SELF-TIMED ARCHITECTURE OF A REDUCED INSTRUCTION SET COMPUTER

Ilana David[1], Ran Ginosar[1,2], and Michael Yoeli[2]

[1]Department of Electrical Engineering
[2]Department of Computer Science
Technion - Israel Institute of Technology
Haifa 32000

ABSTRACT

An advanced Self-Timed Reduced Instruction Set Computer (ST-RISC) architecture is described. It is designed hierarchically, and is formally specified functionally at the various levels by a CSP-like language. The architectural features include decoupled data and branch processors, delayed branches with variable delay, unified data path and control, efficient non-redundant handshaking protocols, and novel self-timed building blocks such as combinational logic, master-slave registers, finite state machines, and FIFO elements.

Keywords: *Asyncronous systems, computer architecture, high-level synthesis, reduce instruction set computer, self-timed.*

1. Introduction

The design and construction of an asynchronous computer is relatively a complex task. Asynchronous design of digital circuits is not as straightforward as synchronous design; the former refrains from the relaxing assumptions and external enforcement of timing available with clocked systems. While synchronous implementations of digital systems prevail today, self-timed asynchronous designs present an attractive alternative, offering the potential for faster systems, correct-by-construction synthesis, and formal verification.

Digital hardware systems consist of basic building blocks, such as combinational logic, registers, and finite state machines. In previous reports we have described the design of self-timed combinational logic, master-slave registers, finite state machines, and first-in-first-out queue elements in a formal manner, while stressing efficient implementations [DGY92a, DGY92b, Da89]. In this paper we present the architecture of a self-timed reduced instruction set computer (ST-RISC). It is described as a hierarchical network of communicating sub-systems; each sub-system is either a basic building block of the above-mentioned types, or is itself a network of sub-components.

ST-RISC, as well as all its subsystems, are self-timed, in the sense that they are (1) asynchronous, (2) delay-insensitive, and (3) they generate completion signals. Delay insensitivity here means that their correct operation is independent of any assumption on the delays of the components or wires.

ST-RISC incorporates advanced architectural concepts, which are designed to enhance its performance. There are two concurrently executing units - the data processor and the branch processor. A novel scheme of delayed branches reduces ‘‘pipeline suspensions.’’ Control and datapath are unified together. The various components intercommunicate over efficient channels, sometimes containing FIFO decoupling buffers. The architecture is presented as a sequence of increasingly-detailed specifications, and as such it yields to formal verification. The architecture and its specification are designed with automatic synthesis in mind.

The pioneer asynchronous processor architecture was published by Martin *et al.* [Ma89a]. A CSP-like specification language was introduced (which has largely been adopted by us), and a synthesis method was presented [Ma89b]. While the emphasis was on automatic compilation, the architecture was rather simple. We further compare our architecture with Martin’s in Section 9.

Other methods for implementing double-rail, self-timed circuits, can be found in references [Se80, Si81, An86, La87 and Ha84], as well as [DGY92a] and [DGY92b]. Extensive research on the synthesis of delay-insensitive circuits is also reported in [Chu87, MFR85, Ma86, Rem85, Sn85, Eb87].

Section 2 presents the building blocks. The processor is functionally defined in Section 3. Section 4 shows how ST-RISC is decomposed hierarchically. Sections 5 and 6 describe the data and branch processors, respectively. The FIFO buffers and the forks are discussed in Section 7. Delayed branches are presented in Section 8, and the significance of this architecture is discussed in Section 9. The Appendix defines the specification language used in the paper.

2. Self-Timed Modules

General methods for efficiently implementing combinational logic (*CL*), master-slave registers (*MS*) and finite state machines (*FSM*) as self-timed logic modules are described in references [DGY92a], [DGY92b], and [DGY89b]. A double-rail code is used to represent the ternary logic: the three values 0, U (*undefined*) and 1 of any line are represented by 10, 00, and 01 respectively.

The *CL* module [DGY92a] implements any family of Boolean functions as a self-timed circuit. In simple *CL* modules, all the inputs become defined before the outputs are generated, and once the outputs are defined, all the inputs must become undefined before any of the outputs disappears (becomes undefined). In more general *CL* modules, these restrictions may apply to predefined subsets of the inputs.

MS [DGY92b] is a self-timed master-slave register. Its operation can be described as follows: When all the inputs become defined, the outputs become undefined, and the inputs are latched into the register; when all the inputs become undefined, the outputs become defined.

The *FSM* [DGY92b] is a self-timed finite state machine. It consists of a *CL*, which implements the logic functions of the state table, and an *MS* for the feedback register. When the inputs (or some predefined subsets thereof) become defined, the outputs and the next state are generated and the next-state is saved in the MS register. When the inputs become undefined, the outputs become undefined, and the saved 'next-state' becomes the output of the *MS* register, serving as the 'present state' for the next cycle.

3. Specification of the Processor

The instruction set of ST-RISC (Table I) contains three types of instructions: ALU (arithmetic, logic and shift), memory (load and store) and flow-control (jump and conditional branch). Separate memories, *DMEM* and *IMEM*, are used for data and instructions. There are 16 16-bit registers in the register file (*RF*), one of which (R_0) is hard-wired zero.

The program in Figure 1 specifies the sequential behavior of the processor. The program notation, which is a modified version of Martin's [Ma89b], is described in the Appendix.

In this program, the next instruction is loaded from the instruction memory *imem* onto variable *b*. The fields of *b* contain the op-code ($b.op$), the two source registers ($b.r_{s1}$, $b.r_{s2}$) and the destination register ($b.r_d$). R_{s1}, r_{s2} and r_d serve as pointers to the register file (rf). In *ldi*,

Arithmetic	ADD	R_{s1}, R_{s2}, R_d	$R_d \leftarrow R_{s1} + R_{s2}$, set CC
	SUB	R_{s1}, R_{s2}, R_d	$R_d \leftarrow R_{s1} - R_{s2}$, set CC
	XOR	R_{s1}, R_{s2}, R_d	$R_d \leftarrow R_{s1}$ *xor* R_{s2}, set CC
	AND	R_{s1}, R_{s2}, R_d	$R_d \leftarrow R_{s1}$ & R_{s2}, set CC
	OR	R_{s1}, R_{s2}, R_d	$R_d \leftarrow R_{s1} \mid R_{s2}$, set CC
Shift	SLL	R_{s1}, R_d	$R_d \leftarrow R_{s1} + R_{s1}$, set CC
	SRL	R_{s1}, R_d	$R_d \leftarrow$ logical_right_shift(R_{s1}), set CC
	SRA	R_{s1}, R_d	$R_d \leftarrow$ arithmetic_right_shift(R_{s1}), set CC
Load and Store	LD	R_{s2}, R_d	$R_d \leftarrow M[\ R_{s2}\ ]$
	LDI	R_d, OPERAND	$R_d \leftarrow$ *OPERAND*
	STORE	R_{s1}, R_{s2}	$M[R_{s2}] \leftarrow R_{s1}$
Flow Control	JUMP	Address	PC ← Address
	CBRANCH	COND,address	if (COND) { PC ← address }

Table 1: ST-RISC Instruction Set

$$
\begin{array}{l}
risc \equiv * [\ b := imem\,(pc);\\
\quad [\ arith\,(b.op) \rightarrow (rf\,(b.r_d)\,,\, cc) := b.op\,(rf\,(b.r_{s1})\,,\, rf\,(b.r_{s2}))\ ,\ \ pc := pc + 1\\
\quad\ [\!]\ shift\,(b.op) \rightarrow (rf\,(b.r_d)\,,\, cc) := b.op\,(rf\,(b.r_{s1})\,)\ ,\ \ pc := pc + 1\\
\quad\ [\!]\ jump\,(b.op) \rightarrow pc := b.addr\\
\\
\quad\ [\!]\ cbranch\,(b.op) \rightarrow [\ \neg cond\,(b.op\,,\, cc) \rightarrow pc := pc + 1\\
\qquad\qquad\qquad\qquad\quad [\!]\ \ cond\,(b.op\,,\, cc) \rightarrow pc := b.addr\ ]\\
\quad\ [\!]\ store\,(b.op) \rightarrow dmem\,(b.r_d) := rf\,(b.r_{s1}),\ pc := pc+1\\
\quad\ [\!]\ ld\,(b.op) \rightarrow rf\,(b.r_d) := dmem\,(b.r_{s1}),\ pc := pc+1\\
\quad\ [\!]\ ldi\,(b.op) \rightarrow rf\,(b.r_d) := b.operand,\ pc := pc+1\ ]\ ]
\end{array}
$$

Figure 1. ST-RISC sequential program

jump, and *cbranch*, *b*'s bits are reorganized to provide the operands, the address, and the address and condition, respectively. The instruction memory *imem* and the data memory *dmem* are represented as arrays; the program counter *pc* points into *imem*, while r_{s1} and r_d index *dmem* (in *ld*, *ldi*, and *store*). *Cc* holds the condition code. The *cond* function determines whether the branch condition holds.

4. Decomposition of ST-RISC into Communicating Processes

ST-RISC consists of two main components, the *branch processor* (*BP*) and the *data processor* (*DP*), and four FIFOs. In Figure 2 it is shown together with the data and instruction memories. Both *BP* and *DP* communicate with *IMEM*, while only *DP* is connected to *DMEM*. The main program of Fig. 1 is decomposed accordingly into the processes *bp* (Fig. 3), *dp* (Fig. 4), and the FIFO programs (Section 7). Figure 5 shows the memory processes. All these processes are active concurrently. Note that *BP* and *DP* are independent, except for the condition code which is generated by *DP* and is used by *BP* for conditional branches.

Not all communication channels in Figure 2 are alike. Some are based on the conventional 4-phase handshake protocol; they comprise double-rail data lines and an acknowledgement line. The communication operation is denoted $X!a$ and $X?b$ in Figures 3-5. Others ($ack-less$ channels) do not require acknowledgement signals, and are denoted $X!!a$ and $X??b$ in the programs. The correct operation of the system is guaranteed nevertheless, and is usually achieved by means of acknowledgement signals on other lines. For example, in Figures 2 and 3 *ADROUT* is an $ack-less$ channel but *DOUT* is not; during memory write operation, acknowledgement on *DOUT* serves to acknowledge *ADROUT* as well. The purpose of $ack-less$ channels is to eliminate redundant overhead. Further, in an efficient design an $ack-less$ channel is treated as such in all cases.

Note that channel *I* in Figure 2 seemingly consists of a fork. Acknowledgements on that channel are resolved by the two FIFOs *F1* and *F3*, as described in Section 7.

It can be proven formally that the two concurrent programs, *dp* and *bp*, together with the programs for the memories and the FIFO queues, are functionally equivalent to the main program *risc* (Figure 1).

In Section 8 we describe an improved architecture of the branch processor, which incorporates delayed branches.

5. The Data Processor

We continue the decomposition procedure, hierarchically. Figure 6 describes the structure of the data processor. The main components of *DP* are the register file (*RF*) , the arithmetic-logic

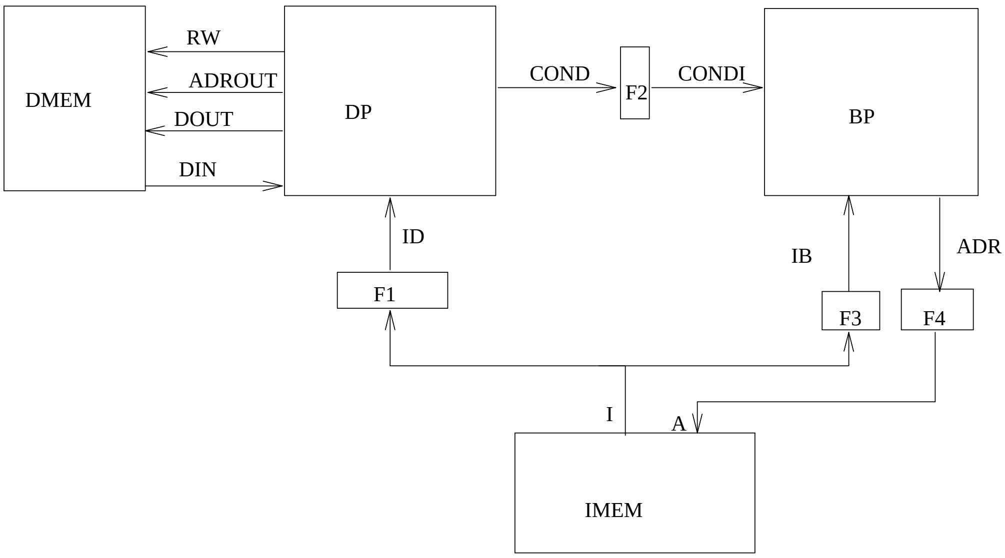

Figure 2. ST-RISC block diagram

$$
\begin{array}{l}
dp \equiv * [\ ID\ ?\ b;\\
\quad [\ arith\ (b.op) \rightarrow (rf\ (b.r_d)\ ,\ cc)\ :=\ b.op\ (rf\ (b.r_{s1})\ ,\ rf\ (b.r_{s2})\ )\\
\quad\ [\!]\ shift\ (b.op) \rightarrow (rf\ (b.r_d)\ ,\ cc)\ :=\ b.op\ (rf\ (b.r_{s1})\ )\\
\quad\ [\!]\ jump\ (b.op) \rightarrow nop\\
\quad\ [\!]\ cbranch\ (b.op) \rightarrow COND\ !\ cond(b.op, cc)\\
\quad\ [\!]\ store\ (b.op) \rightarrow ADROUT\ !!\ rf(b.r_{s1})\ ,\ DOUT\ !\ rf(b.r_{s2})\ ,\ RW\ !!\ 'w'\\
\quad\ [\!]\ ld\ (b.op) \rightarrow ADROUT\ !!\ rf(b.r_{s1})\ ,\ RW\ !!\ 'r'\ ,\ DIN\ ??\ rf(b.r_d)\\
\quad\ [\!]\ ldi\ (b.op) \rightarrow rf(b.r_d) := b.operand\ ]\ ]
\end{array}
$$

Figure 3. The data-processor program

$$
\begin{array}{l}
bp \equiv *[\; IB\;?\;d;\\
\qquad [\;\neg branch(d.op) \rightarrow pc := pc + 1\\
\qquad\quad [\!]\; jump\;(d.op) \rightarrow pc := d.addr\\
\\
\qquad\quad [\!]\; cbranch\;(d.op) \rightarrow CONDI?c;\; [\neg c \rightarrow pc{:=}pc+1\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\!]\, c \rightarrow pc{:=}d.addr]\;]\\
\qquad ADR\;!\;pc]
\end{array}
$$

Figure 4. The branch processor program

$$imem \equiv *[\; A\;??\;addr\;;\; I\;!!\;imem(addr)\;]$$

$$
\begin{array}{l}
dmem \equiv *[\;[ADROUT\;??\;addr\;,\; RW\;??\;rw\;];\\
\qquad\qquad [read(rw) \rightarrow DIN\;!!\;dmem(addr)\\
\qquad\qquad [\!]\; write(rw) \rightarrow DOUT\;?\;data\;;\; dmem(addr){:=}data]\;]
\end{array}
$$

Figure 5. The memory programs

unit (*ALU*) and a decoder (*DECOD*). The instruction word is read from the input channel ID into the decoder which generates the outputs R_{s1}, R_{s2}, R_d and *CMD*, needed to operate units *RF* and *ALU*. If the command is a conditional-branch, the decoder indicates it via BOUT and unit *BC* indicates on its COND output whether the condition is true or false. When the inputs R_{s1}, R_{s2} of *RF* become defined, it outputs the contents of the corresponding registers onto BUSA and BUSB respectively; when R_d and BUSC are defined, *RF* writes the contents of BUSC into register R_d. BUSC comes from a selector *SEL*, which selects a word either from the *ALU* output D (*alu* command), the data-memory input DIN (*ld* command), or the *DECOD* output OP (*ldi* command). The

ALU unit performs an arithmetic or logic operation according to input CMD, and generates a result word D and a condition code CA. The *CC* register holds the condition code to be used in the next instruction.

The register file is implemented as a group of *MS* registers, with additional decoders and selectors. *CC* is also an *MS*. All the other *DP* components, namely *ALU*, *SEL*, *BC*, and *DECOD*, may be implemented as *CL* blocks. Some of them are described as processes in Figure 7.

6. The Branch Processor

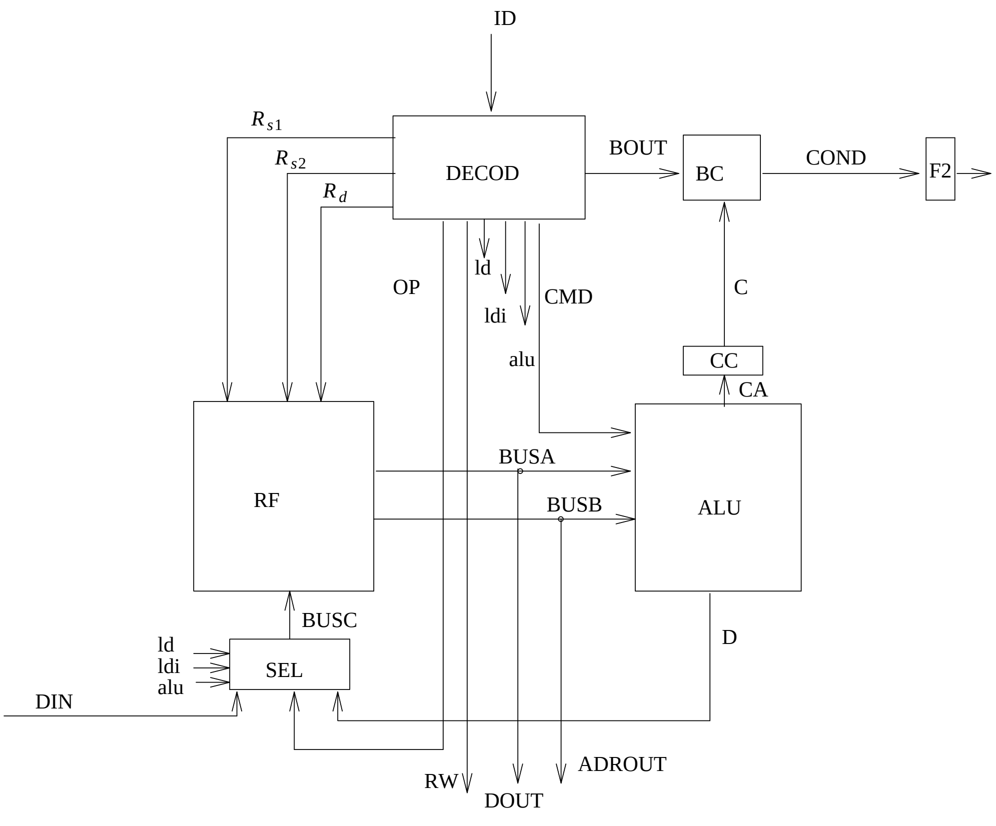

Figure 6. The data processor

$$
\begin{aligned}
decod \equiv\ * [\ & ID\ ?\ b; \\
& [alu\ (b.op) \rightarrow R_{s1}\ !!\ b.r_{s1}\ ,\ R_{s2}\ !!\ b.r_{s2}\ ,\ R_d\ !!\ b.r_d\ ,\ CMD\ !!\ b.cmd\ ,\ alu!!'alu' \\
& \ \ []\ jump\ (b.op) \rightarrow R_{s1}\ !!b.r_{s1}\ ,\ R_{s2}\ !!b.r_{s2}\ ,\ R_d\ !!\ '0'\ ,\ CMD\ !!\ 'nop'\ ,\ alu!!'alu' \\
& \ \ []\ cbranch\ (b.op) \rightarrow BOUT\ !\ condit(b.op) \\
& \ \ []\ store\ (b.op) \rightarrow R_{s1}\ !!\ b.r_{s1}\ ,\ R_{s2}\ !!b.r_{s2}\ ,\ RW\ !!\ 'w' \\
& \ \ []\ ld\ (b.op) \rightarrow R_{s2}\ !!\ b.r_{s2}\ ,\ R_d\ !!b.r_d\ ,\ RW\ !!\ 'r'\ ,\ ld!!'ld' \\
& \ \ []\ ldi\ (b.op) \rightarrow OPERAND\ !!\ b.operand\ ,\ R_d\ !!b.r_d\ ,\ ldi!!'ldi']
\end{aligned}
$$

$$bc \equiv\ * [\ [\ BOUT\ ?\ b\ ,\ C\ ??c\ ];COND\ !\ cond(b,c)]$$

$$rf \equiv\ * [[R_{s1}\ ??\ r_{s1}\ ;\ BUSA\ !!\ rf(r_{s1})]\ ,\ [R_{s2}\ ??\ r_{s2}\ ;\ BUSB\ !!\ rf(r_{s2})]\ ,\ [R_d\ ??\ r_d\ ;\ BUSC\ ?rf(r_{d)}]\ ]$$

$$alu \equiv\ * [(BUSA??op1\ ,\ BUSB??op2\ ,\ CMD??cmd);\ D\ !!\ aluf\ (cmd,\ op1,\ op2)\ ,\ CA\ !!\ condf(cmd,\ op1,\ op2)]$$

Figure 7. The processes of the data processor

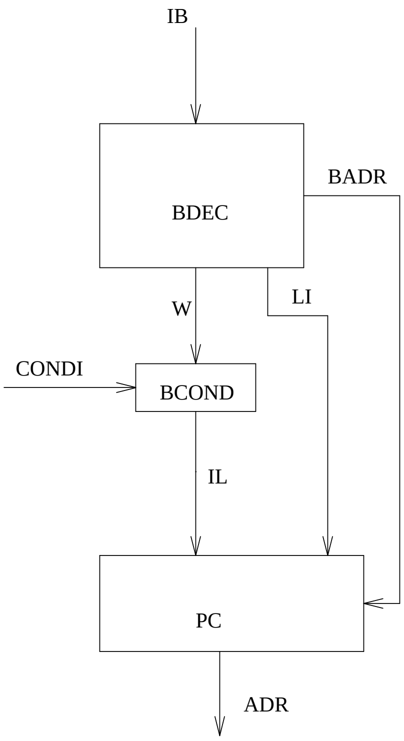

Figure 8. The branch processor

Figure 8 describes the structure of the branch processor. It comprises two decoders (BDEC and BCOND) and a counter (PC). BDEC determines, according to the op-code of the instruction word IB, whether the counter PC should be incremented, or, if the instruction is either *jump* or *cbranch*, loaded with a new address. Recall that the branch address is one of the fields of the instruction word. Unit BCOND is operational only during conditional branch instructions. It waits for the condition bit CONDI to arrive from the data processor and then decides whether the program counter should be incremented or loaded with the jump address. Both BDEC and BCOND are *CL* units. PC is a loadable counter, which is a special form of *FSM*. Figure 9 contains the programs for the BP components.

$$bdec \equiv *[\ IB\ ?\ d;$$

$$[\neg jump(d.op)\ \angle\neg cbranch(d.op)\ \rightarrow\ LI\ !\ \prime incr\prime$$
$$[]\ jump\ (d.op)\ \rightarrow\ LI\ !\ \prime load\prime\ ,\ BADR!!d.addr$$
$$[]\ cbranch\ (d.op)\ \rightarrow\ W\ !\ ,\ BADR!!d.addr]\ ]$$

$$bcond \equiv *[\ [W\ ?\ ,\ CONDI\ ?\ cond];$$

$$[cond\ \rightarrow\ IL\ !\ \prime load\prime$$
$$[]\ \neg cond\ \rightarrow\ IL\ !\ \prime incr\prime]\ ]$$

$$pc \equiv *[\ [\ [LI\ ?\ il\ ]$$

$$[]\ [IL\ ?\ il]\ ];\ [incr(il)\ \rightarrow\ pc:=pc+1$$
$$[]load(il)\ \rightarrow\ BADR\ ??\ pc\ ]\ ;\quad ADR\ !\ pc]$$

Figure 9. The processes of the branch processor

7. The FIFO

The main components of ST-RISC (memories, DP, BP) communicate through self-timed *FIFO*s, which serve as decoupling buffers. The self-timed FIFO element is described in reference [Da89]. A four-stage FIFO is shown in Figure 10. Its behavior may be described as follows: An empty FIFO contains only spacers (*undefined* values, coded 00 in double-rail). As long as there is room in the queue, valid data and spacer words enter the FIFO alternatingly through its input port *Din*. The *ack* line turns "1" as soon as a spacer has entered the FIFO (meaning the FIFO is ready for a data word), and it turns "0" after a valid data word has entered (meaning that the FIFO is ready for a spacer). When the receiving unit is ready, it issues "1" on the *read* line; as a result, the data word at the front of the queue moves to *Dout*, and all data and spacers inside the FIFO are shifted one place towards the output stage. When the *read* line becomes "0," the next spacer moves to the output and again all data and spacers are shifted towards the output.

The FIFO *F1* (Figure 2) holds the n next instructions. Instructions are put on this queue by *IMEM* and are taken out by the data processor *DP*. The length of this FIFO queue is unspecified.

FIFO *F2* is of length 3. The data processor puts the bit COND into it, and the bit is stored until the branch processor takes it out. If *F2* were of length 2 only, *DP* would be delayed after writing into it and until the value was drawn by *BP*. At length 3, *DP* and *BP* are decoupled.

FIFOs *F3* and *F4* are of length 2 each. They connect *IMEM* and *BP*, and implement the fork on channel *I*, as shown in Figure 11. *F3* transfers the instruction word from *IMEM* to *BP*, while *F4* transfers the next address from *BP* to *IMEM*. Once an instruction has been received by *F3* (and *F1*), the address is removed from channel *A*. Once the instruction is removed from channel *I*, the next address is provided. Channels *A* and *I* are $ack-less$, and the whole structure is designed to speed up instruction fetches.

8. Delayed Branch

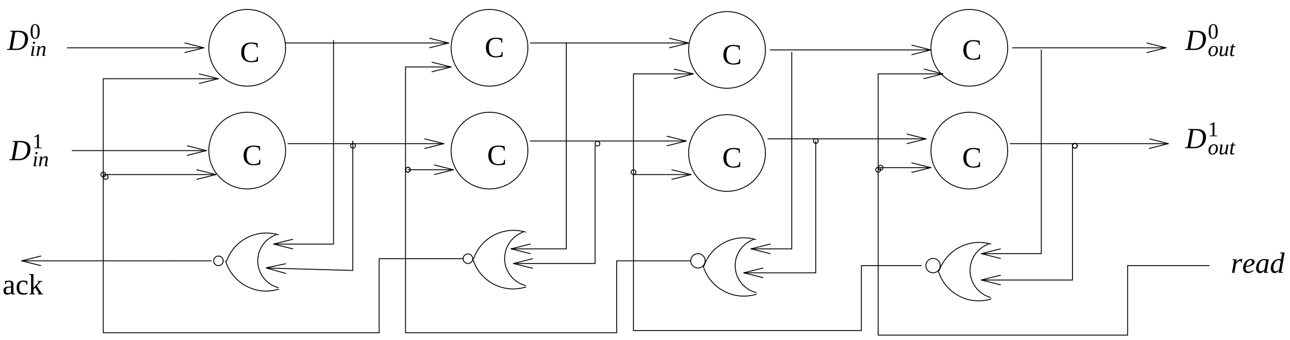

Figure 10: A four-stage self-timed FIFO

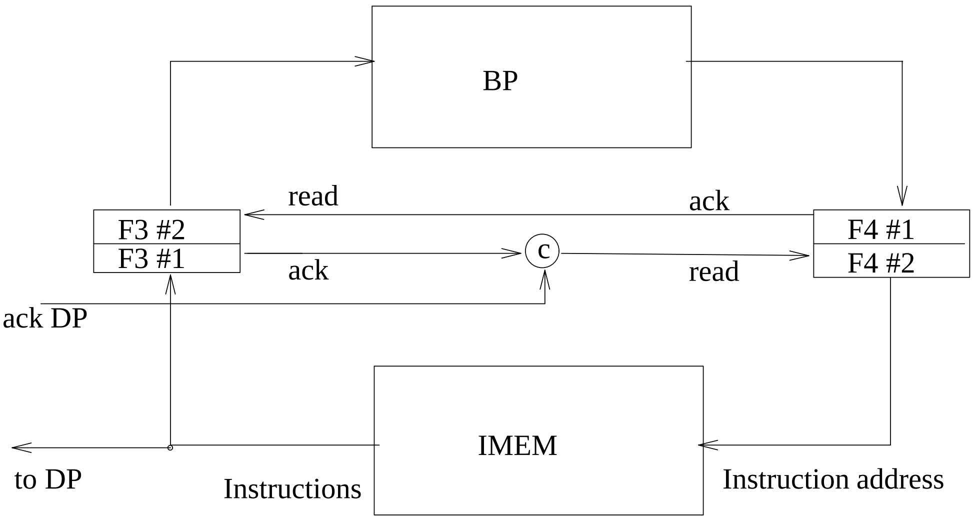

Figure 11: Instruction fetch and fork FIFO organization

Delayed branches have been devised for synchronous processors to eliminate pipeline suspensions due to branches [Ka84]. Asynchronous processors may suffer similar performance degradation. In this section we show how a novel form of delayed branches, and a modified branch processor, improve the ST-RISC efficiency.

With delayed branches, compilers for synchronous systems delay the execution of branch instructions for a constant number of branch-independent instructions. In ST-RISC architecture, the execution of the branch instruction can be delayed for a *variable* number of instructions. The compiler can relocate any number of instructions which do not affect the branch by moving them beyond the branch instruction. The last relocated instruction is marked with a trailer bit. The purpose of this variable-delay delayed branch is to defer the branch as long as is permissible, in hope that by the time the branch is to be executed all relevant conditions will have already been computed and transferred to the branch processor.

The structure of the branch processor is changed as follows (Figures 12-14). A register (*BR*) is added to save the branch address (*BADR*). In addition, the decoder *BDEC* checks the *trailer* bit ($d.t$ in Figures 13 and 14) in each instruction. If this bit is on, *BP* must wait for the *CONDI* bit coming from *DP* before it continues its operation. According to the value of *CONDI*, *BCOND* instructs the *PC* to either load the next address from *BROUT* or just increment the address.

Note that for formal verification of this modified architecture versus the program of Figure 1, the latter has to be modified as well. On the other hand, note that the modular design of ST-RISC allows a substantial modification of *BP* without even touching *DP*.

9. Discussion and Conclusions

We have presented a complete hierarchical architecture of an advanced self-timed processor. Three attributes make ST-RISC self-timed: First, it is asynchronous, i.e. there is no clock to determine timing. Second, it is delay insensitive, i.e. no assumption is made regarding the delays in either the components or the wires. Third, the system, as well as each and every sub-system, generate completion signals. ST-RISC is based on double-rail implementation of ternary logic. Advantages of self-timed architectures include the avoidance of clock-related problems such as clock skews and synchronization failures, and the simplification of the complex task of timing design and verification. In addition, while synchronous systems are timed according to worst propagation delays, self-timed elements may work faster on the average since they signal when results are ready.

The guiding principles in the design of ST-RISC include the elimination of serial bottlenecks, the decomposition into as many concurrently operating modules as possible, and the reduction of interdependencies among the modules. Nine architectural and design features characterize ST-RISC. Most of them are applied to self-timed processor design for the first time:

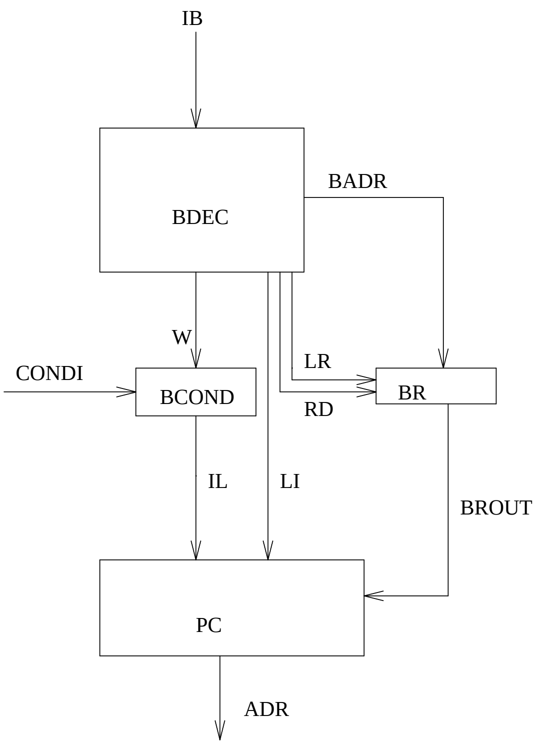

Figure 12. The delayed-branch branch processor (substituting Figure 8)

$$
\begin{array}{l}
bp \equiv *[\ IB\ ?\ d;\\
\quad [\ \neg d.t \angle \neg branch(d.op) \rightarrow pc := pc + 1\\
\quad\ [\,]\ \neg d.t \angle\ jump\ (d.op) \rightarrow pc := d.addr\\
\quad\ [\,]\ \neg d.t \angle cbranch(d.op) \rightarrow br := d.addr\ ,\ pc := pc + 1\\
\\
\quad\ [\,]\ d.t \rightarrow CONDI?c\ \ ;[\ \neg c \rightarrow pc := pc + 1\\
\qquad\qquad\qquad\qquad\qquad\quad c \rightarrow pc := br]\ ]\\
\quad ADR\ !\ pc]
\end{array}
$$

Figure 13. The delayed-branch branch processor program (substituting Figure 4)

$$bdec \equiv \ast[\ IB\ ?\ d;$$

$$\begin{array}{l}
[\neg d.t \angle \neg jump(d.op) \angle \neg cbranch(d.op) \rightarrow LI\ !\ \text{'}incr\text{'} \\
\ [\,]\ \neg d.t \angle\ jump\ (d.op) \rightarrow LI\ !\ \text{'}load\text{'}\ ,\ LR\ !!\ \text{'}l\text{'}\ ,\ RD\ !!\ \text{'}r\text{'}\ ,\ BADR\ !!\ d.addr \\
\ [\,]\ \neg d.t \angle\ cbranch\ (d.op) \rightarrow LI\ !\ \text{'}incr\text{'}\ ,\ LR\ !!\ \text{'}l\text{'}]\ ] \\
\ [\,]\ d.t \rightarrow W\ !\ ,\ RD\ !!\ \text{'}r\text{'}]\ ]
\end{array}$$

$$bcond \equiv \ast[\ [W\ ?\ ,\ CONDI\ ?\ cond];$$

$$\begin{array}{l}
[cond \ \rightarrow\ IL\ !\ \text{'}load\text{'} \\
\ [\,]\ \neg cond \ \rightarrow\ IL\ !\ \text{'}incr\text{'}]\ ]
\end{array}$$

$$pc \equiv \ast[\ [\ [LI\ ?\ il]$$

$$\begin{array}{ll}
[\,]\ [IL\ ?\ il]\ ]\ ; & [incr(il) \rightarrow pc := pc + 1 \\
 & [\,]\, load(il) \rightarrow BROUT\ ??\ pc\ ]\ ;\quad ADR\ !\ pc]
\end{array}$$

$$br \equiv \ast[\ LR\ ??\ l \rightarrow BADR\ ??\ b\ ,\ RD\ ??\ r \rightarrow BROUT\ !!\ b\ ]$$

Figure 14. The processes of the delayed-branch branch processor (substituting Figure 9)

(1) The processor is decomposed into separate *data processor* and *branch processor*. They intercommunicate only during conditional branch instructions. This is similar to the decomposition of synchronous processors into instruction and execution units.

(2) Delayed branches are introduced. Unlike synchronous processors, we take advantage of the increased flexibility and modularity of this architecture and achieve a *variable* length of delay.

(3) In contrast with most other architectures, data-path and control are not separated. While such separation is well understood for general purpose processors, it is a rather special case and is not suitable for general architectural synthesis from behavioral specifications. The

unified architecture resembles the data flow approach.

(4) Novel self-timed FIFO buffers are used to decouple the various parts of the processor.

(5) In addition to conventional four-phase handshaking channels, we employ $ack-less$ channels to remove redundant control signals. They are enclosed within sub-systems that guarantee correct operation. This helps alleviate one of the difficulties of self-timed designs, namely handshaking overhead.

(6) Most channels are point-to-point. When a fork is required, it is achieved with complete self-timed signaling.

(7) In accordance with the above point, ST-RISC does not use any global shared bus to connect the various elements. This helps to avoid possible bottlenecks.

(8) The design is modular. As a result, both types of branch processors (with and without delayed branches) can be used with exactly the same data processor.

(9) The processor is specified hierarchically by functional programs, written in a CSP-like language (adopted from [Ma89b]). This is done in order to facilitate both automatic synthesis and formal verification.

The pioneering work in this field was done by Martin. An asynchronous processor was designed and fabricated [Ma89a], applying novel synthesis and compilation techniques [Ma89b]. Martin describes the communicating processes rather elegantly with the aid of a CSP-like language. While the specification language and the compilation techniques are quite effective, the computer architecture is very basic. Compared to the points above, there is but one processing unit, there are no delayed branches, control is separated from the data path, the various components are tightly coupled, and a global shared bus is used. The more advanced features include the use of probes and lazy-active channels to enhance efficiency.

The goals of this research are twofold - to investigate efficient self-timed processor architectures, and to strive towards automatic synthesis and formal verification of such architectures. ST-RISC is an advanced architecture, and it has been designed and described in an orderly, formal, hierarchical manner, amenable to both algorithmic generation and formal verification.

Appendix: The Program Notation

The first part of our program notation, inspired by Hoare's CSP [Ho78], is taken over from [Ma89b].

The *selection* command $[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$ where G_1 ... G_n are predicates or ‘‘guards’’ (boolean expressions) and S_1 ... S_n are program parts, has the following meaning: wait until one of the G_i holds, then execute the program S_i. In our self-timed systems, guards are true mutually exclusive.

The *repetition* command: $*[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$, where G_1 ... G_n are predicates (boolean expressions) and S_1 ... S_n are program parts, has the following meaning: repeatedly select some i for which G_i holds, then execute S_i. If none of the G_i holds, the repetition command ends.

$*[S]$ stands for "repeat S forever"; $[G]$ stand for "wait until G holds" (this is equivalent to $[G \rightarrow skip]$).

$S1 ; S2$ stands for execute program $S1$, then execute $S2$; $S1 , S2$ means "execute $S1$ and $S2$ in any order".

The following communication commands are particularly tailored towards our self-timed double-rail system.

The processes communicate via double-rail *data* lines and single-rail *acknowledgment* lines. Let process P1 communicate with process P2 via channel X. Let *a* be a variable of process P1 and *b* a variable of process P2. We denote the following output sequence produced by process P1 as $X\ !\ a$:

$$wait \text{ for } ack = 0;\ set\ X := a;\ wait \text{ for } ack = 1;\ set\ X\ undefined;$$

We denote the following input sequence by process P2 as $X\ ?\ b$:

$$wait \text{ for } X\ be\ defined;\ set\ b := X;\ set\ ack = 1;\ wait \text{ for } X\ become\ undefined;\ set\ b\ undefined;\ set\ ack = 0$$

X! stands for X!ϕ where ϕ is any defined value.

X? is defined accordingly; the defined value ϕ is not stored.

Communication between components can take place without acknowledgement signals. We denote as $X\ !!\ a$ the following output sequence produced by process P1:

$$When\ a\ becomes\ defined,\ set\ X := a;\ when\ a\ is\ undefined,\ set\ X\ undefined$$

Similarly, we denote as $X\ ??\ b$ the following input sequence by process P2:

$$When\ X\ becomes\ defined,\ set\ b := X;\ when\ X\ becomes\ undefined,\ set\ b\ undefined$$

References

[An86] Anantharaman T.S., ‘‘A Delay Insensitive Regular Expression Recognizer,’’ *IEEE VLSI Technical Bulletin*, September 86.

[Chu87] Chu, T.A., ‘‘Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications’’, PhD thesis, MIT, 1987.

[DGY92a] David I., Ginosar R. and Yoeli M., ‘‘An Efficient Implementation of Boolean Functions as Self-Timed circuits,’’ IEEE Trans. on Computers, January 1992, pp 2-11.

[DGY92b] David I., Ginosar R. and Yoeli M., ‘‘Implementing Sequential Machines as Self-Timed Circuits,’’ IEEE Trans. on Computers, January 1992, pp 12-17.

[Da89] David I., ‘‘The self-timed FIFO’’ Technical Report No. 731, Dept. Elect. Eng., Technion, Oct. 1989.

[DGY89b] David I., Ginosar R. and Yoeli M., ‘‘An Efficient Implementation of Boolean Functions and Finite State Machines as Self-Timed Circuits’’, Computer Architecture News (CAN), pp. 91-104, Dec 89.

[Eb87] Ebergen J. C., ‘‘Translating Programs into Delay-Insensitive Circuits’’, Ph.D. Thesis ,Eindhoven University of Technology, 1987.

[Ha84] D.S. Ha and S.M. Reddy, ‘‘On Testable Self-timed Logic Circuits,’’ Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers ICCD ’84, pp. 296-301, 1984.

[Ho78] Hoare C.A.R, ‘‘Communicating Sequential Processes’’, Communications of the ACM, Vol. 21 No. 8, pp. 666-677, August 1978.

[Ka84] Katevenis M.G.H., ‘‘Reduced Instruction Set Computer for VLSI,’’ The MIT Press, 1984.

[La87] Lau, C. H., ‘‘Self: a Self-Timed Systems Design Technique,’’ Electronic Letters, Vol. 23, No. 6, March 1987, pp. 269-270.

[Ma86] Martin A. J., ‘‘Compiling Communicating Processes into Delay Insensitive VLSI circuits,’’ in *Distributed Computing*, vol. 1, no 3, 1986.

[Ma89a] Martin A. J., S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus, ‘‘The design of an Asynchronous Microprocessor’’, Caltech-CS-TR-89-02, Computer Science Department, California Institute of Technology, 1989, and Proc. Caltech Conf. on VLSI, 1989.

[Ma89b] Martin A. J., ‘‘Programming in VLSI: From communicating processes to delay-insensitive circuits,’’ in ‘‘UT Year of Programming Inst. on Concurrent Programming,’’ C.A.R. Hoare (ed.), Addison-Wesley, 1989.

[MFR85] Molnar, C.E., Fan, T.P., and Rosenberger, F.U., ‘‘Synthesis of Delay-Insensitive Modules,’’ Journal of Distributed Computing, Vol.1, 1986, pp. 226-234.

[Rem85] Rem M., ‘‘Concurrent Computations and VLSI Circuits’’, in *Control Flow and Data Flow; Concepts of Distributed Computing*, (M. Broy ed.), Springer-Verlag, pp. 399-437, 1985.

[Se80] Seitz, C.L., ‘‘System Timing,’’ in C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980, pp. 218-262.

[Si81] Singh N. P., ''A Design Methodology for Self-Timed Systems,'' M.Sc. Thesis, MIT Laboratory for Computer Science Technical Report TR-258, MIT, Cambridge, Mass., February 1981.

[Sn85] Van de Snepscheut, J. L. A., *Trace Theory and VLSI Design*, LNCS 200, 1985.