The Design of an Asynchronous Microprocessor

Alain J. Martin

Computer Science Department
California Institute of Technology

Caltech-CS-TR-89-2

The Design of an Asynchronous Microprocessor

Alain J. Martin, Steven M. Burns, T.K. Lee,
Drazen Borkovic, Pieter J. Hazewindus

California Institute of Technology
Pasadena CA 91125, USA

to appear in *Proc. Decennial Caltech Conference on VLSI*, 20-22 March, 1989, MIT Press
Caltech-CS-TR-89-2

1 Introduction

Prejudices are as tenacious in science and engineering as in any other human activity. One of the most firmly held prejudices in digital VLSI design is that asynchronous circuits—a.k.a. self-timed or delay-insensitive circuits—are necessarily slow and wasteful in area and logic. Whereas asynchronous techniques would be appropriate for control, they would be inadequate for data paths because of the cost of dual-rail encoding of data, the cost of generating completion signals for write operations on registers, and the difficulty of designing self-timed buses.

Because a general-purpose microprocessor contains a complex data path, a corollary of the previous opinion is that it is impossible to design an efficient asynchronous microprocessor. Since we have been developing a design method for asynchronous circuits that gives excellent results, and since the above objections to large-scale data path designs are genuine but untested, we decided to "pick up the gauntlet" and design a complete processor.

The design of an asynchronous microprocessor poses new challenges and opens new avenues to the computer architect. Hence, the experiment unavoidably developed a dual purpose: We are refining an already well-tested design method, and we are starting a new series of experiments in asynchronous architectures. (As far as we know, this is the first entirely asynchronous microprocessor ever built.) The results we are reporting have a different implication depending on whether they are related to the first or second goal of the experiment. Whereas we are convinced that our design methods have reached maturity, we are quite aware that asynchronous techniques may influence the computer architects in completely new ways that this first design is just starting to explore.

In order to focus the experiment on asynchronous circuit design, we have intentionally excluded optimizations at the high and low ends of the design process. The instruction set is straightforward and no assumption has been made on the code produced by the compiler. No special electrical optimizations other than transistor sizing have been applied; the circuit techniques rarely go beyond those taught in a graduate-level VLSI class, and, apart from the memory interfaces, the circuits are ***delay-insensitive***. Hence, any performance is to be attributed to the design method and to the inherent advantages of asynchronous design.

A circuit is delay-insensitive when its correct operation is independent of any assumption on delays in operators and wires except that the delays be finite. Such circuits do not use a clock signal or knowledge about delays: Sequencing is enforced entirely by communication mechanisms.

The class of entirely delay-insensitive circuits is very limited. Different asynchronous techniques distinguish themselves in the choice of the compromises to delay-insensitivity. ***Speed-independent*** techniques assume that delays in gates are arbitrary, but there are no delays in wires. ***Self-timed*** techniques assume that a circuit can be decomposed into ***equipotential*** regions inside which delays in wires are negligible[11].

In our method, certain local forks are introduced to distribute a variable as inputs of several gates. We assume that the difference between the delays in the branches of such forks are short compared to delays in other gates. We call such forks ***isochronic***[6], [8].

The general method—a complete description of which can be found in the referenced papers [2], [5], [6], [7], [8]—is based on program transformations. The circuit is first designed as a set of concurrent programs. Each program is then compiled (manually or automatically) into a circuit by applying a series of program transformations. Control and data path are first designed separately and then combined in a mechanical way. This important ***divide-and-conquer*** technique is a main innovation of the method.

2 Preliminary Results

As of this writing, the first design is complete, and has been scheduled for fabrication in $2\mu m$ MOSIS SCMOS. The chip was functionally simulated using COSMOS [1], and was found to be functionally correct.

The architecture is a 16-bit processor with offset and a simple instruction set of the RISC type [4]. The data path contains twelve 16-bit registers, four buses, an ALU, and two adders. The chip contains 20,000 transistors and fits within a 5500λ by 3500λ area. We are using an 84-pin $6600\mu m \times 4600\mu m$ frame. An estimate of the critical path suggests processor performance of approximately 15MIPS in $2\mu m$ SCMOS. (A slightly improved $1.6\mu m$ SCMOS version is also being fabricated.)

This experiment, the most challenging one we have conducted so far, promised to be an important test for our method. The results obtained so far have been very encouraging.

The technique for separating control and data path has been extended with a novel asynchronous bus design, and is now robust and general.

The handshaking protocol between circuit elements has also been modified so that half of a protocol sequence overlaps subsequent actions. This protocol makes it possible to "hide" half of delays of the completion trees, the tree of gates that combine the completion signals from the asynchronous elements. In addition, at most two completion trees are in sequence on any path. Thus, completion tree delays are not a serious disadvantage of asynchronous design.

Instruction pipelining has been approached as a concurrent programming problem: Starting with a sequential program for the processor, concurrency is introduced through a series of program transformations. However, although the transformations are guided by the intent to overlap the important phases—fetch, decode, execute—of instruction execution, they are neither mechanical nor unique. The designer decides how to decompose a program into several concurrent ones. We do not claim that our solution in this first design is in any way optimal.

3 Specification of the Processor as a Sequential Program

The instruction set is deliberately not innovative. It is a conventional 16-bit-word instruction set of the ***load-store*** type. The processor uses two separate memories for instructions and data. There are three types of instructions: ALU, memory, and program-counter (***pc***). All ALU instructions operate on registers; memory instructions involve a register and a data memory word. Certain instructions use the following word as ***offset***. (See Table 1 in Appendix 2.)

$$
\begin{array}{l}
*[FETCH : i, pc := imem[pc], pc + 1; \\
\qquad [offset(i.op) \rightarrow offset, pc := imem[pc], pc + 1; \\
\qquad [] \neg offset(i.op) \rightarrow skip \\
\qquad]; \\
\ EXECUTE : [alu(i.op) \rightarrow \langle reg[i.z], f \rangle := aluf(reg[i.x], reg[i.y], i.op, f) \\
\qquad\quad [] ld(i.op) \rightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]] \\
\qquad\quad [] st(i.op) \rightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z] \\
\qquad\quad [] ldx(i.op) \rightarrow reg[i.z] := dmem[offset + reg[i.y]] \\
\qquad\quad [] stx(i.op) \rightarrow dmem[offset + reg[i.y]] := reg[i.z] \\
\qquad\quad [] lda(i.op) \rightarrow reg[i.z] := offset + reg[i.y] \\
\qquad\quad [] stpc(i.op) \rightarrow reg[i.z] := pc \\
\qquad\quad [] jmp(i.op) \rightarrow pc := reg[i.y] \\
\qquad\quad [] brch(i.op) \rightarrow [cond(f, i.cc) \rightarrow pc := pc + offset \\
\qquad\qquad\qquad\qquad [] \neg cond(f, i.cc) \rightarrow skip \\
\qquad\qquad\qquad\qquad] \\
\qquad\quad] \\
].
\end{array}
$$

Figure 1: Sequential program describing the processor

The only important omissions, those of an interrupt mechanism and communication ports, are ones we found to be unnecessary distractions in a first design.

The sequential program describing the processor is a non-terminating loop, each step of which is a *FETCH* phase followed by an *EXECUTE* phase. The complete sequential program for the processor is shown in Figure 1. (The notation, which is an extension of the one we have used in previous work, is described in Appendix 1.) Variable i, which contains the instruction currently being executed, is described in the PASCAL record notation as a structured variable consisting of

several fields. All instructions contain an ***op*** field for the ***opcode***. The parameter fields depend on the types of the instructions, which are found in Table 2 in Appendix 2. The most common ones, those for ALU, load, and store instructions, consist of the three parameters, x, y, and z. Variable ***cc*** contains the condition code field of the branch instruction, and f contains the ***flags*** generated by the execution of an ***alu*** instruction.

The two memories are the arrays ***imem*** and ***dmem***. The index to ***imem*** is the program-counter variable, ***pc***. The general-purpose registers are described as the array ***reg***[0...15]. (Only twelve registers are implemented in the first chip.) Register ***reg***[0] is special: It always contains the value zero.

4 Decomposition into Concurrent Processes

We decompose the previous program into a set of concurrent processes that communicate and synchronize using communication commands on channels. A restricted form of shared variables is allowed. The control channels ***Xs***, ***Ys***, ***ZAs***, ***ZWs***, ***ZRs***, and the bus ***ZA*** are one-to-many; the buses ***X***, ***Y***, ***ZM*** are many-to-many; the other channels are one-to-one. But all channels are used by only two processes at a time. The structure of processes and channels is shown in Figure 2. The final program is shown in Figures 3 and 4.

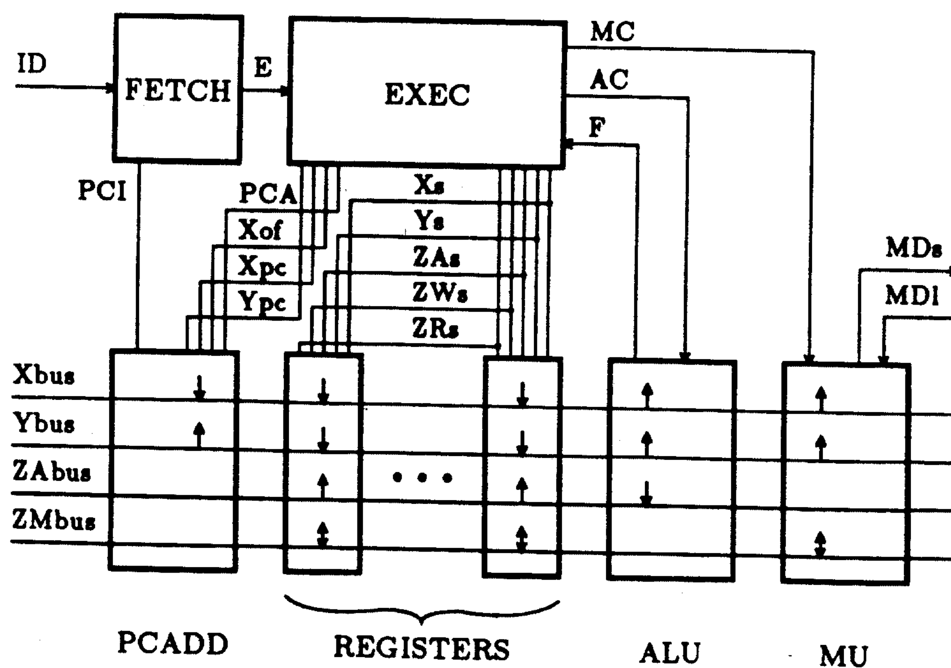

Figure 2: Process and channel structure

```
IMEM ≡ *[ID!imem[pc]]
FETCH ≡ *[PCI1; ID?i; PCI2;
            [offset(i.op) → PCI1; ID?offset; PCI2
            ▯¬offset(i.op) → skip
            ]; E1!i; E2
          ]
PCADD ≡ (*[[‾PCI1‾ → PCI1; y := pc + 1; PCI2; pc := y
             ▯‾PCA1‾ → PCA1; y := pc + offset; PCA2; pc := y
             ▯‾Xpc‾ → X!pc • Xpc
             ▯‾Ypc‾ → Y?pc • Ypc
             ]]
          ‖*[[‾Xof‾ → X!offset • Xof]]
          )
EXEC ≡ *[E1?i;
          [alu(i.op) → E2; Xs • Ys • AC!i.op • ZAs
          ▯ld(i.op) → E2; Xs • Ys • MC1 • ZRs
          ▯st(i.op) → E2; Xs • Ys • MC2 • ZWs
          ▯ldx(i.op) → Xof • Ys • MC1 • ZRs; E2
          ▯stx(i.op) → Xof • Ys • MC2 • ZWs; E2
          ▯lda(i.op) → Xof • Ys • MC3 • ZRs; E2
          ▯stpc(i.op) → Xpc • Ys • AC!add • ZAs; E2
          ▯jmp(i.op) → Ypc • Ys; E2
          ▯brch(i.op) → F?f; [cond(f, i.cc) → PCA1; PCA2
                              ▯¬cond(f, i.cc) → skip
                              ]; E2
          ]
```

Figure 3: The final program, first part

$$
\begin{aligned}
ALU \equiv\ & *[[\overline{AC} \rightarrow AC?op \bullet X?x \bullet Y?y; \\
& \qquad \langle z, f\rangle := aluf(x, y, op, f); ZA!z \\
& \ [\!]\ \overline{F} \rightarrow F!f \\
&]] \\
MU \equiv\ & *[[\overline{MC1} \rightarrow X?x \bullet Y?y \bullet MC1; ma := x + y; MDl?w; ZM!w \\
& \ [\!]\ \overline{MC2} \rightarrow X?x \bullet Y?y \bullet MC2 \bullet ZM?w; ma := x + y; MDs!w \\
& \ [\!]\ \overline{MC3} \rightarrow X?x \bullet Y?y \bullet MC3; ma := x + y; ZM!ma \\
&]] \\
DMEM \equiv\ & *[[\overline{MDl} \rightarrow MDl!dmem[ma] \\
& \ [\!]\ \overline{MDs} \rightarrow MDs?dmem[ma] \\
&]] \\
REG[k] \equiv\ & (*[[\neg bk \wedge k = i.x \wedge \overline{Xs} \rightarrow X!r \bullet Xs]] \\
& \| *[[\neg bk \wedge k = i.y \wedge \overline{Ys} \rightarrow Y!r \bullet Ys]] \\
& \| *[[\neg bk \wedge k = i.z \wedge \overline{ZWs} \rightarrow ZM!r \bullet ZWs]] \\
& \| *[[\neg bk \wedge k = i.z \wedge \overline{ZAs} \rightarrow bk\uparrow; ZAs; ZA?r; bk\downarrow]] \\
& \| *[[\neg bk \wedge k = i.z \wedge \overline{ZRs} \rightarrow bk\uparrow; ZRs; ZM?r; bk\downarrow]] \\
&)
\end{aligned}
$$

Figure 4: The final program, second part

Process *FETCH* fetches the instructions from the instruction memory, and transmits them to process *EXEC* which decodes them. Process *PCADD* updates the address *pc* of the next instruction concurrently with the instruction fetch, and controls the *offset* register. The execution of an ALU instruction by process *ALU* can overlap with the execution of a memory instruction by process *MU*. The *jump* and *branch* instructions are executed by *EXEC*; *store-pc* is executed by the ALU as the instruction "add the content of register *r* to the pc and store it." The array *REG*[*k*] of processes implements the register file. Both *MU* and *PCADD* contain their own adder. Processes *IMEM* and *DMEM* describe the instruction memory and data memory, respectively.

Updating the PC

The variable *pc* is updated by process *PCADD*, and is used by *IMEM* as the index of the array *imem* during the *ID* communication—the instruction fetch.

The assignment $pc := pc{+}1$ is decomposed into $y := pc{+}1; pc := y$, where *y* is a local variable of *PCADD* . The overlap of the instruction fetch, *ID*? (either *ID?i* or *ID?offset*), and the pc increment, $y := pc + 1$, can now occur while *pc* is constant. Action *ID*? is enclosed between the two communication actions *PCI1* and *PCI2*, as follows:

$$PCI1; ID?i; PCI2 \ .$$

In *PCADD*, $y := pc + 1$ is enclosed between the same two communication actions while the updating of *pc* follows *PCI2*:

$$\overline{PCI1} \rightarrow PCI1; y := pc + 1; PCI2; pc := y \ .$$

Since the completions of *PCI1* and *PCI2* in *FETCH* coincide with the completion of *PCI1* and *PCI2* in *PCADD*, respectively, the execution of *ID?i* in *FETCH* overlaps the execution of $y := pc + 1$ in *PCADD*. *PCI1* and *PCI2* are implemented as the two halves of the same communication handshaking to minimize the overhead.

In order to concentrate all increments of *pc* inside *PCADD*, we use the same technique to delegate the assignment $pc := pc + offset$ (executed by the *EXEC* part in the sequential program) to *PCADD*.

The guarded command $\overline{Xof} \rightarrow Xof!offset$ in *PCADD* has been transformed into a concurrent process since it needs only be mutually exclusive with assignment $y := x + offset$, and this mutual exclusion is enforced by the sequencing between $PCA1; PCA2$ and *Xof* within *EXEC*.

5 Stalling the Pipeline

When the pc is modified by *EXEC* as part of the execution of a *pc* instruction, (***store-pc***, ***jump*** or ***branch***), fetching the next instruction by *FETCH* is postponed until the correct value of the pc is assigned to *PCADD.pc*.

When the offset is reserved for *MU* by *EXEC*, as part of the execution of some memory instructions, fetching the next instruction, which might be a new offset, is postponed until *MU* has received the

value of the current offset. In the second design, we have refined the protocol to block ***FETCH*** only when the next instruction is a new offset.

Postponing the start of the next cycle in ***FETCH*** is achieved by postponing the completion of the previous cycle, i.e., by postponing the completion of the communication action on channel E. As in the case of the PCI communication, E is decomposed into two communications, $E1$ and $E2$. Again, $E1$ and $E2$ are implemented as the two halves of the same handshaking protocol.

In ***FETCH***, $E!i$ is replaced with $E1!i; E2$. In ***EXEC***, $E2$ is postponed until after either ***Xof?offset*** or a complete execution of a ***pc*** instruction has occurred.

6 Sharing Registers and Buses

A bus is used by two processes at a time, one of which is a register and the other is ***EXEC***, ***MU***, ***ALU***, or ***PCADD***. We therefore decided to introduce enough buses so as not to restrict the concurrent access to different registers. For instance, ***ALU*** writing a result into a register should not prevent ***MU*** from using another register at the same time.

The four buses correspond to the four main concurrent activities involving the registers.

The X bus and the Y bus are used to send the parameters of an ALU operation to the ALU, and to send the parameters of address calculation to the memory unit. We also make opportunistic use of them to transmit the pc and the offset to and from ***PCADD***.

The ZA bus is used to transmit the result of an ALU operation to the registers. The ZM bus is used by the memory unit to transmit data between the data memory and the registers.

We make a virtue out of necessity by turning the restriction that registers can be accessed only through those four buses into a convenient abstraction mechanism. The ALU uses only the X, Y, and ZA ports without having to reference the particular registers that are used in the communications. It is the task of ***EXEC*** to reserve the X, Y, and ZA bus for the proper registers before the ALU uses them.

The same holds for the ***MU*** process, which references only X, Y, and ZM. An additional abstraction is that the X bus is used to send the offset to ***MU***, so that the cases for which the first parameter is $i.x$ or ***offset*** are now identical, since both parameters are sent via the X bus.

Exclusive Use of a Bus

Commands *Xpc*, *Ypc*, and *Xof* are used by *EXEC* to select the X and Y buses for communication of *pc* and *offset*. Commands *Xs*, *Ys*, and *ZAs* are used by *EXEC* to select the X, Y, and *ZA* buses, respectively, for a register that has to communicate with the ALU as part of the execution of an ALU instruction.

Two commands are needed to select the *ZM* bus: *ZWs* if the bus is to be used for writing to the data memory, and *ZRs* if the bus is to be used for reading from the data memory.

Let us first solve the problem of the mutual exclusion among the different uses of a bus. As long as we have only one ALU and one memory unit, no conflict is possible on the *ZA* and *ZM* buses, since only the ALU uses the *ZA* bus, and only the memory unit uses the *ZM* bus. But the X and Y buses are used concurrently by the ALU, the memory unit, and the pc unit.

We achieve mutual exclusion on different uses of the X bus as follows. (The same argument holds for Y.) The completion of an X communication is made to coincide with the completion of one of the selection actions *Xs*, *Xof*, *Xpc*; and the occurrences of these selection actions exclude each other in time inside *EXEC* since they appear in different guarded commands.

This coincidence is implemented by the *bullet* (•) command : For arbitrary communication commands U and V inside the same process, $U \bullet V$ guarantees that the two actions are completed at the same time. We then say that the two actions coincide. The use of the bullets *X*!*pc* • *Xpc* and *X*!*offset* • *Xof* inside *PCADD* , and *X*!*r* • *Xs* inside the registers enforce the coinidence of X with *Xpc*, *Xof*, and *Xs*, respectively. The bullets in *EXEC*, *ALU*, and *MU* have been introduced for reasons of efficiency: Sequencing is avoided.

7 Register Selection

Command *Xs* in *EXEC* selects the X bus for the particular register whose index k is equal to the field $i.x$ of the instruction i being decoded by *EXEC*, and analogously for commands *Ys*, *ZAs*, *ZRs*, and *ZWs*.

Each register process *REG*[k], for $0 \leq k < 16$, consists of five elementary processes, one for each selection command. The register that is selected by command *Xs* is the one that passes the test $k = i.x$. This implementation requires that the variable $i.x$ be shared by all

registers and *EXEC*. An alternative solution that does not require shared variables uses demultiplexer processes. (The implementations of the two solutions are almost identical.)

The semicolons in the last two guarded commands of *REG*[*k*] are introduced to pipeline the computation of the result of an ALU instruction or memory instruction with the decoding of the next instruction.

Mutual Exclusion on Registers

A register may be used in several arguments (x, y, or z) of the same instruction, and also as an argument in two successive instructions whose executions may overlap. We therefore have to address the issue of the concurrent uses of the same register. Two concurrent actions on the same register are allowed when they are both read actions.

Concurrency within an instruction is not a problem: X and Y communications on the same register may overlap, since they are both read actions, and Z cannot overlap with either X or Y because of the sequencing inside *ALU* and *MU*.

Concurrency in the access to a register during two consecutive overlapping instructions (one instruction is an ALU and the other is a memory instruction) can be a problem: Writing a result into a register (a ZA or a ZR action) in the first instruction can overlap with another action on the same register in the second instruction. But, because the selection of the z register for the first instruction takes place before the selection of the registers for the second instruction, we can use this ordering to impose the same ordering on the different accesses to the same register when a ZA or ZR is involved.

This ordering is implemented as follows: In *REG*[*k*], variable bk (initially false) is set to true before the register is selected for ZA or ZR, and it is set back to false only after the register has been actually used. All uses of the register are guarded with the condition $\neg bk$. Hence, all subsequent selections of the register are postponed until the current ZA or ZR is completed.

We must ensure that bk is not set to true before the register is selected for an X or a Y action ***inside the same instruction***, since this would lead to deadlock. We omit this refinement which does not appear in the program of Figures 3 and 4.

8 Implementation

Control Part

The control part of a process is obtained by the following transformations: First, each communication command involving message input or output is replaced with a "bare" communication on the channel; for instance, $C?x$ and $C!x$ would both be replaced with C.

Second, all assignment statements are delegated to subprocesses. Assignment S is replaced with a communication command on a new channel, say Cs, and the subprocess $*[[\overline{Cs} \rightarrow S \bullet Cs]]$ is introduced. After these transformations, the control part of each process consists only of boolean expressions in conditionals and of communication commands. Thus, the next step is to implement each communication command with a ***handshaking protocol***.

Handshaking Protocols

Consider the matching pair of actions $X!u$ and $X?v$ in processes A and B respectively. We first implement the bare communication on channel X. The channel is implemented by the two handshake wires $(xo \ \underline{w} \ yi)$ and $(yo \ \underline{w} \ xi)$ as indicated on Figure 5.(a). As usual, we use a four-phase, or "return-to-zero" handshaking protocol. Such a protocol is not symmetrical: All communications in one process are implemented as ***active*** and all communications in the other process as ***passive***.

We have shown in [7] and [8] that the implementation of an input action is significantly simpler when combined with an active protocol than with a passive one. Therefore all input actions are implemented as active and all output actions as passive. (In the case of output, the implementation of communication is the same for active and passive protocols.)

The standard active and passive implementations are:

$$[yi]; yo\uparrow; [\neg yi]; yo\downarrow \quad \text{(passive)}$$

$$xo\uparrow; [xi]; xo\downarrow; [\neg xi] \quad \text{(active)} .$$

(The passive protocol starts with the wait action $[yi]$, i.e., "wait until the input wire is set to true." The active protocol starts with $xo\uparrow$, i.e., "set the output wire to true.")

We introduce an alternative active implementation, called ***lazy active***:

$$[\neg xi]; xo\uparrow; [xi]; xo\downarrow \quad \textbf{(lazy active)} .$$

The lazy active protocol differs from the active one in that the last wait action $[\neg xi]$ is postponed until the beginning of the next communication. The difference is important when data communication is involved.

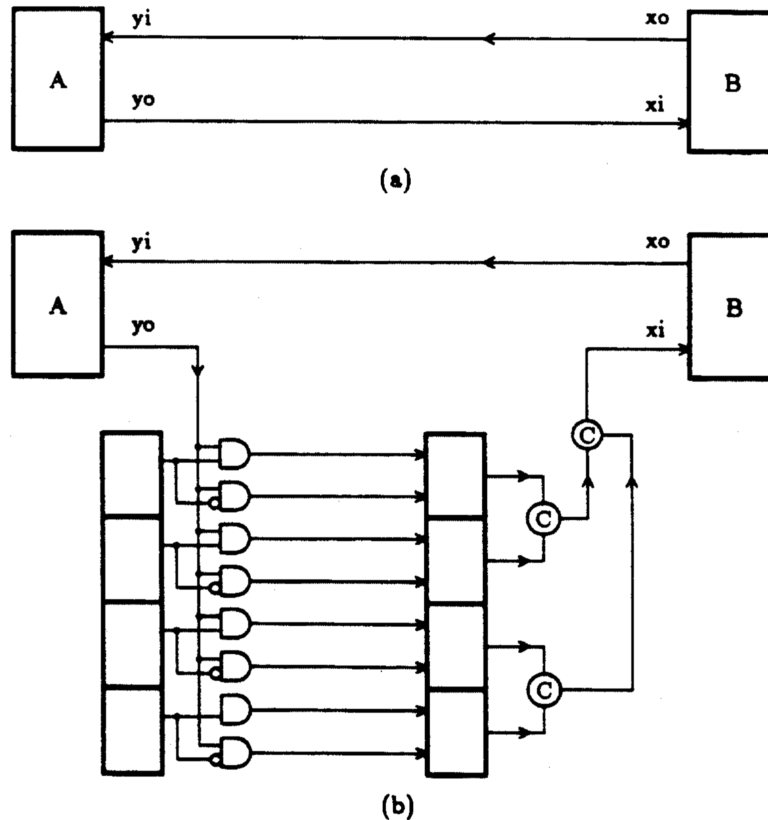

Figure 5: Implementation of communication

Figure 5.(b) shows how the data path is combined with the control. The bits of the communication channel between the two registers (the "data wires") are dual-rail encoded. Wire $(yo \underline{w} xi)$ is "cut open," yo is used to assigned the values of the bits of u to the dual-rail data wires, and xi is set to true when all bits of v have been set to the values of the data wires. Each cell of a register contains an acknowledge wire that is set to true when the bit of the cell has been set to a valid value of the two data wires, and reset to false when the data wires are both

reset to false. Let $vack_i$ be the acknowledge of bit v_i, xi is set and reset as:

$$vack_0 \wedge vack_1 \ldots \wedge vack_{15} \mapsto xi\uparrow$$
$$\neg vack_0 \wedge \neg vack_1 \ldots \wedge \neg vack_{15} \mapsto xi\downarrow$$

Since a 16-input C-element would be prohibitively slow to implement, the implementation is a tree of smaller C-elements, which we call a ***completion tree***. Figure 5.(b) shows a tree of binary C-elements. In the actual processor, we use a two-level tree of 4-input C-elements.

When data is transmitted via a bus, and when the completion tree is large, the gain of using a lazy-active protocol can be very important, since half of the data transmission delays and half of the completion-tree delays can overlap with the rest of the computation. Therefore, all input actions are implemented as lazy active.

The case when data is transmitted from process A to process B via a bus is only slightly more complicated. No arbitration is necessary: A and B are allowed to communicate via a bus only after the bus has been reserved for these two processes. The chief problem in implementing the buses is the distributed implementation of large multi-input OR-gates.

The lazy-active protocol cannot be used when an input action is probed—such as action $AC?op$ in the ALU—because the probe requires a passive protocol. For those cases, we have designed a special protocol that requires two control wires.

9 ALU

ALU control

In the ALU process, variable z is not needed to store the result of an ALU operation: the result can be put directly on the ZA bus. The first guarded command of the ALU process can be rewritten:

$$\overline{AC} \rightarrow AC?op \bullet X?x \bullet Y?y; \langle ZA, f\rangle := aluf(x, y, op, f).$$

Hence, the control part is simply:

$$*[[\overline{AC} \rightarrow AC \bullet X \bullet Y; AL$$
$$[]\overline{F} \rightarrow F$$
$$]].$$

(The assignment to f is omitted.) Communication command AL is the call of the subprocess evaluating ***aluf***. The handshaking protocol of AL is passive because it includes an output action on the ZA bus: $[ali]; alo\uparrow; [\neg ali]; alo\downarrow$. Hence, $alo\uparrow$ is the "go" signal for the ALU computation proper.

The first guarded command has the structure of a canonical stage of the pipeline. Parameters are simultaneously received on a set of ports, and the result is sent on another port as in:

$$*[L?x; R!f(x)].$$

Such a process is called a ***buffer***. Since L is implemented as lazy active, and R as passive, it is a ***lazy-active/passive buffer***. In the second design, where we have decomposed both the ALU and the memory processes into two processes in order to improve the pipeline, each stage of the pipeline is a lazy-active/passive buffer.

ALU data path

The output Z of the subprocess is dual-rail encoded. When the subprocess is called, variables x, y, and op have stable and valid values. Moreover, the content of op has been encoded in a KPG ("kill, propagate, generate") form which is used to produce the carry-out for each bit, and also for the result. The length of the carry chain is variable, which is an advantage in a fully asynchronous execution.

Since the carry-out of each bit is inverted relative to the carry-in, we alternate the logic encoding of the stages in the carry chain: A carry-in that has a true value when high generates a carry-out that has a true value when low, and vice-versa for the next stage. With this coding, only one CMOS gate delay is incurred per stage. Although the acknowledge from the ZA bus is used as completion signal, a completion tree is needed at the output of the subprocess for the computation of the flags.

The elapsed time between the activation of the ALU subprocess by $alo\uparrow$ and the appearance of the results on the output Z depends on the number of stages in the carry chain. Add, substract, and other logical functions typically take between 13 and $25ns$ in $2\mu m$ SCMOS.

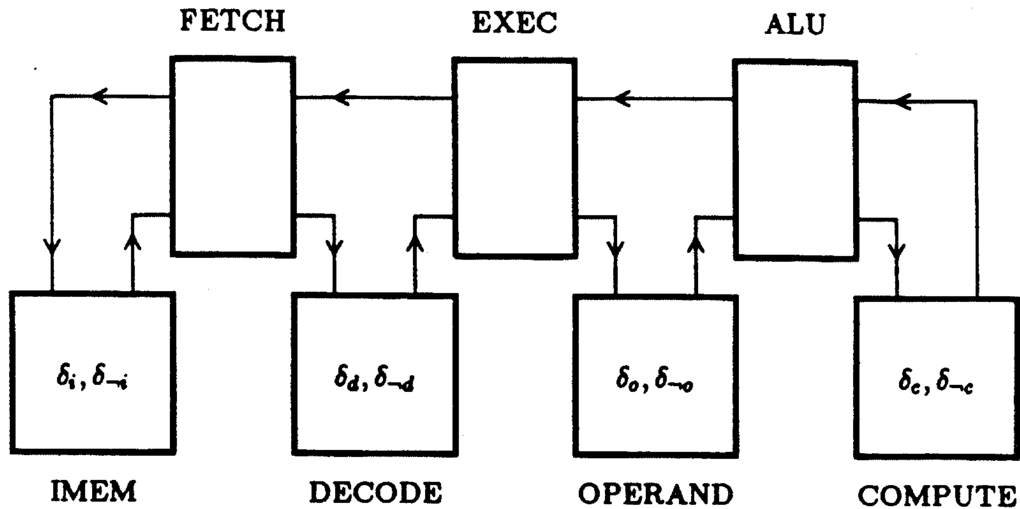

Figure 6: Abstract Pipeline for ALU Instructions

10 Performance

In this processor, an instruction is executed in a varying amount of time, depending in part on the type of instruction and the values of its operands, and on the sequence surrounding the instruction. Because of this data dependence, an analysis of the "real" performance of the processor, i.e., the performance of the processor when executing "real" programs, is quite complex and most probably must be determined by simulation. The performance analysis can be simplified by assuming an infinite sequence of identical instructions with typical operand values. (The results obtained through this analysis do not include the potential benefits of interleaving ALU and memory instructions.) Here, we analyze the performance of the processor executing an infinite sequence of ALU instructions.

In this case, the processor can be viewed as the three-stage pipeline shown in Figure 6. By assuming the ALU operations are performed on distinct registers, the register locking mechanism need not be introduced and the control for the ***EXEC*** process and the ***ALU*** process reduces to lazy-active/passive buffers. The ***fetch*** process is complicated by the increment of the ***pc***, but if the instruction memory is assumed to be slower than the increment, control for this process also reduces to a lazy-active/passive buffer. By first assuming negligible control delays compared with datapath delays (denoted δ_D and $\delta_{\neg D}$ for the upgoing and downgoing propagation delays of datapath unit D, respectively),

the cycle time, c_P, of each process P is determined by the datapath delays that must be sequenced. A lazy-active/passive buffer sequences only the upgoing transitions of the two datapath units and, separately, the upgoing and downgoing transitions of the individual units, resulting in cycle time $\max(\delta_{D1} + \delta_{D2}, \delta_{D1} + \delta_{\neg D1}, \delta_{D2} + \delta_{\neg D2})$.

Since each process in the pipeline is a lazy-active/passive buffer, and since the throughput of the pipeline is determined by the slowest process:

$$\begin{aligned} c_{FETCH} &= \max(\delta_m + \delta_d, \delta_m + \delta_{\neg m}, \delta_d + \delta_{\neg d}) \\ c_{EXEC} &= \max(\delta_d + \delta_o, \delta_d + \delta_{\neg d}, \delta_o + \delta_{\neg o}) \\ c_{ALU} &= \max(\delta_o + \delta_c, \delta_o + \delta_{\neg o}, \delta_c + \delta_{\neg c}) \\ c_{PROC} &= \max(c_{FETCH}, c_{EXEC}, c_{ALU}) \ . \end{aligned}$$

Timing simulations suggest that the dominant constraints are the memory and decode sequence in the *FETCH* process ($\delta_m + \delta_d$), and the operand and compute sequence in the ALU process ($\delta_o + \delta_c$). For the $2\mu m$ SCMOS processor, the delays introduced by the control parts increase the cycle time by **10** to **20*ns***, bringing the cycle time for an infinite stream of ALU instructions up to $\max(35ns + \delta_m, 65ns)$. We expect the processor to achieve **15** MIPS if the access delay of the instruction memory (δ_m) is no longer than **30*ns***.

11 Correctness by Construction and CAD Tools

Since the method is based on semantics-preserving program transformations, the object code generated by the compilation procedure is correct by construction.

The object code is a set of potentially concurrent ***production rules*** that are constructs of the form $B1 \mapsto x\uparrow$ or $B2 \mapsto x\downarrow$, where $B1$ and $B2$ are mutually exclusive boolean expressions, and $x\uparrow$ and $x\downarrow$ stand for "set x to true" and "set x to false," respectively. The compilation procedure guarantees the absence of hazards by ensuring that the conditions $B1$ and $B2$ are ***stable***, i.e., if $B1$ is true, it remains true until x as been set to true.

If the production rules of the object code can be matched with the production rules that describe the standard cells of a cell library, a standard-cell-layout program can be used to generate a layout corresponding to the object code. We have been using such a standard cell approach in our previous designs, and indeed all chips fabricated in this way have been found to be functional on "first silicon."

However, most of the processor was designed manually. First, since the control section introduces significant overhead, we decided to

compile its object code manually. Second, because the data path was expected to be the critical part with respect to size and because of the difficulty of adjusting the pitch of the different registers automatically, the automatic layout program was used for the control part but not for the data path. This decision was later justified by the fact that, whereas the data path was hardly changed after the first design, the control part went through a series of drastic modifications. We observed that, again, our method for separating control and data path permitted us to implement completely different pipelines by changing the control without significant alterations of the data path.

As usual, the disadvantage of manual compilation was that the design was not shielded from clerical errors at which humans excel.

While the difficult optimization problem that is at the core of a high-performance processor design is probably still beyond automatic compilation technology, the designer should be assisted with CAD tools that perform the mechanical translation steps. Other CAD tools that we found useful include a program that estimates the critical path of a circuit. The program, which was developed by Steve Burns, gives excellent results. It estimates the delays of each path by a simulation of the execution based on the production rules.

Magic was used for the manual layout [10].

12 Conclusion

Although the chips are still in fabrication, we are very satisfied with the preliminary results of the experiment.

First, the chip layout is obviously not large. The control is surprisingly small despite our use of an automatic layout tool; also, the anticipated nightmare of data path layout did not materialize. The register pitch is 80λ, which is quite reasonable given that four buses have to be placed.

Second, the predicted performance is quite remarkable, given that the experiment is a first in two ways: It is our first experience as computer architects, and it is the first asynchronous microprocessor ever built.

Third, the complete design took five persons (one joined in the middle of the project) five months.

Since the choice of an instruction set was not part of the experiment, our design should be judged in two ways: the choice of the concurrent program of Figure 3, and its implementation.

The implementation is satisfactory, but not optimal. The sizing of transistors can be improved and the number of transitions can be decreased, mainly by a better placement of inverters. For instance, the delays due to a completion tree and to the control for a buffer are both about twice their theoretical minimum.

The program of Figure 3 represents the choice of a pipeline, and of synchronization techniques to implement it. We have deliberately chosen a simple pipeline. In particular, the mechanism for stalling, which places part of the decoding in series with the fetch on the critical path, sacrifices efficiency for simplicity. However, performance evaluations show that the pipeline is well-balanced since the different stages have comparable average delays. Improving the critical path by overlapping fetch and decode requires improving the ALU and memory instruction execution stages by pipelining parts of these stages.

The practicality of overlapping ALU and memory instruction executions remains an open issue. It is not clear whether the gain in performance is worth the complexity of the synchronization involved and the requirement of two separate Z buses.

We find the synchronization techniques used to implement the concurrent activities between the different stages of the pipeline particularly elegant and efficient, since the delays incurred in a synchronization can be of arbitrary length and vary from instruction to instruction.

We foresee excellent performances for asynchronous processors as the feature size keeps decreasing. But the designer must be ready to learn and apply new design methods based on concurrent programmming, that are required to exploit asynchronous techniques to their fullest.

Acknowledgment

We are indebted to Bill Athas and Bill Dally for useful discussions in the preliminary stage of the design. Chuck Seitz, Nanette Boden, and Dian De Sha made excellent comments on the manuscript. The first author enjoyed numerous discussions with Chuck Seitz on the general topic of asynchronous design.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, DARPA Order numbers 3771 & 6202, and monitored by the Office of Naval Research under contract numbers N00014-79-C-0597 & N00014-87-K-0745.

References

[1] R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. In ***24th Design Automation Conference***, pp.9–16. ACM and IEEE, 1987.

[2] Steven M. Burns and Alain J. Martin, Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. In J. Allen and F. Leighton (ed), ***Fifth MIT Conference on Advanced Research in VLSI,*** pp 35-40, MIT Press, 1988.

[3] C.A.R. Hoare, Communicating Sequential Processes. ***Comm. ACM*** 21,8, pp 666-677, August, 1978.

[4] Mark Horowitz ***et al.***, MIPS-X: A 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache. ***IEEE Journal of Solid-State Circuits***,SC-22(5):790-799,October, 1987.

[5] Alain J. Martin, The Design of a Self-timed Circuit for Distributed Mutual Exclusion. In Henry Fuchs (ed), ***1985 Chapel Hill Conf. VLSI***, Computer Science Press, pp 247-260, 1985.

[6] Alain J. Martin, Compiling Communicating Processes into Delay-insensitive VLSI Circuits. ***Distributed Computing***, 1,(4), Springer-Verlag, pp 226-234 1986.

[7] Alain J. Martin, A Synthesis Method for Self-timed VLSI Circuits. ***ICCD 87: 1987 IEEE International Conference on Computer Design***, IEEE Computer Society Press, pp 224-229, 1987.

[8] Alain J. Martin, Programming in VLSI: From Communicating Processes to Delay-insensitive Circuits. In C.A.R. Hoare (ed), ***UT Year of Programming Institute on Concurrent Programming***, Addison-Wesley, Reading MA, 1989.

[9] Carver Mead and Lynn Conway, ***Introduction to VLSI Systems***, Addison-Wesley, Reading MA, 1980.

[10] J. K. Ousterhout ***et al.***, The Magic VLSI layout system, ***IEEE Design Test Comput.***, 2, (1), pp 19-30, February, 1985.

[11] Charles L. Seitz, System Timing, Chapter 7 in Mead & Conway, ***Introduction to VLSI Systems***, Addison-Wesley, Reading MA, 1980.

Appendix 1: Notation

The program notation, which is inspired by C.A.R. Hoare's CSP [3], is briefly described.

$b\uparrow$ stands for $b :=$ **true**, $b\downarrow$ stands for $b :=$ **false**.

The execution of the *selection* command $[G_1 \to S_1 [] \ldots [] G_n \to S_n]$, where G_1 through G_n are boolean expressions, and S_1 through S_n are program parts, (G_i is called a "guard," and $G_i \to S_i$ a "guarded command") amounts to the execution of an arbitrary S_i for which G_i holds. If $\neg(G_1 \vee \ldots \vee G_n)$ holds, the execution of the command is suspended until $(G_1 \vee \ldots \vee G_n)$ holds.

The execution of the *repetition* command $*[G_1 \to S_1 [] \ldots [] G_n \to S_n]$, where G_1 through G_n are boolean expressions, and S_1 through S_n are program parts, amounts to repeatedly selecting an arbitrary S_i for which G_i holds and executing S_i. If $\neg(G_1 \vee \ldots \vee G_n)$ holds, the repetition terminates.

For communication actions X and Y, "$X \bullet Y$" stands for the coincident execution of X and Y, i.e., the completions of the two actions coincide.

$[G]$ where G is a boolean expression, stands for $[G \to \textbf{skip}]$, and thus for "wait until G holds."

(Hence, "$[G]; S$" and $[G \to S]$ are equivalent.)

$*[S]$ stands for $*[\textbf{true} \to S]$, and thus for "repeat S forever."

From (iii) and (iv), the operational description of the statement $*[[G_1 \to S_1 [] \ldots [] G_n \to S_n]]$ is "repeat forever: wait until some G_i holds; execute an S_i for which G_i holds."

Communication commands: Let two processes, *p1* and *p2*, share a channel with port X in *p1* and port Y in *p2*. (In the processes of Figure 3, the same name is used for all the ports of the same channel.) If the channel is used only for synchronization between the processes, the name of the port is sufficient to identify a commnication on this port. If the communication is used for input and output of messages, the CSP notation is used: $X!u$ outputs message u, and $X?v$ inputs message v.

At any time, the number of completed X-actions in *p1* equals the number of completed Y-actions in *p2*. In other words, the completion of the nth X-action "coincides" with the completion of the n-th Y-action. If, for example, *p1* reaches the nth X-action before *p2* reaches the nth Y-action, the completion of X is suspended until *p2* reaches Y. The X-action is then said to be *pending*. When, thereafter, *p2* reaches Y, both X and Y are completed. It is possible (and even advantageous) to define communication actions as coincident and yet implement the actions in completely asynchronous ways. For an explanation, see [8].

Probe: Since we need a mechanism to select a set of pending communication actions for execution, we provide a general boolean command on ports, called the ***probe***. In process **p1**, the probe command $\overline{X}$ has the same value as the predicate "Y is pending in **p2**."

Appendix 2: Instruction Set

ALU	op rx ry rz	rz,f := rx op ry
MEM	op rx ry rz	rz := mem[rx+ry] (for load) mem[rx+ry] := rz (for store)
MEMOFF	op ao ry rz offset	rz := mem[ry + offset] (for load) mem[ry + offset] := rz (for store) rz := ry + offset (for load address)
BRANCH	op ao — cc offset	if cond(f,cc) then pc := pc + offset
JUMP	op ao ry —	pc := ry
STPC	op ao — rz	rz := pc

Table 1: Instruction Types

inst	n_3 $b_{15}b_{14}b_{13}b_{12}$	n_2 $b_{11}b_{10}b_9b_8$	n_1 $b_7b_6b_5b_4$	n_0 $b_3b_2b_1b_0$
alu	0011	rx	ry	rz
	0100	rx	ry	rz
	⋮			
	1111	rx	ry	rz
ld	0010	rx	ry	rz
st	0001	rx	ry	rz
ldx	0000	0000	ry	rz
stx	0000	0001	ry	rz
lda	0000	0010	ry	rz
brc	0000	0011	—	cc
jmp	0000	0100	ry	—
stpc	0000	0101	—	rz

Table 2: Opcode Assignments