The Design of an Asynchronous MIPS R3000 Microprocessor

Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström
Paul Penzes, Robert Southworth, Uri Cummings, Tak Kwan Lee
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

Abstract

The design of an asynchronous clone of a MIPS R3000 microprocessor is presented. In $0.6\mu m$ *CMOS, we expect performance close to 280 MIPS, for a power consumption of 7 W. The paper describes the structure of a high-performance asynchronous pipeline, in particular precise exceptions, pipelined caches, arithmetic, and registers, and the circuit techniques developed to achieve high throughput.*

1 Introduction

This paper describes the architectural algorithms and circuit techniques used in the design of an asynchronous MIPS R3000 microprocessor. The project has two main goals. First, we are investigating issues in asynchronous processor architecture that we have not tackled in the Caltech Asynchronous Microprocessor [6]: caches, precise exceptions, register bypassing, branch-delay slot and branch prediction. Secondly, we are developing new techniques for asynchronous digital VLSI—based on very fine pipelining—that can meet high throughput requirements without sacrificing the low-power advantage of asynchronous circuits.

At the moment of writing, HSPICE simulations indicate that we can achieve these goals: In 0.6μm MOSIS SCMOS, the instruction execution frequency is expected to be approximately 280 MIPS for a power consumption of 7 W at 75°C. (This first prototype is "single-scalar;" it fetches one instruction at a time.)

We want to optimize the average instruction execution time, also called "cycle time", τ, and the average energy per instruction, E. Since these two parameters can be traded against each other through voltage adjustments, we combine them into one figure of merit, the product $E\tau^2$, which is independent of the voltage in first approximation. Minimizing this measure is the theoretical goal of our architectural and circuit decisions, and it is through this measure that we compare our designs against others. In particular, it makes it possible to compare designs at opposite ends of the performance curve. Furthermore, once we have obtained a good $E\tau^2$, we can exploit the robustness of asynchronous designs to voltage variations to achieve both high speed at high voltage and low power at low voltage. For instance, we expect the asynchronous MIPS to deliver 100 MIPS for less than 0.5 W of power at 1.5 V, which is a better performance than that of low-power designs like the StrongARM in equivalent technologies. (See the section on performance comparisons.)

At the architectural level, we start with a clean slate. We want to implement the MIPS R3000 instruction set, as defined in [3], including the precise exception mechanism, and the branch delay slot. But we don't require any similarity whatsoever with the architecture of existing MIPS microprocessors. We could have replicated exactly some standard architecture by implementing each pipeline stage asynchronously. But such a solution would grossly underutilize the possibilities of an asynchronous architecture. Instead, a significant part of this project's research has been to invent new asynchronous solutions for issues associated with pipelined microprocessors.

The asynchronous circuits we use are called *quasi delay-insensitive* or QDI. A QDI circuit does not use any assumption on, or knowledge of, delays in operators and wires, with the exception of some forks, called *isochronic forks*, in which the delays on the different branches of the fork are assumed to be similar. (See [8] for a more precise definition.) QDI circuits are the most conservative asynchronous circuits in terms of the use of delays. But they are also the most robust to physical parameters variations because the circuit's dependence on delays is minimal. It is this robustness that makes it possible to exchange energy and throughput against each other through voltage adjustments. Another goal of this project was to show that delay assumptions are not necessary to achieve high performance in asynchronous circuits.

The paper is organized in two parts. Part 1 deals with architectural issues. We briefly introduce the MIPS instruction set. We describe the general structure of the asynchronous pipeline, and we present in more detail the solutions introduced for the main architectural issues. Part 2 deals with new asynchronous circuit techniques introduced to improve both the throughput and the latency of the pipeline. Finally, we discuss the expected performance of the first prototype. We compare the performance, in terms of both speed and power, to other processor architectures and draw some conclusions.

2 Part 1: Architecture

2.1 Specification of the MIPS R3000

We chose the MIPS R3000 as target for our project because it is the archetype of a commercial RISC microprocessor, without being overly complicated. The R3000 consists of two tightly-coupled processors, usually implemented on a single chip: a full 32-bit RISC CPU, and a memory-management unit, called CP0. The processor can be extended with three off-chip coprocessors. Coprocessor 1 (CP1) is specified by the instruction set to be the floating-point unit. The other two are unspecified.

The CPU has 32 32-bit general-purpose registers, a program counter, and two special-purpose registers for multiplication and division. It has two operating modes—user and kernel—providing different levels of hardware protection. Branches have a single delay-slot. Although the MIPS ISA allows both little-endian and big-endian modes to be selected on reset, we have chosen to implement little-endian mode only.

There are three types of instructions: *Immediate*, *jump*, or *register*, with the following structures.

- immediate: $\langle op, rs, rt, immediate\rangle$
- jump: $\langle op, target\rangle$
- register: $\langle op, rs, rt, rd, sa, func\rangle$

The field *op* is the opcode;the fields *rs*, *rt*, and *rd* are register indices, *immediate* is an immediate offset, and *target* is an offset used in jump target calculation: *sa* is a shift amount, and *func* is a function-field supplementing the opcode, when necessary.

The memory management system provides hardware for address translation in the form of an on-chip 64-entry Translation Lookaside Buffer. The R3000 has write-through direct-mapped caches with one-word cache lines and 4-word blocks for refill on a cache miss.

The chip we are sending now for fabrication is not the complete MIPS but a somewhat reduced "MiniMIPS." We have left out the TLB implementation (we have implemented the TLB down to the gates but we don't have layout yet), the partial-word memory operations, and the external interrupt mechanism. We are implementing the internal precise exceptions and the on-chip caches.

2.2 General Pipeline Structure

Let us first clarify one possible source of confusion. The final design is very finely pipelined, in order to achieve high throughput. So, what we are describing now as a pipeline stage is later on refined as a large number of small pipeline stages. For the moment, we describe the pipeline at a macroscopic level: a stage is meant to be one of the main units of the architecture.

The simplest structure for the MIPS consists of three sequential steps: *compute the next program counter*, *fetch the instruction*, *execute the instruction*.

The first step in pipelining this program consists in decomposing it into a 3-stage pipeline consisting of a PC unit that computes the next pc, a $FETCH$ unit fetching the instructions, and an $EXEC$ unit executing the instruction. Since the MIPS has a one-delay slot, this pipeline can allow the execution of an instruction to overlap with the program counter calculation and the fetching of the next instruction: we have two instructions in different stages of processing in this loop.

Next, we decompose the $EXEC$ unit to allow concurrent and out-of-order execution of instructions. The general pipeline structure is shown in Figure 1. A $DECODE$ unit decodes the instruction to be executed, a number of execution units (EUs) execute the decoded instructions, a register unit (REG) controls the general-purpose registers (GPR): the unit sends the values of registers to the execution units via the two operand buses X and Y, and receives the result of an instruction execution from the execution units via one of the two result buses $Z0$ or $Z1$. The register unit also decides when to do register bypassing.

A write-back unit, WB, decides whether the result from an execution unit should be written into the registers (GPR, Hi/Lo in the multiplier/divider unit, or CP0 registers) depending on whether an exception has occurred in a previously executed valid instruction. The execution units are:

- an adder/subtracter/comparator
- a functional unit for logical operation
- a memory unit for load/stores and manipulation of CP0 registers
- a shifter
- a multiplier/divider

There are two distinct parts in the pipeline: the "fetch-loop" consisting of the PC unit, the $FETCH$, and the $DECODE$, and the "execution pipeline" consisting of the execution units, the register unit, and the write-back.

A *token* is a piece of data in transfer in the pipeline: pc, instruction, opcode, operands, etc. At any moment, there are exactly two tokens in the fetch-loop. But there is a variable number of tokens in the execution pipeline.

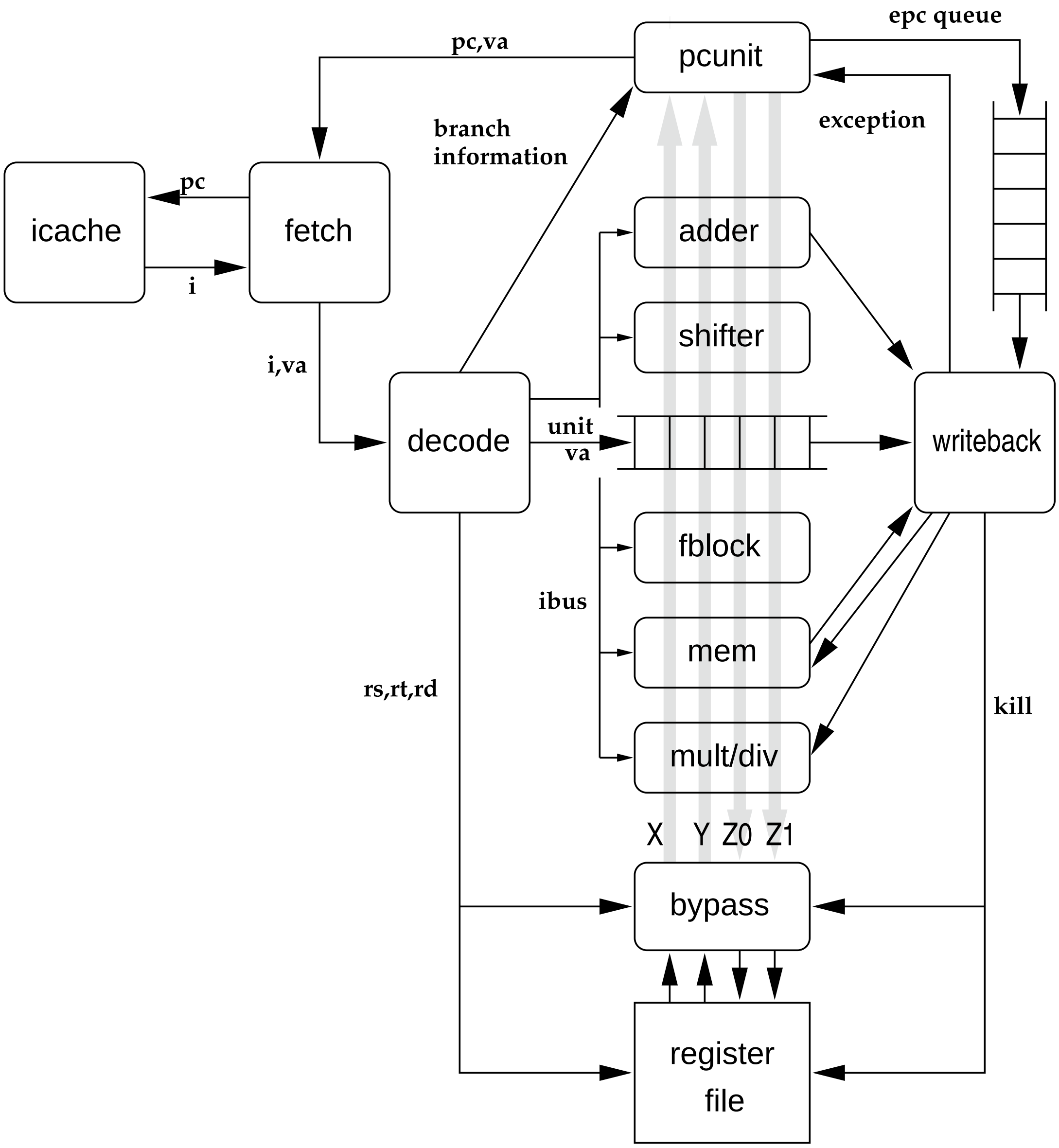

Figure 1. The MiniMIPS pipeline

The execution pipeline of the MiniMIPS is very different from a synchronous one. In a synchronous pipeline, the EU's and the WB are aligned in some specific sequence, and all the tokens go through all the units of the sequence in order. In the MiniMIPS, all EU's are in parallel in the pipeline, and can execute concurrently. For example, a result coming out of the adder will not go through any other EU or the WB, but directly to the register unit.

Although all the stages of the execution pipeline are finely pipelined and can contain a large number of tokens—as we will explain, the maximal number of tokens that a pipeline can contain is called the *slack* of the pipeline—the actual number of tokens is limited to achieve the maximum throughput. For the pipeline to process the tokens at optimal throughput, the tokens have to be spaced through the pipeline. The optimal number of stages per token for maximum throughput is determined by the ratio of the cycle period over the forward latency of a pipeline stage. As we will see, the circuit characteristics of the pipeline stages we have chosen for the MiniMIPS are such that the optimal throughput is achieved for 8 stages per token. As a consequence, the execution pipeline contains 3 tokens on average.

2.3 An Instruction Execution

As an illustration of the pipeline, let us follow the execution of a typical instruction, for example an addition: ADD rd, rs, rt. The net effect is defined as: " if $GPR[rs]+GPR[rt] < 2^{32}$ then $GPR[rd] := GPR[rs] + GPR[rt]$ else overflow exception."

The pc of the instruction is calculated by the PC unit and sent to the $FETCH$, together with a "valid again" boolean label used in the exception handling phase of the pipeline. The $FETCH$ unit fetches the instruction from the instruction cache and sends it to the predecode part of the $DECODE$ while the tags in the instruction cache are compared for hit or miss. If it is a hit, the decoding of the instruction is pursued normally. If it is a miss, the partially decoded instruction is killed, and the $FETCH$ fetches the instruction from memory. (In the complete MIPS, address translation may cause an exception, but not in the MiniMIPS.)

The $DECODE$ finishes the decoding of the instruction: the opcode is passed to the ADD execution unit, and the *rs, rt, rd* fields are passed to the REG unit to select and reserve the operand registers *rs, rt*, and the result register *rd.* The $DECODE$ also adds to the "unit-number" queue the entry that an add instruction has been dispatched.

From now on, the execution of the instruction involves several units operating concurrently: The adder decodes the opcode. The REG unit sends the operandss over the operand buses either from the registers $GPR[rs]$ and $GPR[rt]$ or from the bypass if it happens that the previous instruction has produced a result to be written in $GPR[rs]$ or $GPR[rt]$. Once the adder has received the operands from the X and Y buses, it performs the addition, which may either lead to an overflow exception or produce a valid result.

In both cases, the adder communicates with the write-back (WB) unit which polls each EU in order of program execution. The program order is provided by the unit-number queue. If the addition terminated normally, and if the addition is not part of a sequence of instructions that have to be canceled after an exception occurred, the WB allows the result of the addition to be written back in the registers, through one of the two result buses. (The decode decides which result bus is used, and this information is passed to the result bus interface of the adder.)

The writing of the result in a register is delayed. The result is kept in the bypass unit until the next cycle. If the addition has to be canceled either because it caused an overflow exception, or because it is in the sequence of "canceled instructions" following an exception, then the *WB* sends a cancel command to the register unit.

On exception, the *WB* sends an exception message to the *PC* unit and gathers the necessary information about the exception. The *PC* unit probes the exception channel once per iteration. If there is a pending exception message from the *WB*, it updates the pc accordingly. Observe that probing the exception channel when there is no exception is practically free since it does not involve any synchronization of the writeback with the *PC* unit.

This completes the brief description of a typical instruction. Branch/jump instructions are different since they do not go to an execution unit but directly from the *DECODE* to the *PC* unit.

2.4 Precise Exceptions

The handling of precise exceptions is one of the main innovations of this project. Since an asynchronous pipeline is a totally distributed system, reconstructing the precise global state of the computation at which an exception occurs is different from, and more complicated than, a synchronous pipeline. On the other hand, the overhead of checking for precise exceptions is almost negligible in an asynchronous system when no exception occurs.

An exception occurs either during the execution of an instruction in an execution unit, during the fetching of the instruction from memory, or because of an external interrupt. In the complete MIPS, when a memory exception is detected during the fetching of the instruction, the decoder passes the instruction to the memory/CP0 unit. (Such an exception does not occur in the MiniMIPS.) The unit recognizes it as a memory-exception instruction, and deals with it as if it were a normal *EU* exception.

Consider an exception detected in an execution unit. First, by definition of precise exceptions, the exception mechanism has to guarantee that the instruction that caused the exception, and all instructions following it in the pipeline before the first instruction of the exception handler, do not change the global state of the computation. We will refer to these instructions as the *canceled instructions*. Canceling instructions amounts to canceling the "write-back" of the results of these instructions' executions to the permanent registers (GPR, Hi/Lo, and CP0 registers) and to memory.

Second, the command to interrupt the normal pc update has to be sent to the *PC* unit.

Third, the information about the exception (the cause of the exception, the pc of the exception instruction, whether the exception occured in a branch-delay slot) has to be stored in the proper CP0 registers.

In order to determine the sequence of canceled instructions, the program order of instruction execution must be recorded before the *DECODER* dispatches the instructions to the different execution units. To that effect, the decoder maintains a queue of unit numbers (the unit-number queue) that represents the order in which the *DECODER* dispatches the instructions to the *EU*'s.

The central mechanism for exception handling is the write-back unit. Each execution unit that can cause an exception communicates to the *WB* whether the instruction just executed resulted in an exception. The *WB* cancels the writing of the instruction result into the registers if that instruction or a preceding instruction caused an exception. The *WB* reconstructs the program order by polling the execution units in the order of the unit numbers in the queue.

The interaction between the *EU*'s and the *WB* is very different from what usually takes place in a synchronous pipeline. Because the *EU*'s are not aligned in any order in the pipeline, the *WB* is not placed "at the end of the pipeline," but rather "on the side." Each

EU concurrently sends its result to the bypass and communicates with the WB. If the result has to be canceled because the instruction is one of the canceled instructions, the WB sends a "kill" message to the register unit, and the result is canceled in the register unit. This solution is obviously more efficient than the synchronous one.

The WB needs one additional bit of information in order to stop canceling instructions. We introduce a "valid-again" bit generated by the PC unit. When a pc is sent by the PC unit to the $FETCH$, a valid-again bit is attached to it. This valid-again bit is then attached to the fetched instruction sent to the $DECODE$, and in turn to the unit-number queue. When the WB reads a unit number with the valid-again bit set, it stops discarding instruction write-backs. The first pc tagged with the valid-again bit set is not the pc of first instruction of the exception handler but that of a pseudo-instruction preceding it. This pseudo-instruction is decoded as a $CP0$ instruction and is executed by the $CP0$ unit. The purpose of this pseudo-instruction is to store the information about the exception in the proper $CP0$ registers, before the exception handler instructions can use these $CP0$ registers.

2.5 Register Unit

The register unit consists of the register file, the register lock, the execution buses, and the register bypass. (See Figure 2.) The register file has two read ports and two write ports. The register bypass connects the register file's ports to the X and Y operand buses, and the $Z0$ and $Z1$ result buses. The X and Y buses are 1-to-6: they are used to send an operand from the X or the Y bypass to 1 of 6 units. The $Z0$ and $Z1$ buses are 6-to-1: they are used to send the results from the units to the $Z0$ or $Z1$ bypass. The data on a result bypass may be written back into a register, or both written back into a register and sent to an operand bus in the case of a bypass. A bypass takes place when the operand of an instruction is the result of the previous instruction. Again because the EU's are not aligned in any particular order in the pipeline, each unit communicates directly with the bypass unit, and therefore bypassing can take place between any two EU's.

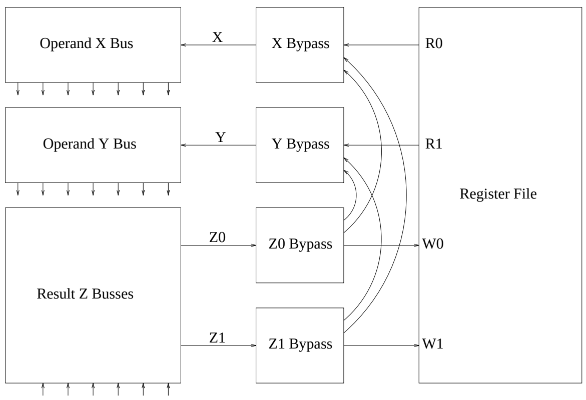

Figure 2. Register File and Execution Buses

The ports of the register file are also buses: The $R0$ and $R1$ ports are 32-to-1 buses, and the $W0$ and $W1$ ports are 1-to-32 buses. Each unit is connected to a bus through a bus-interface process. An additional asynchronous channel delivers the selection control to these processes in the form of a 1-of-N code.

We use the two result buses in a strictly interleaved scheme in order to reduce pipeline stalls caused by discrepancies between execution unit latencies. For instance, if an instruction is assigned to write back to result bus $Z0$, but has a large latency (like a cache miss), the next instruction can still write back on result bus $Z1$ in parallel if there is no data dependency. Interleaving the result buses permits a latency mismatch of the execution units equal to the processor's cycle-time without reducing the processor's throughput.

2.6 Low-latency Addition

Apart from the shifting and logical operations, there are two types of arithmetic operations in the MiniMIPS: addition and comparison. There are four types of addition and comparison instructions, depending on whether the instruction uses the immediate field or a register as one operand, and whether the operation is signed or unsigned. Subtraction is either signed or unsigned, and cannot use immediates.

All the addition/subtraction and comparison instructions are executed in a single four-stage pipelined execution unit which we will refer to as the "adder." The adder can generate an overflow exception on addition and subtraction instructions. When an instruction for this unit is received by the decode, the unit receives operands from the X and Y buses and possibly the immediate buses and additional control information from the decode. The adder produces its output on one of the two Z buses as well as a bit indicating whether the instruction raised an exception.

The standard techniques for binary addition include using a carry-lookahead adder (a kill-propagate-generate adder), a carry-select adder, or a conditional-sum adder. In an asynchronous system, we have the option of using a simple ripple-carry binary adder, since the average-case latency (assuming the inputs are bitwise independent) for binary addition is $O(\log N)$, where N is the number of bits in the input. Since the adder is used in latency-critical parts of the processor such as address calculation in the memory unit, we decided to use an adder with a worst-case latency of $O(\log N)$ instead of $O(N)$.

Most fast binary adders are based on the kill-propagate-generate (kpg) technique. Assuming that we do not use a large number of n-type transistors in series (not more than six for the process we are using), a pipelined full-throughput kpg adder would take seven stages of logic, including the part of the adder that conditionally reads from different buses and conditionally inverts the input for subtraction. Carry-select and conditional-sum adders speculatively compute results based on the different inputs and therefore waste energy. We would like to keep the speculation used in the adder to a minimum.

The adder we use is a hybrid between a kpg adder and carry-select adder. The first stage of the adder computes the kpg code for each input bit. Next, a kpg tree adder is used to compute the two possible sums for each 8-bit block. We compute the carry-in for each 8-bit block in parallel with the block addition. These two operations constitute the next two stages of the adder. The final stage of the adder selects the appropriate sum for each 8-bit block and produces the output.

HSPICE measurements indicate that this execution unit can execute the required arithmetic instructions at a throughput of 280 MHz at 75°C, with a worst-case input to output latency of 1.6ns. The current architecture will scale to a 64-bit adder without significant latency or throughput penalty.

The adder used to add immediates to a register value for address calculation in the memory unit has been optimized to take advantage of the fact that the immediate operands are usually small, and that the immediate field is known early since it is part of the instruction. The adder is optimized to produce the bottom 12 bits of the output in 0.5ns for small immediates. These bits are needed early because they are used to access the data cache.

2.7 Pipelined Caches

The MiniMIPS cache system consists of two four-kilobyte (1 page) caches: an instruction cache (I-cache) and a direct-mapped data cache (D-cache). The caches are identical except that the D-cache has provisions for writing (using a write-through mechanism with a write buffer) and the I-cache has provisions for branch prediction and predecode. Cache refills are carried out with a 128-bit (4 line) refill block.

Some functions that are part of the MIPS cache specification have been omitted for the sake of simplicity: the isolate-cache and swap-cache mechanisms, the explicit invalidation mechanism, and the ability to change the size of the refill block. The MiniMIPS instruction set also omits partial-word operations. The cache architecture has been designed so as to make it easy to add partial-word operations while maintaining low latency and high throughput on the more common word loads and stores.

2.7.1 Cache Lines, Segments, and Invalidation

The MiniMIPS caches use 48-bit cache lines: 32 bits of data and 16 bits of tags. The tags are generated from bits 12 through 27 of the address. Address bit 31 is hard-wired to zero in the cache; this technique is used to implement an "uncacheable" two gigabyte segment since tags with address bit 31 equal to one always miss. Furthermore, this allows flushing the caches by reserving a four-kilobyte block of zeros in the uncacheable segment. Reading this block as data or executing it (for the I-cache) will flush the caches.

2.7.2 Cache Pipelining

In order to achieve high density, the static RAM of the cache core was designed with an average access time of 3 ns, almost an entire instruction fetch cycle. Allowing an additional 1.5 ns for tags comparison and the subsequent decision to accept or discard the value read from the cache core puts the latency through the entire cache system at 4.5 ns for a cache *hit.* The maximum cache performance required is obtained with one cache operation per instruction per cache—one instruction fetch from the I-cache and one load from the D-cache. Given the average latency of the cache cores, this performance could not be achieved without pipelining the caches.

An unpipelined cache would look up the contents of a cache line, compare the tags, and then decide whether to output the data as its result and go ahead with the next operation or to read the data from memory, refill the cache core, and continue. The design is pipelined so that the cache control issues a lookup to the cache core before deciding whether the previous lookup was a hit or a miss. The cache core is pipelined internally, allowing the two lookups to proceed independently and concurrently unless they refer to the same block in which case they are strictly sequentialized. The cache array is implemented as a pipelined tree fanning out to 4 leaf-cells.

This pipelined cache implementation introduces problems analogous to "structural hazards." Assume that a load word at an address is immediately followed by a store to that address. The store instruction updates the value in the cache and in the main memory. If the load hits in the cache, the update in the cache will take place after the load and the execution is correct. If, however, the load misses, the store will already have been issued to the cache core (since the cache is pipelined), and will update the cache line immediately without checking any further conditions. Meanwhile, the refill of the line has begun, and the final state will be that the value that was written to the cache core will be overwritten by the old value from main memory!

Another, less serious, problem with the pipelined cache implementation concerns multiple consecutive loads in the same refill block—something likely to happen in programs with locality of data reference. If the first load misses, the second is highly likely to miss as well, needlessly causing two cache refill operations of the same block.

We use the same solution for the write-after-write hazard and the double refill problem. The cache is given the ability to repeat certain operations that are affected by the dependencies between successive cache lookups. A cache store is repeated when it follows another load miss to the same line, since the load following the store may have overwritten the stored data by a refill. A load is repeated when it is a miss, and it follows another load miss to the same line.

The implementation uses an additional eight-bit comparator to check whether a read miss or write occurring after a read miss is to the same block. If the operations refer to the same block, the operation following the miss is repeated (repeated writes are not repeated to main memory but only to the cache core); if not, they are allowed to proceed. The logic to detect these cases is added to the R process. Interestingly, it would be possible to repeat more operations, e.g., by omitting the comparator and *always* repeating the instruction after a load miss; this would lead to a simpler but slightly less efficient solution.

The MiniMIPS cache system has a variable latency. When a cache miss occurs, the effect on latency is hidden from the rest of the system simply by allowing the handshakes between the cache and the main CPU pipeline to stretch. Internally, the cache array has also a variable latency—the throughput of a leaf of the tree of cache cells is only one half that of the CPU as a whole. But, by interleaving the accesses to different leaves, the total throughput of the cache array is brought up to match that of the main CPU pipeline. This means, on the other hand, that the access time through the cache depends on the address of the data. If two reads are dispatched that refer to data in the same leaf, the latency of the second read will be greater than if they had referred to separate blocks. Again, the delay-insensitivity of the pipeline makes these latency variations transparent to the rest of the system.

3 Part 2: Microarchitecture and Circuit Techniques

In this part, we describe the novel asynchronous pipeline implementation and the related circuit techniques that we have introduced in order to achieve high performance.

All circuits were designed using the synthesis method described in [7]. This method leaves the designer great freedom in choosing between speed, size, or power, as long as the circuit is quasi delay-insensitive. However, to minimize $E\tau^2$, we found that the best results are obtained for very fine pipelines. In order to achieve the high-throughput and low latency we are aiming at, we had to improve our design techniques significantly, by introducing several innovations compared to previous asynchronous designs: low forward-latency pipeline stage, slack matching, and pipelined completion. We pipeline all of our lowest level processes; all cells combine computation and buffering (latching) of the data. This approach to asynchronous pipeline design was first proposed in [4] and used in the design of a digital filter[2].

3.1 Handshaking Protocols

Processes communicate through channels implemented with a four-phase handshake protocol. The data is encoded using two types of DI codes: 1-of-N code or dual-rail code. (In a 1-of-N code, one rail is raised for each value of the data. In a dual-rail code, two rails are used to encode each bit of the binary representation of the value.)

A DI code is characterized by the fact that the data rails alternate between a neutral state that doesn't represent a valid encoding of a data value, and valid state that represents a valid encoding of a data value. See [9]. A channel communication goes through four phases. The sending process waits for an "enable" from the receiving process to be true, then it sets the data rails to a valid value. The receiving process lowers the "enable" after it has latched the data. The sender waits for the "enable" to be false, then sets the data rails to the neutral value. Finally, the receiver raises the "enable" to allow the next communication to start. In HSE notation, sending the value of a one bit variable x over channel C and receiving it in variable y looks like:

$$\begin{aligned}
&\mathit{Sender} \equiv \; [C^{\mathrm{e}}];\; [x^0 \longrightarrow C^0\!\uparrow \,\square\, x^1 \longrightarrow C^1\!\uparrow];\; [\neg C^{\mathrm{e}}];\; C^0\!\downarrow, C^1\!\downarrow;\\
&\mathit{Receiver} \equiv \; [C^0 \longrightarrow y^0\!\uparrow \,\square\, C^1 \longrightarrow y^1\!\uparrow];\; C^{\mathrm{e}}\!\downarrow;\; [\neg C^0 \wedge \neg C^1];\; C^{\mathrm{e}}\!\uparrow;
\end{aligned}$$

In brief, $*[S]$ repeats statement S infinitely, the $[B]$ waits for a boolean expression to be true, and the $[B_0 \rightarrow S_0 \square B_1 \rightarrow S_1]$ waits for one of the boolean expression B_0 or B_1 to become true, then executes the corresponding statement S_0 or S_1. The semicolon sequences two statements, and the comma lets two statements execute in parallel. The $v\!\uparrow$ and $v\!\downarrow$ set a boolean variable v to true and false respectively. The superscripts are used to indicate the data rails and enable of a channel.

3.2 Handshaking Reshuffling

Consider a simple buffer stage that receives one bit of data x on channel L, and sends it (without computation) on channel R:

$$*[\; L?x;\;\; R!x]$$

Using the send and receive hanshakes, we implement the buffer stage as:

$$\begin{aligned}
*[&[L^0 \longrightarrow x^0\!\uparrow \,\square\, L^1 \longrightarrow x^1\!\uparrow];\; L^{\mathrm{e}}\!\downarrow;\; [\neg L^0 \wedge \neg L^1];\; L^{\mathrm{e}}\!\uparrow;\\
&[x^0 \longrightarrow R^0\!\uparrow \,\square\, x^1 \longrightarrow R^1\!\uparrow];\; [\neg R^{\mathrm{e}}];\; R^0\!\downarrow, R^1\!\downarrow;\; [R^{\mathrm{e}}]]
\end{aligned}$$

For the MIPS, we used only three different types of HSE reshufflings that are all directly derived from three reshufflings of the above HSE for the buffer. A reshuffling of a HSE rearranges the non data-dependent portions of the four-phase communication to improve speed and size. The three reshufflings are called HB (half-buffer), PCHB (precharge-logic half-buffer), and PCFB (precharge-logic full-buffer). All three reshuffling eliminate the explicit variable x by computing the output data directly from the input data.

For a half-buffer, the communication phases of the inputs alternate with the phases of the outputs.

$$\begin{aligned}
\mathit{HB} \equiv \; *[&[R^{\mathrm{e}}];\; [L^0 \longrightarrow R^0\!\uparrow \,\square\, L^1 \longrightarrow R^1\!\uparrow];\; L^{\mathrm{e}}\!\downarrow;\\
&[\neg R^{\mathrm{e}}];\; [\neg L^0 \wedge \neg L^1];\; R^0\!\downarrow, R^1\!\downarrow;\; L^{\mathrm{e}}\!\uparrow]
\end{aligned}$$

This circuit is well-known, and is widely used as an asynchronous QDI buffer. By adding more inputs and more outputs, and by including more complex logic in the selection statements, this type of cell may also do computation as well as buffering.

For large numbers of inputs, the half-buffer is inefficient as the wait for the neutrality of the inputs ($[\neg L^0 \wedge \neg L^1]$) grows unmanageably large, and must be done before the outputs are reset ($R^0\!\downarrow, R^1\!\downarrow$). For most cells with computation, we use the second reshuffling (*precharged half-buffer*). This reshuffling postpones the wait for the input neutrality. Since

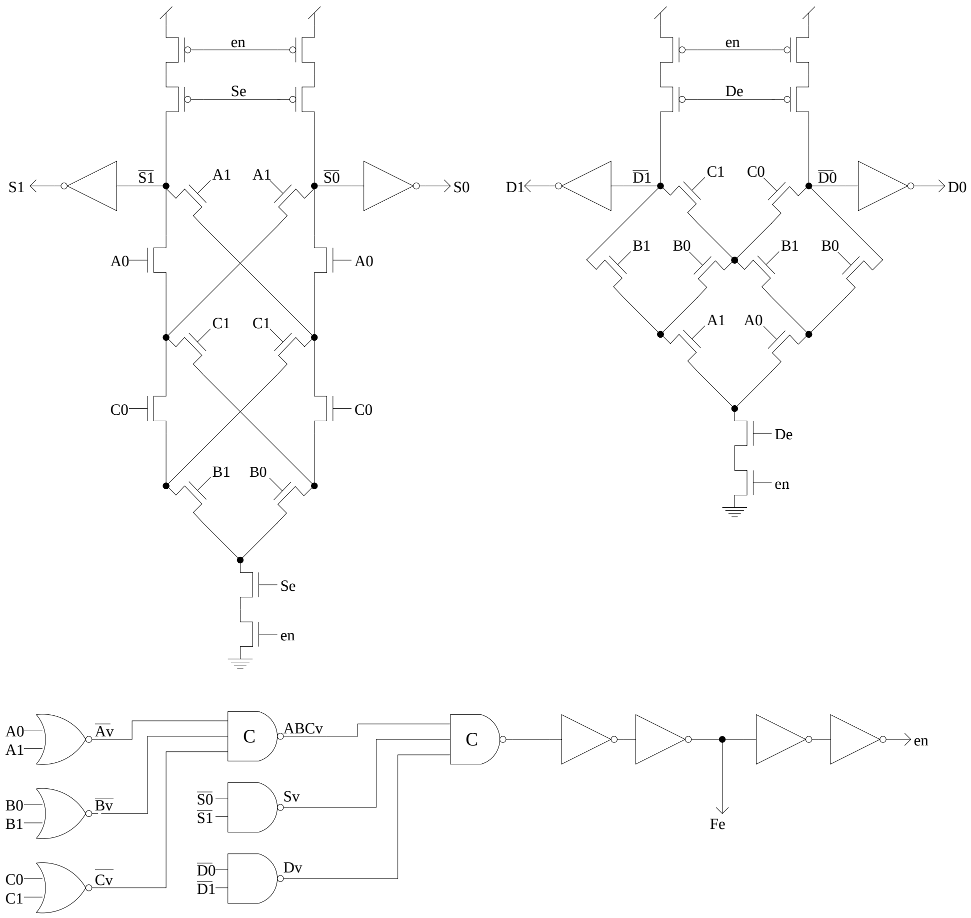

Figure 3. PCHB Fulladder

there is no data transmitted on the second half of the handshake protocol, this is allowed. The PCHB reshuffling for a buffer is:

$$\begin{aligned} PCHB \equiv\ & *[[R^{e}];\ [L^{0} \longrightarrow R^{0}\uparrow [\!] L^{1} \longrightarrow R^{1}\uparrow];\ L^{e}\downarrow; \\ & [\neg R^{e}];\ R^{0}\downarrow, R^{1}\downarrow;\ [\neg L^{0} \wedge \neg L^{1}];\ L^{e}\uparrow] \end{aligned}$$

Adding more input and output channels simply enlarges the expressions that $L^{e}\uparrow$ and $L^{e}\downarrow$ must wait for. These expressions can be broken down into completion trees of OR gates followed by C-elements to check the validity and neutrality of the input and output channels. In the circuit implementation, the main block of computation looks like precharge dual-rail domino-logic (hence the name). We usually do the logic computation in a network of n-transistors, precharged by one or two p-gates in series. These inverted rails are sent through inverters to produce the output, so that these cells can be easily composed. A PCHB implementation of a one-bit full adder is shown in Figure 3.

In several circumstances requiring higher speed (mainly the buses, register bypass, and register file), the PCFB reshuffling is used. This reshuffling allows the reset phases of the input and output communications to execute in parallel. An internal state variable en is added so that the PRS can be compiled. The HSE for a one bit buffer is:

$$
\begin{aligned}
PCFB \equiv\ & *[[R^e];\ [L^0 \longrightarrow R^0\!\uparrow [] L^1 \longrightarrow R^1\!\uparrow];\ L^e\!\downarrow;\ en\!\downarrow; \\
& ([\neg R^e];\ R^0\!\downarrow, R^1\!\downarrow),\ ([\neg L^0 \wedge \neg L^1];\ L^e\!\uparrow);\ en\!\uparrow]
\end{aligned}
$$

Although this reshuffling requires a state variable, it leads to a very efficient CMOS implementation. Essentially both the output data and the input enables are produced by precharge logic blocks as soon as possible. The *en* must check that the output data has actually become neutral, but the input enables don't have to. This allows the neutrality test of the output data to overlap with raising the left enables, thus saving a few transitions on the handshake cycle, and reducing the load on the circuits for the input enables.

These types of processes produce a deep pipeline when composed, but each process has the forward latency of only a N-logic pulldown followed by an inverter. Since no extra latches are required, these processes are energy and area efficient, compared to non-pipelined QDI alternatives. In forward latency, these cells are superior to synchronous dual rail domino logic, since there are no latches at the end of a block of logic, and no extra timing margin is necessary. The intrinsic pipelining also yields a very competitive throughput. Although restricted, this design style has produced high performance circuits for caches, buses, registers, control, and arithmetic.

3.3 Slack Matching

The *slack* of a communication channel in an asynchronous system refers to the amount of buffering present in the channel. The slack of a channel indicates the maximum difference possible between the number of messages sent and received on the channel. This slack is implemented with buffers that are pipeline stages.

Given a linear array of pipeline stages, the steady-state cycle time and latency through the pipeline stages are governed by the cycle time of individual stages and constraints introduced due to interactions between neighboring stages [1]. The pipeline stages described above introduce constraints on the cycle time τ of the form $\tau \geq t_0$. When we have a ring of N pipeline stages, each with a steady-state cycle time of τ and latency l with one token flowing through the ring, we have an additional constraint, namely that $\tau = lN$. We also have latency constraints of the form $l \geq l_0$. The optimal cycle time t_0 can be attained when $N = N_0 = t_0/l_0$.

If a ring of pipeline stages in the system does not have N_0 stages per token, it will not operate at the maximum possible throughput. We can increase its throughput by modifying the number of stages in the ring until we do have N_0 stages per token. This can be done in various ways, including modifying the extent to which a unit in the ring has been pipelined and explicitly introducing buffer stages in the ring. We call this process of adjusting the slack on channels to optimize the throughput of the system *slack matching*[4],[2] .

The pipeline stages used in the MiniMIPS typically have $t_0 \approx 3.5$ns and $l_0 \approx 0.44$ns. Therefore, the optimal cycle time in a ring of these pipeline stages is obtained at $N_0 \approx 8$.

Rings of pipeline stages are present in a number of places in the MiniMIPS. Some of the critical loops we optimized were the pcunit-fetch-decode loop, the loop through an execution unit to the register file and back to another execution unit that corresponds to data-dependent instructions, and the loop in the pipelined cache implementation. For example, the pcunit-fetch-decode loop contains two tokens; it was optimized to contain 16 pipeline stages so as to operate at 280MHz.

The execution units in the MiniMIPS have different latencies as a result of different degrees of pipelining, ranging from 1/4th of a cycle for logical operations to over 1 cycle for a load from data memory. We alternate the use of the result buses to compensate for this latency variation. Suppose a load from memory is writing its result back on bus

$Z0$. If the following instruction is not data-dependent on it (indeed, the MIPS architecture specifies that the compiler should guarantee that this is not the case since loads have a one instruction data-hazard), it can write back its results in parallel on bus $Z1$. Observe that interleaving the use of the result buses permits us to tolerate a latency mismatch of upto one cycle without dropping below peak throughput.

3.4 Slack Elasticity

The high degree of pipelining makes the processor a highly concurrent program, increasing the complexity of the design. As we have seen, increasing the slack on channels can be used to increase the performance. As we will show, increasing the amount of pipelining in an asynchronous system can also be thought of as increasing the slack on channels in the computation.

But increasing the slack on channels can modify the synchronization behavior of the computation, leading to an incorrect implementation. In this section, we introduce a property of the computation, *slack elasticity*, which is necessary and sufficient to maintain the correctness of the computation under modification of the slack on channels.

Increasing pipelining in an asynchronous system corresponds to increasing the slack on a communication channel in the system. For example, suppose we have a single function computation block described by the following program:

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

If we can increase the slack on communication channel R by 1, we can pipeline the computation to obtain:

$$*[\ L?x;\ \ I!f(x)\ ]\ \|\ *[\ I?y;\ \ R!g(y)\ ]$$

As far as the environment of this unit is concerned, the unit reads in values x and computes $g(f(x))$; the only thing that has changed is the amount of buffering on the channel R. We would like to determine the exact conditions under which we can modify the amount of slack on a channel in an asynchronous computation, since this corresponds to conditions under which we can increase the amount of pipelining in an asynchronous system.

In a deterministic computation, we can always increase the slack on a communication channel without affecting correctness. The reason for this is straightforward: the sequence of sends and receives are predetermined, and adding buffering on a channel doesn't affect the result of the computation. The only problem arises when a program probes a channel to determine whether a communication on the channel can complete, and then performs different computations depending on the result of the probe. We define a *decision point* to be a point in the computation where a program makes such a choice. The detailed description of this concept is provided in [5]. Decision points correspond to those points in the system where arbitration is needed. When the computation reaches a decision point, some unit in the system uses an arbiter to pick between a number of alternatives.

The main result that characterizes when we can increase the amount of slack on a channel is [5]: *Increasing the slack of a channel by 1 does not affect the correctness of the computation if (and only if) it does not introduce additional decision points in the system.*

Computations in which we can add slack without affecting the correctness are said to be *slack elastic*. The implementation of the MiniMIPS has only one arbitration device—the one in the exception-handling mechanism. Therefore, the only decision points that might be introduced lie in this part of the processor. The arbitration device in the exception handling mechanism is exercised whenever an exception occurs. Since an exception could potentially be raised by every program counter value, the set of possible points in the computation where arbitration is necessary cannot be increased. This implies that we can increase the slack on any communication channel in the system without affecting the correctness of the resulting design.

Therefore, the design of the MiniMIPS is highly modular—individual execution units can be pipelined without affecting the correctness of the processor so long as they maintain their input/output characteristics in terms of the values they compute.

In the next section, we introduce a transformation, *pipelined completion*, that removes the delay attached to the completion of a datapath operation. Pipelined completion increases the slack on communication channels between the control and datapath, and between the different bits of the datapath. Because of the slack elasticity of the MiniMIPS design, we can always use pipelined completion in the MiniMIPS datapath.

4 Pipelined Completion

In an asynchronous system, since there is no clock signal, a pipeline stage or functional unit needs to generate its own "completion signal" to indicate the completion of that stage and the availability of the result for the next stage. The generation of a completion signal takes a time proportional to $logN$, where N is the number of bits of output data. Since a critical cycle usually contains 2 completion signals, for 32 or 64 bits of output, the performance of a pipeline including such a mechanism is seriously limited. The completion detection problem is the cause of the remaining skepticism against asynchronous techniques.

We have solved this problem by pipelining the completion mechanism. The data path is decomposed into small units—say, 4 bits wide. Each unit generates a completion signal through a small completion tree that has a small, constant, delay. The collection of all the unit completion signals is pipelined, and therefore does not appear on any critical cycle. Now, the completion detection overhead is reduced to a small constant, that of a single unit. By increasing the pipelining, this method could increase the latency on pipeline stalls, and therefore requires additional care in the design of the datapath, and the reduction of the forward latency.

In our current design style, the forward latency of a pipeline stage is only 1/8 of the cycle time, and, as we explained, the cycle time is decreased by a combination of deep pipelining and pipelined completion. We think that the frequencies we can obtain with this approach are very difficult to achieve for clocked designs due to clock frequency limitations. For example, the latency of a bare pipeline stage in 0.6CMOS is 500 picosecs, which corresponds to a 2GHz clock frequency. Similarly, the low forward latencies we achieve can be difficult to match in a clocked design because of the margins required for proper clock operation.

5 Performance Comparisons

We estimate that, at 3.3 V and 75 °C, the MiniMIPS will dissipate 7 W and run at 280 MIPS. At 2.0 V and 75 °C, it will dissipate 1 W and run at 150 MIPS. It should exceed 300 MIPS at 25 °C. The transistor count is 1.25M for the caches, and 250K for the datapath and control. The overall floorplan is shown in Figure 4. The gate design is very conservative: There are no delay assumptions, and all gates are "staticized." The transistor count is inflated by the large number of staticizers, and by the fact that long transistor gates are cut into sections. The power consumption is somewhat higher than we expected, and breaks down as follows. The pcunit/memory/decode part consumes 3 W;

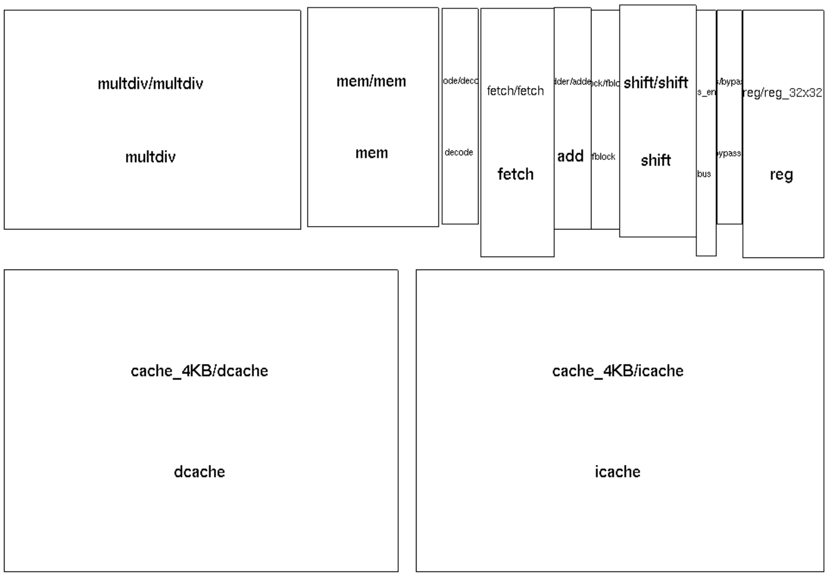

Figure 4. The floorplan for the MiniMIPS

the registers/bus/bypass consumes 3 W; a typical execution unit operation consumes 1 W as a rough average.

Comparing the performance of the Caltech MiniMIPS to that of existing synchronous and asynchronous processors is not straightforward. High-performance versions of the MIPS R3000 do not exist, and high-performance processors fabricated in similar technologies are quite different. Furthermore, the technologies used in current microprocessor designs are not always directly comparable to the 0.6 μm SCMOS process (Hewlett-Packard via MOSIS) used for the MiniMIPS. A comparison of the MiniMIPS to the Orion R4600, the DEC StrongARM, and the AMULET2 is summarized in the table below.

By MIPS, we mean millions of R3000/R4000 instructions per second for the MiniMIPS and the Orion. The numbers for the ARM architecture processors (AMULET and StrongARM) refer to their published Dhrystone MIPS figures. Since errors in the performance are, given P, reflected to the third power in the $E\tau^2$ figures, we caution that $E\tau^2$ figures really are only comparable between processors that do similar amounts of work per "operation."

Processor	f.s. μm	$1/\tau$ MHz	MIPS	P W	word size	MIPS^2/E $10^{21}\,\mathrm{J}^{-1}\mathrm{s}^{-2}$
MiniMIPS @3.3V	0.6	280	280	7	32	3100
MiniMIPS @2V	0.6	150	150	1	32	3375
AMULET2e	0.5	35	38	0.150	32	365
Orion R4600	0.64	150	150	3.0	64	1125
StrongARM SA110	0.35	200	235	0.9	32	14420

In spite of the huge architectural differences between the MiniMIPS and the DEC Alpha, it is worth mentioning that the 280 MHz execution rate of the MiniMIPS compares well with the 300 MHz execution rate of the Digital Alpha 21164 processor in a 0.5 μm technology. The newer 0.35 μm 21164 demonstrates the power and speed advantages of scaling to about the same extent that we would expect to see in a scaled MiniMIPS processor.

The Digital StrongARM SA110 processor is a low-power commercial microprocessor[11]. At 2.0 V Vdd, a power consumption of about 0.5 W and performance of about 185 MIPS was expected from the StrongARM. At the same voltage, we expect performances around 150 MIPS and 1 W for the MiniMIPS. We note that the StrongARM currently beats the MiniMIPS by a factor of three on the $E\tau^2$ metric. To attempt to correct for the difference in fabrication technology, we can use Digital's experience in scaling the Alpha 21164 processor, which indicates that the $E\tau^2$ metric improves by approximately a factor of 9.5 when going from the old 0.5 μm technology to the newer generation's 0.35 μm. Assuming the MiniMIPS would scale similarly, we would achieve twice the performance of the SA110 in the same manufacturing technology, using the $E\tau^2$ metric.

The Orion R4600 is another low-power commercial microprocessor. Although it contains a floating-point unit, the precautions taken by the designers to isolate the floating-point unit when not in use make it possible to compare it to the MiniMIPS on integer code.

The AMULET2e is a current self-timed processor from the University of Manchester. It was announced in 1996 at a performance of 38 MIPS in 0.5 μm CMOS. The expected performance for the MiniMIPS in a similar technology exceeds the AMULET2e's performance by almost an order of magnitude!

6 Conclusion

With a total transistor count of 1.5 million, this project stretches the limit of what a small research group can (and should?) do in a university environment, especially when every cell is laid out manually. Yet, we believe an experiment of that scale was necessary to demonstrate the capability of asynchronous VLSI for high-performance microprocessors. Assuming that fabrication will confirm the performance estimated by SPICE simulation, we can draw the following conclusions from the experiment.

Provided that each stage is finely decomposed, asynchronous pipelines can deliver a throughput at least equivalent—and we believe even superior—to that of the best clocked designs. Furthermore, the same design, run at the optimal voltage for energy consumption, compares favorably in terms of energy consumption with the best low-power architectures.

At the circuit level, the challenge was to partition each complex cell into a large collection of elementary communicating cells. Each cell should be small enough to have only 14 to 18 transitions on a cycle. Yet, the decomposition should not increase the forward latency of the pipeline stages since that would degrade the throughput of the whole pipeline through the feedback loops. Not only did the cells have to be designed very carefully, but we also had to deal with problems related to the pipeline dynamics of the system. A typical example was the frustrating situation where two cells show significant degradation in throughput once they are composed together. To deal with pipeline dynamics problems, we introduced what we call slack-matching techniques.

At the architecture level, we had to invent new solutions for problems like precise exceptions, pipelined caches, register allocation and bypassing that would be usable in the context of a totally distributed system. The structure of the asynchronous pipeline distinguishes itself from that of a synchronous one by its flexibility, i.e., its automatic adaptation to the type of instruction being executed, and by the ability to take advantage of data-dependent execution times.

The requirement of decomposing the whole system into a network of, say, ten thousand communicating cells, required refining and extending our techniques for synthesis by program decomposition. Starting from a rather simple sequential "seed" program that describes the whole processor, we were able to decompose it into the final network of cells by the successive application of a surprisingly small number of transformations. In order to argue that the final program was semantically equivalent to the original one, we had to introduce a new notion of equivalence: slack elasticity.

In conclusion, the project has been an excellent catalyst and testbed for a new design method which, we believe, constitutes an order-of-magnitude improvement on the state-of-the-art in asynchronous design. However, we knew at the start of the project that the standard RISC instruction set of the MIPS family would not allow us to take full advantage of the possibilities offered by an asynchronous approach.

Acknowledgments

We wish to thank Marcel van der Goot and Peter Hofstee for their excellent comments and suggestions, and Cindy Ferrini for her help in the preparation of the manuscript. Acknowledgement is also due to Marc Renaudin who participated in the design of the TLB.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, and monitored by the Army Research Office. T. K. Lee is partially supported by a Direct Grant from the Chinese University of Hong Kong.

References

[1] S.M.Burns and A.J. Martin. Performance Analysis and Optimization of Asynchronous Circuits. *Proceedings Advanced Research in VLSI 1991*, ed. C.H. Sequin, MIT Press, 1991.

[2] U.V. Cummings, A.M. Lines, A.J. Martin. An Asynchronous Pipelined Lattice Structure Filter. *Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, 1994.

[3] G. Kane and J. Heinrich. MIPS RISC Architecture. Prentice-Hall, 1992.

[4] Andrew M. Lines. Pipelined Asynchronous Circuits. MS Thesis, Caltech-CS-TR-95-21, 1995.

[5] Rajit Manohar. The Impact of Asynchrony on Computer Architecture. PhD Thesis, Caltech, In preparation, 1997.

[6] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus. The Design of an Asynchronous Microprocessor. *Decennial Caltech Conference on VLSI*, ed. C.L. Seitz, MIT Press, 351-273, 1989.

[7] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. *Formal Methods for VLSI Design*, ed. J. Staunstrup, North-Holland, 1990.

[8] Alain J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conference on Advanced Research in VLSI*, ed. W.J. Dally, MIT Press, 1990.

[9] Alain J. Martin. Asynchronous Datapaths and the Design of an Asynchronous Adder. *Formal Methods in System Design*, 1:1, Kluwer, 117-137, 1992.

[10] Mika Nyström. Pipelined Asynchronous Cache Design. MS Thesis, Caltech-CS-TR-97-21, 1997.

[11] S. Santhanam. StrongARM 110: A 160MHz 32b 0.5W CMOS ARM processor. *Proceedings of HotChips VIII*, 119-130, 1996.