

**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

# **Precision Timed Machines**

Copyright 2012  
by  
Isaac Liu

**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

|   |           |
|---|-----------|
| <b>List of Figures</b>                                  | <b>iv</b> |
| <b>List of Tables</b>                                   | <b>v</b>  |
| <b>1 Introduction</b>                                   | <b>1</b>  |
| 1.1 Background . . . . .                                | 1         |
| 1.2 Intro Section Header 2 . . . . .                    | 1         |
| <b>2 Precision Timed Machine</b>                        | <b>2</b>  |
| 2.1 Pipelines . . . . .                                 | 2         |
| 2.1.1 Pipeline Hazards . . . . .                        | 2         |
| 2.1.2 Pipeline Multithreading . . . . .                 | 6         |
| 2.1.3 Thread-Interleaved Pipelines . . . . .            | 9         |
| 2.2 Memory System . . . . .                             | 13        |
| 2.2.1 Scratchpad . . . . .                              | 13        |
| 2.2.2 DRAM memory controller . . . . .                  | 13        |
| 2.3 Programming Models . . . . .                        | 13        |
| 2.3.1 PRET Programming model Section Header . . . . .   | 13        |
| 2.3.2 Pret Programming model Section Header 2 . . . . . | 13        |
| <b>3 Implementation of PTARM</b>                        | <b>14</b> |
| 3.1 PTARM Architecture . . . . .                        | 15        |
| 3.2 Instruction Implementations . . . . .               | 17        |
| 3.2.1 Data-Processing . . . . .                         | 17        |
| 3.2.2 Branch . . . . .                                  | 17        |
| 3.2.3 Loads and Stores . . . . .                        | 18        |
| 3.2.4 Timing Instructions . . . . .                     | 18        |
| 3.2.5 Synthesis Results . . . . .                       | 18        |
| 3.3 PTARM Simulator . . . . .                           | 18        |
| 3.4 Worst Case Execution Time Analysis . . . . .        | 18        |
| <b>4 Related Work</b>                                   | <b>19</b> |
| 4.1 Architectural Modifications . . . . .               | 19        |
| 4.2 Related Section Header 2 . . . . .                  | 19        |

|          |   |           |
|----------|---|-----------|
| <b>5</b> | <b>Applications</b>   | <b>20</b> |
| 5.1      | Eliminating Side-Channel-Attacks . . . . .                    | 20        |
| 5.2      | Real Time 1D Computational Fluid Dynamics Simulator . . . . . | 21        |
| <b>6</b> | <b>Conclusion and Future work</b>                             | <b>23</b> |
| 6.1      | Summary of Results . . . . .                                  | 23        |
| 6.2      | Future Work . . . . .   | 23        |
|          | <b>Bibliography</b>   | <b>24</b> |

# List of Figures

|     |  |    |
|-----|--|----|
| 1.1 | Image Placeholder . . . . .  | 1  |
| 2.1 | Sample code with data dependencies . . . . .   | 2  |
| 2.2 | Handling of data dependencies in single threaded pipelines . . . . .   | 3  |
| 2.3 | Sample code for GCD with conditional branches . . . . .  | 4  |
| 2.4 | Handling of conditional branches in single threaded pipelines . . . . .                                      | 5  |
| 2.5 | Simple Multithreaded Pipeline . . . . .  | 7  |
| 2.6 | Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5<br>pipeline stages . . . . . | 8  |
| 2.7 | Execution of 5 threads thread-interleaved pipeline when 2 threads are inactive . . .                         | 11 |
| 3.1 | Block Level View of the PTARM 5 stage pipeline . . . . .   | 15 |
| 3.2 | Branch Instruction Execution in the Ptarm Pipeline . . . . .   | 17 |
| 3.3 | Load/Store Instruction Execution in the Ptarm Pipeline . . . . .   | 18 |
| 4.1 | Image Placeholder . . . . .  | 19 |
| 5.1 | Block Level View of the PTARM 6 stage pipeline . . . . .   | 22 |
| 5.2 | Image Placeholder . . . . .  | 22 |

# 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-interleaved pipeline and memory hierarchy, Chapter ??, Chapter 5, Chapter 6,

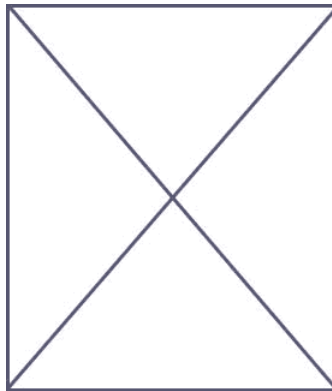


Figure 1.1: Image Placeholder

## 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

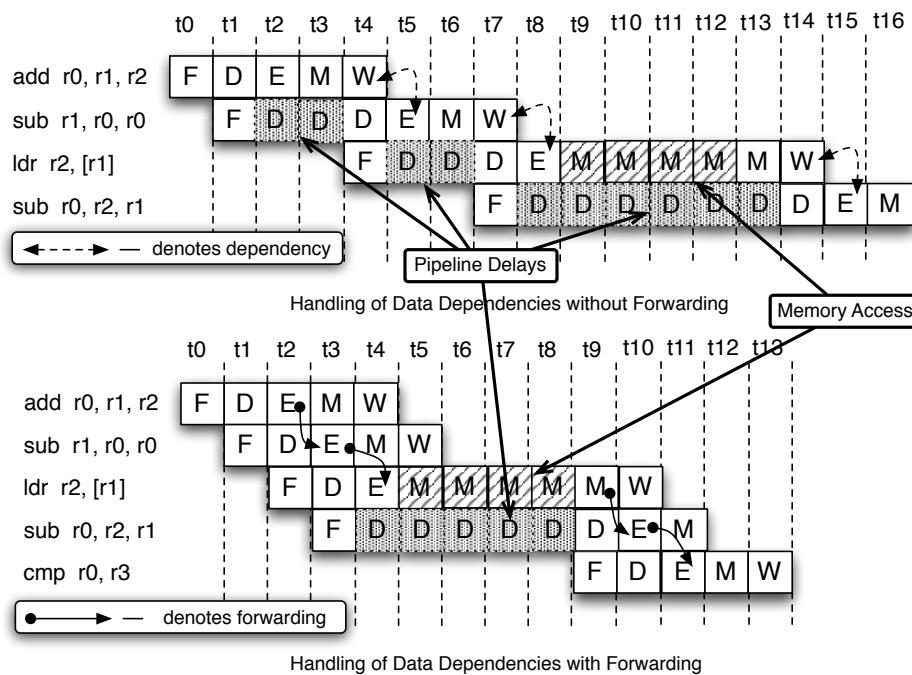


Figure 2.2: Handling of data dependencies in single threaded pipelines

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

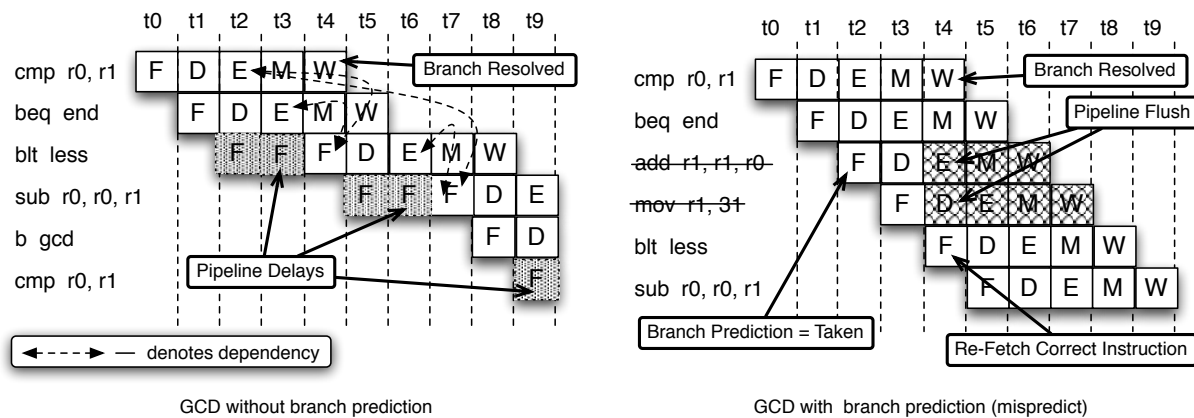


Figure 2.4: Handling of conditional branches in single threaded pipelines

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 it is 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

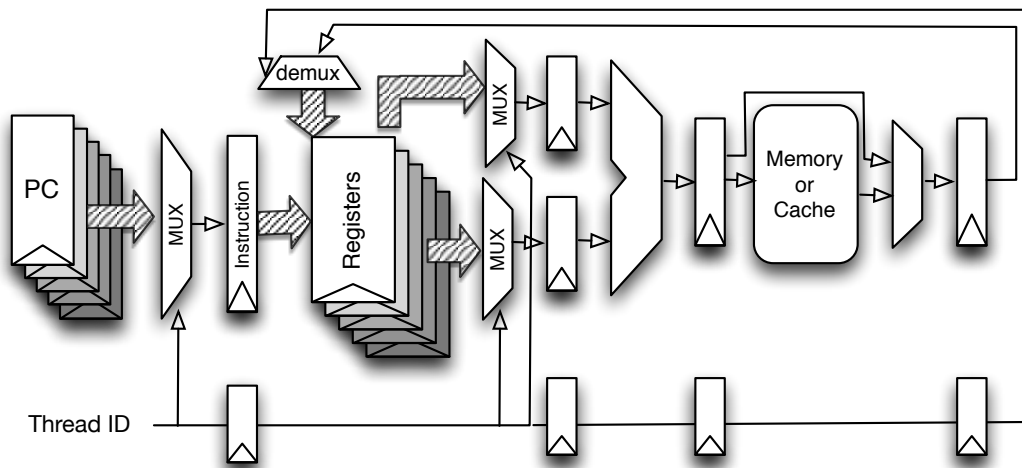


Figure 2.5: Simple Multithreaded Pipeline

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. [23] 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 Simultaneous Multithreaded (SMT) architectures, can fetch and execute instructions from several hardware threads in the same cycle.



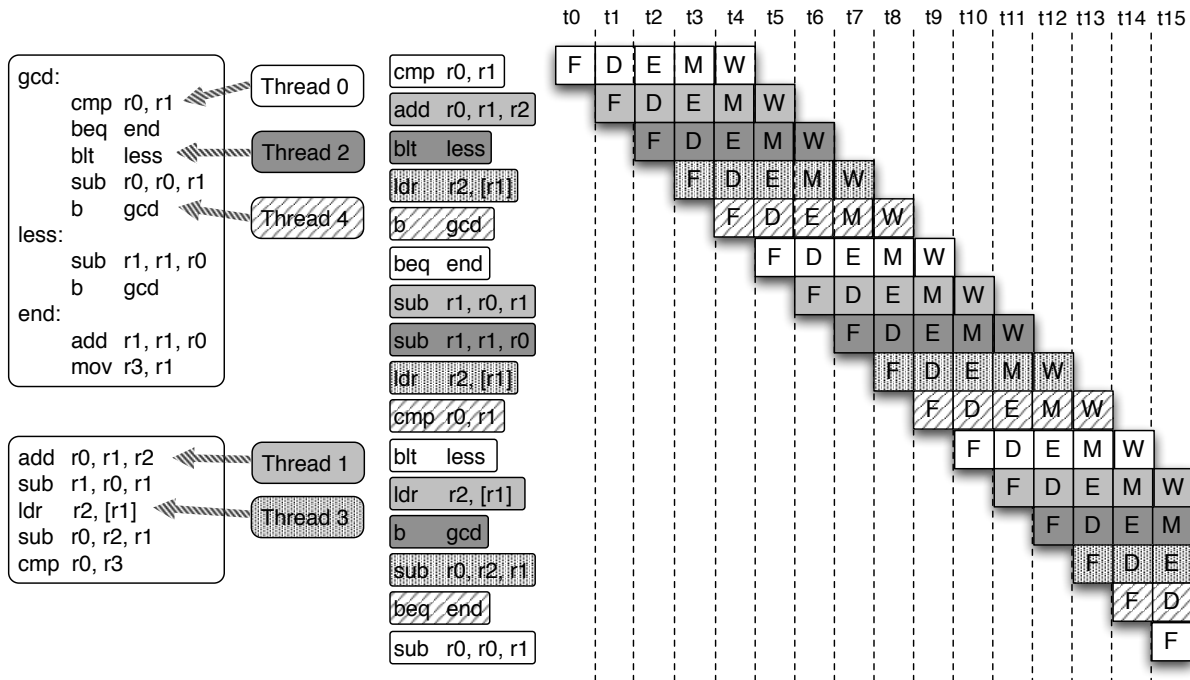


Figure 2.6: Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5 pipeline stages

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 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 with 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  $t_0$ . 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

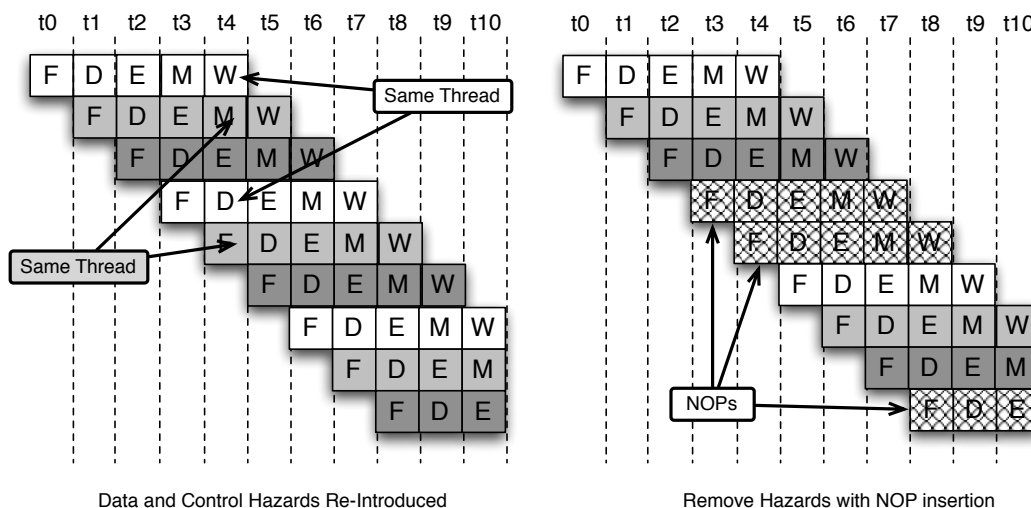


Figure 2.7: Execution of 5 threads thread-interleaved pipeline when 2 threads are inactive

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 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 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 a 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 thread's 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 to 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, the worst case execution time of a TDMA access may seem like the same as the basic queuing scheme.

It is important to understand that it is not the thread scheduling that is non-predictable, but how thread switches are triggered. It is still possible for a predictable architecture to support the

scheduling of hardware threads, but it must be done statically in software, instead of dynamically in hardware. If the hardware can provide a means of explicitly setting thread schedules in software, then it is possible to make the thread schedules transparent to any timing analysis. The software still needs to be carefully constructed, since it is still possible for the software to switch hardware thread schedules dynamically, complicating the timing analysis. One could envision a program that contains different program states, each state executing a different number of threads. As long as the state transitions are statically analyzable, the timing analysis complexity is not increased.

In summary, blah blah blah. . .

## **2.2 Memory System**

### **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 build a single cycle hardware shifter along with an 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 base address. 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 read 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 registers (R0 to R15) to be accessed with its instructions, with register 15 being the program counter. Any data-operation done on register 15 triggers a branch.

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 includes hardware support for several floating point operations, and implements the timing instructions introduced in chapter 2.3. We will first discuss the architectural details of PTARM, then present the C++ software simulator of PTARM.

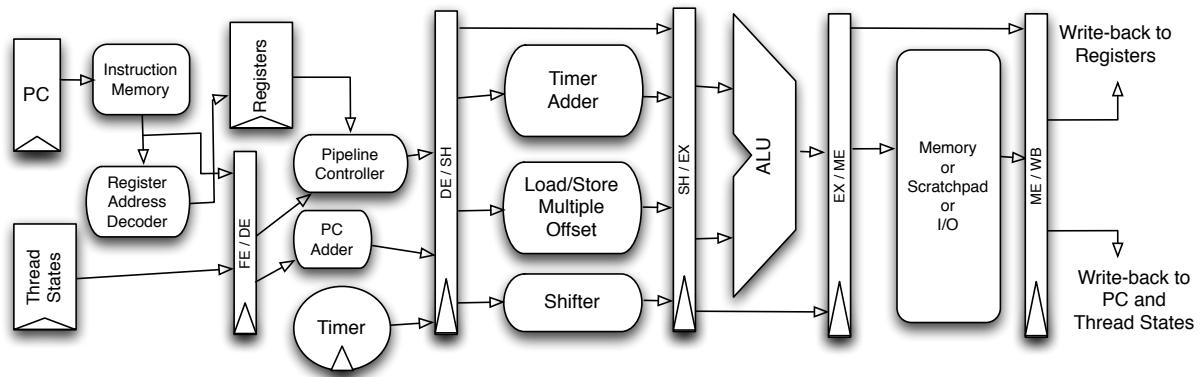


Figure 3.1: Block Level View of the PTARM 5 stage pipeline

### 3.1 PTARM Architecture

The pipeline of PTARM implements a thread-interleaved pipeline for the ARM instruction set. PTARM is written in VHDL and targets Xilinx Virtex-5 Family FPGAs, thus the design decisions made tailored the PTARM architecture towards optimizing the design for the Xilinx V5 FPGA. It has a 32 bit datapath and consists of a five stage pipeline with four threads interleaving through the pipeline. In chapter 2 we discussed the timing and hardware benefits of a typical thread-interleaved pipeline that has at least as many threads as pipeline stages. However, have more threads in the pipeline means decreased performance for each thread, since the more threads are utilizing the same pipeline resources. Thus, we design our pipeline to support a minimum of one less thread than the number of pipeline stages, and create a four thread five stage thread interleaved pipeline. As discussed in chapter 2, interleaving  $N$  threads through the an  $N$  stage pipeline helps remove data and control hazards in the pipeline. With  $N - 1$  threads interleaving through the  $N$  stage pipeline, data hazards are removed because the data committed at the last stage is not used for the next thread until the decode stage of the pipeline. However, control hazards are present if the PC is committed at the writeback stage, because the PC needed for the thread in the fetch stage has not been committed yet. In most cases, the next PC can be determined as early as the decode stage (data-processing and load/store instructions). For branch instructions, the address calculation is completed after the execute stage.

Figure 3.1 shows a block diagram view of the pipeline. The 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, which are not shown in the figure. 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.

The fetch stage of the pipeline selects the correct PC according to which thread is executing, and passes the address to instruction scratchpad. A simple  $\log(n)$  bit upcounter is used to keep track of which thread current to fetch. At this point we assume the source code fits entirely on the instruction scratchpad. Later when we present the memory hierarchy of PTARM we will discuss further about the implications of larger code bases. Once the instruction is received from the scratchpad, it goes through a register address decoder to quickly determine what bits to send to



the register. Typical RISC instruction sets such as MIPS have encoding of instruction bits where the register operands have a fixed location for all instruction types. Thus, once the instruction is received from instruction memory, the selected bits can be propagated as addresses to the register file. In the ARM instruction set however, not all instruction encoding have the register addresses at the same location. Thus, we insert in a simple and small logic block for a quick decoding of register addresses.

The decode stage of the pipeline consists of the pipeline controller which does the full decoding of instructions and sets the correct pipeline signals to be propagated down the pipeline. Typically the controller needs to know the current instructions in the pipeline to detect the possibility of pipeline hazards. However, in a thread-interleaved pipeline, other instructions in the pipeline belong to other 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. These signals get propagated down the pipeline stages, and they control the execution units to ensure the correct actions are taken, and signal the multiplexers to select the correct data operands. Because most of ARM instructions are conditionally executed, the pipeline controller also checks the condition bits to determine whether the instruction is to be executed or not. The PC Adder is almost just a simple adder that increments the PC. The only addition is instead only outputting an address incremented by four, it outputs two addresses, the PC incremented by both four and eight. The address calculation of ARM branch instructions add the offset to PC+8, instead of just the current PC, so we need to store the additional PC offset of eight in case of branch instructions. The Timer logic block is a hardware counter clocked to the processor clock which is used to implement the timing instructions mentioned in chapter 2.3. The Timer counts time in nanoseconds, and its hardware logic does not reside in the decode stage. The time value however is latched in the decode stage as the subsequent stages use it for timer manipulation, so in figure 3.1 we include it in the decode stage. We will discuss in more detail how the timing instructions are implemented later in this chapter.

The shift stage of the pipeline is the additional stage on top of a conventional five stage pipeline. The ARM ISA data-processing instructions include shifter bits and even a shifter operand for data-processing register shift instructions to shift the operands before the logical or arithmetic operations. Thus, an extra 32 bit shifter is included to shift the operands before the execution stage. 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. We will later describe in detail how the load/store multiple instructions are executed within the pipeline. The timer adder logic block is a 32 bit add/subtract unit. Since time is represented in 64 bit values, we add an additional 32 bit add/subtracter so the execution stage doesn't need to incorporate a 64 bit ALU.

The execute stage simply just contains the 32 bit ALU that can do both logical and arithmetic operations. The memory stage interacts with the data scratchpad and memory controller, and the write back stage writes the correct data back to the registers and hardware thread states.

The floating point hardware units are pipelined floating point computation units generated

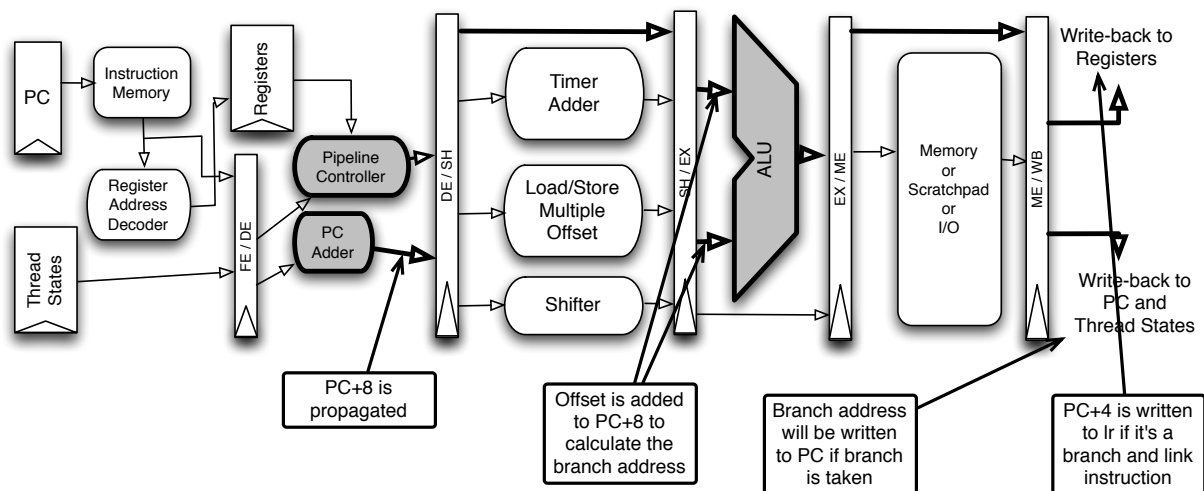


Figure 3.2: Branch Instruction Execution in the Pfarm Pipeline

with the Xilinx Coregen tool.

## 3.2 Instruction Implementations

In this section we will go into more details about how each instruction type is implemented and how the hardware blocks shown in the previous sections are used. We will go through different instruction types and also discuss the timing implications of our implementation. We will summarize with a table with all instructions and the cycle count it takes to execute them.

### 3.2.1 Data-Processing

#### 3.2.2 Branch

Branch instructions in the ARM can conditionally branch forward or backwards by up to 32MB. Because the general purpose register 15 is the PC, any write to R15 is also considered a branch instruction that branches to the address written into R15. Figure 3.2 show how the branch instruction is executed in the thread-interleaved pipeline. The hardware units involved are highlighted in gray and the lines are in bold. The branch instructions for the ARM ISA calculate the address based upon an offset added to PC incremented by 8. Thus, the PC adder, in addition to incrementing the PC by 4, also increments the PC by 8 to pass to the ALU for correct address calculation. Once the address is calculated, it is written back to the PC for the corresponding thread. If the instruction is a branch and link (*bl*), PC+4 will be written back to the link register (R14). In the case of a conditional branch, the PC is only written back if the condition holds true, or else PC+4 is written back to the PC. Because of the thread interleaved pipeline, we are not concerned that the pipeline will be stalled waiting for result of the branch to execute the next instruction. Instead, instructions from other threads will be fetched before the results of the branch is needed. Thus, we can write back to the PC at the end of the pipeline without creating hazards.

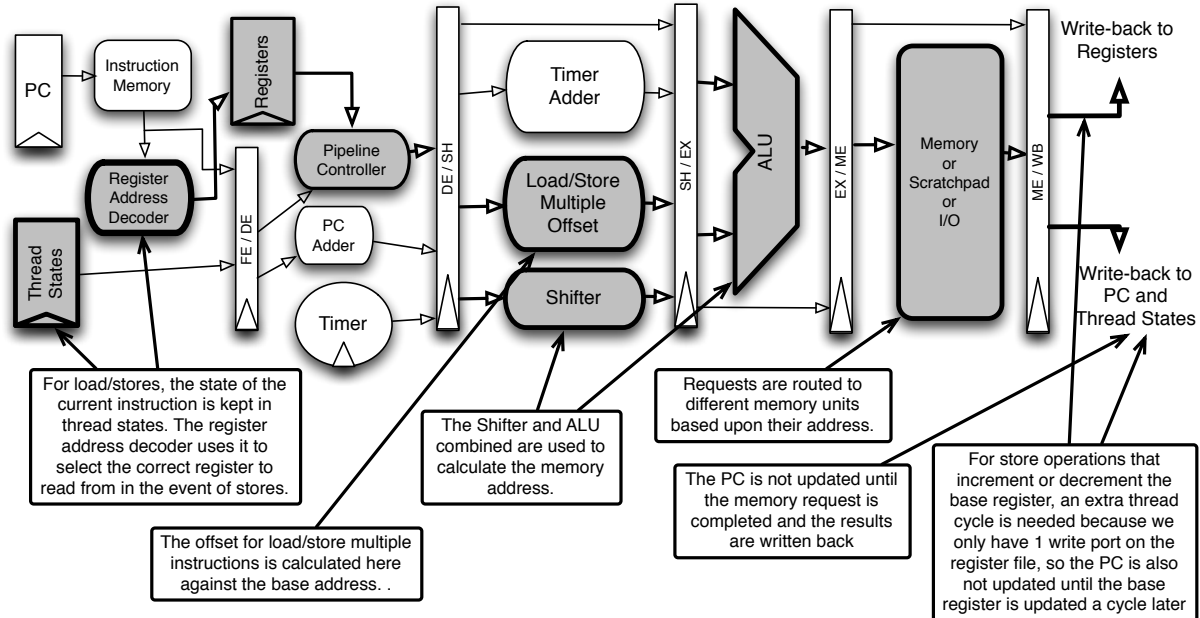


Figure 3.3: Load/Store Instruction Execution in the Ptarm Pipeline

### 3.2.3 Loads and Stores

For load and store instructions, the memory access address is formed by combining a base register and an offset value. ARM also specifies 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. ARM calls these pre-indexed and post-indexed addressing modes. 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. The base register could be incremented or decremented when it is being updated. Because there is only one write port to our register file however, we cannot simultaneously write back a load result from memory and the updated base register to the register file in a single cycle. As a result,

### 3.2.4 Timing Instructions

### 3.2.5 Synthesis Results

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

## 3.3 PTARM Simulator

## 3.4 Worst Case Execution Time Analysis

## Chapter 4

# Related Work

### 4.1 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 Related Section Header 2

Here is another header

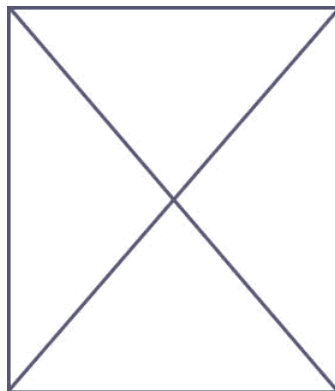


Figure 4.1: Image Placeholder

## 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, 26] 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, 21, 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 [18, 14], fault injection attacks [4, 11], and many others [26].

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, 25, 24]. 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, 25, 24] 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 [19, 26] 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 [17] 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 [17], 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 [22], which are a good fit for PRET’s embedded nature. We demonstrate the effectiveness of our approach by running both the RSA and DSA [20] 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

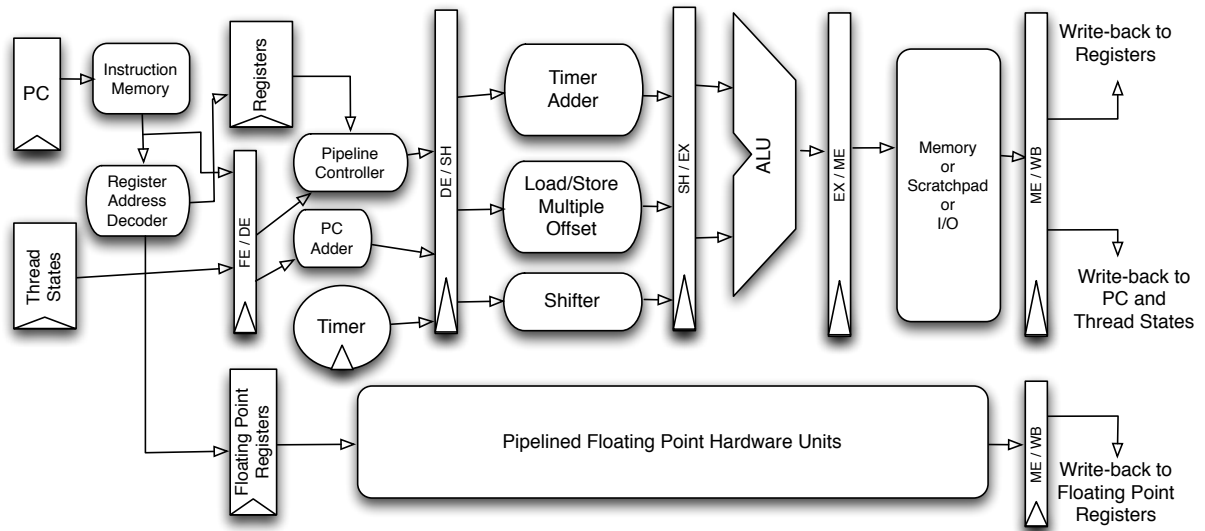


Figure 5.1: Block Level View of the PTARM 6 stage pipeline

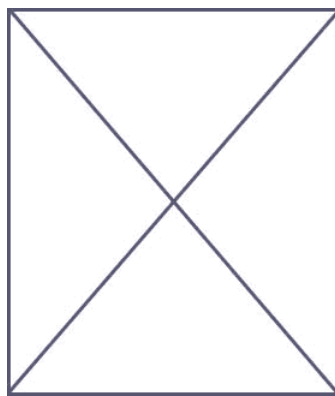


Figure 5.2: Image Placeholder

## **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



# Bibliography

- [1] O. Aciicmez, 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] 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.
- [18] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *USENIX Workshop on Smartcard Technology*, pages 151–162, 1999.
- [19] 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 *Cryptology ePrint Archive, Report 2005/368*, 2005.
- [20] National Institute of Standards and Technology. "Digital Signature Standard". Federal Information Processing Standards Publication 186, 1994.
- [21] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, page 05, 2005.
- [22] 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.
- [23] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35:29–63, March 2003.
- [24] 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.
- [25] 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.
- [26] Yongbin. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.