

# Decoupled Pipelines: Rationale, Analysis, and Evaluation

Frederick A. Koopmans, Sanjay J. Patel  
Department of Electrical and Computer Engineering  
University of Illinois At Urbana-Champaign  
{fkoopman@csrd.uiuc.edu, sjp@crhc.uiuc.edu}

## Abstract

*This paper analyzes the potential of asynchronous, decoupled pipelines from an architectural viewpoint. Previous work in this area has been implementation oriented and has concentrated on developing circuit models and design tools needed to actually build a simple clockless processor. This past work has shown that asynchronous processing is a viable approach, with several inherent advantages over synchronous processing. This paper breaks from this implementation focus and instead considers what is gained architecturally when the globally synchronous clock is removed. We first present the design of a decoupled, self-timed asynchronous architecture in which each pipeline component is able to determine its own operating speed. Our design includes micropipelines between stages to provide extra elasticity, allowing the pipeline to dynamically adjust to long latency operations. We then show how this organization enables a new style of prediction-based optimizations that is not possible in a synchronous processor. Finally, we provide preliminary results of a complete VHDL simulator which demonstrate that these optimizations can carry a significant performance advantage.*

## 1 Introduction

In the past decade, there has been a rebirth of interest in asynchronous design as a potential framework for building highly efficient microprocessors. Much of this regained interest into what some considered a dead field can be attributed to emerging design challenges faced by synchronous processors. These include the difficulty of generating and distributing a global high frequency clock as well as the increased importance of power efficiency. Initial research has shown that asynchronous processors have the potential to solve these problems in a very elegant manner. First, asynchronous processors do not have a clock and therefore do not suffer from problems related to clock skew. Second, an asynchronous design has an inherent power advantage because all transactions and computations are requested explicitly. Thus, an asynchronous circuit only consumes dynamic power when it is performing useful work. At all other times, unused circuits are implicitly turned off [1], as opposed to a synchronous processor in which unused circuits can consume a significant amount of power by merely

reacting to a toggling clock. [2] estimates that synchronous processors expend as much as 40% of total power consumption generating and distributing the clock. Previous research has also identified several secondary advantages of asynchronous processing including better technology migration and increased design modularity.

There is now a litany of proposed asynchronous architectures [3][4][5][6][7] including at least one commercial implementation [3]. For the most part, these projects have focused on low-level implementation issues such as developing new techniques and tools for asynchronous circuit design and validation [8][9]. As a result, these processors have generally targeted modest architectures, comparable to simple embedded processors. In this paper, we focus on developing a high-performance asynchronous architecture. Our approach is to start with the basic infrastructure of a modern high-end superscalar processor and then gradually incorporate asynchronous behavior into the pipeline. We believe that an evolution from synchronous to asynchronous is a more natural approach than trying to redesign an asynchronous architecture from scratch. With this goal in mind, this paper sets out to analyze the fundamental differences between synchronous and asynchronous architectures.

At a high level, our proposed architecture looks similar to the organization of a generic out-of-order processor. The pipeline is divided into a number of individual stages to provide high throughput, concurrency, and ease of design. In keeping the pipeline structure of a traditional synchronous processor, our decoupled processor is not purely globally asynchronous. In a synchronous processor, all stages operate at the same speed such that instructions march down the pipeline in lock step. However, in our decoupled design, each stage is disconnected from its neighbors by an explicit go/acknowledge protocol. This allows each component to operate independently and determine its own latency, which can be different for each operation. This flexibility is the most important architectural difference between synchronous and asynchronous processors because it allows each pipeline operation to be individually optimized for its own critical path. The primary contributions of this paper include our

collective analysis of decoupled pipelines plus several specific hardware optimizations that are enabled by our architecture, along with a preliminary performance evaluation based on a VHDL simulator.

The rest of the paper is organized as follows: Section 2 briefly describes the communication protocols used in our design and states our assumptions of the underlying self-timed circuitry. Section 3 presents an overview of our asynchronous processor with extended descriptions of several key features. In Section 4, we analyze this design to uncover specific architectural advantages made possible by the decoupled architecture. Section 5 presents our experimental environment and the preliminary results that we have achieved. Finally, Section 0 discusses a few related projects and offers some concluding remarks.

## 2 Background

### 2.1 Asynchronous Communication

In this section, we introduce the communication protocols used to coordinate data transactions between pipeline stages. The most general asynchronous communication protocol consists of a bidirectional 4-way handshake between participants. However, this generality is not needed when there is only one sender and one receiver, as is frequently the case with consecutive pipeline stages. For this relationship, we use a simple level-insensitive 2-way handshake to pass instruction bundles from one stage to the next. In this protocol, the literal values of the two coordinating signals do not convey any actual meaning. Instead, communication occurs through a series of discrete events encoded by signal transitions. An *event* is produced simply by toggling a signal from one state to the other. For instance, when the sender wants to transfer data to the receiver, it sets the data lines and generates a go event. When the receiver observes that the go signal has transitioned, it will read the data lines and send an acknowledgement event. The sender is then free to modify the data and begin the next transaction. Figure 1 shows two of these asynchronous transactions.

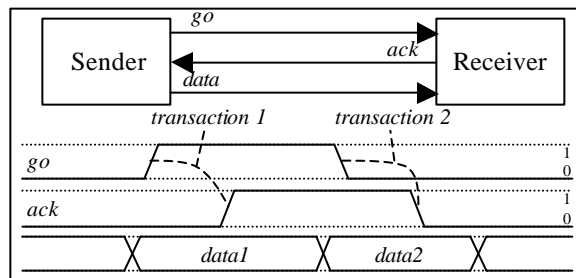


Figure 1: This timing diagram depicts two asynchronous transactions administered by the 2-way level insensitive communication protocol.

This protocol is ideal for simple producer-consumer relationships frequently found in pipelined circuitry. However, there are also more complicated situations in which other protocols are better suited. For instance, our design uses the standard 4-way protocol to access the caches because data flows in both directions. More information on these communications protocols is available in [10].

Within a single stage, we use three types of event-based logic gates to make control related decisions. The C, XOR, and SEL gates shown in Figure 2 provide the asynchronous equivalence of common synchronous logic gates. The **C-gate**, also called the Celement, performs the logical AND of its input event signals. This gate waits for events to arrive on both inputs before generating an event on the output. The **XOR-gate** performs the logical OR of its input event signals. This gate generates an event on its output whenever an event arrives on either of its inputs. The **SEL-gate** performs the role of a logical demultiplexer. An event on the input is transferred to exactly one of the outputs as determined by the select input, which is a traditional level-sensitive signal. Dual input/output versions of these gates are shown in Figure 2, although larger gates can also be constructed in a straightforward manner. As we will see, these simple gates are powerful enough to construct sophisticated control circuits for a pipelined processor. A more detailed description of these and other event-based logic gates is available in [10], which also includes CMOS implementations.

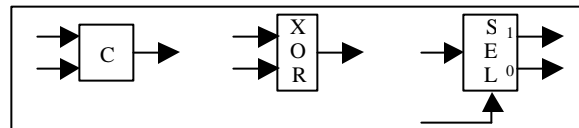


Figure 2: These three gates are the fundamental building blocks of event-based logic design.

### 2.2 Self-Timed Circuitry

The final piece of the asynchronous puzzle is the ability to drive computational circuits with these event signals. In Figure 3, the *start* event is provided when the inputs are available. The circuit's job is then to produce the *done* event when the outputs become stable. This behavior is known as self-timed circuitry. By chaining several self-timed components together, we can create very complex devices such as a pipelined processor.

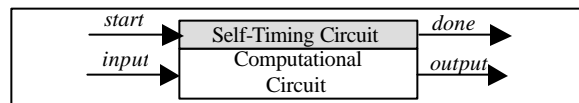


Figure 3: This circuit represents a generic self-timed circuit that automatically produces the done signal based on when it receives the start signal.

There are a variety of different ways to implement self-timed circuitry ranging from simple/conservative to complex/aggressive approaches. A simple approach, known as the bounded-delay method, is to statically determine the computational circuit's critical path, specified by a number of logic gates. The start event can then be fed through a series of dummy gates to mimic the critical path of the circuit and produce the done event when the outputs are known to be stable. One drawback of this approach is that it must typically overestimate the critical path to ensure correctness for all situations. More sophisticated methodologies include I-net and trace-based models which accurately generate the done signal at approximately the same time that the outputs become available. These methods are also advantageous because they allow the circuit's latency to vary with the input data. However, delay-insensitive approaches such as these require that the computational circuit be completely redesigned in asynchronous logic, which includes eliminating data hazards that do not affect bounded-delay or synchronous circuits. With the bounded-delay model, the computational circuit can essentially be copied directly from a synchronous design. A thorough evaluation of these and other self-timed implementation strategies is available in [11]. For the purposes of this paper, the design of the self-timed circuits will remain implementation dependent and will not be discussed further. We will instead focus on the architectural potential of a self-timed pipeline.

### 3 Design

#### 3.1 The DSEP Organization

Figure 4 shows the microarchitecture of the Decoupled, Self-Timed, Elastic Pipeline, or simply the DSEP. This design supports out-of-order execution with in-order retirement and also is configurable for multiple instruction issue. This organization was intentionally chosen to resemble the Intel P6 microarchitecture [13]. The principle difference between the DSEP and the P6 is that each pipeline stage is completely decoupled from its neighboring stages via the asynchronous protocols discussed earlier. This allows each stage to determine its own latency, without regard to the global critical path or pipeline stage balancing. The remainder of this section briefly discusses the design and operation of the DSEP.

**Fetch** reads a block of instructions from the instruction cache and passes them to *Decode*. At the same time, *Fetch* accesses the BTB/predictor to determine the next fetch address. Depending on the specific configuration of the processor, the instruction block may contain anywhere from 1 to 4 instructions.

**Decode** identifies the instruction's type and format, the source and destination registers, and the functional unit that will execute each instruction. Additionally, *Decode* assigns each instruction a sequence number to identify the instruction as it flows through the pipeline.

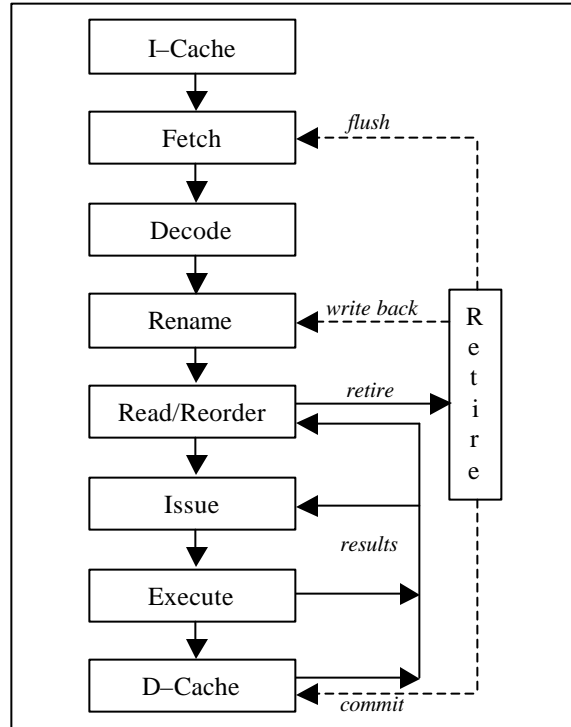


Figure 4: This figure shows the high-level organization of the Decoupled, Self-Timed, Elastic Pipeline. Each box represents a decoupled component of the processor.

**Rename** contains both the retired register file (RRF) and the register alias table (RAT) in addition to keeping track of which logical registers are currently retired and which have been renamed. For retired operands, the rename stage reads the 32-bit value from the RRF. For renamed operands, the sequence number of the instruction that will produce the desired value is read from the RAT. Additionally, each instruction renames its destination register by writing its sequence number to the RAT.

**Read** contains the reorder buffer which stores the results of completed instructions and is indexed by sequence number. Instructions coming from *Rename* check this buffer for missing operands and perform a read only if the value is available. Additionally, each instruction reserves its position in the reorder buffer by clearing the valid and dirty bits of its entry.

**Issue** contains the reservation stations to hold instructions until their remaining operands arrive from the execution units. In our base design, the instruction window contains 1 private entry per functional unit.

When an instruction has obtained all of its operands, it is queued for execution to its assigned functional unit. The overall makeup of *Issue* is similar to its synchronous counterpart with an important distinction to account for the asynchronous arrival of input operands and asynchronous dynamic scheduling of instructions. We provide a discussion of this particular issue in Section 4.

*Execute* consists of four functional units that perform the desired operation specified by the decode unit. Conditional and unconditional branches are sent to the *branch unit*, loads and stores are sent to the *address unit*, and all remaining instructions are sent to one of the two *integer units*. Upon completion, branch and integer instructions are sent to the reorder buffer, and address instructions are sent to the data cache.

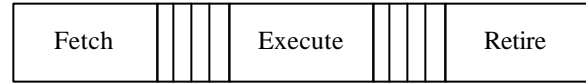
*Retire* reads instructions from the reorder buffer in the original program order and performs any necessary final actions. Most instructions simply need to write their result back to the RRF. However, store instructions must send a commitment to the data cache, and mispredicted branches must flush the pipeline and send a new address to the fetch unit.

### 3.2 Pipeline Elasticity

As stated, each of these 7 stages implements a 2-way asynchronous handshake with its neighboring stages. This protocol allows each stage to determine its own latency but is not sufficient for the stages to sustain different rates of operation. Consider for example that when Fetch sends an instruction to Decode, it cannot begin fetching the next instruction until it receives an acknowledgement for the previous instruction. Thus, the performance of each stage is still limited by its surrounding stages. We resolve this problem by placing buffers between stages to further decouple the pipeline. This allows a stage to send an instruction to its output buffer and begin processing the next instruction immediately, therefore reducing the frequency of pipeline stalls.

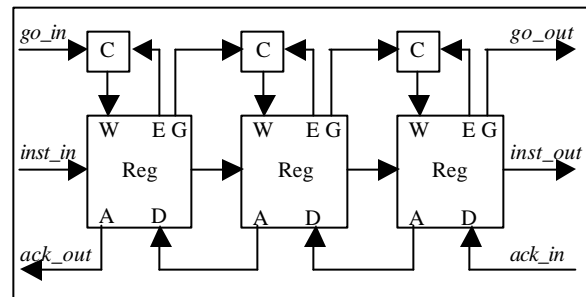
As a concrete example, consider the simple 3stage, decoupled pipeline in Figure 5. Although equilibrium requires that each stage will average the same operating speed, certain events may occur that allow one or more stages to temporarily fall behind the others. For example, if Execute stalls on a cache miss, Fetch can continue generating instructions until it fills the first buffer. Similarly, Retire can continue committing instructions until the second buffer runs empty. Execute can then catch back up to the other stages by executing instructions in parallel, or when one of the other stages stalls on a long latency operation. The buffers are therefore able to hide some

of the latency of the cache miss by preventing or postponing stalls in the Fetch and Retire stages. This behavior is known as pipeline elasticity, because the pipeline stretches and shrinks to absorb the latency of the instruction stream.



**Figure 5:** This figure depicts an elastic 3-stage pipeline with two decoupling buffers. This organization divides the pipeline into its three core components.

In typical out-of-order processors (whether synchronous or asynchronous), this form of global elasticity is provided by the instruction window and the reorder buffer. In a synchronous processor, there is no need to provide additional elasticity because the rest of the pipeline has a fixed latency. However, the DSEP does benefit from additional elasticity because the buffers are able to smooth the operation of the variable pipeline. Although the instruction window and reorder buffer are typically implemented in expensive circular arrays to allow non-sequential access, the DSEP uses *micropipelines* to implement efficient FIFO buffering between stages. Figure 6 shows the layout of an example 3-stage micropipeline.



**Figure 6:** This diagram depicts the operation of a generic 3-stage micropipeline. The C-elements control access to the registers to construct an efficient FIFO buffering device.

This example contains three registers connected in series, each of which is controlled by a C-element (presented earlier in Section 2). The two inputs of each C-element are the go signal and the empty signal, which is just an inverted copy of done. As shown in the diagram, instructions flow with the go events from left to right, while acknowledgments flow from right to left. When an instruction enters the queue, it will move to the last unoccupied register and bubble from left to right until it exits the queue. However, when the queue is empty, the C-elements will open the registers, allowing incoming instructions to quickly slide through the queue with minimal delay. A complete description and analysis of micropipelines is provided in [10] along with several example uses. In the context of building a general purpose asynchronous processor, the micropipeline is an effective method of reducing the

number of pipeline stalls due to variable operating speeds. In Section 5, we present experiments to support this claim and explain where elasticity is most important.

### 3.3 Flushing the Pipeline

This section describes how the DSEP asynchronously flushes and restarts after a mispredicted branch. Because there is no global synchronizing mechanism, this area of the microarchitecture is quite different than its synchronous counterpart. When Retire encounters a mispredicted branch, its first task is to setup a *barrier* to prevent subsequent instructions from completing. Retire then sends the new address to Fetch so that it may begin fetching from the new instruction stream. The first instruction from the new stream is marked *clean* by setting a single bit in its control word. The clean instruction identifies the start of the new instruction stream and serves as an implicit acknowledgement that the flush has been processed by all previous stages. Thus, the flush does not completely finish until the clean instruction reaches the retire stage. A new flush cannot begin until the previous flush has finished, and therefore it is guaranteed that the pipeline will only contain one clean instruction at a time.

In addition to jumping to the new instruction stream, the pipeline must squash all outstanding instructions and preserve the retired pipeline state. To do this, the flush event is asynchronously broadcast to the entire pipeline. When each stage receives the flush event, it will finish processing its current instruction and then perform any actions needed to restore its retired state. The rename stage will validate the RRF and invalidate the RAT, and the reorder stage will invalidate all entries in the reorder buffer. Each stage then sets up a barrier to protect its state from subsequent instructions of the dead instruction stream. This is necessary to catch instructions that may be between stages (e.g. in a buffer) when the flush event is seen. A barrier remains active until it sees the clean instruction, at which point it is removed and the new instructions are allowed to proceed without interference. In summary, the interaction of the clean instruction and the various dead stream barriers allow the pipeline to safely flush in an asynchronous manner.

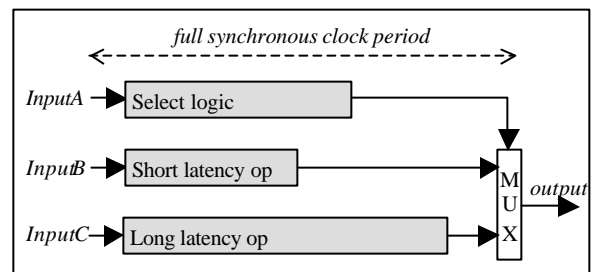
## 4 Analysis

In this section, we analyze the architectural advantages and potential limitations of the DSEP approach. The primary advantage of this design is the ability to exploit the mismatch between average-case delay and worst-case delay. In a fully synchronous processor, every stage in the pipeline runs at the speed of the worst-case stage running in the worst-case input

scenario. Thus, synchronous designers must focus on reducing critical paths and developing good stage balancing to achieve high performance. An asynchronous designer can instead optimize for common case operations and only suffer a minimal performance hit when the worst-case scenario occurs. This strategy is not as applicable in a synchronous design because clock boundaries are often a limiting factor. For example, there is often no benefit gained from optimizing a path beyond the expected cycle time of the design. But by removing the clock, the processor can always benefit from reduced latency. Another advantage of the DSEP approach is that delays are not fixed, even within a single stage. Thus, we can also implement prediction or speculation based optimizations that can ease a critical path without requiring complex recovery mechanisms.

### 4.1 DSEP Optimization Advantages

We begin by presenting the general form of optimizations that are made possible by a decoupled pipeline. Figure 7 shows a generic pipeline stage that contains three logic blocks: a short operation, a long operation and a circuit that selects between the two. If the latency of the long operation is the critical path of the processor, then the minimum cycle time of a synchronous processor (and thus the maximum throughput) is limited by the long logic block. However, in a decoupled approach, the stage can operate *on average* closer to the speed of the select block, depending on how frequently the short latency operation is chosen over the long latency operation. Furthermore, logic speculation techniques such as cache hit prediction or way prediction (i.e., guessing the outcome of the select logic in Figure 7) are often used in heavily pipelined machines to reduce the impact of long latency operations. In a synchronous pipeline, this technique is used to drive the machine cycle time below that of the long-latency operation.



**Figure 7: A case in which the average case optimization can be applied. If the short latency operation is selected often, then the output of this stage can be generated on average at the latency of the select logic.**

The DSEP approach allows this optimization to be used more generally by allowing *intra-stage* optimizations. Thus, if the short operation is

significantly more frequent than the long operation, the maximum throughput of this stage will be based on the latency of the short operation. In the following subsections, we provide several examples of how this type of optimization is applied in the DSEP microarchitecture.

### Fetch Alignment

One of the more significant architectural limitations on fetch bandwidth is taken control flow instructions (i.e. branches and jumps). A taken control flow instruction causes the fetch stream to be non-sequential and disrupts the flow of instructions through the pipeline. For this reason, many schemes have been devised to overcome these effects, such as trace caches and compiler transformations to reduce the frequency of taken branches. Therefore, in the average case, the group of instructions fetched in a cycle will be sequential relative to the previous group. This behavior can be exploited by providing a fast logic path for sequential fetches. Conversely, when a (predicted) taken branch is encountered, a slower logic path can handle masking and aligning the discontinuous cache lines and generating the taken fetch address. Thus, this optimization allows sequential instruction fetches to stream out of the fetch unit faster than discontinuous fetches.

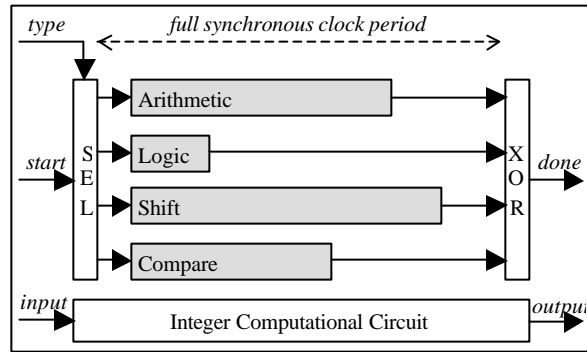
### Non-Uniform Decoders

In RISC architectures, one of the goals of the instruction set encoding is to provide uniform decode latencies for all instructions. Conversely, CISC instructions sets such as x86 were designed so that common instructions can be decoded very quickly, while less frequent instructions may take significantly longer. CISC processors exploit this disparity by decoding simple instructions in one cycle and more complicated operations in multiple cycles [13]. This strategy is even more applicable in a decoupled processor because the decoder is not limited by the granularity of a full clock period. For example, the decoder can be optimized by instruction frequencies so that the most common instructions are decoded in a half “cycle”, while less frequent instructions require multiple cycles. This optimization is likely to remove the decoder from the processor’s critical path.

### Average-Case Execution

In the execution unit, instruction delays are highly variable and there is much potential for improvement through average-case optimizations. For example, consider an integer unit that handles arithmetic, logical, shift, and compare instructions. In a synchronous processor, all of these instructions are likely to have a latency of 1 cycle, even though none of them would probably need the full cycle to complete. In a

decoupled processor, we can improve performance by implementing separate delay circuits for each instruction class as shown in Figure 8. In this circuit, the instruction’s type is used to route the start signal through the proper delay circuit before generating the done signal when the computation has completed. This optimization can significantly improve the throughput of the often overloaded integer units.

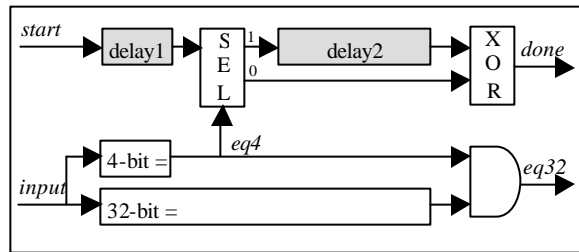


**Figure 8: An average-case integer functional unit that determines its own latency based on instruction type.**

A more aggressive approach to average-case execution is to implement optimizations based on operand values. This strategy is based on the fact that a circuit’s latency is often dependent on its particular inputs. Consider for example the behavior of a 32-bit carry lookahead adder. In the worst-case scenario, the carry bit can propagate all the way from the lowest bit to the highest bit. However, this behavior rarely occurs in practice. In real programs, add instructions are most frequently used for small increments or decrements, computing an address offset, or copying a value from one register to another. In all of these cases, one of the operands is likely to be small, implying that there will be few carries. Therefore, we can reduce the adder’s average latency by speculating that only a small number of ripples will actually be needed, as described in [12]. Additionally, we can exploit the fact that the lower bits of an addition are available sooner than the higher bits. As pointed out in [4], the lower (index) bits of a computed address could be used to begin accessing the cache before the higher (tag) bits have finished computing, potentially reducing a critical path. In all of these situations, the circuit designer must take care to adjust the self-timing circuit model to account for this type of variable latency.

Another important operation that can benefit from this type of input driven optimization is the comparison instruction. In this case, the circuit can be optimized to generate a negative answer very quickly by checking the lower bits first, since these bits frequently reveal a difference. For example, the branch at the end of an iterative single-step loop can be resolved 15 out of 16

times, or 94% of the time by comparing only the lower 4 bits. Also, when the set-if-less-than instruction is used, the branch can always be resolved by comparing only the lowest bit. In a straightforward implementation, a 32-bit comparator can reduce its latency by more than half when only the lowest 4 bits need to be checked, as shown in Figure 9. If the lower 4 bits are not equal, then a false answer is generated after *delay1*. Otherwise, the circuit waits for the result of the full 32-bit comparison, which is available after *delay2*.



**Figure 9:** This 32-bit average-case comparator is more than 50% faster when the comparison can be resolved by the lower 4 bits.

### Cache Access

Average-case optimization can be applied to cache accesses by exploiting spatial and temporal locality. For instance, a cache will frequently have a small number of lines that are temporally hot (i.e., lines that are likely to be accessed again in the very near future). A specific example of this involves the instruction cache lines of a tight loop. In a synchronous processor, it is difficult to implement speculation based on fine-grain temporal locality because doing so may increase the critical path. However, in the DSEP approach, we can easily speculate that a memory reference will access the same cache line as the previous reference. When this occurs, the cache can return the data faster than if it had to perform a full lookup.

The optimizations presented in this section exemplify the intrinsic potential of decoupled pipelines to dynamically adjust to the instruction stream. Certainly, there are many other operations that could also benefit from this style of optimization. The best candidates are likely to be circuits in which the average-case can be implemented significantly faster than the worst-case. With the emergence of 64-bit architectures, this style of optimization may become even more important as average-case and worst-case scenarios become even more unequal.

### 4.2 Asynchronous Dynamic Scheduling

One of the most challenging areas of any high performance processor is the design of the out-of-order scheduler. In a synchronous processor, the job is much easier because execution latencies are generally fixed. This allows an aggressive scheduler to predict when

operands will become available and schedule dependent instructions before the source instruction has even started execution. This is extremely important in a synchronous processor because it allows dependent operations to execute in consecutive cycles. In an asynchronous processor, the scheduler cannot accurately predict when operands will become available because execution latencies are variable. Furthermore, the elastic nature of an asynchronous pipeline can also add unpredictable delay to an instruction's result. Although designing a high performance asynchronous scheduler is still very much an open topic of research, we discuss a few possible approaches to this problem.

The first solution is to simply not allow dependent instructions to execute back to back. The scheduler would then have to wait for an instruction's results to arrive before issuing a dependent operation. This approach might be reasonable for a low-end processor but is unacceptable for a high performance design.

A better approach is for each instruction to produce an early wakeup event at some point before completing execution. This would give the scheduler a head start to locate and issue dependent instructions. As in a synchronous processor, the scheduler can still begin waking up instructions before the source instruction begins execution. However, the wakeup event provides only a hint as to when an operand will become available. Therefore, when a dependent instruction reaches the head of its functional unit queue, it must be sure to wait until its operands arrive before proceeding. If the early wakeup event was delivered accurately, the amount of head-of-line blocking should be kept to a minimum. This solution makes it possible for dependent operations to execute back-to-back, but it is unclear how frequently this will actually occur.

The previous solution is feasible but unlikely to be competitive with an aggressive synchronous scheduler. Therefore, it seems plausible that a commercial implementation may initially choose to synchronize the scheduler and execution units with a local clock. This would unfortunately defeat many of the benefits of an asynchronous architecture, but may be necessary until high performance asynchronous scheduling becomes a reality. Alternatively, it is also possible that back-to-back execution may become less important in the future due to new architectural models such as multithreaded processors.

## 5 Results

### Simulation Environment

To evaluate the performance of the DSEP microarchitecture, we created a detailed VHDL simulator using the Renoir development suite [14]. Our simulator implements the MIPS I instruction set, minus system calls and floating point instructions. The pipeline has the same basic design that was presented in Section 3 with the following specifications. Instructions are fetched and retired one a time, but the functional units may execute up to 4 independent instructions simultaneously. There is a 64-entry reorder buffer and 4-entry split instruction window, such that each functional unit has a single private entry. The instruction and data caches are both 1KB direct mapped caches with 16-byte lines. The branch predictor contains 128 entries and uses standard 2-bit counters. Finally, the pipeline is entirely self-timed using the simple bounded delay model described in Section 2. We have designed this simulator to reflect a simple out-of-order processor and believe that it is realistic and sophisticated enough to provide meaningful evaluation of our decoupled architecture.

### Benchmarks

Unfortunately, our simulator does not provide operating system support, limiting our evaluation to relatively simple benchmarks. With this in mind, we chose two benchmarks that represent common integer codes. The first benchmark is a bubble sort of a 50-element array, and the second is a 10x10 integer matrix multiply. These problem sizes were specifically chosen so that the working set would fit into the cache for both benchmarks. Each benchmark was compiled with GCC 3.0 for the MIPS I instruction set and with O2 optimizations turned on.

#### 5.1 Pipeline Elasticity

The first experiment evaluates the need for extra elasticity in the pipeline. Recall that the instruction window and the reorder buffer already provide global elasticity as explained in Section 3. These two structures effectively decouple the pipeline into its three core components. However, within each of these components, there are several decoupled stages that will suffer from stalling if the pipeline is not able to accommodate variable latencies. To demonstrate this behavior, we have configured the pipeline with the Variable latencies of Appendix A, which include many of the optimizations presented Section 4. At a minimum, we must have a queue of length one between every pair of consecutive stages for the sake of concurrency. To determine the ideal length of each queue, we experimented with many different configurations of the form  $A \times B \times C$ , where A equals the length of the queues in Decode, Rename, and Retire, B

equals the length of the queues in Read, and C equals the length of the queues in Execute. Thus, the minimum possible configuration is  $1 \times 1 \times 1$ . Note that these queues are in addition to the 4-entry instruction window and the 64-entry reorder buffer. We then ran the benchmarks under several configurations and obtained the following results.

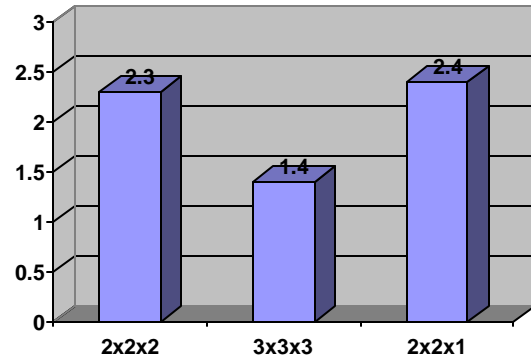


Figure 10: This graph shows the percent speedup of Bubble Sort for 3 micropipeline configurations, calculated as the improvement over the minimum  $1 \times 1 \times 1$  configuration.

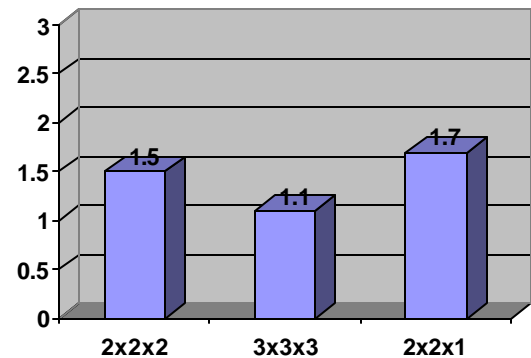


Figure 11: This graph shows the percent speedup of Matrix Multiply for 3 micropipeline configurations, calculated as the improvement over the minimum  $1 \times 1 \times 1$  configuration.

The results show that there is moderate benefit from increasing the length of the queues, about 2.4% and 1.7% in the best case. For both benchmarks, the fastest configuration was  $2 \times 2 \times 1$ , which indicates that the execution units were primarily limited by instruction dependences and not by inelasticity. The percentage of stalls also decreased dramatically from  $74 \times 22 \times 3\%$  to  $30 \times 29 \times 0.3\%$  on average for the two benchmarks. These stalls were even further reduced in the larger configurations, but this did not lead to increased performance, indicating that the average variation between stages is approximately 2x for this pipeline. If the variations were more substantial, we would continue to see improvement in the larger configurations.

## 5.2 Out-of-Order Utilization

As a second experiment, we measured the utilization of the fixed length instruction window and reorder buffer to determine the influence of the pipeline configuration on the performance of the out-of-order engine. Utilization was measured by calculating the number of active instructions in the buffers at increments of 10 time units. Figure 12 shows the results for the two configurations, which show that utilization increased 75% and 40% on average for the IW and RB respectively. Thus, we see that the extra elasticity provided by the micropipelines actually increases the efficiency of the processor's existing resources.

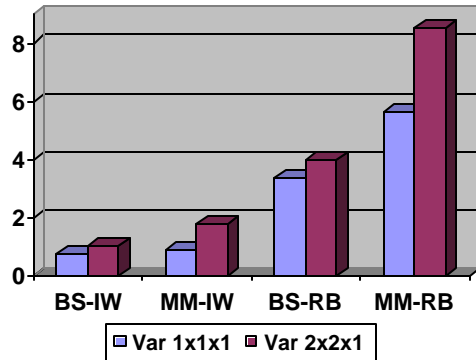


Figure 12: This graph shows how the pipeline configurations influence the utilization of the Instruction Window and the Reorder Buffer. The maximum utilization of the instruction window and reorder buffer is 4 and 64 respectively.

## 5.3 Average-Case Performance

The final experiment evaluates the performance advantage of allowing the pipeline to have variable latencies. We did this by comparing a fixed pipeline, in which every stage has the same latency, to the variable pipelines from the last experiment. To construct the fixed pipeline, we took the variable latencies of Appendix A and rounded each operation up to the nearest multiple of 120 time units, which

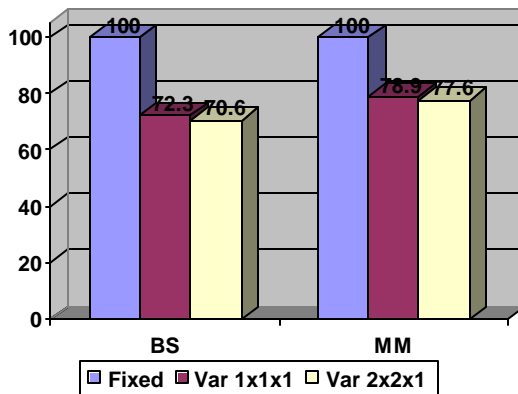


Figure 13: This graph shows the performance advantage of a decoupled, variable pipeline with average-case optimizations.

represents the period of a synchronous clock. We then compared the fixed pipeline to 1x1x1 and 2x2x1 configurations of the variable pipeline. The normalized results, shown in Figure 13, demonstrate that the variable pipelines have a significant advantage over the fixed pipeline, 29.4% and 22.4% in the best case for Bubble Sort and Matrix Multiply respectively.

## 6 Discussion

### 6.1 Asynchronous vs. Multiple Clock Domains

An alternative approach to constructing a decoupled processor is to partition the pipeline into multiple clock domains. As in the case of a self-timed processor, this approach is characterized by how data passes between unsynchronized components. Typically, data crosses domain boundaries by passing through synchronizers, which are effectively synchronous latches with extremely small setup and hold times [15]. For a processor with multiple clock domains, these devices play a similar role as the micropipelines of the DSEP. However, they do not provide elasticity and therefore do not truly decouple the pipeline.

### 6.2 Related Work

The past decade has produced numerous projects that prove that it is feasible to implement a general purpose processor without relying on a global synchronizing clock. The two designs most similar to the DSEP are the MiniMIPS from the California Institute of Technology and the Amulet3 from the University of Manchester. MiniMIPS [4] issues instructions in-order, but allows instructions to finish execution slightly out of order (by 1 instruction). This choice was made to simplify precise exception support, but is unlikely to perform well for programs without a great deal of compiler exposed ILP. Interestingly, the MiniMIPS processor implements a very similar technique for flushing the pipeline.

The Amulet3 [3] also issues instructions in order but allows memory operations to complete out of order with respect to non-memory operations. In the DSEP, all instructions are allowed to begin and end execution out of order, subject to the size of the instruction window and the number of functional units. The strongest similarity between the DSEP and the Amulet is the sequence number approach to register renaming.

### 6.3 Conclusion

In this paper, we staged the framework for an architectural evaluation of asynchronous processors by stating a specific model of self-timed operation. From this point of reference, we designed a decoupled, elastic pipeline and analyzed its unique capabilities as compared to traditional synchronous processors. Based on the organization of a common synchronous out-of-

order processor, our design demonstrates that it is possible to achieve many of the benefits of an asynchronous design without having to build a fully asynchronous processor. Specifically, this design exposes a new category of speculation-based optimizations that are only possible in a decoupled pipeline. Finally, we provided an initial performance evaluation of our design including an empirical determination of the best micropipeline configuration. Our results are encouraging and suggest that asynchronous processors may eventually develop a performance advantage over synchronous processors, in addition to other advantages currently being perused.

### Acknowledgments

The authors would like to thank Ajay Ladsaria and Michael Urman for their participation on an early version of this work as part of a class project.

### References

[1] Efthymiou, J.D. Garside, and S. Temple. A Comparative Power Analysis of an Asynchronous Processor. Intl. Workshop on Power And Timing Modeling, Optimization and Simulation, 2001.

[2] V. Tiwari, et al. Reducing Power in High-Performance Microprocessors. Proc of the Design Automation Conference, pp. 732-737, 1998.

[3] S.B. Furber, D.A. Edwards, J.D. Garside. AMULET3: A 100 MIPS Asynchronous Embedded Processor. ICCD, September 2000.

[4] A.J. Martin, A. Lines, et al. The Design of an Asynchronous MIPS R3000 Microprocessor. Advanced Research in VLSI, pp164-181, September 1997.

[5] T. Werner, V. Akella. Asynchronous Processor Survey. Computer, vol. 30, no. 11, November 1997.

[6] W. F. Richardson, E. Brunvand. Fred: An Architecture for a Self-Timed Decoupled Computer. IEE Proceedings on Computers and Digital Techniques, vol. 143, no. 5, September 1996

[7] A. Takamura, M. Kuwako, et al. TITAC-2: An Asynchronous 32-bit Microprocessor based on Scalable -Delay-Insensitive Model. ICCD 1997, 288-294.

[8] P.B. Endecott, S.B. Furber. Modeling and Simulation of Asynchronous Systems Using the LARD Hardware Description Language. Proc. of the 12<sup>th</sup> European Simulation Multiconference, pp 39-43, June 1998.

[9] S.B. Furber. Validating the AMULET Microprocessors. The Computer Journal, vol.45, no.1, pp.19-26, 2001.

[10] I.E. Sutherland. Micropipelines. Communications of the ACM, vol. 32, no. 6, June 1989.

[11] S. Hauck. Asynchronous Design Methodologies: An Overview. Proc. of the IEEE, vol. 83, no. 1, pp. 69-93, January 1995.

[12] F.-C. Cheng, S. Unger, M. Theobald. Self-Timed Carry-Lookahead Adders. IEEE Trans. on Computers, vol 49, no.7, July 2000.

[13] The IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture, available at from <http://developer.intel.com>

[14] Mentor Graphics Corporation. Renoir Version 99.1 (Build 26).

[15] T. Chelcea and S. M. Nowick, A Low-Latency FIFO for Mixed-Clock Systems. Proceedings of the IEEE Workshop on VLSI, pp. 119-126, 2000.

### Appendix A

Operation	Variable	Fixed
L1 Cache	100	120
Main Memory	1000	1000
BTB	100	120
Decode	50/80/120	120
RAT/RRF	80/120/150	120
Reorder Buffer	120	120
MEM Address	100	120
INT Arithmetic	130	120
INT Logical	20	120
INT Shift	150	120
INT Multiply	360	360
INT Divide	600	600
BR Compare	40/80	120
Queue Register	5	5