Precision Timed Machines

by

Isaac Liu

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Edward A. Lee, Chair
Professor John Wawrzynek
Professor Alice Agogino

Spring 2012

The dissertation of Isaac Liu, titled Precision Timed Machines is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2012

Precision Timed Machines

Abstract

Precision Timed Machines

by

Isaac Liu
Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

This is my abstract

To my wife Emily Cheung, my parents Char-Shine Liu and Shu-Jen Liu, and everyone else whom I've had the privilege of running into for the first twenty-seven years of my life.

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank my wife
I want to thank my parents
I want to thank my advisor, Edward A. Lee
I want to thank the committee members
I want to thank all that worked on the PRET project with me:
Ben Lickly
Hiren Patel
Jan Rieneke
Sungjun Kim
David Broman
I would also like to thank the ptolemy group especially Christopher and Mary, Jia for providing me the template
I would like to thank everyone else that made this possible

Chapter 1

Introduction

Outline
(Todo: make sure to add in timing anomalies)

1.1 Background

- Discuss the problem
- show the difficulty in execution time analysis of a simple c code

Fig. 1.1 shows an image

1.2 Intro Section Header 2

Here is another header

The remaining chapters are organized as follows. Chapter 4 surveys the related research that has been done on architectures to make them more analyzable. Chapter 2 explains the architecture of PRET including the thread-interleavedpipeline and memory hierarchy, Chapter **??**, Chapter 5, Chapter 6,

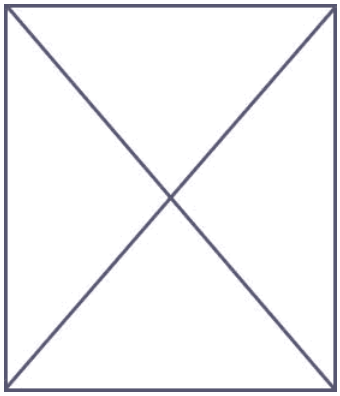

Figure 1.1: Image Placeholder

Chapter 2

Precision Timed Machine

In this chapter we present the guidelines of designing a PREcision Timed (PRET) Machine. It is important to understand why and how current architectures fall short of timing predictability and repeatability. Thus, we first discuss common architectural designs and their effects on execution time, and point out some key issues and trade-offs when designing architectures for predictable and repeatable timing.

2.1 Pipelines

The introduction of pipelining vastly improved the average-case performance of programs. It allows faster clock speeds, and improves instruction throughput compared to single cycle architectures. Pipelining begin executing subsequent instructions while prior instructions are still in execution. Ideally each processor cycle one instruction completes and leaves the pipeline as another enters and begins execution. In reality, different pipeline hazards occur which reduce the throughput and create stalls in the pipeline. Different techniques were introduced to handle the effects of pipeline hazards, and greatly effect to the timing predictability and repeatability of an architecture. To illustrate this point, we discuss some basic hardware additions proposed to reduce performance penalty from hazards, and show how they effect the execution time and predictability.

2.1.1 Pipeline Hazards

Data hazards occur when instructions need the results of previous instructions that have not yet committed. The code segment shown in figure 2.1 contains instructions that each depend on the result of its previous instruction. Figure 2.2 shows two ways data hazards can be handled in a single-threaded pipeline. In the figure, time progresses horizontally towards the right, each time step, or column, represents a processor cycle. Each row represents an instruction that is fetched and executed within the pipeline. Each block represents the instruction entering the different stages of

```
add  r0, r1, r2        # r0 = r1 + r2
sub  r1, r0, r1        # r1 = r0 - r1
ldr  r2, [r1]          # r2 = mem[r1]
sub  r0, r2, r1        # r0 = r2 - r1
cmp  r0, r3            # compare r0 and r3
```

Figure 2.1: Sample code with data dependencies

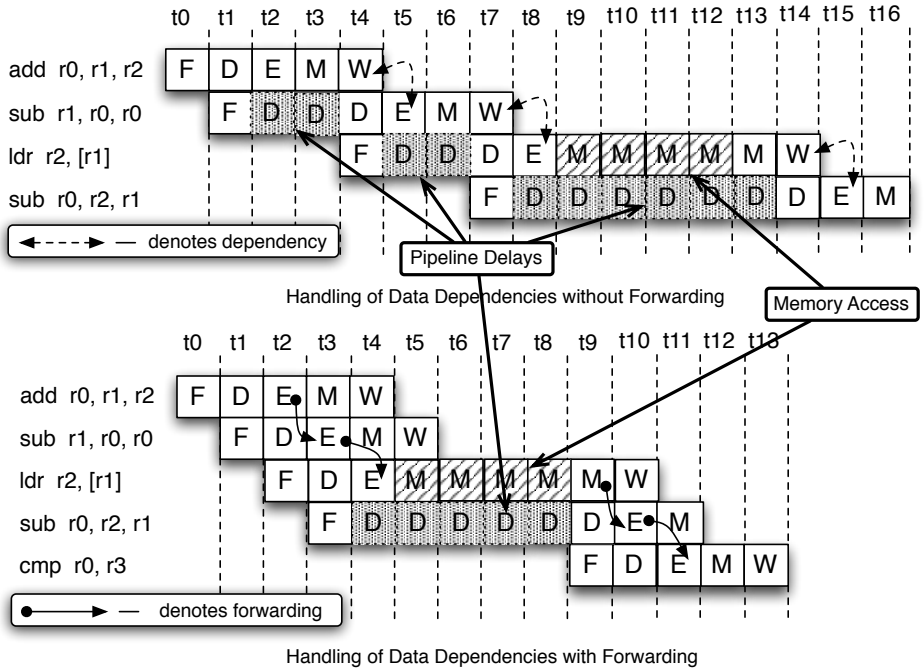

Figure 2.2: Handling of data dependencies in single threaded pipelines

the pipeline – fetch (F), decode (D), execute (E), memory (M) and writeback (W). The pipelines here are assumed to have a similar design to the five stage pipeline mentioned in Hennessy and Pattern (Todo: cite hennessy and patterson).

A simple but effective way of handling data hazards is by simply stalling the pipeline until the previous instruction completes. This is shown in the top of figure 2.2. Pipeline delays (or bubbles) are inserted for instructions to wait until the previous instruction is complete. The dependencies between instructions are shown in the figure to make clear why the pipeline bubbles are necessary. The performance penalty incurred in this case is the pipeline delays inserted to wait for the previous instruction to complete. Data forwarding was introduced to remove the need for inserting bubbles into the pipeline. Data forwarding relies on the fact that the results of the previous instruction is typically available before the it commits. A data forwarding circuitry consists of backwards paths for data from later pipeline stages to the inputs of earlier pipeline stages, and multiplexers to select amongst all data signals. Because it provides a way to directly access computation results from the previous instruction before the previous instruction finishes, it removes the need to wait for the previous instruction to commit. The pipeline controller dynamically detects whether a data-dependency exists, and changes the selection bits to the multiplexers accordingly so the correct operands are selected. The bottom of figure 2.2 shows the execution with forwarding in the pipeline. No pipeline bubbles are needed for the first *sub* instruction and *ld* instruction because the data they depend on are forwarded with the forwarding paths. However, the second *sub* instruction after the *ld* instruction still stalls. As mentioned earlier, forwarding relies on the results from the previous instruction being available before the previous instruction commits. In the case of longer latency operations, such as memory accesses, the data cannot be forwarded until it becomes available, so stalls are still required. The memory access latency in the figure is arbitrarily chosen to be 5 cycles

so the figure is not too long and instructions after the *ld* instruction can be shown. We purposely leave out the details regarding memory accesses at this point, and will discuss it extensively in section 2.2. We merely use the *ld* instruction to illustrate the limitations of data forwarding. They can address the data-dependencies caused by pipelining – the read-after-write of register computations. However, they cannot address the data-dependencies caused by other long latency operations such as memory operations, so pipeline stalls are still needed. More involved techniques such as the introduction of out-of-order execution or superscalar pipelines are used to mitigate the effects of long latency operations. We will discuss more of these in chapter 4 when we mention the related works.

To understand the timing effects of handling data hazards, we discuss how to determine execution time for instructions using both methods of handling data hazards. With the simple method of inserting stalls, we need to know when the stalls will be inserted and how long the instruction will need to stall for. This information can be determined by simply checking the previous instruction since stalls are inserted only if this instruction depends on the results of the previous instruction. Within pipelines, the execution of most instructions are deterministic, so for the most part we can determine how long the stall will be by checking the previous instruction. Memory access instructions are an exception to instructions that have deterministic execution time, but as mentioned before, we will discuss these extensively in section 2.2. For pipelines with data forwarding, we need to know in what situations the data forwarding circuitry cannot correctly forward the data to the next instruction. Although the pipeline dynamically forwards the data during run-time, the logic in the pipeline controller that enables and selects the correct forwarding bits only needs to keep track of a small set of previous instructions to detect data-dependencies. The set of instructions it needs to check usually depends on the depth of the pipeline. Thus, static execution time analysis can detect forwarding by simply checking a short window of previous instructions to account for stalls accordingly. (Todo: find papers to back this up) We simplified greatly the execution time analysis discussed above to ignore effects from other pipeline mechanisms. We wanted to simply focused on the effects of handling data-hazards through stalling or data-forwarding. We can see that both methods of handling data-hazards cause instruction execution time to depend previous instruction execution history. But the execution history that instruction execution time is dependent upon is small and temporary enough to be accounted for.

Branches cause control-flow hazards in the pipeline; the instruction after the branch, which should be fetched the next cycle, is unknown until after the branch instruction is completed. Conditional branches further complicates matters, as whether or not the branch is taken depends on an additional condition that could possible be unknown when the conditional branch is in execution. The code segment in figure 2.3 shows assembly instructions from the ARM instruction set architecture (ISA) that implement the Greatest Common Divisor (GCD) algorithm using conditional branch instructions *beq* (branch equal) and *blt* (branch less than). Conditional branch

```
gcd:
        cmp  r0, r1        # compare r0 and r1
        beq  end           # branch if r0 == r1
        blt  less          # branch if r0 < r1
        sub  r0, r0, r1    # r0 = r0 - r1
        b    gcd           # branch to label gcd
less:
        sub  r1, r1, r0    # r1 = r1 - r0
        b    gcd           # branch to label gcd
end:
        add  r1, r1, r0    # r1 = r1 + r0
        mov  r3, r1        # r3 = r1
```

Figure 2.3: Sample code for GCD with conditional branches

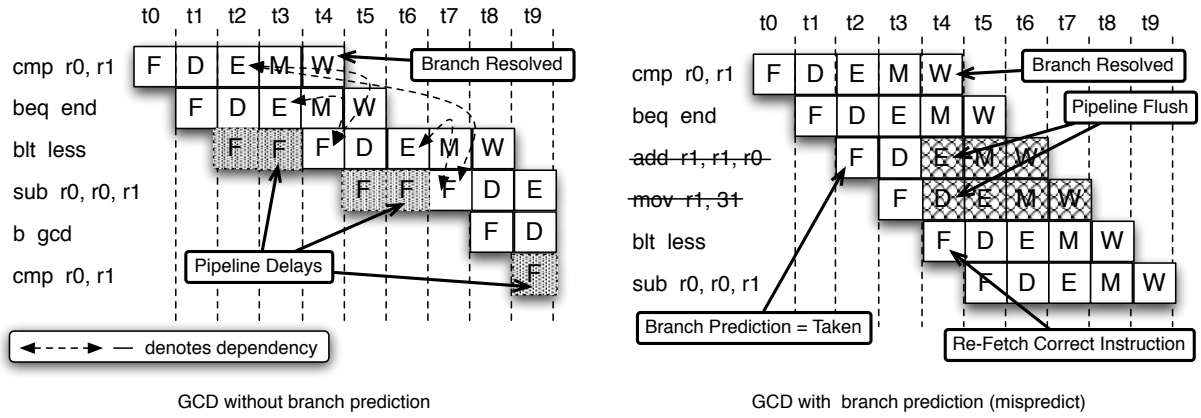

Figure 2.4: Handling of conditional branches in single threaded pipelines

instructions in ARM branch based on conditional bits that are stored in a processor state register and set with special compare instructions(Todo: cite arm manual). The *cmp* instruction is one such compare instruction that subtracts two registers and sets the conditional bits according to the results. The GCD implementation shown in the code uses this mechanism to determine whether to continue or end the algorithm. Figure 2.4 show two ways branches can be handled in a single-threaded pipeline.

Similar to handling data-hazards, a simple but effective way of handling control-flow hazards is by simply stalling the pipeline until the branch instruction completes. This is shown on the left of figure 2.4. Two pipeline delays (or bubbles) are inserted after each branch instruction to wait until address calculation is completed. The dependencies between instructions are also drawn out to make clear why the pipeline bubbles are necessary. In order for the *blt* instruction to be fetched, its address must be calculated during the execution stage of the *beq* instruction. At the same time, because *beq* is a conditional branch, whether or not the branch is taken depends on the *cmp* instruction. The pipeline here is assumed to have forwarding circuitry, so the addresses calculated by the branch instructions and the results of the *cmp* instruction can be used before the instructions are committed. The performance penalty incurred is the pipeline delays inserted to wait for the branch address calculation to complete. Conditional branches will also incur extra delays for deeper pipelines if the branch condition cannot be resolved in time. Some architectures enforce the compiler to insert one or more non-dependent instructions after a branch that is always executed before the change in control-flow of the program. These are called branch delay slots and can mitigate the branch penalty, but become less effective as pipelines grow deeper because the longer delay slots are required.

In attempt to remove the need of inserting pipeline bubbles, branch predictors were invented to predict the results of a branch before it is resolved(Todo: citation). Many clever branch predictors have been proposed, and they can accurately predict branches up to 93.5%(Todo: citation). Branch predictors predict the condition and target addresses of branches, so pipelines can speculatively continue execution based upon the prediction. If the prediction was correct, no penalty occurs for the branch, and execution simply continues. However, when a mispredict occurs, then

the speculatively executed instructions need to be flushed and the correct instructions need to be refetched into the pipeline for execution. The right of figure 2.4 shows the execution of GCD in the case of a branch misprediction. After the *beq* instruction, the branch is predicted to be taken, and the *add* and *mov* instructions from the label *end* is directly fetched into execution. When the *cmp* instruction is completed, a misprediction is detected, so the *add* and *mov* instruction are flushed out of the pipeline while the correct instruction *blt* is immediately re-fetched and execution continues. The misprediction penalty is typically the number of stages between fetch and execute, as those cycles are wasted executing instructions from an incorrect execution path. This penalty only occurs on a mispredict, thus branch prediction typically yields better average performance and is preferred for modern architectures. Nonetheless, it is important to understand the effects of branch prediction on execution time.

Typical branch predictors predict branches based upon the history of previous branches encountered. As each branch instruction is resolved, the internal state of the predictor, which stores the branch histories, is updated and used to predict the next branch. This implicitly creates a dependency between branch instructions and their execution history, as the prediction is affected by its history. In other words, the execution time of a branch instruction will depend on the branch results of previous branch instructions. During static execution timing analysis, the state of the branch predictor is unknown because is it often infeasible to keep track of execution history so far back. There has been work on explicitly modeling branch predictors for execution time analysis(Todo: citation), but the results are (Todo: the results of branch predictor modeling for execution time analysis). The analysis needs to conservatively account for the potential branch mispredict penalty for each branch, which leads to overestimated execution times. To make matters worse, as architectures grow in complexity, more internal states exist in architectures that could be affected by the speculative execution. For example, cache lines could be evicted when speculatively executing instructions from a mispredicted path, changing the state of the cache. This makes a tight static execution time analysis extremely difficult, if not impossible; explicitly modeling all hardware states and their effects together often lead to an infeasible explosion in state space. On the other hand, although the simple method of inserting pipeline bubbles for branches could lead to more branch penalties, the static timing analysis is precise and straight forward, as no prediction and speculative execution occur. The timing analysis simply adds the branch penalty to the instruction after a branch. Additional penalties from a conditional branch can be accounted for by simply checking for instructions that modify the conditional flag above the conditional branch. We explicitly showed this simple method of handling branches to point out an important trade-off between speculative execution for better average performance and consistent stalling for better predictability. Average-case performance can be improved by speculation at the cost of predictability and potentially prolonging the worst-case performance. The challenge remains to maintain predictability while improving worst-case performance, and how pipeline hazards are handled play an integral part of tackling this challenge.

2.1.2 Pipeline Multithreading

Multithreaded architectures were introduced to improve instruction throughput over instruction latency. The architecture optimizes thread-level parallelism over instruction-level parallelism to improve performance. Multiple hardware threads are introduced into the pipeline to fully utilize thread-level parallelism. When one hardware thread is stalled, another hardware thread can be fetched into the pipeline for execution to avoid stalling the whole pipeline. To lower the context

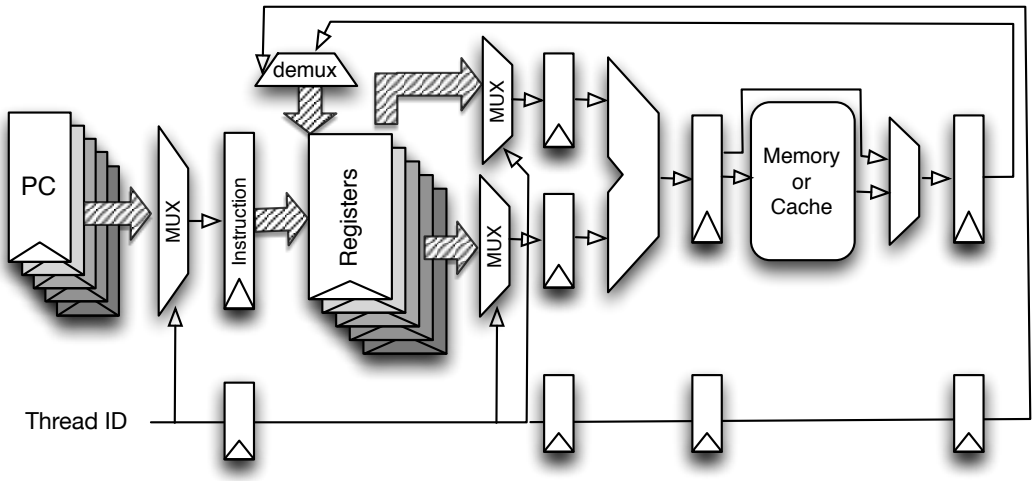

Figure 2.5: Simple Multithreaded Pipeline

switching overhead, the pipeline contains physically separate copies of hardware thread states, such as registers files and program counters etc, for each hardware thread. Figure 2.5 shows a architectural level view of a simple multithreaded pipeline. It contains 5 hardware threads, so it has 5 copies of the Program Counter (PC) and Register files. Once a hardware thread is executing in the pipeline, its corresponding thread state can be selected by signaling the correct selection bits to the multiplexers. The rest of the pipeline remains similar to a traditional 5 stage pipeline as introduced in Hennessy and Pattern(Todo: citation). The extra copies of the thread state and the multiplexers used to select them thus contribute to most of the hardware additions needed to implement hardware multithreading.

Ungerer et al. [24] surveyed different multithreaded architectures and categorized them based upon the (Todo: thread selection?) policy and the execution width of the pipeline. The thread selection policy is the context switching scheme used to determine which threads are executing, and how often a context switch occurs. Coarse-grain policies manage hardware threads similar to the way operation systems manage software threads. A hardware thread gain access to the pipeline and continues to execute until a context switch is triggered. Context switches occur less frequently via this policy, so less hardware threads are required to fully utilize the processor. Different coarse-grain policies trigger context switches with different events. Some trigger on dynamic events, such as cache miss or interrupts, and some trigger on static events, such as specialized instructions. Fine-grain policies switch context much more frequently – usually every processor cycle. Both coarse-grain and fine-grain policies can also have different hardware thread scheduling algorithms that are implemented in a hardware thread scheduling controller to determine which hardware thread is switched into execution. The width of the pipeline refers to the number of instructions that can be fetched into execution in one cycle. For example, superscalar architectures have redundant functional units, such as multipliers and ALUs, and can dispatch multiple instructions into execution in a single cycle. Multithreaded architectures with pipeline widths of more than one, such as Sumultanous Multithreaded (SMT) architectures, can fetch and execute instructions from several hardware threads in the same cycle.

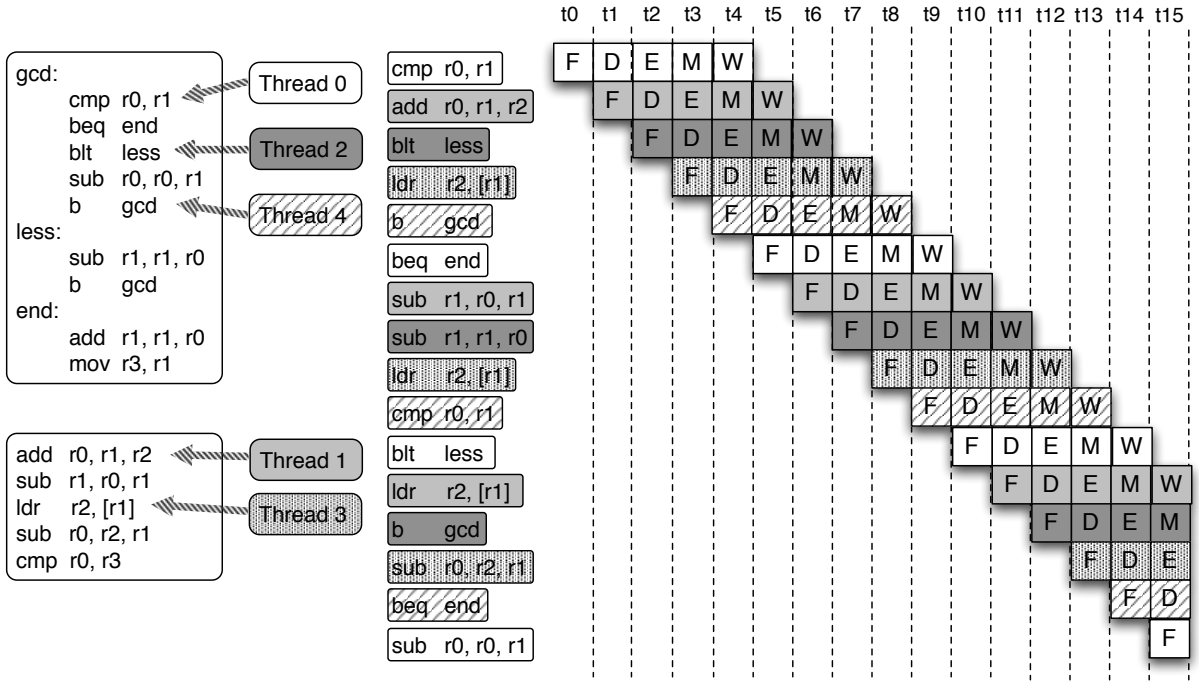

Figure 2.6: Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5 pipeline stages

Multithreaded architectures typically bring additional challenges to execution time analysis of software running on them. Any timing analysis for code running on a particular hardware thread needs to take into account not only the code itself, but also the thread selection policy of the architecture and sometimes even the execution context of code running on other hardware threads. For example, if dynamic coarse-grain multithreading is used, then a context switch could occur at any point when a hardware thread is executing in the pipeline. This not only has an effect on the control flow of execution, but also the state of any hardware that is shared, such as caches or branch predictors. Thus, it becomes nearly impossible to estimate execution time without knowing the exact execution state of other hardware threads and the state of the thread scheduling controller. However, it is possible to for multithreaded architectures to fully utilize thread-level parallelism while still maintaining timing predictability. Thread-interleaved pipelines use a fine-grain thread switching policy with round robin thread scheduling to achieve high instruction throughput while still allowing precise timing analysis for code running on its hardware threads. Below, its architecture and trade-offs are described and discussed in detail along with examples and explanation of how timing predictability is maintained. Through the remainder of this chapter, we will use the term "thread" to refer to explicit hardware threads that have physically separate register files, program counters, and other thread states. This is not to be confused the common notion of "threads", which is assumed to be software threads that is managed by operating systems with thread states stored in memory.

2.1.3 Thread-Interleaved Pipelines

The thread-interleaved pipeline was introduced to improve the response time of handling multiple I/O devices (Todo: citation). I/O operations often stall from the communication with the I/O devices. Thus, interacting with multiple I/O devices leads to wasted processor cycles that are idle waiting for the I/O device to respond. By employing multiple hardware thread contexts, a hardware thread stalled from the I/O operations does not stall the whole pipeline, as other hardware threads can be fetched and executed. Thread-interleaved pipelines use fine-grain multithreading; every cycle a context switch occurs and a different hardware thread is fetched into execution. The threads are scheduled in a deterministic round robin fashion. This also reduces the context switch overhead down to nearly zero, as no time is needed to determine which thread to fetch next. Barely any hardware is required to implement round robin thread scheduling; a simple $log(n)$ bit up counter (for n threads) would suffice. Figure 2.6 shows an example execution sequence from a 5 stage thread-interleaved pipeline with 5 threads. The thread-interleaved pipelines shown and presented in this thesis are all of single width. The same code segments from figure 2.3 and figure 2.1 are being executed in this pipeline. Threads 0, 2 and 4 execute GCD (figure 2.3) and threads 1 and 3 execute the data dependent code segment (figure 2.1). Each hardware thread executes as an independent context and their progress is shown in figure 2.6 with thick arrows pointing to the execution location of each thread at t0. We can observe from the figure that each time step an instruction from a different hardware thread is fetched into execution and the hardware threads are fetched in a round robin order. At time step 4 we begin to visually see that each time step, each pipeline stage is occupied by a different hardware thread. The fine-grained thread interleaving and the round robin scheduling combine to form this important property of thread-interleaved pipelines, which provides the basis for a timing predictable architecture design.

For thread-interleaved pipelines, if there are enough thread contexts, for example – the same number of threads as there are pipeline stages, then at each time step no dependency exists between the pipeline stages since they are each executing on a different thread. As a result, data and control pipeline hazards, the results of dependencies between stages within the pipelines, no longer exist in the thread-interleaved pipeline. We've already shown from figure 2.4 that when executing the GCD code segment on a single-threaded pipeline, control hazards stem from branch instructions because of the address calculation for the instruction after the branch. However, in a thread-interleaved pipeline, the instruction after the branch from the same thread is not fetched into the pipeline until the branch instruction is committed. Before that time, instructions from other threads are fetched so the pipeline is not stalled, but simply executing other thread contexts. This can be seen in figure 2.6 for thread 0, which is represented with instructions with white backgrounds. The *cmp* instructions, which determines whether next conditional branch *beq* is taken or not, completes before the *beq* is fetched at time step 5. The *blt* instruction from thread 0, fetched at time step 10, also causes no hazard because the *beq* is completed before *blt* is fetched. The code in figure 2.1 is executed on thread 1 of the thread interleave pipeline in figure 2.6. The pipeline stalls inserted from top of figure 2.2 are no longer needed even without a forwarding circuitry because the data-dependent instructions are fetched after the completion of its previous instruction. In fact, no instruction in the pipeline is dependent on another because each pipeline stage is executing on a separate hardware thread context. Therefore, the pipeline does not need to include any extra logic or hardware for handling data and control hazards in the pipeline. This gives thread-interleaved pipelines the advantage of a simpler pipeline design that requires less hardware logic, which in

turns allows the pipeline clock speed to increase. Thread-interleaved pipelines can be clocked at higher speeds since each pipeline stage contains significantly less logic needed to handle hazards. The registers and processor states use much more compact memory cells compared to the logic and muxes used to select and handle hazards, so the size footprint of thread-interleaved pipelines are also typically smaller.

For operations that have long latencies, such as memory operations or floating point operations, thread-interleaved pipelines hides the latency with its execution of other threads. Thread 3 in figure 2.6 shows the execution of a *ld* instruction that takes the same 5 cycles as shown in figure 2.2. We again assume that this *ld* instruction accesses data from the main memory. While the *ld* instruction is waiting for memory access to complete, the thread-interleaved pipeline executes instructions from other threads. The next instruction from thread 3 that is fetched into the pipeline is again the same *ld* instruction. As memory completes its execution during the execution of instructions from other threads, we replay the same instruction to pick up the results from memory and write it into registers to complete the execution of the *ld* instruction. It is possible to directly write the results back into the register file when the memory operation completes, without cycling the same instruction to pick up the results. This would require hardware additions to support and manage multiple write-back paths in the pipeline, and a multi write ported register file, so contention can be avoided with the existing executing threads. In our design we simply replay the instruction for write-backs to simplify design and piggy back on the existing write-back datapath. Multithreaded pipelines typically mark threads inactive when they are waiting for long latency operations. Inactive threads are not fetched into the pipeline, since they cannot make progress even if they are scheduled. This allows the processor to maximize throughput by allowing other threads to utilize the idle processor cycles. However, doing so has non-trivial effects on thread-interleaved pipelines and the timing of other threads.

First, if the number of "active" threads falls below the number of pipeline stages, then pipeline hazards are reintroduced; it is now possible for the pipeline to be executing two instructions from the same thread that depend on each other simultaneously. This can be circumvented by inserting pipeline bubbles when there aren't enough active threads. For example, as shown in figure 2.7, for our 5 stage thread-interleaved pipeline that has 5 threads, if two threads are waiting for main memory access and are marked inactive, then we insert 2 NOPs every round of the round-robin schedule to ensure that no two instructions from the same thread exists in the pipeline. Note that if the 5 stage thread-interleaved pipeline contained 7 threads, then even if 2 threads are waiting for memory, no NOP insertion would be needed since instructions in each pipeline stage in one cycle would still be from a different thread. NOP insertions only need to occur when the number of active threads drops below the number of pipeline stages.

The more problematic issue with setting threads inactive whenever long latency operations occur is the effect on the execution frequencies of other threads in the pipeline. When threads are scheduled and unscheduled dynamically, the other threads in the pipeline would dynamically execute more or less frequently depending on how many threads are active. This complicates timing analysis since the thread frequency of one thread now depends on the program state of all other threads. In order for multithreaded architectures to achieve predictable performance, *temporal isolation* must exist in the hardware between the threads. Temporal isolation is the isolation of timing behaviors of a thread from other thread contexts in the architecture. With temporal isolation, the timing analysis is greatly simplified, as software running on individual threads can be analyzed

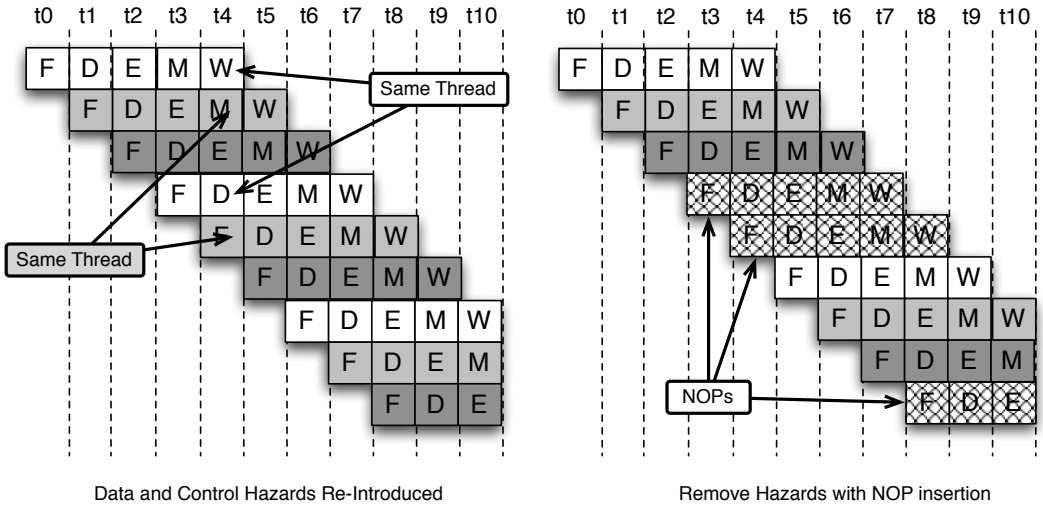

Figure 2.7: Execution of 5 threads thread-interleaved pipeline when 2 threads are inactive

separately without worry about the effects of integration. If temporal isolation is broken, any timing analysis needs to model and explore all possible combinations of program state of all threads, which is typically infeasible. The round-robin thread scheduling of thread-interleaved pipelines is a way of achieving temporal isolation for a multithreaded architecture. Unlike coarse-grain dynamically switched multithreaded architectures, thread-interleaved pipelines can maintain the same round-robin thread schedule despite the execution context of each thread within the pipeline. This is a step towards achieving temporal isolation amongst the threads, as the execution frequency of threads does not change dynamically. However, dynamically scheduling and unscheduling threads based upon long-latency operations breaks temporal isolation amongst threads. Thus, our thread-interleaved pipeline does not mark threads inactive on long latency operations, but simply replays the instruction whenever the thread is fetched. Although this slightly reduces the utilization of the thread-interleaved pipeline, but threads are decoupled and timing analysis can be done individually for each thread without interference from other threads. At the same time, we still preserve most of the benefits of latency hiding, as other threads are still executing during the long latency operation.

(Todo: talk about xmos handling exceptions and our handling of exceptions here)

Shared hardware units within multithreaded architectures could also easily break temporal isolation amongst the threads. Two main issues arise when a hardware unit is shared between the threads. The first issue arises when shared hardware units share the same state between all threads. If the state of hardware unit is shared and can be modified by any thread, then it is nearly impossible to get a consistent view of the hardware state from a single thread during timing analysis. Shared branch predictors and caches are prime examples of how a shared hardware state can cause timing inference between threads. If a multithreaded architecture shares a branch predictor for all threads, then the branch table entries can be overwritten by branches from any thread. This means that each thread's branches can cause a branch mispredict for any other thread. Caches are especially troublesome when shared between threads in a multithreaded architecture. Not only does it make the execution time analysis substantially more difficult, it also decreases overall performance for each thread due to cache thrashing, an event where threads continuously evict each other threads cache lines in the cache(Todo: citation). To achieve temporal isolation between the threads, the

hardware units in the architecture must not share state between the threads. Each thread must have its own consistent view of the hardware unit states, without the interference from other threads. For example, each thread in our thread-interleaved pipeline contains its own private copy of the registers and thread states. We already showed why thread-interleaved pipelines do not need branch predictors because they remove control-hazards, and we will discuss a timing predictable memory hierarchy that uses scratchpads instead of caches in section 2.2. The sharing of hardware state between threads also increases security risks in multithreaded architectures. Side-channel attacks on encryption algorithms(Todo: cite) take advantage of the shared hardware states to disrupt and probe the execution time of threads running the encryption algorithm to crack the encryption key. We will discuss this in detail in section 5.1 and show how a predictable architecture can prevent timing side-channel attacks for encryption algorithms.

The second issue that arises is that shared hardware units create structural hazards – hazards that occur when a hardware unit needs to be used by two or more instructions at the same time. Structural hazards typically occur in thread-interleaved pipelines when the shared units take longer than one cycle to access. The ALU, for example, is shared between the threads. But because it takes only one cycle to access, there is no contention even when instructions continuously access the ALU in subsequent cycles. On the other hand, a floating point hardware unit typically takes several cycles to complete its computation. If two or more threads issue a floating point instruction in subsequent cycles, then contention arises, and the the second request must be queued up until the first request completes its floating point computation. This creates timing interference between the threads, because the execution time of a floating point instruction from a particular thread now depends on if other threads are also issuing floating point instructions simultaneously. If the hardware unit can be pipelined to accept inputs every processor cycle, then we can remove the the contention caused by the hardware unit, since accesses no longer need to be queued up. The shared memory system in a thread-interleaved pipeline also creates structural-hazards in the pipeline. In section 2.2 we will discuss and present our memory hierarchy along with a redesigned DRAM memory controller that supports pipelined memory accesses. If pipelining cannot be achieved, then any timing analysis of that instruction must include a conservative estimation that accounts for thread access interference and contention management. Several trade-offs need to be considered when deciding how to manage the thread contention to the hardware unit.

A time division multiplex access (TDMA) schedule to the hardware unit can be enforced to decouple the access time of threads remove timing interference. A TDMA access scheme certainly creates an non-substantial overhead compared to conventional queuing schemes, especially if access to the hardware unit is rare and sparse. However, in a TDMA scheme, each threads wait time to access the shared resource depends on the time offset in regards to the TDMA schedule, and is decoupled from the accesses of other threads. Because of that, it is possible obtain a tighter worst case execution time analysis per thread. For a TDMA scheme, the worst case access time occurs when an access just missed its time slot and must wait a full cycle before accessing the hardware unit. For a conventional queuing scheme where each requester can only have one outstanding request, the worst case happens when every other requester has a request in queue, and the first request is just beginning to be serviced. At first, it may seem that the worst case execution time of a TDMA scheme may seem similar to the basic queuing scheme. For timing analysis at an unknown state of the program, no assumption can be made on the TDMA schedule, thus the worst case time must be used for conservative estimations. However, because the TDMA access schedule is static, and

access time is decoupled from other threads, there is potential to obtain tighter timing analysis for accesses by inferring access slot hits and misses for future accesses. For example, based upon the execution time offsets of a sequence of accesses to the shared resource, we may be able to conclude that at least one access will hit its TDMA access slot and get access right away. We can also possibly derive more accurate wait times for the accesses that do not hit its access slots based upon the elapsed time between accesses. An in depth study of WCET analysis of TDMA access schedules is beyond the scope of the thesis. But these are possibilities now because there is no timing interference between the threads. A queue based mechanism would not be able to achieve better execution time analysis without taking into account the execution context of all other threads in the pipeline.

It is important to understand that we are not proclaiming that all dynamic behavior in systems are harmful. But only by achieving predictability in the hardware architecture can we begin to reason about more dynamic behavior in software. For example, we discussed that dynamically scheduling threads in hardware causes timing interference. However, it is not the switching of threads that is unpredictable, but how the thread switching is triggered that makes it predictable. For example, the Giotto(Todo: cite) programming model specifies a periodic software execution model that can contain multiple program states. If such a programming model was implemented on a thread-interleaved pipeline, different program states might map different tasks to threads or have different number of threads executing within the pipeline. But by explicitly controller the thread switches in software, the execution time variances introduced is transparent at the software level, allowing potential for timing analysis.

In this section we introduced a predictable thread-interleaved pipeline design that provides temporal isolation for all threads in the architecture. The thread-interleaved pipeline favors throughput over single thread latency, as multiple threads are executed on the pipeline in a round robin fashion. We will present in detail our implementation of this thread-interleaved pipeline in chapter 3, and show how the design decisions discussed in this chapter are applied.

2.2 Memory System

The memory hierarchy

2.2.1 Scratchpad

2.2.2 DRAM memory controller

2.3 Programming Models

Intro text here

2.3.1 PRET Programming model Section Header

2.3.2 Pret Programming model Section Header 2

Here is another header

Chapter 3

Implementation of PTARM

The Precision Timed ARM (PTARM) architecture is a realization of the PRET principles on an ARM ISA architecture(Todo: Citation). In this chapter we will describe in detail the implementation details of the timing-predictable ARM processor and discuss the worst-case execution time analysis of code running on it. We show that with the architectural design principles of PRET, the PTARM architecture is easy analyzable with repeatable timing.

The architecture of PTARM closely follows the principles discussed in chapter 2. This includes a thread-interleaved pipeline with scratchpads along with the timing predictable memory controller. The ARM ISA was chosen not only for its popularity in the embedded community, but also because it is a Reduced Instruction Set Computer (RISC), which has simpler instructions that allow more precise timing analysis. Complex Instruction Set Computers (CSIC) on the other hand adds un-needed complexity to the hardware and timing analysis. RISC architectures typically features a large uniform register file, a load/store architecture, and fixed-length instructions. In addition to these, ARM also contains several unique features. ARM's ISA requires a built in hardware shifter along with the arithmetic logic unit (ALU), as all of its data-processing instructions can shift its operands before passed onto the ALU. ARM's load/store instructions also contain auto-increment capabilities that can increment or decrement the value stored in the base address register. This is useful to compact code that is reading through an array in a loop, as one instruction can load the contents and prepare for the next load in one instruction. In addition, almost all of the ARM instructions are conditionally executed. The conditional execution improves architecture throughput with potential added benefits of code compaction(Todo: Citation). ARM programmer's model specifies 16 general purpose registers (R0 to R15) to be accessed with its instructions, with register 15 being the program counter (PC). Writing to R15 triggers a branch, and reading from R15 reads the current PC plus 8.

ARM has a rich history of versions for their ISA, and PTARM implements the ARMv4 ISA, currently without support for the thumb mode. PTARM uses scratchpads instead of caches, and a DDR2 DRAM for main memory managed by the timing predictable memory controller. PTARM also implements the timing instructions introduced in chapter 2.3.

3.1 Thread-Interleaved Pipeline

PTARM implements a thread-interleaved pipeline for the ARM instruction set. PTARM was initially written to target Xilinx Virtex-5 Family FPGAs, thus several design decisions were made to optimized the PTARM architecture for Xilinx V5 FPGAs. PTARM has a 32 bit datapath in a five stage pipeline with four threads interleaving through the pipeline. Chapter 2 discussed the timing and hardware benefits of a typical thread-interleaved pipeline which removes pipeline hazards with multiple threads. Section 2.1.3 mentioned that conventional thread-interleaved pipelines typically have at least as many threads as pipeline stages to keep the pipeline design simple and maximize the clock speed. However, having more threads in the pipeline increases single thread latency, since all threads are essentially time-sharing the pipeline resource. Lee and Messerschmitt [17] showed that the minimum number of threads required to remove hazards is actually less than the number of pipeline stages in the pipeline. In our design, we implement a five stage thread-interleaved pipeline with four threads by carefully designing the PC writeback mechanism one pipeline stage earlier.

Figure 3.1 shows a block diagram view of the pipeline. Some multiplexers within the pipeline have been omitted in the figure for a simplified view of the hardware components that make up the pipeline. There contains four copies of the Program Counter(PC), Thread States, and Register File. Most of the pipeline design follows a typical Hennessy and Patterson(Todo: citation) five stage pipeline, with the five stages in the pipeline being – Fetch, Decode, Execute, Memory, Writeback. We will briefly describe the functionality of each stage, and leave more details when we discuss how instructions are implemented in section 3.3.

The *fetch stage* of the pipeline selects the correct PC according to which thread is executing, and passes the address to instruction memory. The PC forward path forwards a loaded address from main memory for instructions that load to R15, which causes a branch. We will discuss the need for the forwarding path below when we describe the *writeback stage*. A simple $log(n)$ bit upcounter is used to keep track of which thread current to fetch.

The *decode stage* contains the *pipeline controller* which does the full decoding of instructions and sets the correct pipeline signals to be propagated down the pipeline. Most of ARM instructions are conditionally executed, so the pipeline controller first checks the condition bis to determine whether the instruction is to be executed or not. Typically the *pipeline controller* needs to

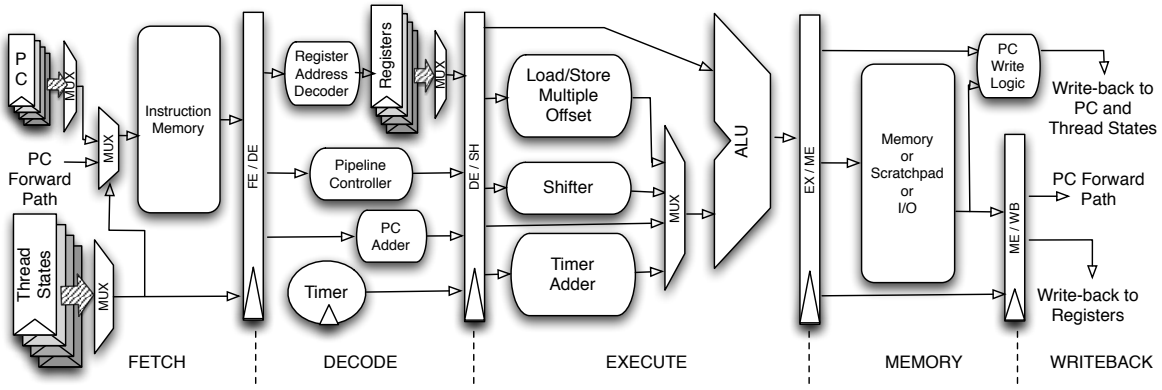

Figure 3.1: Block Level View of the PTARM 5 stage pipeline

know the current instructions in the pipeline to detect the possibility of pipeline hazards and stall the current instruction. However,in a thread-interleaved pipeline, other instructions down the pipeline are from threads, thus the controller logic is greatly simplified. It simply decodes the instruction to determine the correct signals to send to the data-path and multiplexers down the pipeline. It does not need to know any information about instructions already in flight. A small decoding logic, the *register address decoder*, is inserted in parallel with the controller to decode the register addresses from the instruction bits. Typical RISC instruction sets, such as MIPS, set the encoding of instruction bits so the register operands have a fixed location for all instruction types. However, in the ARM instruction set, certain instructions encode the register read address at different bit locations of the instruction. For example, ARM data-processing register shift instructions reads a third operand from the register to determine the shift amount. Store instructions also read a third register to obtain the register value that is stored to memory. However, both instructions have different bit locations in the instruction encoding to determine what register to read from. Thus, a small register address decoding logic is inserted for a quick decoding of the register addresses from the instruction bits. The *PC Adder* is used to increment the PC. The ARM ISA programmer's model states that reading from R15 reads the current PC+8, the PC adder not only increments the PC by 4 to get the potential next PC, but it also increments the current PC by 8 to be used as an operand. Single threaded pipelines need to increment the PC immediately in the fetch stage to prepare for the next instruction fetch. For thread-interleaved pipelines, since the next PC from the current thread is not needed until several cycles later, it doesn't need to be in the fetch stage. But because we need the results of PC+8 as a data operand, it is placed in the decode stage. The *timer* is a hardware counter clocked to the processor clock which is used to implement the timing instructions mentioned in chapter 2.3. The timer contains a 64 bit value that represents nanoseconds, and starts at 0 when the pipeline starts up. The time value is latched in the decode stage as the subsequent stages use it for timer manipulation.

The *execute*, *memory* and *writeback* stages execute the instruction and commits the result. The *execute* stage contains mostly execution units and muxes that select the correct operand and feeds it to the ALU. The ARM ISA assumes a built in shifter to shift the operands before operations, so a 32 bit *shifter* is included to shift the operands before the ALU. The *load/store multiple offset* logic block is used to calculate the offset of load/store multiple instructions. The load/store multiple instruction uses a 16 bit vector to represent each of the 16 general purpose registers. The bits that are set in that bit vector represents a load/store on that register. The an offset is added to the base memory address for the instruction, and that offset depends on how many bits are set. Thus, the load/store multiple offset logic block does a bit count on the bit vector and adjusts the offset to be passed into the ALU for load/store multiple instructions. The *timer adder* logic block is a 32 bit add/subtract unit. Time in the pipeline is a 64 bit value representing nanoseconds. Thus, any timing instruction that interacts with the timer in the pipeline needs to operate on 64 bit values. We could have reuse the existing ALU at the expense of having all timing instructions take an additional pass through the pipeline. But we chose to include an addition add/subtract unit specifically for the implementation of the delay_until instruction so it can check for deadline expiration every cycle, which we will discuss in detail in section 3.3 when we show how delay_until is implemented. A 32 bit *ALU* does most of the logical and arithmetic operations, including data-processing operations and branch address calculations. The results is passed to the *memory stage*, which either uses it as an address to interact with the data memory, or forwards it along to the *writeback stage* to commit back to the registers.

Figure 3.2 shows an execution sequence of the four thread five stage pipeline. The instruction in the fetch stage belongs to the same thread as the instruction in the writeback stage. This does not cause any data hazards because the data from the registers will not be read until the decode stage. But committing the PC at the writeback stage would result in a control hazard because the PC would not be ready for the subsequent fetch. For most instructions, the next PC calculation is completed before the memory stage, so we move the PC commit one stage earlier so the next instruction can be fetched. However, the ARM ISA allows instructions to write to register 15 (PC), which acts as a branch to the value written to R15. This means a load instruction can write to R15 and cause a branch whose target is not known until after the memory read. Thus, a PC forwarding path is added to forward the PC back from memory if a load instruction writes to R15. The forwarding path does not cause any timing analysis difficulties because the statically the forwarding path is only used when a load instruction writes to R15, which can be statically determined. Also, this causes no stall in the pipeline, and does not effect the timing of any following instructions. This allows us to interleave four threads in our five stage pipeline instead of five.

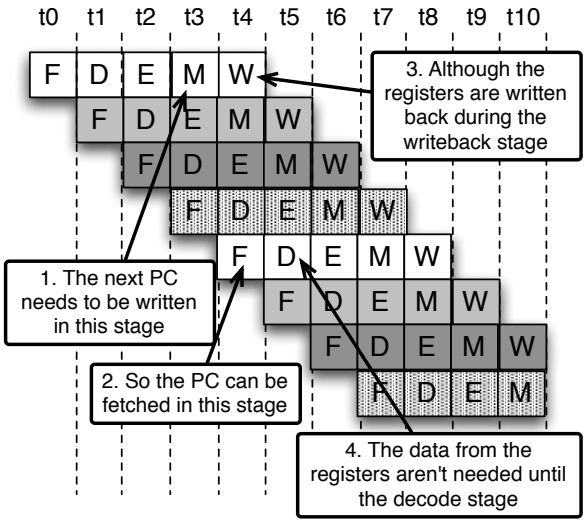

Figure 3.2: Four thread execution in PTARM

3.2 Memory Hierarchy

The instruction memory is currently composed of an instruction scratchpad and a boot ROM. The boot ROM (Todo: mention bootROM size) is shared between all the threads and contains the initialization code for each thread. It also contains the exception vector table that stores entries for handling different exceptions that occur in the pipeline, along with some of the exception handlers. The instruction scratchpad (Todo: size?) is currently logically divided into five regions. Each of the threads contains its own private instruction region, and a shared region where all threads can access. Both the boot ROM and instruction scratchpad are also synthesized to FPGA block RAMs, and both give single cycle access latencies.

The data memory

Also talk about I/O bus connections and the protocol to determine if data is ready

3.3 Instruction Implementations

In this section we go into more details on how each instruction type is implemented and how each hardware block in the pipeline shown in figure 3.1. We will go through different instruction types and discuss the timing implications each instruction in our implementation. We will summarize with a table with all instructions and the cycle count it takes to execute them.

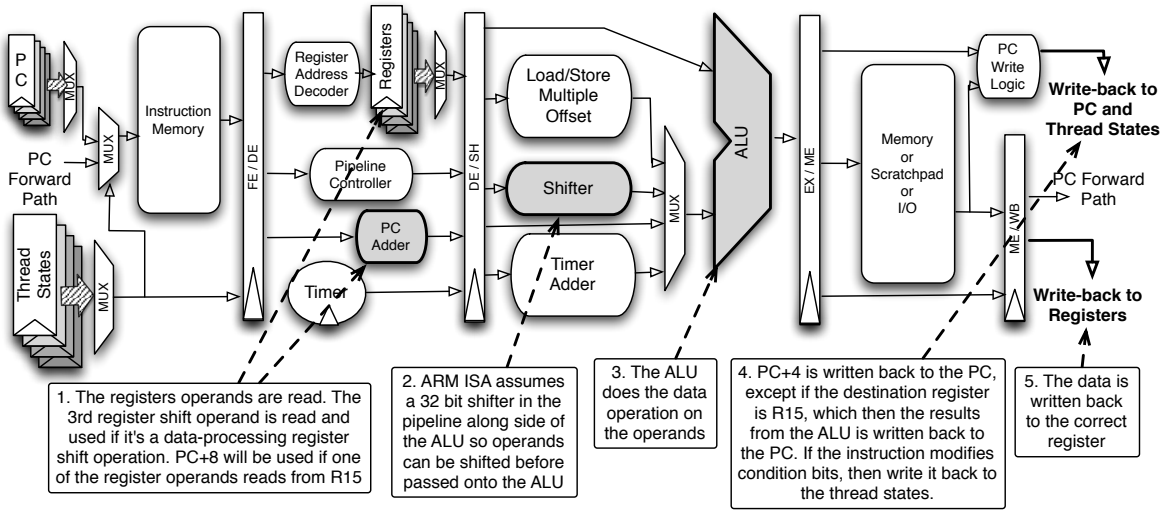

Figure 3.3: Data Processing Instruction Execution in the PTARM Pipeline

3.3.1 Data-Processing

We begin by explaining how data-processing instructions are implemented. These instructions are used to manipulate register values by executing register to register operations. Most data-processing instructions take two operands. One operand is always a register value, the second operand is labeled the shifter operand. The shifter operand could be an immediate value or a register value, both which can be shifted to form the final operand that is fed into the ALU. Figure 3.3 explains how data-processing instructions are executed through the pipeline.

Because R15 is PC, so data-processing instructions that use R15 as an operand will read the value of PC+8 as the operand. Any instruction that uses R15 as the destination register will trigger a branch to the result of the computation. As discussed earlier, our pipeline commits the next PC in the memory stage, so to trigger a branch from data-processing instructions simply means storing back the results from the ALU as the next PC. In our thread-interleaved pipeline, when the next PC from the current thread is fetched, it will contain already contain the target address to branch to when we issue a data-processing instruction that writes to R15.

Data processing instructions can also update the program condition code flags that are stored in the thread state. The condition code flags are used to predicate execution for ARM instructions, and consists of four bits: Zero (Z), Carry (C), Negative (N) and Overflow (V). The high four bits of each instruction forms a conditional field that is checked against the thread state condition code flags to determine whether or not the instruction is executed. The conditional execution for each instruction is checked in the pipeline controller. Data-processing instructions provide a mechanism to update the condition code flags according to the results of data operations. The instructions that update the flags do not write any data back to the registers, they simply update the condition code flags.

All data-processing instructions only take one pass through the pipeline, even instructions that read from or write to R15, so all data-processing instructions take one thread cycle to execute.

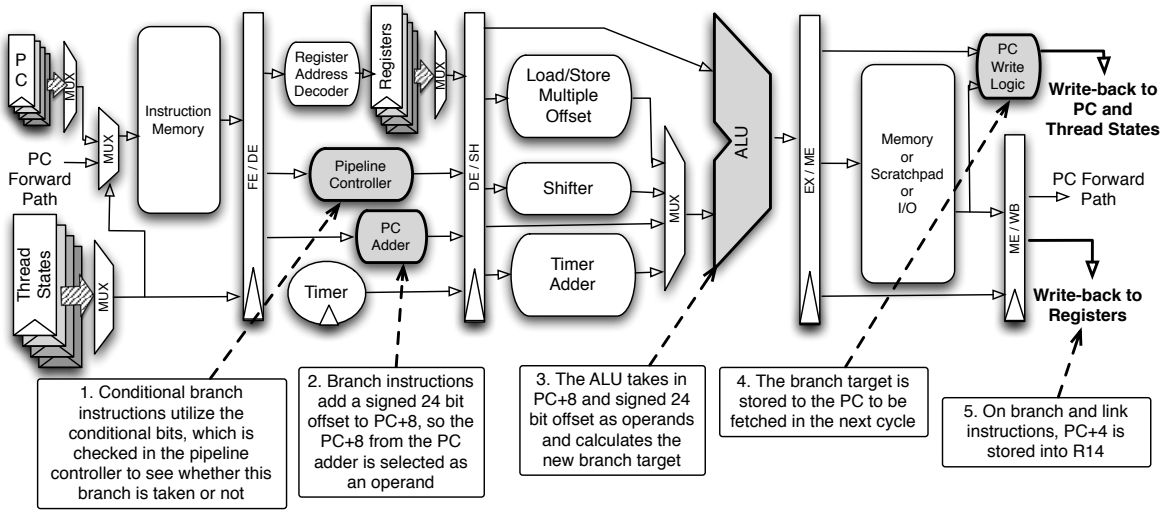

Figure 3.4: Branch Instruction Execution in the PTARM Pipeline

3.3.2 Branch

Branch instructions in the ARM can conditionally branch forward or backwards by up to 32MB. There is no explicit conditional branch instruction in ARM. Conditional branches are implemented using the ARM predicated instruction mechanism. So the condition used to determine if a conditional branch is taken is simply the condition code flags in the thread state. Figure 3.4 show how branch instructions are executed in the thread-interleaved pipeline.

The branch instructions for the ARM ISA calculate the branch target address by adding a 24 bit signed offset, specified in the instruction, to the current PC incremented by 8. Thus, the PC adder, in addition to incrementing the PC by the conventional offset of 4, also increments the PC by 8, to be used as an operand for the ALU to calculate the target branch address. Once the address is calculated, it is written back to its thread's next PC ready to be fetched. If the instruction is a branch and link (*bl*) instruction, PC+4 is propagated down the pipeline and written back to the link register (R14).

All branch instructions, whether conditionally taken or not, all take only one thread cycle to execute. But more importantly, the next instruction after the branch, whether it is a conditional branch or not, is not stalled or speculatively executed. The execution time of instructions from the same thread after the branch is not stalled nor affected by the branch instruction. The thread-interleaved pipeline simplified the implementation of the branch instruction and control hazard handling logic, as the pipeline will not need the results of the branch target address calculation the very next processor cycle. Instead, instructions from other threads will be fetched before the results of the branch is needed.

3.3.3 Memory Instructions

There are two type of memory instructions implemented in PTARM from the ARM ISA: Load/Store Register and Load/Store Multiple. We discuss both type of memory instructions, and in particular, the interaction of the pipeline with the memory hierarchy presented earlier. We also present a special case when the load instruction loads to R15, which loads a branch target address

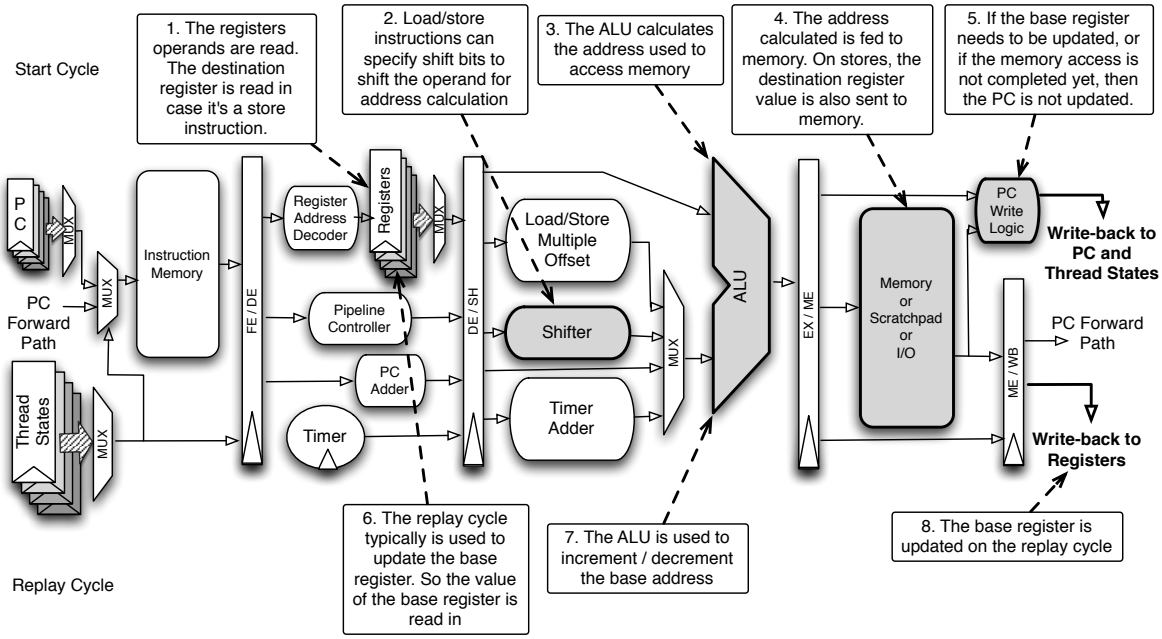

Figure 3.5: Load/Store Instruction Execution in the Ptarm Pipeline

from memory and triggers a branch. This slightly complicates our pipeline design, but we show that it does not affect the timing and execution of the instruction and subsequent instructions. Currently load/store halfword doubleword is not implemented in PTARM, as they fall under the miscellaneous instructions category. These instructions can easily be implemented using the same principles described below without significant hardware additions.

Load/Store Register

Load instructions load data from the memory and writes them into the registers. Store instructions store data from the registers into memory, thus store instructions utilize the extra register read port to read in the register value to be stored into memory. The address used to access memory is formed by combining a base register and an offset value. The offset value can be a 12 bit immediate supplied from the instruction, or a register operand that can be shifted. The current load/store instructions can support word operations or byte operations. Figure 3.5 shows how the load/store instruction is executed in the pipeline.

All load and store instructions in ARM have the ability to update the base register after any memory operation. This compacts code that reads arrays, as a load or store instruction can access memory and updates the base register so the next memory access is done on the updated base register. Different address modes differentiate how the base address register is updated. Pre-indexed addressing mode calculates the memory address by first using the value of the base register and offset, then updating the base register. Post-indexed addressing mode first updates the base register, then uses the updated base register value along with the offset to form the memory address. Offset addressing mode simply calculates the address from the base register and offset, and does not update the base register. The base register could be either incremented or decremented. When pre and post-indexed addressing modes are used, memory operations require at least an additional thread

cycle to complete. Because the register file only contains one write port, we cannot simultaneously write back a load result from memory and the updated base register to the register file. Thus, we need to spend an extra pass through the pipeline to update the base register.

When the memory address is accessing the scratchpad memory region, memory operations can be completed in a single cycle, and the data is ready by the next (*writeback*) stage to be written back to the registers. However, if the memory read/write operation is accessing the memory region of the DRAM, the request must go through the DRAM memory controller to access the DRAM. DRAM operations typically take three or four thread cycles to complete. As discussed in chapter 2, our thread-interleaved pipeline implementation does not dynamically switch threads in and out of execution when they are stalled waiting for memory access to complete. Thus, the when a memory instruction accesses the DRAM memory region, the same instruction is replayed by withholding the update for the next PC, until the data from DRAM arrives and is ready to be written back in the next stage. For memory instructions accessing I/O regions, the access latency depends on the I/O accessed and the connection of the bus. As mentioned in section 3.2, the actual access time to I/O devices is device dependent, and a discussion of time-predictable buses is outside the scope of this thesis. In the hardware implementation, for memory instructions that access the DRAM or I/O region, it is possible to update the base register earlier during the cycles where the instruction is waiting for access to complete. However, the current PTARM implementation uses the same logic and datapath for all memory accesses (scratchpad, DRAM, I/O etc) to minimize hardware resources, so an additional cycle is used to update the base register for all memory accesses regardless of the address region they are accessing.

Load/Store Multiple

The load/store multiple instruction is used to load (store) a subset, or possibly all, of the general purpose registers from (to) memory. This instruction is often used to compact code that pushes or pops registers from the program stack. The list of registers that are used in this instruction is specified in the register list as a 16 bit field in the instruction. The 0th bit of the bit field representing R0 and the 15th bit representing R15. A base register supplies the base memory address that is loaded from or stored to, which then is sequentially incremented or decremented by 4 bytes for each register that is operated on. Figure 3.6 shows how the load/store multiple instruction is executed in the pipeline.

The load/store multiple instruction is inherently a multi-cycle instruction, because each thread cycle we can only write back one value to the register or store one value to memory. Thus, the execution state and remaining register list of the load/store multiple instruction is stored in thread state. After the decoding the instruction, the remaining thread cycles load the register field from the thread state and clears it as registers are being operated on. The instruction completes when all registers have been operated on. Each iteration the *register address decoder* in the pipeline decodes the register list and determines the register being operated on. For load multiple, this indicates the destination register that is written back to. For store multiple, this indicates the register whose value will be stored to memory. The *load/store multiple offset* block is used to obtain the current memory address offset depending on how far we are in the execution of this instruction. The offset is added to the base register to form the memory address fed into memory.

The execution time of this instruction depends on the number of registers specified in the register list and the memory region that is being accessed. For accesses to the scratchpad,

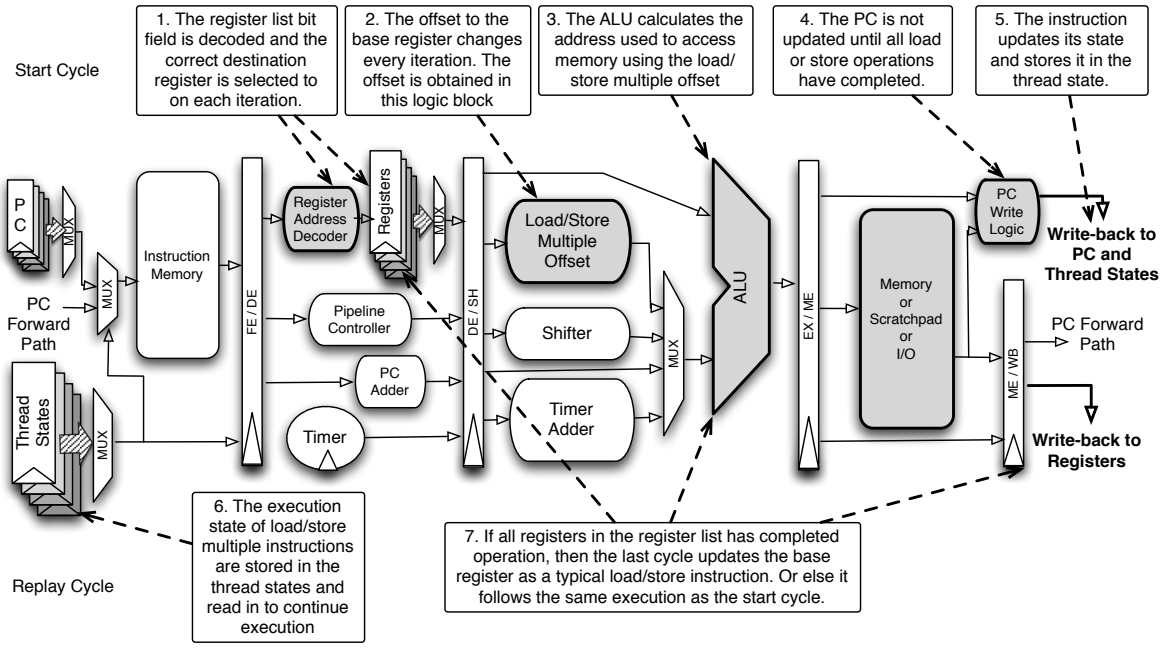

Figure 3.6: Load/Store Multiple Instruction Execution in the PTARM Pipeline

each register load or store takes only a single cycle. However, if memory accesses are to the DRAM region, then register load/store will take multiple cycles. It is also possible for the load/store multiple instruction to update the base register after all the register operations completes. Similar to the load/store register instruction, an additional thread cycle will be used to update the base register. Although the execution time of this instruction seems to be dynamic depending on the number of registers specified in the register list, but the instruction binary will allow us to statically determine that number by parsing the bit field of the instruction. Thus, the execution time of this instruction can still be statically analyzed.

Load to PC

When load/store operations load to the destination register R15, it triggers a branch in the pipeline. This also holds true for the load multiple instruction if the 15th bit is set in the register list. In our five stage pipeline, we commit the next PC in the memory stage so the next instruction fetch from the same thread can fetch the updated PC. However, when the branch target address is loaded from memory, the address is not yet present in the memory stage to be committed, but only at the beginning of the writeback stage will it be present. Thus, we introduce a forwarding path that forwards the PC straight from the writeback stage to the fetch stage. Figure 3.7 shows how this is implemented in our pipeline.

An extra multiplexer is placed in the fetch stage before the instruction fetch to select the forward path. When a load to R15 is detected, it will signal the thread state to use the forwarded PC on the next instruction fetch, instead of the one stored in next PC. Because the data from memory will be ready at the beginning of the writeback stage, the correct branch target address will be selected and used. We discussed in section 2.1.1 the timing implications of data-forwarding logic in the pipeline. Those same principles are applied in this situation. Although it seems the selection of

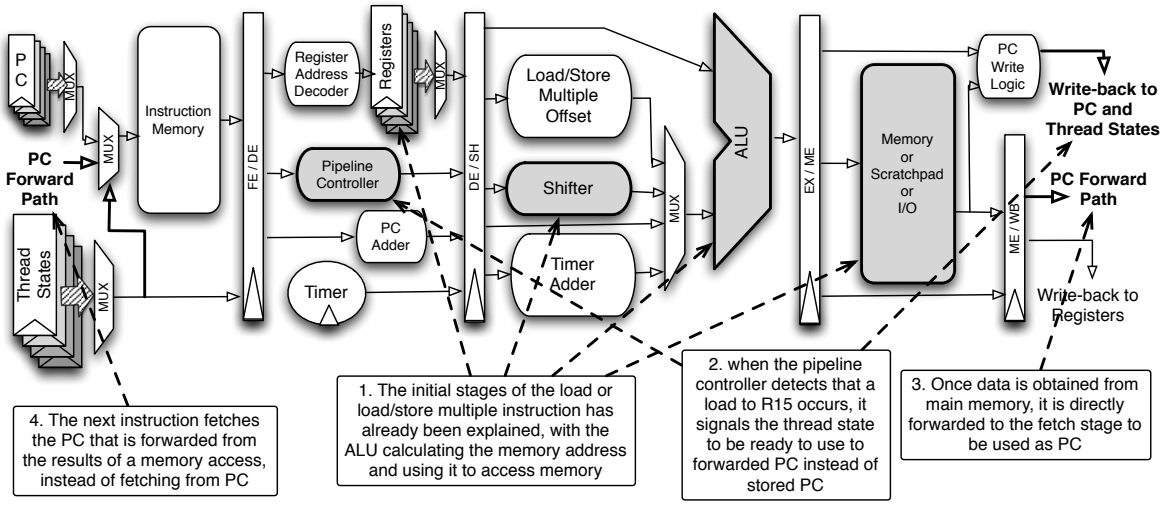

Figure 3.7: Load to R15 Instruction Execution in the PTARM Pipeline

PC is dynamic, but when forwarding occurs is actually static; this PC forwarding only and always occurs when instructions load from memory to R15. This mechanism has no additional timing effects on any following instructions, as no stalls are needed to wait for the address to be ready. Even if the load to R15 instruction is accessing the DRAM region, the timing of this instruction does not deviate from a load instruction destined for other registers. Although the target address will not be known until after the DRAM access completes, a load instruction that does not load to R15 also needs to wait until the DRAM access completes before the thread fetches the next instruction. So this extra forwarding mechanism does not cause load to R15 instructions to deviate from other load timing behaviors.

If the load to R15 instruction updates the base register, then the forwarding path is not used and not needed. The extra cycle used to update the base register will allow us to propagate the results from memory to be committed in the memory stage. The timing behavior still conforms to a regular load to registers instruction.

3.3.4 Exception Handling

When exceptions occur in a single threaded pipeline, the whole pipeline must be flushed because of the control flow shift in the program. The existing instructions in the pipeline become invalid, and the pipeline overwrites the PC to jump to a specified exception handler. The ARM ISA specifies seven types of exceptions, and an exception vector table that points the PC to specified handler addresses when those exceptions occur. The exception vector table is stored in the BootROM in our implementation. Exceptions can be triggered by external or internal events that occur in the pipeline, such as an toggling an external interrupt signal or using a software interrupt instruction to trigger the exception programmatically. But no matter how the exceptions are generated, they must be handled predictably in the pipeline.

In the context of a thread interleaved pipeline, all threads are temporally isolated. Thus, an exception that occurs on one thread must not effect the execution of other threads in the pipeline. In our pipeline, any exceptions or interrupts that occur during execution are latched at each stage

and propagated down the pipeline with the instruction. An instruction flush signal is toggled to ensure that this instruction does not commit any state to memory or registers. The exception type is checked at the memory stage in the PC write logic before the next PC is committed. According to the exception type, the program state register bits are set, and the PC is redirected to the correct entry in the exception vector table. The current PC is passed on to the writeback stage to store in the link register (R14). This provides a mechanism for the program return to the initial instruction where the exception occurs and re-execute it if desired, since the instruction did not complete its execution. If the exception is generated from after a memory access and detected in the writeback stage, the PC forwarding path is used to fetch the exception vector entry for data memory exceptions. Note that it is up the each exception handler to save the register states and stack of the program.

None of the instructions that are executing in the pipeline are flushed when an exception occurs in our pipeline. As shown in figure 3.8, the instructions that are executing in other pipeline stages all belong to other threads, so no flushing of the pipeline is required because no instruction was executed speculatively. This simplifies the timing analysis of exceptions in our pipeline, as the timing behavior of other threads in the pipeline are unaffected. For the thread which the exception occurs, the only overhead to handling the exception is that the current instruction does not complete its execution this thread cycle. The next thread cycle the pipeline will be handling the exception already, resulting in no additional stalls for the thread.

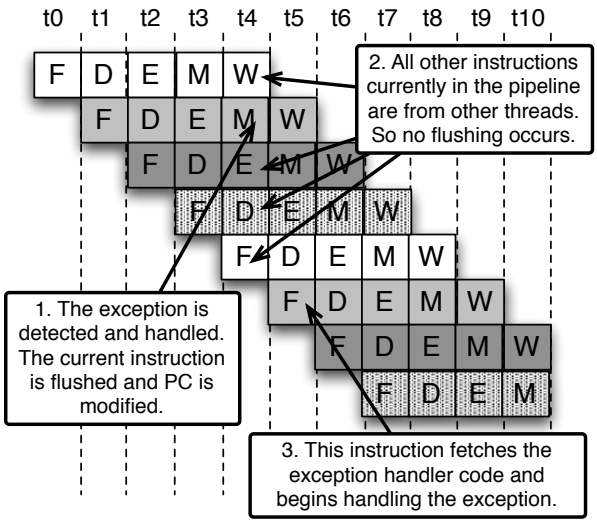

Figure 3.8: Handling Exceptions in PTARM

It is possible that an exception occurs during a memory access instruction that is waiting for the results from DRAM to complete. In this case, because the memory request is also sent to the DRAM controller, and possible already being serviced by the DRAM, we cannot cancel the memory request abruptly. In the case that the interrupted instruction was a load, we can simply disregard the results of the load, but if the instruction was a store, we cannot cancel the store request that is writing data to the memory. So it is up to the programmer to disable interrupts before writing to critical memory locations that require a consistent program state. By interrupting an instruction that is waiting on memory access to complete, we also potentially complicate the interaction with our DRAM controller. The DRAM controller can only service one request from each thread at a time for predictable performance(Todo: elaborate on this?). This normally is not an issue because our pipeline does not reorder instructions or speculatively execute while there are outstanding memory requests, but the pipeline waits until the request is finished before continuing execution. However, if a memory instruction is interrupted, the pipeline flushes the current instruction and continues execution of the exception handler in the Boot ROM. If at this point, the exception handler contains a memory request instruction to the DRAM, a memory request would be issued to the DRAM controller that is still servicing the previous request prior to the exception from this thread. The current memory request in this case would need to wait

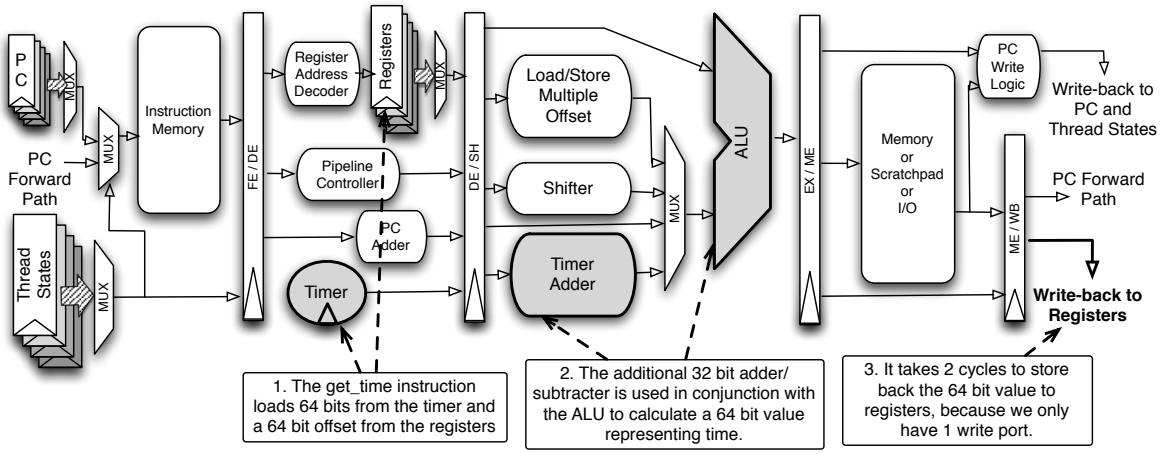

Figure 3.9: Get_Time Instruction Execution in the PTARM Pipeline

until the previous "canceled" memory request to complete its service by the DRAM before it can begin being serviced. This creates timing variability to the load instructions because the execution time of load instructions would be different depending on whether an exception just occurred or not. Because this situation can only occur in the exception handler, because it contains the instructions executed right after an exception, so we leave it to the compiler to ensure that the first few instructions in the exception handler code does not access the DRAM memory region. In PTARM, the compiler simply needs to ensure that the first three instructions executed from an exception handler are not instructions that access the DRAM.

Currently PTARM does not implement an external interrupt controller to handle external interrupts. But when implementing such an interrupt controller, each thread should be able to register specific external interrupts that it handles. For example, we might have a hard real-time task that is executing on one thread, while another thread without timing constraints is executing on another thread waiting for an interrupt to signal the completion of a UART transfer. In this case the thread running the hard real-time task should not be effected even if it is in execution when the interrupt occurs. Only the specific thread handling the UART transfers should be interrupt by this interrupt. So we envision an interrupt controller that allows each thread to register specific interrupts that it handles, without affecting other threads in the pipeline.

3.3.5 Timing Instructions

Get_Time

get time blah blah

Delay_Until

delay until blah blah

Exception_on_Expire, Deactivate_Exception

exception on expire blah blah

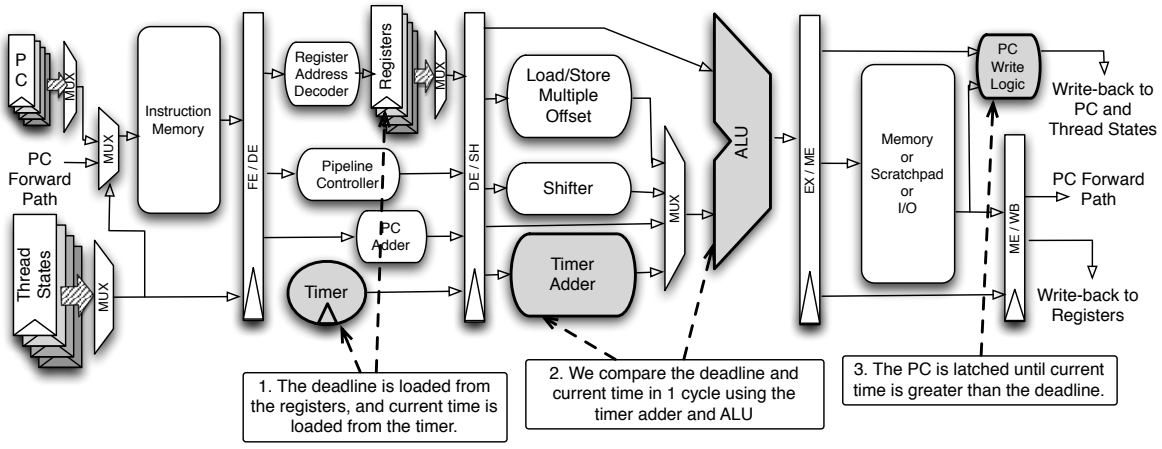

Figure 3.10: Delay_Until Instruction Execution in the PTARM Pipeline

3.4 Worst Case Execution Time Analysis

3.5 PTARM VHDL Soft Core

Our pipeline can be clocked up to $100MHz$ when synthesized to a Virtex-5 lx110t FPGA.

Talk about I/O devices, including UART, DVI controller and Interface with DDR2 DRAM controller

3.6 PTARM Simulator

Talk about experimentation with DMA and memory hierarchy

Chapter 4

Related Work

4.1 Academia

4.1.1 Architectural Modifications

Craven et. al [9] implements PRET thread interleaved pipeline as an open source core using OpenFire

Fig. 4.1 shows an image

4.2 Industry

Here is another header

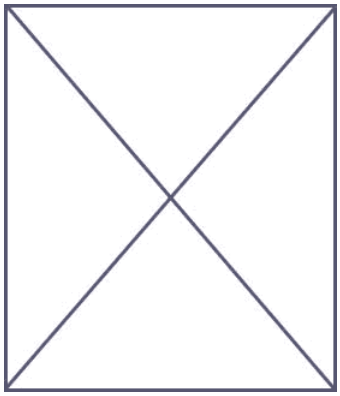

Figure 4.1: Image Placeholder

Chapter 5

Applications

5.1 Eliminating Side-Channel-Attacks

Encryption algorithms are based on strong mathematical properties to prevent attackers from deciphering the encrypted content. However, their implementations in software naturally introduce varying run times because of data-dependent control flow paths. Timing attacks [15] exploit this variability in cryptosystems and extract additional information from executions of the cipher. These can lead to deciphering the secret key. Kocher describes a timing attack as a basic signal detection problem [15]. The "signal" is the timing variation caused by the key's bits when running the cipher, while "noise" is the measurement inaccuracy and timing variations from other factors such as architecture unpredictability and multitasking. This signal to noise ratio determines the number of samples required for the attack – the greater the "noise," the more difficult the attack. It was generally conceived that this "noise" effectively masked the "signal," thereby shielding encryption systems from timing attacks. However, practical implementations of the attack have since been presented [6, 10, 27] that clearly indicate the "noise" by itself is insufficient protection. In fact, the architectural unpredictability that was initially believed to prevent timing attacks was discovered to enable even more attacks. For example, computer architects use caches, branch predictors and complex pipelines to improve the average-case performance while keeping these optimizations invisible to the programmer. These enhancements, however, result in unpredictable and uncontrollable timing behaviors, which were all shown to be vulnerabilities that led to side-channel attacks [3, 22, 2, 8].

In order to not be confused with Kocher's [15] terminology of *timing attacks* on algorithmic timing differences, we classify all above attacks that exploit the timing variability of software implementation *or* hardware architectures as *time-exploiting attacks*. In our case, a *timing attack* is only one possible *time-exploiting attack*. Other time-exploiting attacks include branch predictor, and cache attacks. Examples of other side-channel attacks are power attacks [19, 14], fault injection attacks [4, 11], and many others [27].

In recent years, we have seen a tremendous effort to discover and counteract side-channel attacks on encryption systems [4, 8, 16, 12, 1, 13, 7, 26, 25]. However, it is difficult to be fully assured that all possible vulnerabilities have been discovered. The plethora of research on side-channel exploits [8, 4, 16, 12, 1, 13, 7, 26, 25] indicates that we do not have the complete set of solutions as more and more vulnerabilities are still being discovered and exploited. Just recently, Coppens et al. [8] discovered two previously unknown time-exploiting attacks on modern x86 pro-

cessors caused by the out-of-order execution and the variable latency instructions. This suggests that while current prevention methods are effective at *defending* against their particular attacks, they do not *prevent* other attacks from occurring. This, we believe, is because they do not address the root cause of time-exploiting attacks, which is that run time variability *cannot be controlled* by the programmer.

It is important to understand that the main reason for time-exploiting attacks is *not* that the program runs in a varying amount of time, but that this variability *cannot be controlled* by the programmer. The subtle difference is that if timing variability is introduced in a controlled manner, then it is still possible to control the timing information that is leaked during execution, which can be effective against time-exploiting attacks. However, because of the programmer's *lack of control* over these timing information leaks in modern architectures, noise injection techniques are widely adopted in attempt to make the attack infeasible. These include adding random delays [15] or blinding signatures [15, 7]. Other techniques such as branch equalization [20, 27] use software techniques to rewrite algorithms such that they take equal time to execute during each conditional branch. We take a different approach, and directly address the crux of the problem, which is the *lack of control* over timing behaviors in software. We propose the use of an embedded computer architecture that is designed to allow predictable and controllable timing behaviors.

At first it may seem that a predictable architecture makes the attacker's task simpler, because it reduces the amount of "noise" emitted from the underlying architecture. However, we contend that in order for timing behaviors to be controllable, the underlying architecture *must* be predictable. This is because it is meaningless to specify any timing semantics in software if the underlying architecture is unable to honor them. And in order to guarantee the execution of the timing specifications, the architecture must be predictable. Our approach does not attempt to increase the difficulty in performing time-exploiting attacks, but to eliminate them completely.

In this paper, we present the PREcision Timed (PRET) architecture [18] in the context of embedded cryptosystems, and show that an architecture designed for predictability and controllability effectively eliminates all time-exploiting attacks. Originally proposed by Lickly et al [18], PRET provides instruction-set architecture (ISA) extensions that allow programmers to control an algorithm's temporal properties at the software level. To guarantee that the timing specifications are honored, PRET provides a predictable architecture that replaces complex pipelines and speculation units with multithread-interleaved pipelines, and replaces caches with software-managed fast access memories. This allows PRET to maintain predictability without sacrificing performance. We target embedded applications such as smartcard readers [16], key-card gates [5], set-top boxes [16], and thumbpods [23], which are a good fit for PRET's embedded nature. We demonstrate the effectiveness of our approach by running both the RSA and DSA [21] encryption algorithms on PRET, and show its immunity against time-exploiting attacks. This work shows that a disciplined defense against time-exploiting attacks requires a combination of software and hardware techniques that ensure controllability and predictability.

5.2 Real Time 1D Computational Fluid Dynamics Simulator

Here is another header

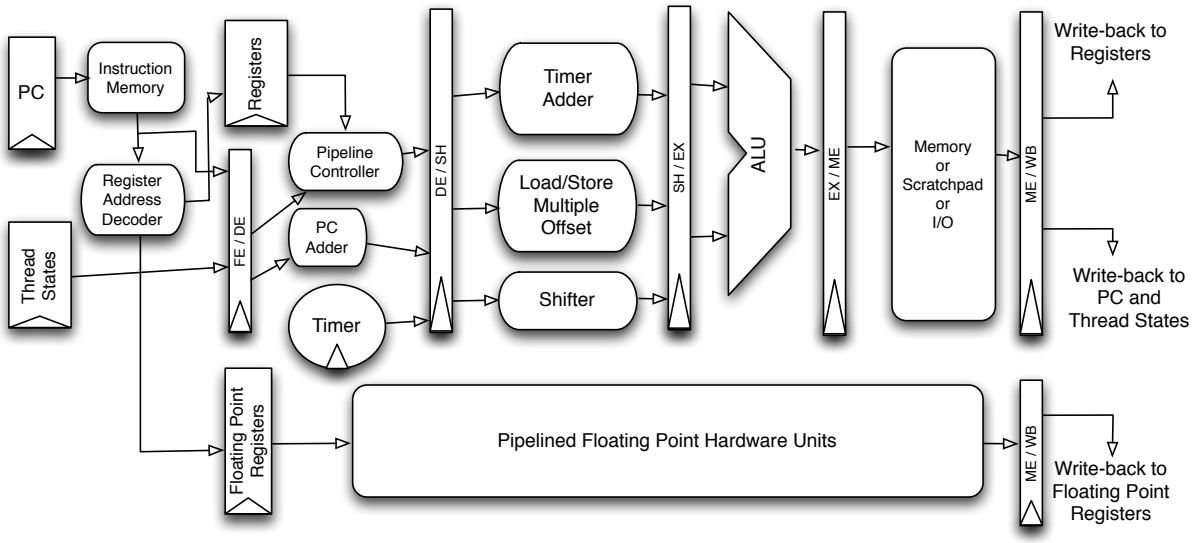

Figure 5.1: Block Level View of the PTARM 6 stage pipeline

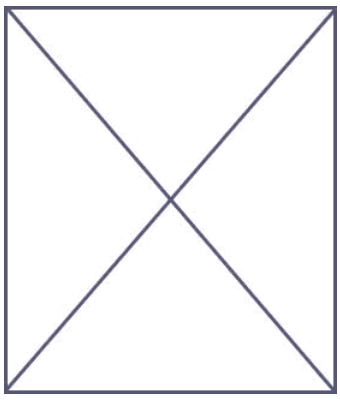

Figure 5.2: Image Placeholder

Chapter 6

Conclusion and Future work

6.1 Summary of Results

This is my summary

6.2 Future Work

Here is what you can keep doing
Talk about future research challenges for a predictable architecture.

- synchronization of threads, atomic primitives and memory barrier?
- Bus and I/O architectures

Bibliography

[1] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. On the Power of Simple Branch Prediction Analysis. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320, New York, NY, USA, 2007. ACM.

[2] O. Aclicmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *in Cryptology CT-RSA 2007, The Cryptographers Track at the RSA Conference 2007*, pages 225–242. Springer-Verlag, 2007.

[3] D. J. Bernstein. Cache-timing Attacks on AES, 2004.

[4] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. *Lecture Notes in Computer Science*, 1294:513–525, 1997.

[5] S. C. Bono, M. Green, A. Stubblefield, A. Juels, A. D. Rubin, and M. Szydlo. Security analysis of a cryptographically-enabled rfid device. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[6] D. Brumley and D. Boneh. Remote timing attacks are practical. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[7] D. Chaum. Blind Signatures for Untraceable Payments. In *Advances in Cryptology: Proceedings of Crypto 82*, pages 199–203. Plenu Press, 1983.

[8] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors, 2009.

[9] S. Craven, D. Long, and J. Smith. Open source precision timed soft processor for cyber physical system applications. In *Proceedings of the 2010 International Conference on Reconfigurable Computing and FPGAs*, RECONFIG '10, pages 448–451, Washington, DC, USA, 2010. IEEE Computer Society.

[10] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater, and J.-L. Willems. A Practical Implementation of the Timing Attack. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of the Third Working Conference on Smart Card Research and Advanced Applications (CARDIS 1998)*. Springer-Verlag, 1998.

[11] M. Feng, B. B. Zhu, M. Xu, S. Li, B. B. Zhu, M. Feng, B. B. Zhu, M. Xu, and S. Li. Efficient Comb Elliptic Curve Multiplication Methods Resistant to Power Analysis, 2005.

[12] R. Karri, K. Wu, P. Mishra, and Y. Kim. Fault-Based Side-Channel Cryptanalysis Tolerant Rijndael Symmetric Block Cipher Architecture. In *DFT '01: Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, page 427, Washington, DC, USA, 2001. IEEE Computer Society.

[13] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. In *Journal of Computer Security*, pages 97–110. Springer-Verlag, 1998.

[14] P. Kocher, J. J. E, and B. Jun. Differential Power Analysis. In *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[15] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[16] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *USENIX Workshop on Smartcard Technology proceedings*, pages 9–20, 1999.

[17] E. Lee and D. Messerschmitt. Pipeline interleaved programmable DSP's: Architecture. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 35(9):1320–1333, 1987.

[18] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable Programming on a Precision Timed Architecture. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 137–146, New York, NY, USA, 2008. ACM.

[19] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *In USENIX Workshop on Smartcard Technology*, pages 151–162, 1999.

[20] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *In Cryptology ePrint Archive, Report 2005/368*, 2005.

[21] National Institute of Standards and Technology. "Digital Signature Standard". Federal Information Processing Standards Publication 186, 1994.

[22] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, page 05, 2005.

[23] P. Schaumont, K. Sakiyama, Y. Fan, D. Hwang, S. Yang, A. Hodjat, B. Lai, and I. Verbauwhede. Testing ThumbPod: Softcore bugs are hard to find. In *Eighth IEEE International High-Level Design Validation and Test Workshop, 2003*, pages 77–82, 2003.

[24] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35:29–63, March 2003.

[25] Z. Wang and R. B. Lee. Covert and Side Channels Due to Processor Architecture. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 473–482, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494 – 505, San Diego, CA, June 2007 2007.

[27] Yongbin. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.