zero or more input values. There are four types of port, each type corresponding to a different feature of the XC6200 architecture:

1. **compute port:** corresponds to the function unit of a cell within the array;
2. **i/o port:** corresponds to an input/output block on the perimeter of the array;
3. **multiplexor port:** corresponds to a routing multiplexor; and
4. **source port:** corresponds to a global wire or an internal control wire.

The first two types of port could be described in a more primitive way using multiplexor components, but this less abstract approach is neither necessary nor particularly desirable. The overall classification of ports makes the data structure amenable to use with other FPGA technologies, as well as specifically for the XC6200 family.

All ports have some associated information: the (x, y) coordinates of the port on the array, a further sub-position at the (x, y) coordinate, and the type of the port. In fact, the type of the port can be directly inferred from its sub-position, but is stored separately for convenience. Ports contained within the $n \times n$ cell array have coordinates in the range $(0, 0)$ to $(n-1, n-1)$. Ports contained within perimeter i/o blocks extend the coordinate range down to -1 to the left and below, and up to n to the right and above. Other ports (those corresponding to global wires) use $(-1, -1)$ as their coordinates. There are a few other fields for use in timing analysis and simulation; these are described in Section 4.

Compute ports have some extra associated information. There are values for $Y2$ and $Y3$, which select the function computed, and for RP and CS, which determine the behaviour of the D-type register associated with the function unit. I/O ports also have some extra associated information, specifying their configurable items: array enable, data delay, pull up, pull down and slew. Both compute and i/o ports have one other field for use in timing analysis and simulation, and this is described later in Section 4.

In graph terms, the different types of port have different in-degrees, that is, different numbers of edges directed at the port. A compute port has three in-edges, corresponding to the three function unit inputs $X1$, $X2$ and $X3$. An i/o port also has three in-edges, corresponding to the three inputs to an i/o block that come from the array. These arrive from the neighbouring cell, and the adjacent length-4 and length-16 fastlanes. A multiplexor port has a single in-edge, and a source port (as its name suggests) has no in-edges. For each in-edge, a pointer to the source port is stored — this allows edges to be traversed in a backwards direction, which is useful for the tools decribed in Sections 3 and 4

Each type of port has a variable out-degree, that is, it can have zero or more edges directed towards other ports. At each port, the out-edges are represented by a list of the destination ports that are reachable from the port. This allows edges to be traversed in a forward direction, although this capability is not in fact required by the tools described in this paper. The maximum out-degree allowed varies not only between the different types of port, but also between different instances of the four basic port types.

2.3 Lazy optimisation of representation size

When representing a configured XC6200 array, in principle, all possible ports should be present, and all in-edges to ports should be present. This is because, when the array is initialised, the configuration memory is zeroed, which results in a safe configuration of all function units, routing and input/output blocks. Subsequent writing of configuration data results in this initial configuration being modified as required. In order to keep down the memory space required for the internal representation of circuitry, there is a different notion of an initial configuration. In this, ports initially have a 'not present' characteristic, which means that the memory required for a full port representation is not used. Whenever a port is required, either because it is being added to a circuit, or because it is touched by a signal path during circuit analysis or simulation, the full port representation is dynamically created. This lazy evaluation approach allows significant economies for almost all circuitry.

3 Circuit specification functions

The SPODE[2] circuit specification library is the component of CHASTE that allows circuitry to be created at a level that allows complete control over the resulting array configuration. That is, functions, placement and routing can be precisely specified. In particular, specific circuitry can be

[2] **S**oftware **P**erson **O**riented **D**esign **E**nvironment.

dynamically generated by an operating system as required. The functions in the SPODE library are called from a C program; when the program is run, the SPODE functions output configuration data for the XC6200 part. This data is in exactly the same format as that generated by the Xilinx XACT*step* development system, so circuitry can be generated using a mixture of both tools.

3.1 The SPODE library

There are only nine different functions in the SPODE library, which was designed to provide simple, but powerful, primitives. First, there is a pair of functions that supply an enclosing context for a circuit specification. All other function calls made within this context contribute to the specification. There are three pairs of functions, concerned with creating array cells, input/output blocks and fastlane wiring respectively. Each of these is used to establish a sub-context within the overall circuit context. Finally, one function is used for creating specific connectivity between ports (to use the terminology of Section 2). This function can be used within any of the three contexts: array cells, i/o blocks and fastlane wiring. Complete details of the nine functions, and their parameters, can be found in [3]. The following contrived example illustrates how these different functions might be called in an appropriate program sequence:

```
begin_block ("contrived")
   begin_cell (not_d__b_in__f_out, a_xnor_b, 19, 13);
      to_from (a_in, right);
      to_from (right, f_out);
   end_cell ();
   begin_iob (1, 1, 0, 1, 0, 0, bottom, 19);
      to_from (clock, G1);
      to_from (top_16, io_pad);
   end_iob ();
   begin_slice (bottom_16, 15, 0);
      to_from (bottom_16, top);
   end_slice ();
end_block ();
```

The details of the function parameter values are not important here, just the overall structure. As the names of the functions hint, the `begin_cell/end_cell` context is used for defining a cell, the `begin_iob/end_iob` context is used for defining an i/o block and the `begin_slice/end_slice` context is used for defining fastlane wiring. A slice is a group of adjacent cells, the example here in fact being a 1×16 group, which can include length-16 vertical fastlane wires.

The example here does not illustrate the main advantage of the C function library approach. This is that the full power of the C language can be used in combination with the SPODE functions. In particular, constants, arithmetic expressions, conditional statements, looping statements and function definition can be used. This allows structures of repeated identical, or similar, circuitry to be easily specified. Also, libraries of useful derived functions can be constructed for standard, and parameterised circuitry. Functions can also be used as a way of structuring circuit specifications. It is possible to produce complete circuit designs using SPODE. For example, a demonstration implementation of the Huffman decoding circuitry described in [4] has been produced. However, this 'machine code' style of design is more appropriate for producing smaller component pieces of circuitry.

As a small illustration of the use of SPODE, the C function `top_river_router` shown in Figure 2 generates a river-routed bus connecting a vector of points (`x1`) with y-coordinate `y1` to a vector of points (`x2`) above with y-coordinate `y1+width`. This does not look a pleasant piece of programming work, but it only needs be done once, and it is then available as a parametrised library function for other designs to use.

The `to_from` function is used for creating wiring, and is at the heart of most circuit specifications. There are 46 different wiring positions that can be specified as sources and/or destinations for wires. The allowable positions for a given call depend on the surrounding context: cell, iob or

```
/* Auxiliary function to generate a cell containing one wire */
void wire_cell (location_type to, location_type from, int x, int y)
{
   begin_cell (no_register, no_function, x, y);
      to_from (to, from);
   end_cell ();
}

void top_river_router (int width, int x1 [], int x2 [], int y1)
{
   int i, j;

   if (y1+width >= ARRAY_HEIGHT) return;
   for (i = 0; i < width; i++) /* For each point in vectors */ {
      /* First, wire upwards to horizontal channel */
      for (j = 0; j < i; j++) wire_cell (top, bottom, x1[i], y1+j+1);
      if (x1[i] > x2[i]) /* Need to go left horizontally */ {
         wire_cell (left, bottom, x1[i], y1+i+1);
         for (j = x1[i]-1; j > x2[i]; j--) wire_cell (left, right, j, y1+i+1);
         wire_cell (top, right, x2[i], y1+i+1);
      } else if (x1[i] < x2[i]) /* Need to go right horizontally */ {
         wire_cell (right, bottom, x1[i], y1+i+1);
         for (j = x1[i]+1; j < x2[i]; j++) wire_cell (right, left, j, y1+i+1);
         wire_cell (top, left, x2[i], y1+i+1);
      } else /* No horizontal move necessary */ {
         wire_cell (top, bottom, x1[i], y1+i+1);
      }
      /* Finally, wire upwards from horizintal channel */
      for (j = i+1; j < width; j++) wire_cell (top, bottom, x2[i], y1+j+1);
   }
}
```

Figure 2: Example of SPODE-based function for river-routed wiring

slice. At run-time, SPODE checks that the actual parameters supplied to each call of `to_from` are consistent with the current context, as given by the most recent call of `begin_cell`, `begin_iob` or `begin_slice`, and reports an error in cases of inconsistency.

The `begin_cell` function uses a slightly higher level of abstraction, compared with the level used in the internal representation described in Section 2. Function units are not described in terms of the RP and CS variables that determine D-type register behaviour, and the $Y2$ and $Y3$ variables that determine the computed function. Instead, the register type is described in a higher level way, and the function type is described in terms of the logic gate or multiplexor implemented. Apart from convenience for a designer, this choice makes SPODE more amenable to use with other FPGA technologies. Register types incorporate three separate characteristics:

1. use: constant 0, constant 1, D-type, or unused;
2. whether loopback, i.e., register can supply one function unit input; and
3. whether sequential, i.e., function unit computation result is fed through register.

For example, the "`not_d__b_in__f_out`" type that appeared in the contrived SPODE example earlier means that the register is a D-type ("`not_d`") with loopback ("`b_in`") and is sequential ("`f_out`"). Examples of computed function types include "`one`" (constant 1), "`not_b`" (inverse of input B) and "`a_or_not_b`" (disjunction of A input and inverted B input).

3.2 SPODE implementation

There is a fairly close correspondence between the information supplied by parameters to SPODE function calls and the configuration data that has to be generated for the XC6200 part. Ideally, a simple implementation route would be for SPODE functions to do little more than print out appropriate data. Unfortunately, the situation is not so straightforward, the basic problem being the nature of the multiplexors used to implement the XC6200 technology. In many cases, the multiplexors output the inversion of the selected input value. The result is that wiring which passes through one or more multiplexors may invert the signal it is carrying. The solution to this problem is to compensate for the inversion when the signal is used as an input. This is possible in almost all cases for XC6200 circuitry.

To do the inversion correction, it is necessary to have the complete circuit, and then investigate it for inverting wiring. Therefore, SPODE function calls have the effect of constructing an internal representation of a circuit using the data structure described in Section 2. This circuit is structurally correct, in that ports and connections are accurate. However, it may be functionally inaccurate, because all wiring is assumed to be non-inverting in the first instance. After the circuit is complete, i.e., when the `end_block` function is called, it is analysed to find inverting wiring, and such wiring is corrected by adjusting function unit and i/o block configurations.

The heart of the correction process resides in a table-driven function that, for any connection between two ports, reveals whether or not that connection is inverting. This is used in a search algorithm that begins from each compute port, and from each i/o port.

For a compute port, a backwards traversal of edges is performed for each of the three in-edges. The traversal of edges continues until it reaches the non-multiplexor port that is the source of the signal to the function unit. During the traversal, it is noted whether the total number of inverting edges on the path is odd or even. If this number proves to be odd, then that function input is treated as negated. For example, for an `a_and_b` function, if the signal corresponding to `a` is inverted, then a `not_a_and_b` function is configured instead.

For an i/o port, the three in-edges are traced back similarly. However, inversion compensation is only possible for the length-4 fastlane in-edge, by flipping the array enable configuration of the i/o block. There is no way of compensating for inversions on the other two inputs. For the these in-edges, a warning message is output to the user, who must perform hand correction, if necessary, by adjusting the circuitry.

Two points about the searching and correction process are worth making. First, although a recursive graph traversal function is tempting, the traversal is implemented iteratively with an explicit stack. This is to avoid internal stack overflow due to over-deep recursion when traversing long wiring paths — a problem that became evident on the PC implementation of SPODE. Note also that the traversal process checks for wiring loops, and reports these, rather than indulging in infinite traversal. Second, correction must be done at the end of the wiring where a signal is received; it cannot be done at the end where the signal is generated because, in general, there may be fan-out.

Once inversion correction has been performed, configuration data for the XC6200 is generated. The data is output in increasing address order, except where impossible (in cases where control registers have to be changed prior to continuing with array configuration). This enables maximum use to be made of multi-byte write operations at configuration time. It also gives scope for the automatic identification of potential wildcarded addressing — a feature not currently implemented, however.

The configuration data generated by SPODE for a variety of circuits has been tested on a real XC6216 part, and found to work correctly. Further, the SPODE tool has been tested on XC6200 novices, who were both able to produce working circuitry within short periods of time [5,8].

4 Timing analysis and simulation functions

The basic facilities needed for timing analysis and simulation are very closely related and, indeed, can be used in tandem. They are not intended for use with completely arbitrary XC6200 circuitry. Two restrictions are made on circuits, both completely acceptable for the work which the CHASTE toolkit is designed to support, and also acceptable for other XC6200 circuitry produced elsewhere. The restrictions are:

- there are no combinatorial loops within the circuit; and
- input signals are synchronised with the global clock input signal.

The effect of these restrictions is that, for a given value (high or low) of the global clock signal, an input vector can be supplied, and this will result in the circuit reaching a new stable state of register and output values.

The purpose of timing analysis is to identify critical paths within circuits, in order to determine the minimum time required for the circuit to reach a stable state. From this, appropriate global clock rates can be determined for normal operation. This analysis is purely structural, given that accurate timing information is available for the components of the XC6200 part.

The purpose of simulation is to determine the new stable state of a circuit, given a new setting of the global clock signal. In fact, given a circuit design style with registers clocked by the global clock, and also the fact that the SRAM read/write interface uses the global clock, it is the most interesting to simulate a single clock cycle. A cycle consists of the global clock going low, and then going high. At this point, register contents will be stable. Two such cycles are required to carry out an SRAM access to the array, so the array is in an intermediate state between the cycles. Simulation is not just structural, but depends on input values and stored values in the circuit.

Timing analysis and simulation are combined into one function called `time_step`. This simulates one step of the circuit, for a constant value of the global clock. As it does so, it computes the real time required for the step, and this is returned as a result. One use of the function is for it to be called repeatedly, with paired calls for each clock cycle. This can give a time-driven simulator (also known as an oblivious simulator) for the circuit. Such a simulator is usually regarded as inferior to an event-driven simulator, since time steps without input changes might be simulated unnecessarily. It would not be hard to construct an event-driven simular based on the `time_step` function using well-known methods, if lengthy periods of circuit operation were to be simulated, but so far this has not proved necessary for the experimental work carried out.

4.1 Implementation of `time_step` function

The `time_step` function operates on the circuit representation described in Section 2. Circuits can be stored either using SPODE, or using the decoder to read in standard XC6200 configuration data files. The `time_step` function makes use of some extra information fields associated with ports, already alluded to in the description of the representation. First, a one-bit field is used to contain the value associated with the port. Recall that the idea is that each port generates a value that can propagate to other ports. For example, the value associated with a compute port is the output of its function unit (or of its register, if its operation is sequential). During a simulation step, stable output values for ports are determined. When a port output values does become stable, a second extra field is updated. This gives the number of picoseconds that necessarily elapse from the beginning of the simulation step until the output value is stable.

One further field is needed for compute ports, and for i/o ports, in order to carry out the simulation. In the case of a compute port, the field contains the one-bit value held in its register at the previous simulation step. In the case of an i/o port, the field contains the one-bit value from its input pad. The pad input values are supplied as the input vector for the simulation step.

A *demand-driven* simulation approach is used. This is a well-known approach to circuit simulation (for example, see [6,11,12]), which works by simulating only the circuitry needed to determine the visible outputs. Here, the visible outputs are the signals on output pins, and also the values contained in registers. With this lazy approach, simulation time might be saved by not dealing with redundant circuitry.

The overall algorithm is very similar to that used in the inversion compensation algorithm described in Section 3. That is, paths are traced backwards through the circuit wiring. Further, these paths similarly begin at compute ports — which house the registers — and at i/o ports — which house the output pins. The traversal must also take account of inversions caused by multiplexors, but in this case simulating the inversions rather than counting them.

When the traversal process visits a port, it first checks whether the port output value has already been determined. This is indicated by the value field being 0 or 1, rather than 'undefined'. If it is defined then the visit ends immediately, and the result of the visit is the port's value *and* the earliest time at which it becomes stable. If the value is not yet defined, then some work must be done. The type of work depends on the type of port:

- compute port: visit the three ports that supply function inputs, by following the in-edges backwards; compute the new port value from the three input values, and set the stable time to be the earliest time at which all three input values are stable and can propagate to the function unit.
- i/o port: if the i/o block is configured in input mode, set the new port value to the input pin value associated with the port and the stable time to be zero; if not, visit the port that supplies the output pin value by following the neighbour in-edge backwards, then set the new port value to this port's value and the stable time to be the earliest time that this value can propagate to the output pin.
- multiplexor port: visit the port that supplies the input to the multiplexor; set the new port value to the input value, and set the stable time to be the earliest time at which the input value can propagate to the multiplexor.
- source port: set the new port value to the fixed value associated with the port (this is either a constant, or an input pin value, depending on the particular port instance), and set the stable time to be zero.

Although this description is in a recursive form, it is implemented iteratively, as for the inversion compensation algorithm.

After all of the visible circuit outputs have been determined, the simulation step is complete. The required minimum duration of the step is determined by finding the maximum stable time over all of the compute ports and i/o ports in the circuit.

The remaining details of the `time_step` function concern the input and output pins. These ensure that the extra control functions associated with pins, in particular the read and write access to the array, are implemented correctly. A complication is that i/o blocks can be configured so that control signals to the array are supplied by the array itself, rather than the external input pins. An auxiliary data structure is used for the extra i/o information that is not stored with the main circuit representation. In essence, this data structure represents the 'hard state' of the device, i.e., the state information that cannot be configured. Further information on this aspect of the simulation can be found in [8].

The time analysis and simulation function were exercised using non-trivial example circuits that had already been tested on a real XC6200 part, in order to ensure that the correct results were obtained. These circuits included use of i/o blocks in non-trivial ways, and so exercised a fairly tricky part of the simulation environment. The results of the timing analysis were consistent with results computed by hand. It was not possible to check that the claimed clock rates were indeed practicable, since the hardware available did not allow the selection of arbitrary clock rates. However, this will be possible in the future.

The main application of the simulation function has been in supporting the virtual hardware operating system experiments described in [2]. The simulator has allowed detailed investigation of the behaviour of the XC6200 array, including the timings for crucial operations. A further luxury of the simulation approach was to allow experimentation with slightly modified forms of the XC6200 architecture. Such experiments are useful in order to investigate possible future FPGA improvements to support operating system requirements more efficiently.

5 Conclusions

The CHASTE toolkit has proved invaluable for experimenting in detail with the novel features of the new Xilinx XC6200 technology. The circuit specification tool has been used for producing tailored circuitry and, as a general point, has confirmed that there is a place for design tools embedded within normal programming languages. If one is designing a piece of random logic, to be implemented using an FPGA, then conventional high level circuit design tools are more appropriate. However, for very regular logic, device-specific logic or dynamically-generated logic, the SPODE approach is worth considering. The low level of specification might be made more attractive in the future by supplying a graphical front end that generates SPODE function calls.

The timing analysis and simulation tools have been used principally for experimental work to explore the relationship between operating system software and the XC6200 FPGA. Although real experiments are planned, using both a Xilinx PCI bus board and also the HADES co-processor board [9], the simulator will still have a rôle when investigating novel forms of interface and extended FPGA architectures. Such investigation is important for finding good, and efficient, models for hardware/software co-design.

Aside from the research applications of CHASTE, recent experience has shown that it provides a way for novice students to rapidly produce working XC6200 circuitry, without any previous experience of hardware design. This is encouraging in a computer world where the natures of hardware and software are being revisited, with FPGA technology acting as an important catalyst.

6 Acknowledgements

Many thanks are due to Damian Kwok for his implementation work on both the simulator and the configuration data file decoder, as well as his infinite patience when faced with frequent respecification of the overall CHASTE environment. The technical assistance of the Xilinx Development Corporation has been essential to underpin this work.

7 References

[1] Bertin and Touati, "PAM Programming Environments: Practice and Experience," Proc. IEEE Workshop on FPGAs for Custom Computing Machines, IEEE 1994, pp.133–138.

[2] Brebner, "A virtual hardware operating system for the Xilinx XC6200," Proc. 6th International Workshop on Field Programmable Logic and Applications, Springer 1996, pp.327–336.

[3] Brebner "The SPODE circuit specification library for the XC6200," Internal report, Department of Computer Science, University of Edinburgh, 1996.

[4] Brebner and Gray, "Use of reconfigurability in variable-length code detection at video rates," Proc. 5th International Workshop on Field Programmable Logic and Applications, Springer 1995, pp.429–438.

[5] Campbell, "An investigation of virtual hardware," Masters dissertation, Department of Computer Science, University of Edinburgh 1996.

[6] Charlton, Jackson and Leng, "Lazy simulation of digital logic," *Computer-Aided Design* **23** September 1991, pp.506–513.

[7] Kean, Churcher and Wilkie, "XC6200 fastmap processor interface," Proc. 5th International Workshop on Field Programmable Logic and Applications, Springer 1995, pp.36–43.

[8] Kwok, "An investigation of virtual hardware using FPGA technology," Honours dissertation, Department of Computer Science, University of Edinburgh 1996.

[9] Ludwig, "The design of a co-processor board using Xilinx's XC6200 FPGA — an experience report," Proc. 6th International Workshop on Field Programmable Logic and Applications, Springer 1996, pp.77–86.

[10] Monier and Vuillemin, "Using Silicon Assemblers," Proc. IFIP International Conference on VLSI, Elsevier 1985, pp.315–324.

[11] Smith, *Progress in computer-aided VLSI design* Vol. 1, Ablex Publishing Corporation 1989: chapter on demand-driven simulation, pp.191–233.

[12] Smith, Mercer and Brock, "Demand driven simulation: BACKSIM," Proc. 24th ACM/IEEE Design Automation Conference, ACM/IEEE 1987, pp.181–187.

[13] Xilinx Inc., XC6200 advance product specification, Xilinx 1996.