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

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
- show the variability different architecture improvements have introduced to improve average case execution time (Sami's graph)

With designs being pushed to higher and higher levels of abstraction, we need lower levels to provide robust, non brittle fundamentals in which we can reason about timing guarantees.

1.2 Intro Section Header 2

Here is another header
Talk about timing anoma

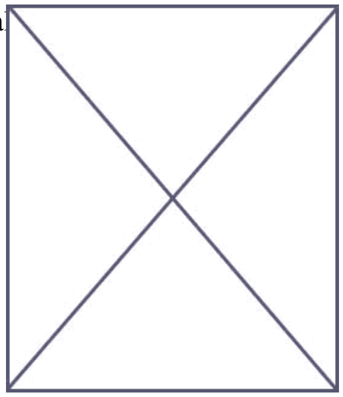


Figure 1.1: Image Placeholder

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,

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

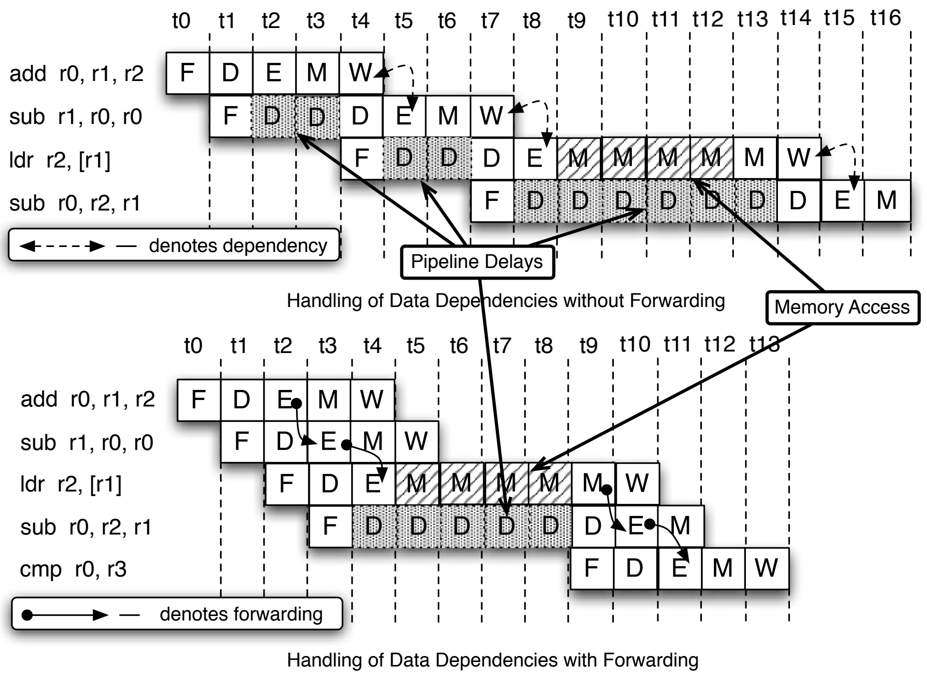

t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16
add r0, r1, r2
sub r1, r0, r0
ldr r2, [r1]
sub r0, r2, r1
cmp r0, r3
F D E M W
denotes dependency
Pipeline Delays
Memory Access
Handling of Data Dependencies without Forwarding
denotes forwarding
Handling of Data Dependencies with Forwarding

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).

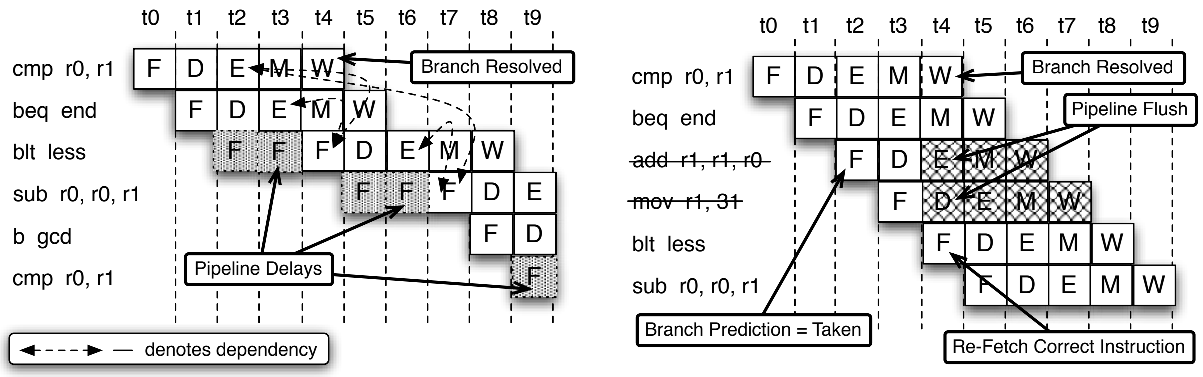
A simple but effective way of handling data hazards is by simply stalling the pipeline until the previous instruction completes. This is shown in the top of figure 2.2. Pipeline delays (or bubbles) are inserted for instructions to wait until the previous instruction is complete. The dependencies between 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



GCD without branch prediction   GCD with branch prediction (mispredict)

Figure 2.4: Handling of conditional branches in single threaded pipelines

instructions in ARM branch based on condi-
tional 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

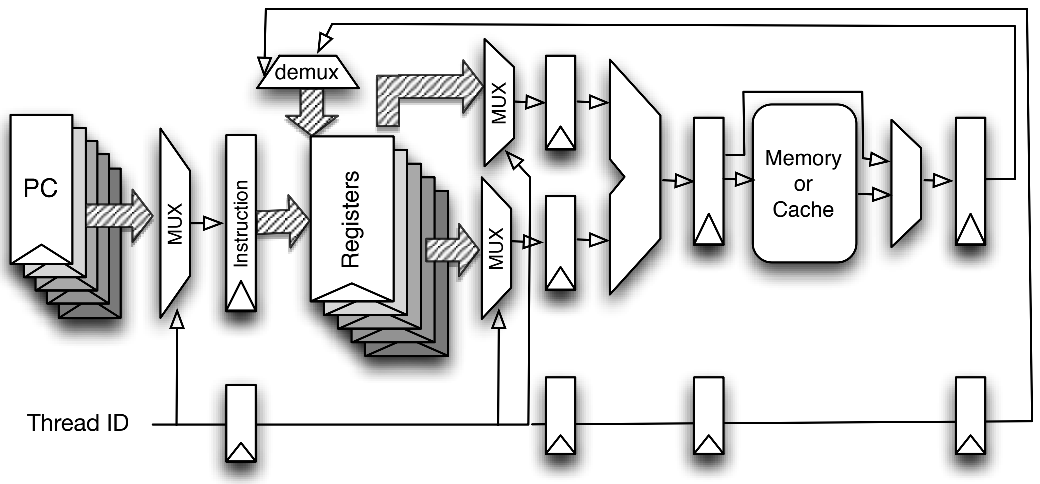


Figure 2.5: Simple Multithreaded Pipeline

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. [48] 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.

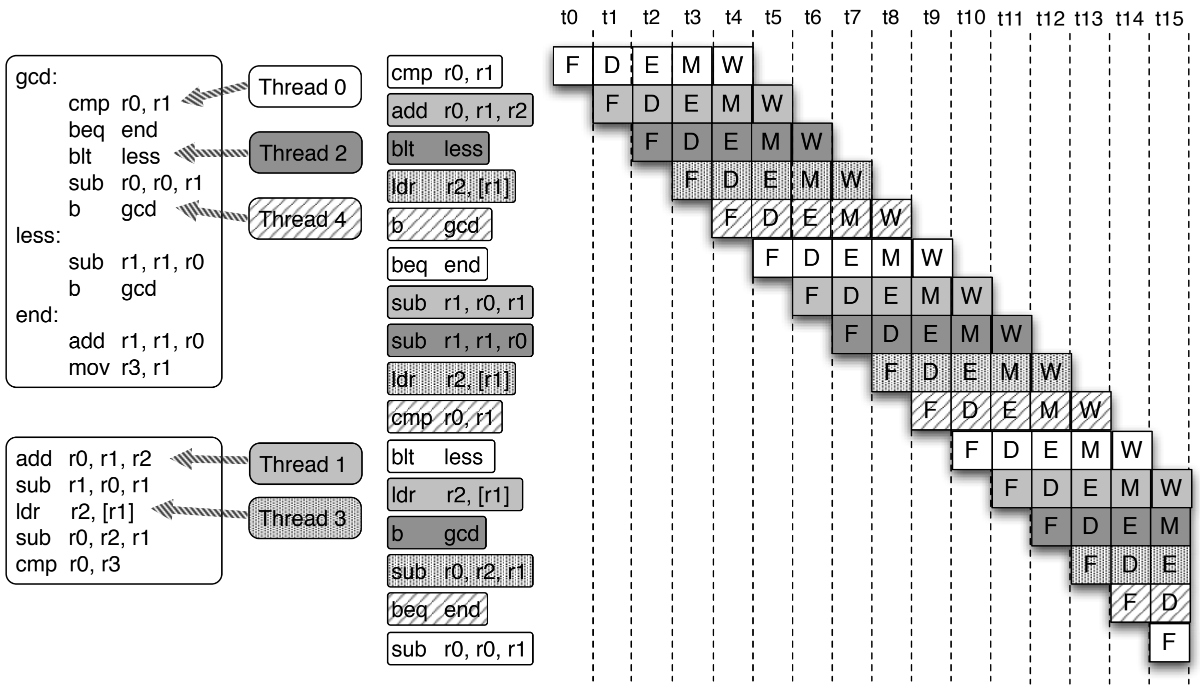


Figure 2.6: Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5 pipeline stages

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

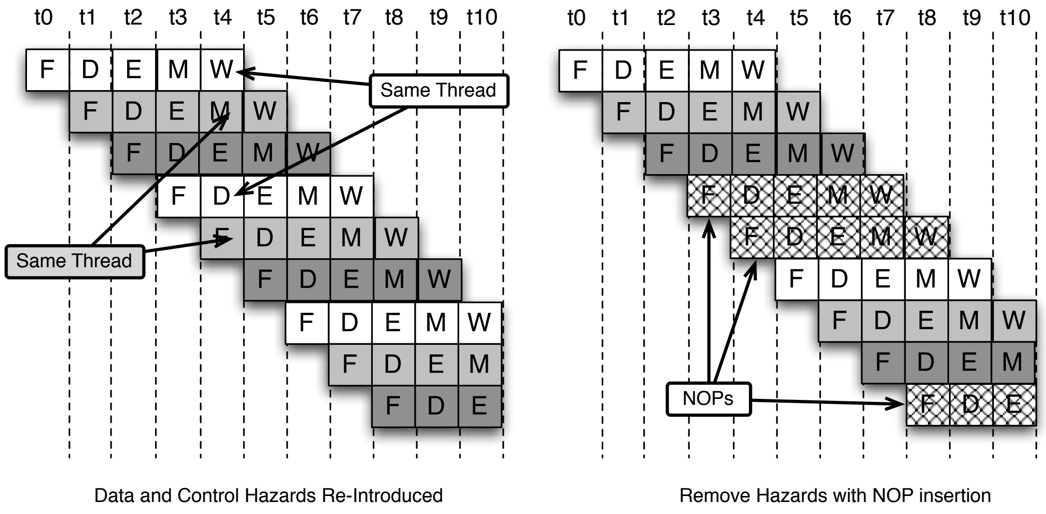

t0
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
F
D
E
M
W
Same Thread
Same Thread
NOPs
Data and Control Hazards Re-Introduced
Remove Hazards with NOP insertion

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

While pipelines designs continue to improve, memory technology has been struggling to keep up with the increase in clock speed and performance. Even though memory bandwidth can be improved with more bank parallelization, the memory latency remains the bottle neck to really improving memory performance. Common memory technologies used in embedded systems contain a significant trade off between access latency and capacity. Static Random-Access Memories (SRAM) provides sufficient latency that allows single cycle memory access latencies from the pipeline. However, the hardware cost to implement each memory cell prohibits large capacities to be implemented close to the processor. On the other hand, Dynamic Random-Access Memories (DRAM) uses a compact memory cell design that can easily be designed into larger capacity memory blocks. But the memory cell of DRAMs must be constantly refreshed due to charge leakage, and the large capacity of DRAM cells in a memory block design often prohibit faster access latencies. To bridge the latency gap between the pipeline and memory, smaller memories are placed in between the pipeline and larger memories to act as a buffer, forming a memory hierarchy. The smaller memories give faster access latencies at the cost of lower capacity, while larger memories make up for that with larger capacity but slower access latencies. The goal is to speed up program performance by placing commonly accessed values closer to the pipeline, while less access values are placed farther away.

2.2.1 Caches vs Scratchpads

A *CPU Cache* (or cache) is commonly used in the memory hierarchy to manage the smaller fast access memory made of SRAMs. The cache manages of contents of the fast access memory in hardware by leveraging the spatial and temporal locality of data accesses. The main benefits of the cache is that it abstracts away the memory hierarchy from the programmer. When a cache is used, all memory accesses is routed through the cache. If the data from the memory access is on the cache, then a cache hit occurs, and the data is returned right away. However, if data is not on the cache, then a cache miss occurs, and the cache controller fetches the data from the larger memory and adjusts the memory contents on the cache. The replacement policy of the cache is used to determine which cache line, the unit of memory replacement on caches, to replace. A variety of cache replacement policies have been researched (Todo: cite) and used to optimize for memory access patterns of different applications. In fact, modern memory hierarchies often contain multiple layers of hierarchy to balance the trade-off between speed and capacity. A commonly used memory hierarchy is shown in figure 2.8. If the data value is not found in the L1 cache, then it is searched for in the L2 cache. If the L2 cache also misses, then the data is retrieved from main memory, and sent back to the CPU while the L1 and L2 cache updates its contents. Often times different replacement policies are used at different levels of the memory hierarchy to optimize the hit rate or miss latency of the memory access. Benchmarks have show that caches can have hit rates up to (Todo: find number)% (Todo: and cite).

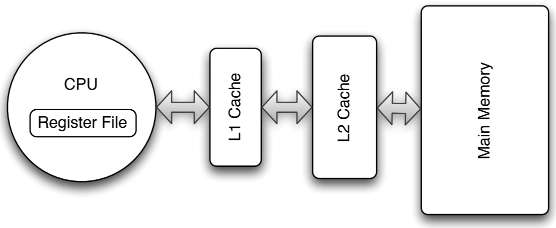


Figure 2.8: Memory Hierarchy w/ Caches

As sophisticated as this seems, the program is oblivious to the different levels of memory hierarchy, and whether or not an access hits the cache or goes all the way out to main memory. The memory hierarchy is abstracted away from the program, and the cache gives its best-effort to optimize memory access latencies. This is one of the main reasons for the cache's popularity; the programmer does not need to put in extra effort to get a reasonable amount of performance. A program can be run on any memory hierarchy configuration without modification and still obtain reasonable performance from the hardware. Thus, for general purpose applications, caches give the ability to improve design times and decrease design effort. However, the cache makes no guarantees on actual memory access latencies and program performance. The execution time of programs could highly vary depending on a number different factors, cold starts, the execution context previously running, interrupt routines, and even branch mispredictions that cause unnecessary cache line replacements. Thus, when execution time is important, the variability and uncontrollability of caches may outweigh the benefits it provides.

The cache uses internal states stored in hardware to manage the replacement of contents on it. As the programmer cannot control the states of the cache explicitly, it is extremely difficult to analyze the execution time of a program running with caches. At an arbitrary point in the program, the state of the cache is unknown to the software. Whether a memory access hits or misses the cache cannot to determined easily, so the conservative worst-case execution time analysis well need to assume the worst case as if the memory access was directly to main memory. In theory, (Todo: Cite and summarize Jan's thesis on cache analysis) showed that for certain cache replacement policies, it is possible to obtain tighter execution time analysis. However, the complexity of modern memory hierarchies with caches make timing analysis extremely difficult, and introduce high variability in program execution time.

Even outside of real-time applications, caches present side effects that it were not intended for. For applications that require extremely high speed, the best-effort memory management that caches offer simply is not good enough. In those situations, programs need to be hand tuned and tailored to specific cache architectures and parameters. Algorithm designers tune the performance of algorithms by reaching beneath the abstracted away memory architecture to enforce data access patterns to conform to the cache replacement policies and cache line sizes. Blocking [26], for example, is a well-known technique to optimize algorithms for high performance. Instead of operating on entire rows or columns of an array, algorithms are rewritten to operate on a subset of the data at a time, or blocks, so the faster memory in the hierarchy can be reused. (Todo: talk about LAPACK? Libraries that tune programs to caching). In this case, we see that the hidden memory hierarchy actually could degrade program performance. Multithreaded threaded architectures with shared caches amongst the hardware threads can suffer from *cache thrashing*, an effect where different threads' memory accesses evict the cached lines of others. In this situation, it is impossible for hardware threads have any knowledge on the state of the cache, because it is simultaneously being modified by other threads in the system. As a result, the hardware threads have no control over which level in the memory hierarchy they are accessing, and the performance highly varies depending on what is running on other hardware threads. For multicore architectures, caches create a data coherency problem when keeping data consistent between the multiple cores. When the multiple cores are sharing memory, each core's private cache may cache the same memory address. If one core writes to a memory location that is cached in its private cache, then the other core's cache would contain stale data. Various methods such as bus snooping(Todo: cite) or implementing a directory protocol(Todo: cite) have been proposed to keep the data consistent in all caches. However, doing this scalably and efficiently is still a hot topic of research today(Todo: cite all work on cache coherency).

We cannot argue against the need for a memory hierarchy, as there is an undeniable gap between processor and DRAM latency. However, instead of abstracting away the memory hierarchy from the programmer, we propose to expose the memory layout to the software. *Scratchpads* were initially proposed for their power saving benefits over caches(Todo: cite). (Todo: Papers that show scratchpads were already popular? Mention that they are already used. The Cell processor, Nvidia's 8800 GPU) Scratchpads uses the same memory technology as caches, but does not implement the hardware controller to manage its memory contents. Scratchpads have reduced access latency, area and power consumption compared to caches. Memory operations that access the scratchpad region take only a single cycle to complete, which is the same as a cache hit. Thus, scratchpads may serve as the fast-access memory in a memory hierarchy, instead of caches. Unlike caches that overlay address space with main memory, scratchpads occupy a distinct address space in memory, so it does not need to check whether the data is on the scratchpad or not. Furthermore, the memory access time of each memory request is only depend on the memory address it is accessing. This drastically improves the predictability of memory access times, and reduces the variability of execution time introduced with caches. By using scratchpads, we explicitly expose the memory hierarchy to software, giving the programmer full control over the management of memory contents within the memory hierarchy.

Two allocation schemes are commonly employed to manage the contents of scratchpads in software. Static allocation schemes allocate data on the scratchpad during compile time, and the contents allocated on the scratchpad does not change throughout program execution. Static scratchpad allocation schemes [44, 37] often use heuristics or compiler-based static analysis(Todo: citation) of program to find the most commonly executed instructions or data structures, and allocate them statically on the scratchpad to improve program performance. Dynamic allocation schemes modify the data on the scratchpad during run time in software through DMA mechanisms. The allocation could either be automatically generated and inserted by the compiler, or explicitly specified by the user programmatically. Embedded system designs typically deal with limited resources and other design constraints, such as less memory or hard timing deadlines. Thus, the design of these systems often contain analysis on memory usage etc to ensure that the constraints are met. Actor oriented models of computations, such as Dataflow(Todo: cite) or Giotto(Todo: cite), allow users to design systems at a higher level. These higher level actor oriented programming models exposes the structure and semantics of the model for better analysis, which can be used to optimize scratchpad allocation dynamically. Bandyopadhyay [5] presented an automated memory allocation of scratchpads for the execution of Heterochronous Dataflow models. The Heterochronous Dataflow (HDF) model is an extension to the Synchronous Dataflow (SDF) model with finite state machines (FCM). The HDF models contain different program states, each state executing a SDF model that contains actors communicating with each other. Bandyopadhyay analyzed the actor code and the data that was being communicated in each HDF state. The dynamic scratchpad allocation is inserted during state transitions, and the memory allocated is optimized for each HDF state. This allocation not only showed roughly 17% performance improvement compared to executions using LRU caches, but also more predictable program performance.

The underlying memory technology that is used to make both scratchpads and caches is not inherently unpredictable, as SRAMs provide constant low-latency access time. However, caches manage the contents of the SRAM in hardware. By using caches in the memory hierarchy, the hierarchy is hidden from the programmer, and hardware managed memory contents creates highly variable execution times with unpredictable access latencies. Scratchpads on the other hand exposes the memory hierarchy to the programmer, allowing more predictable and repeatable memory access performances. Although the allocation of scratchpads could be challenging, but it also provides opportunity for high efficiency, as it can be tailored to specific applications.

2.2.2 DRAM Memory Controller

Main memory requests that access the DRAM present a different challenge to obtaining predictable access times. In this section we will present a DRAM memory controller designed to achieve predictable memory accesses. The contributions from this section is research done jointly with the several co-authors from [41]. We do not claim sole credit for this work, and it is included in this thesis for completeness purposes. We will first give some basic background on DRAM memories, then present the predictable DRAM controller designed.

DRAM Basics

We present the basics of DRAM, and the structure of modern DRAM modules. Inherent properties of DRAM and the structure of DRAM modules impose several timing constraints, which all memory controllers must obey. For more details on different DRAM standards, we refer the reader to Jacob et al. [19].

Dynamic RAM Cell A DRAM cell consists of a capacitor and a transistor as shown in 2.9. The charge of the capacitor determines the value of the bit, and by triggering the transistor, one can access the capacitor. However, the capacitor leaks charge over time, and thus, it must be refreshed periodically. According to the JEDEC [20] capacitors must be refreshed every 64 ms or less.

DRAM Array A DRAM array contains a two-dimensional array structure of DRAM cells. Accesses to a DRAM array proceed in two phases: row accesses followed by one or more column accesses. A row access moves one of the rows of the DRAM array into the row buffer. As the capacitance of the capacitors in the DRAM cells is low compared with that of the wires connecting them to the row buffer, sense amplifiers are needed to read out their values. In order for the sense amplifiers to read out the value of a DRAM cell, the wires need to be precharged close to the voltage threshold between 0 and 1. Once a row is in the row buffer, columns can be read from and written to quickly. Columns are small sets of consecutive bits within a row.

DRAM Devices DRAM arrays form banks in a DRAM device. Figure 2.9 illustrates a bank's structure, and the location of banks within DRAM devices. Modern DRAM devices have multiple banks, control logic, and I/O mechanisms to read from and write to the data bus as shown in the center of 2.9. Different banks within a device can be accessed concurrently. This is known as bank-level parallelism. However, the data, command, and address busses are shared among all banks, as is the I/O gating, connecting the banks to the data bus.

A DRAM device receives commands from its memory controller through the command and address busses. The following table lists the four most important commands and their function:

Command	Abbr.	Description
Precharge	PRE	Stores back the contents of the row buffer into the DRAM array, and prepares the sense amplifiers for the next row access.
Row access	RAS	Moves a row from the DRAM array through the sense amplifiers into the row buffer.
Column access	CAS	Overwrites a column in the row buffer or reads a column from the row buffer.
Refresh	REF	Refreshes several[1]rows of the DRAM array. This uses the internal refresh counter to determine which rows to refresh.

To read from or write to the DRAM device, the controller needs to first precharge (PRE) the bank containing the data that is to be read. It can then perform a row access (RAS = row access strobe), followed by one or more column accesses (CAS = column access strobe). Column accesses can be both reads and writes. For higher throughput, column accesses are performed in bursts. The length of these bursts is usually configurable to four or eight words. In a x16-device, columns consist of 16 bits. A burst of length four will thus result in a transfer of 64 bits. To decrease the latency between accesses to different rows, column accesses can be immediately followed by precharge operations, which is known as auto-precharge (aka closed-page policy).

There are two ways of refreshing DRAM cells within the 64 ms timing constraint:

1. Issue a refresh command. This refreshes all banks of the device simultaneously. The DRAM device maintains a refresh counter to step through all of the rows. To refresh every row and thus every DRAM cell every 64 ms, the memory controller has to issue at least 8192 refresh commands in every interval of 64 ms. Earlier devices had exactly 8192 rows per bank. However, recent higher-density devices have up to 65536 rows per bank. As a consequence, for such devices, a refresh command will refresh several rows in each bank, increasing refresh latency considerably.

2. Manually refresh rows. The memory controller performs a row access on every row in every bank every 64 ms. This forces the memory controller to issue more commands, and it requires a refresh counter outside of the memory device. Each refresh takes less time because it only accesses one row, but refreshes have to be issued more frequently.

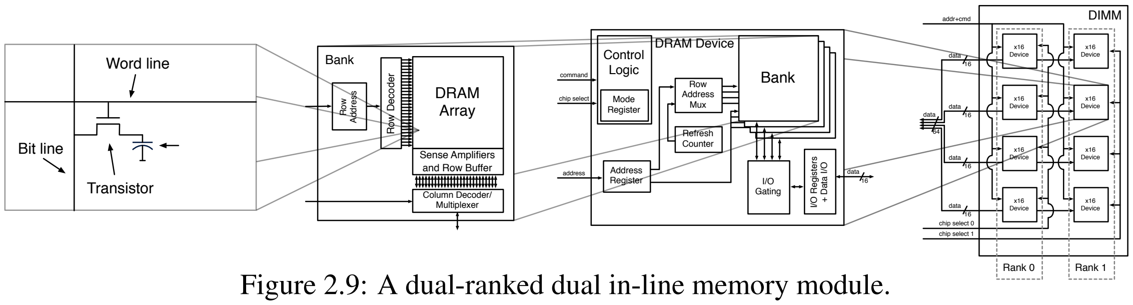


Figure 2.9: A dual-ranked dual in-line memory module.

DRAM Modules To achieve greater capacity and bandwidth, several DRAM devices are integrated on a memory module. The right side of 2.9 depicts a high-level view of the dual-ranked dual in-line memory module (DIMM) that the PRET DRAM controller uses. The DIMM has eight DRAM devices that are organized in two ranks of four x16 DRAM devices each. The two ranks share the address and command inputs, and the 64-bit data bus. The chip select input determines which of the two ranks is addressed. DRAM devices within a rank operate in lockstep: they receive the same address and command inputs, and read from or write to the data bus at the same time.

Logically, the four x16 DRAM devices that comprise a rank can be viewed as one x64 DRAM device. This is how we view them for the remainder of this paper. When referring to a bank i in one of the two ranks, we are referring to bank i in each of the four x16 DRAM devices that comprise that rank. A burst of length four results in a transfer of $4 \cdot 16 \cdot 4 = 256$ bits $= 32$ bytes.

[1]The number of rows depends on the capacity of the device.

Table 2.1: Overview of DDR2-400 timing parameters at the example of the Qimonda HYS64T64020EM-2.5-B2.

Parameter	Value (in cycles at 200 MHz)	Description
t_{RCD}	3	Row-to-Column delay: time from row activation to first read or write to a column within that row.
t_{CL}	3	Column latency: time between a column access command and the start of data being returned.
t_{WL}	$t_{CL} - 1 = 2$	Write latency: time after write command until first data is available on the bus.
t_{WR}	3	Write recovery time: time between the end of a write data burst and the start of a precharge command.
t_{WTR}	2	Write to read time: time between the end of a write data burst and the start of a column-read command.
t_{RP}	3	Time to precharge the DRAM array before next row activation.
t_{RFC}	21	Refresh cycle time: time interval between a refresh command and a row activation.
t_{FAW}	10	Four-bank activation window: interval in which maximally four banks may be activated.
t_{AL}	set by user	Additive latency: determines how long posted column accesses are delayed.

Due to the sharing of I/O mechanisms within a device, consecutive accesses to the same rank are more constrained than consecutive accesses to different ranks, which only share the command and address as well as the data bus. We later exploit this subtle difference by restricting consecutive accesses to different ranks to achieve more predictable access latencies. Our controller makes use of a feature from the DDR2 standard known as posted-CAS. Unlike DDR or other previous versions of DRAMs, DDR2 can delay the execution of CAS commands (posted-CAS). After receiving a posted-CAS, DDR2 waits for a user-defined latency, known as the additive latency AL, until sending the CAS to the column decoder. Posted-CAS can be used to resolve command bus contention by sending the posted-CAS earlier than the corresponding CAS needs to be executed. We explain this in more detail in 2.2.2.

DDR2 Timing Constraints The internal structure of DRAM modules described above as well as properties of DRAM cells incur a number of timing constraints, which DRAM controllers have to obey. 2.1 gives an overview of timing parameters for a DDR2-400 memory module and brief explanations. These parameters constrain the placement of commands to be send to a DDR2 module. Some of the constraints (t_{RCD}, t_{RP}, t_{RFC}) are solely due to the structure of DRAM banks, which are accessed through sense amplifiers that have to be precharged. Others result from the structure of DRAM banks and DRAM devices: $t_{CL}, t_{WR}, t_{WTR}, t_{WL}$. The four-bank activation window constraint t_{FAW} constrains rapid activation of multiple banks which would result in too high a current draw. The additive latency, t_{AL}, can be set by the user and determines how many cycles after a posted-CAS a CAS is executed.

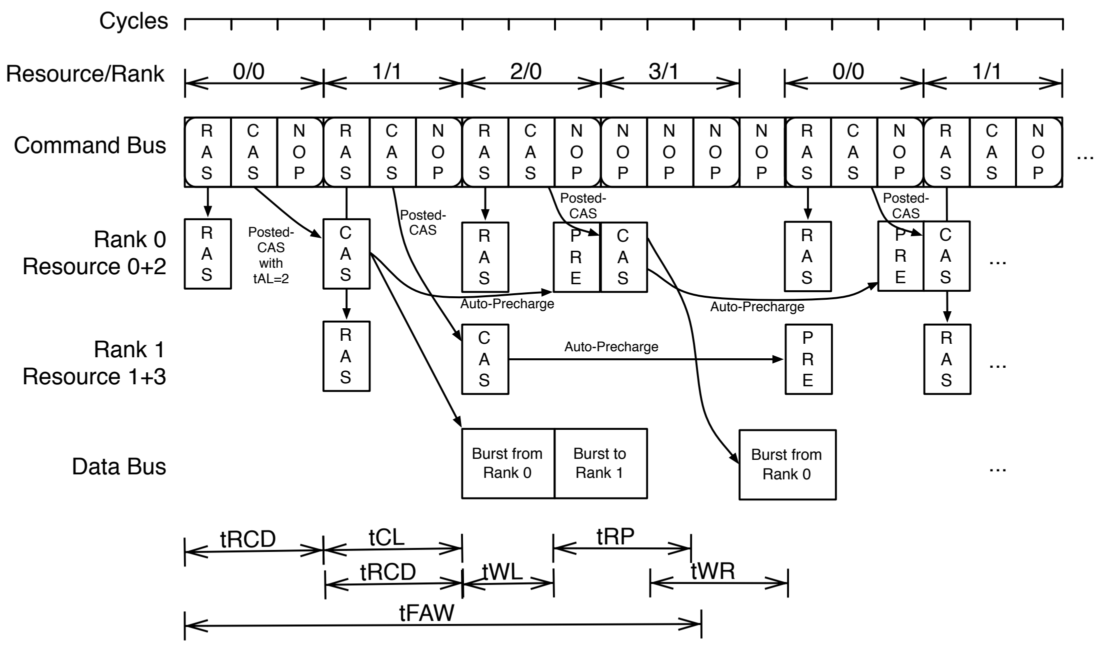


Figure 2.10: The periodic and pipelined access scheme employed by the backend. In the example, we perform a read from resource 0 (in rank 0), a write to resource 1 (in rank 1), and a read from resource 2 (in rank 0).

Predictable DRAM Controller Backend

The backend views the memory device as four independent resources: each resource consisting of two banks within the same rank. By issuing commands to the independent resources in a periodic and pipelined fashion, we exploit bank parallelism and remove interference amongst the resources. This is unlike conventional DRAM controllers that view the entire memory device as one resource. Other partitions of the eight banks would be possible, as long as all of the banks that are part of a resource belong to the same rank of the memory module, and each of the two ranks contains two resources.

2.10 shows an example of the following access requests from the frontend: read from resource 0 in rank 0, write to resource 1 in rank 1, and read from resource 2 in rank 0. The controller periodically provides access to the four resources every 13 cycles. In doing so, we exploit bank parallelism for high bandwidth, yet, we avert access patterns that otherwise incur high latency due to the sharing of resources within banks and ranks.

The backend translates each access request into a row access command (RAS), a posted column access command (posted-CAS) or a NOP. We refer to a triple of RAS, CAS and NOP as an access slot. In order to meet row to column latency shown in 2.1, the RAS command and the first CAS command need to be 3 cycles apart. However, we can see from 2.10 that if we waited for 3 cycles before issuing the CAS to access the first resource, it would conflict with the RAS command for accessing the second resource on the command bus. Instead, we set the additive latency t_{AL}

to 2. This way, the posted-CAS results in a CAS two cycles later within the DRAM chip. This is shown in 2.10 as the posted-CAS appears within its rank 2 cycles after the CAS was issued on the command bus, preserving the pipelined access scheme.

The row access command moves a row into the row buffer. The column access command can be either a read or a write, causing a burst transfer of $8 \cdot 4 = 32$ bytes, which will occupy the data bus for two cycles (as two transfers occur in every cycle). We use a closed-page policy (also known as auto-precharge policy), which causes the accessed row to be immediately precharged after performing the column access (CAS), preparing it for the next row access. If there are no requests for a resource, the backend does not send any commands to the memory module, as is the case for resource 3 in 2.10.

There is a one cycle offset between the read and write latencies. Given that requests may alternate between reads and writes, the controller inserts a NOP between any two consecutive requests. This avoids a collision on the data bus between reads and writes. By alternating between ranks, no two adjacent accesses go to the same rank. This satisfies the write-to-read timing constraint t_{WTR} incurred by the sharing of I/O gating within ranks. In addition, we satisfy the four-bank activation window constraint because within any window of size t_{FAW} we activate at most four banks due to the periodic access scheme.

With the closed-page policy, in case of a write, we need 13 cycles to access the row, perform a burst access, and precharge the bank to prepare for the next row access. This is the reason for adding a NOP after four access slots: to increase the distance between two access slots belonging to the same resource from 12 to 13 cycles. The backend does not issue any refresh commands to the memory module. Instead, it relies on the frontend to refresh the DRAM cells using regular row accesses.

Longer Bursts for Improved Bandwidth Depending on the application, bandwidth might be more important than latency. Bandwidth can be improved by increasing the burst length from 4 to 8. Extending the proposed access scheme to a burst length of 8 is straightforward with the insertion of two additional NOP commands after each request to account for the extra two cycles of data being transfered on the data bus. In this case, the access slot latency for each request is increased from three to five to include the extra two NOP commands, and data will be transferred in four out of five cycles rather than in two out of three. Then, of course, latency of transfers of size less than or equal to 32 bytes increases, but the latency of large transfers decreases and higher bandwidth is achieved.

DRAM Controller Frontend

In this section, we discuss our integration of the backend within the PTARM PRET architecture [**?**]. We also discuss how the PRET DRAM controller could be integrated into other predictable architectures, such as those proposed by the MERASA [**?**], PREDATOR [**?**], JOP [**?**], or CoMPSoC [**?**] projects, which require predictable and composable memory performance.

Integration with the PTARM Architecture

PTARM [**?**], a PRET machine [**?**], is a thread-interleaved implementation of the ARM instruction set. Thread-interleaved processors preserve the benefit of a multi-threaded architecture

– increased throughput, but use a predictable fine-grained thread-scheduling policy – round robin. If there is the same number of hardware threads as there are pipeline stages, at any point in time, each stage of the pipeline is occupied by a different hardware thread; there are no dependencies between pipeline stages, and the execution time of each hardware thread is independent of all others. PTARM has four pipeline stages and four hardware threads. Each hardware thread has access to an instruction scratchpad and a data scratchpad. The scratchpads provide single-cycle access latencies to the threads. The two scratchpads are shared among the four threads, allowing for shared memory communication among the threads. However, due to the thread-interleaving, only one thread can access the scratchpad at any time. Each hardware thread is also equipped with a direct memory access (DMA) unit, which can perform bulk transfers between the two scratchpads and the DRAM. Both scratchpads are dual-ported, allowing a DMA unit to access the scratchpads in the same cycles as its corresponding hardware thread. In our implementation of thread-interleaving, if one thread is stalled waiting for a memory access, the other threads are unaffected and continue to execute normally.

The four resources provided by the backend are a perfect match for the four hardware threads in the PTARM thread-interleaved pipeline. We assign exclusive access to one of the four resources to each thread. In contrast to conventional memory architectures, in which the processor interacts with DRAM only by filling and writing back cache lines, there are two ways the threads can interact with the DRAM in our design. First, threads can initiate DMA transfers to transfer bulk data to and from the scratchpad. Second, since the scratchpad and DRAM are assigned distinct memory regions, threads can also directly access the DRAM through load and store instructions.

Whenever a thread initiates a DMA transfer, it passes access to the DRAM to its DMA unit, which returns access once it has finished the transfer. During the time of the transfer, the thread can continue processing and accessing the two scratchpads. If at any point the thread tries to access the DRAM, it will be blocked until the DMA transfer has been completed. Similarly, accesses to the region of the scratchpad which are being transferred from or to will stall the hardware thread[2]. 2.11 shows a block diagram of PTARM including the PRET DRAM controller backend and the memory module. The purpose of the frontend is to route requests to the right request buffer in the backend and to insert a sufficient amount of refresh commands, which we will discuss in more detail.

When threads directly access the DRAM through load (read) and store (write) instructions, the memory requests are issued directly from the pipeline. **??**, which we will later use to derive the read latency, illustrates the stages of the execution of a read instruction in the pipeline. At the end of the memory stage, a request is put into the request buffer of the backend. Depending on the alignment of the pipeline and the backend, it takes a varying number of cycles until the backend generates corresponding commands to be sent to the DRAM module. After the read has been performed by the DRAM and has been put into the response buffer, again, depending on the alignment of the pipeline and the backend, it takes a varying number of cycles for the pipeline to reach the write-back stage of the corresponding hardware thread. Unlike the thread-interleaved pipeline, the DMA units are not pipelined, which implies that there are no “alignment losses”: the DMA units can fully utilize the bandwidth provided by the backend.

Store Buffer Stores are fundamentally different from loads in that a hardware thread does not have to wait until the store has been performed in memory. By adding a single-place store buffer to

[2]This does not affect the execution of any of the other hardware threads.

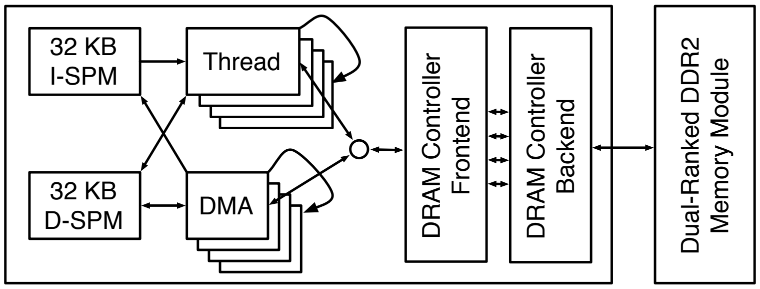


Figure 2.11: Integration of PTARM core with DMA units, PRET memory controller and dual-ranked DIMM.

the frontend, we can usually hide the store latency from the pipeline. Using the store buffer, stores which are not preceded by other stores can be performed in a single thread cycle. By *thread cycle*, we denote the time it takes for an instruction to pass through the thread-interleaved pipeline. Other stores may take two thread cycles to execute. A bigger store buffer would be able to hide latencies of successive stores at the expense of increased complexity in timing analysis.

Scheduling of Refreshes DRAM cells need to be refreshed at least every 64 ms. A refresh can either be performed by a hardware refresh command, which may refresh several rows of a device at once[3], or by performing individual row accesses "manually". We opt to do the latter. This has the advantage that a single row access takes less time than the execution of a hardware refresh command, thereby improving worst-case latency, particularly for small transfers. The disadvantage, on the other hand, is that more manual row accesses have to be performed, incurring a slight hit in bandwidth.

In our device, each bank consists of 8192 rows. Thus, a row has to be refreshed every $64ms/8192 = 7.8125\mu s$. At a clock rate of 200 MHz of the memory controller, this corresponds to $7.8125\mu s \cdot (200 cycles/\mu s) = 1562.5$ cycles. Since each resource contains two banks, we need to perform two refreshes every 1562.5 cycles, or one every 781.25 cycles. One round of access is 13 cycles at burst length 4, and includes the access slots to each resource plus a nop command. So we schedule a refresh every $\lfloor 781.25/13 \rfloor^{th} = 60^{th}$ round of the backend. This way, we are scheduling refreshes slightly more often than necessary. Scheduling a refresh every $60 \cdot 13$ cycles means that every row, and thus every DRAM cell, is refreshed every $60 \cdot 13\, cycles \cdot 8192 \cdot 2/(200000\ cycles/ms) \leq 63.90ms$. We are thus flexible to push back any of these refreshes individually by up to $0.1ms = 20000$ cycles without violating the refreshing requirement.

We make use of this flexibility for loads from the pipeline and when performing DMA transfers: if a load would coincide with a scheduled refresh, we push back the refresh to the next slot. Similarly, we skip the first refresh during a DMA transfer and schedule an additional one at the end of the transfer. This pushes back the refresh of a particular row by at most $60 \cdot 13$ cycles. More sophisticated schemes would be possible, however, we believe their benefit would be slim.

[3]Internally, this still results in several consecutive row accesses.

Following this approach, two latencies can be associated with a DMA transfer:

1. The time from initiating the DMA transfer until the data has been transferred, and is, e.g., available in the data scratchpad.
2. The time from initiating the DMA transfer until the thread-interleaved pipeline regains access to the DRAM.

Our conjecture is that latency 1 is usually more important than latency 2. Furthermore, our approach does not deteriorate latency 2. For loads sent from the pipeline, the pushed back refreshes become invisible: as the pipeline is waiting for the data to be returned and takes some time to reach the memory stage of the next instruction, it is not able to use successive access slots of the backend, and thus it is unable to observe the refresh at all. With this refresh scheme, refreshes do not affect the latencies of load/store instructions, and the refreshes scheduled within DMA transfers are predictable so the latency effects of the refresh can be easily analyzed.

Integration with Other Multi-Core Processors

Several recent projects strive do develop predictable multi-core architectures [**?**, **?**, **?**, **?**]. These could potentially profit from using the proposed DRAM controller. The frontend described in the previous section makes use of specific characteristics of the PTARM architecture. However, when integrating the backend with other architectures, we cannot rely on these characteristics. A particular challenge to address is that most multi-core processors use DRAM to share data, while local scratchpads or caches are private. This can be achieved by sharing the four resources provided by the backend within the frontend. A particularly simple approach would first combine the four resources into one: an access to the single resource would simply result in four smaller accesses to the resources of the backend. This single resource could then be shared among the different cores of a multi-core architecture using predictable arbitration mechanisms such as Round-Robin or CCSP [**?**] or predictable and composable ones like time-division multiple access (TDMA). However, sharing the DRAM resources comes at the cost of increased latency. We investigate this cost in **??**.

2.3 Programming Models

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 [28] 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

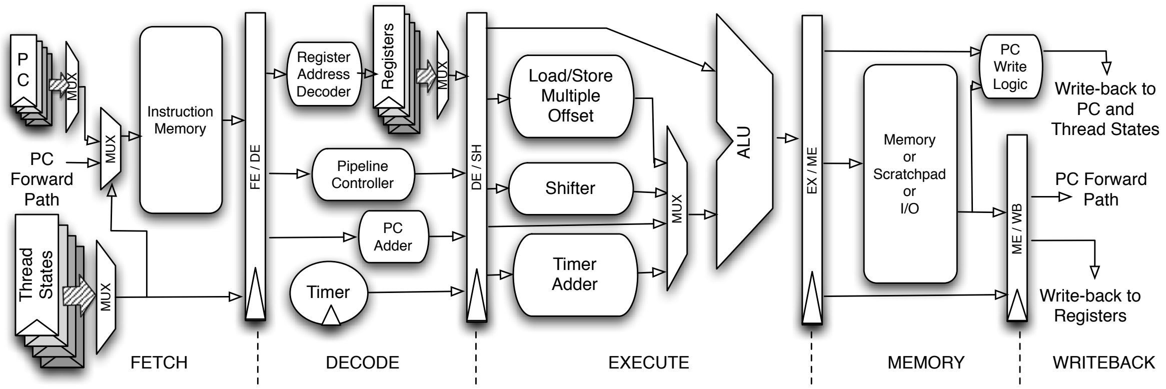


Figure 3.1: Block Level View of the PTARM 5 stage pipeline

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 **??**. 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.

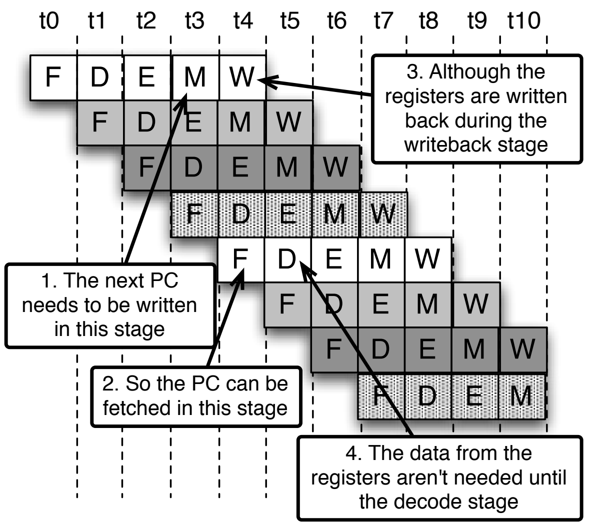

t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10
F D E M W
3. Although the registers are written back during the writeback stage
1. The next PC needs to be written in this stage
2. So the PC can be fetched in this stage
4. The data from the registers aren't needed until the decode stage

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.

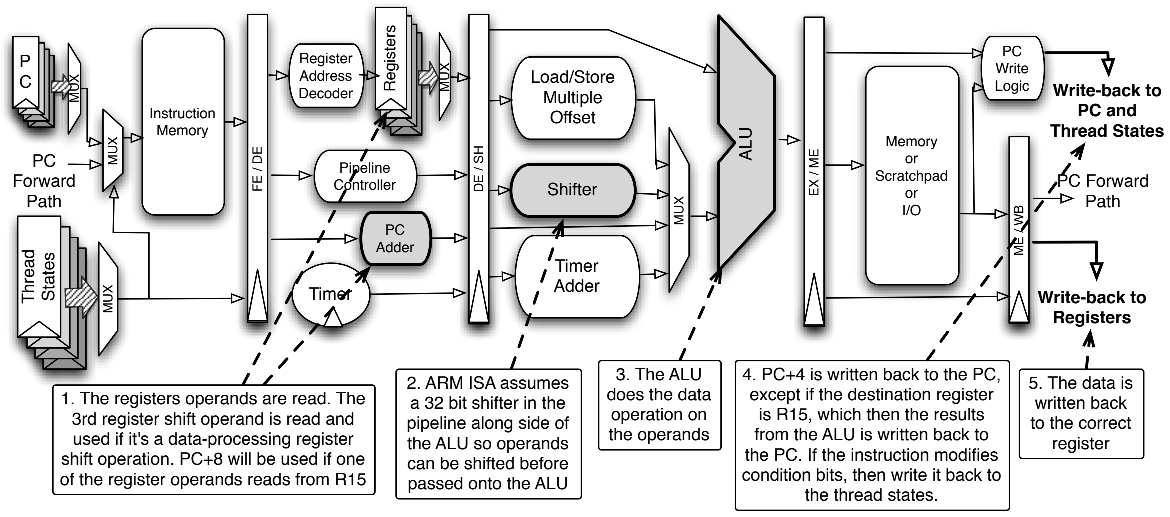

P
C
MUX
PC
Forward
Path
MUX
Instruction
Memory
Thread
States
MUX
FE / DE
Register
Address
Decoder
Registers
MUX
Pipeline
Controller
PC
Adder
Timer
DE / SH
Load/Store
Multiple
Offset
Shifter
Timer
Adder
MUX
ALU
EX / ME
Memory
or
Scratchpad
or
I/O
PC
Write
Logic
Write-back to
PC and
Thread States
PC Forward
Path
ME / WB
Write-back to
Registers
1. The registers operands are read. The 3rd register shift operand is read and used if it's a data-processing register shift operation. PC+8 will be used if one of the register operands reads from R15
2. ARM ISA assumes a 32 bit shifter in the pipeline along side of the ALU so operands can be shifted before passed onto the ALU
3. The ALU does the data operation on the operands
4. PC+4 is written back to the PC, except if the destination register is R15, which then the results from the ALU is written back to the PC. If the instruction modifies condition bits, then write it back to the thread states.
5. The data is written back to the correct register

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.

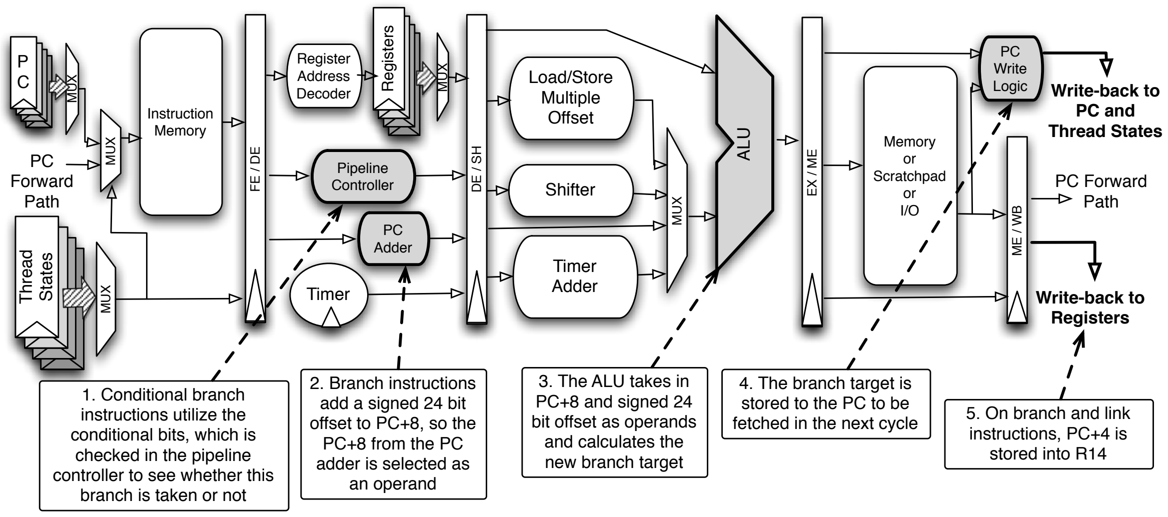

P
C
MUX
MUX
PC Forward Path
Thread States
MUX
Instruction Memory
FE / DE
Register Address Decoder
Registers
MUX
Pipeline Controller
PC Adder
Timer
DE / SH
Load/Store Multiple Offset
Shifter
Timer Adder
MUX
ALU
EX / ME
Memory or Scratchpad or I/O
PC Write Logic
Write-back to PC and Thread States
PC Forward Path
ME / WB
Write-back to Registers
1. Conditional branch instructions utilize the conditional bits, which is checked in the pipeline controller to see whether this branch is taken or not
2. Branch instructions add a signed 24 bit offset to PC+8, so the PC+8 from the PC adder is selected as an operand
3. The ALU takes in PC+8 and signed 24 bit offset as operands and calculates the new branch target
4. The branch target is stored to the PC to be fetched in the next cycle
5. On branch and link instructions, PC+4 is stored into R14

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

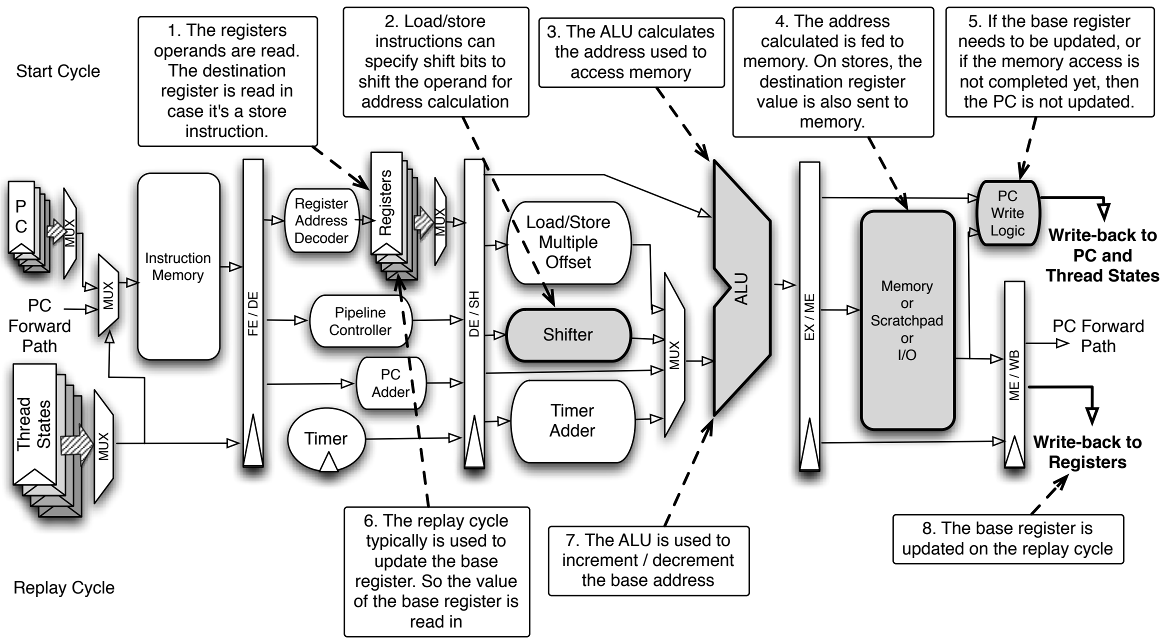

Start Cycle
1. The registers operands are read. The destination register is read in case it's a store instruction.
2. Load/store instructions can specify shift bits to shift the operand for address calculation
3. The ALU calculates the address used to access memory
4. The address calculated is fed to memory. On stores, the destination register value is also sent to memory.
5. If the base register needs to be updated, or if the memory access is not completed yet, then the PC is not updated.
PC
MUX
PC Forward Path
Instruction Memory
FE / DE
Register Address Decoder
Registers
Pipeline Controller
PC Adder
Timer
DE / SH
Load/Store Multiple Offset
Shifter
Timer Adder
ALU
EX / ME
Memory or Scratchpad or I/O
PC Write Logic
Write-back to PC and Thread States
PC Forward Path
ME / WB
Write-back to Registers
Thread States
6. The replay cycle typically is used to update the base register. So the value of the base register is read in
7. The ALU is used to increment / decrement the base address
8. The base register is updated on the replay cycle
Replay Cycle

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,

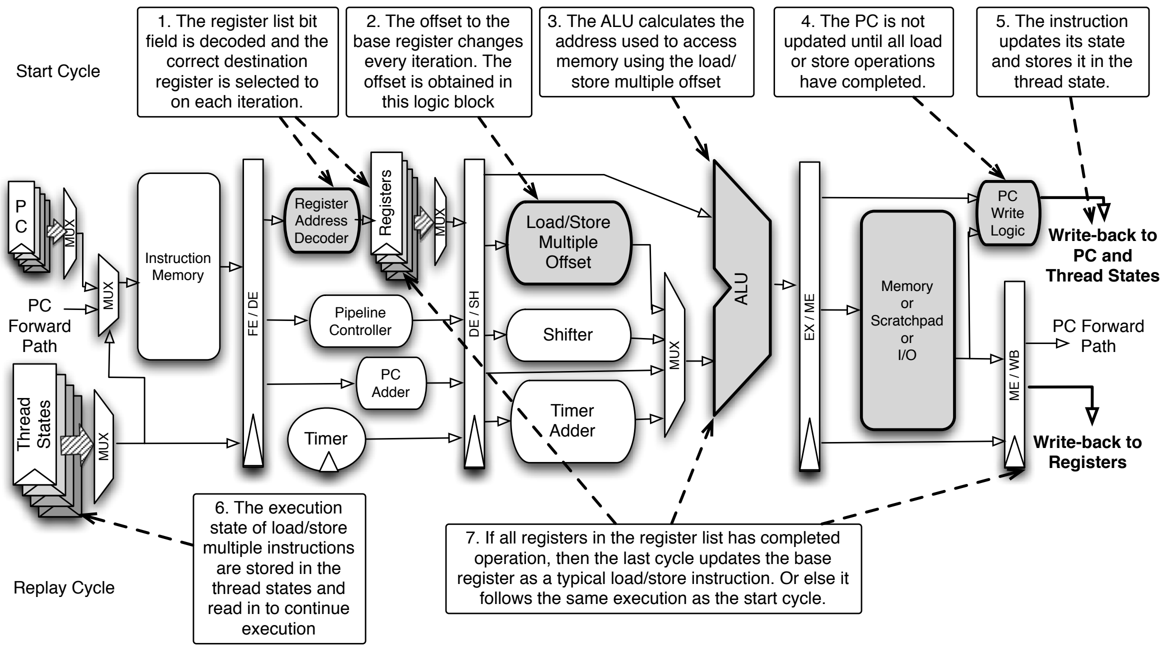

Start Cycle
1. The register list bit field is decoded and the correct destination register is selected to on each iteration.
2. The offset to the base register changes every iteration. The offset is obtained in this logic block
3. The ALU calculates the address used to access memory using the load/store multiple offset
4. The PC is not updated until all load or store operations have completed.
5. The instruction updates its state and stores it in the thread state.
PC
MUX
PC Forward Path
Instruction Memory
Thread States
FE / DE
Register Address Decoder
Registers
Pipeline Controller
PC Adder
Timer
DE / SH
Load/Store Multiple Offset
Shifter
Timer Adder
ALU
EX / ME
Memory or Scratchpad or I/O
PC Write Logic
Write-back to PC and Thread States
ME / WB
PC Forward Path
Write-back to Registers
6. The execution state of load/store multiple instructions are stored in the thread states and read in to continue execution
7. If all registers in the register list has completed operation, then the last cycle updates the base register as a typical load/store instruction. Or else it follows the same execution as the start cycle.
Replay Cycle

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

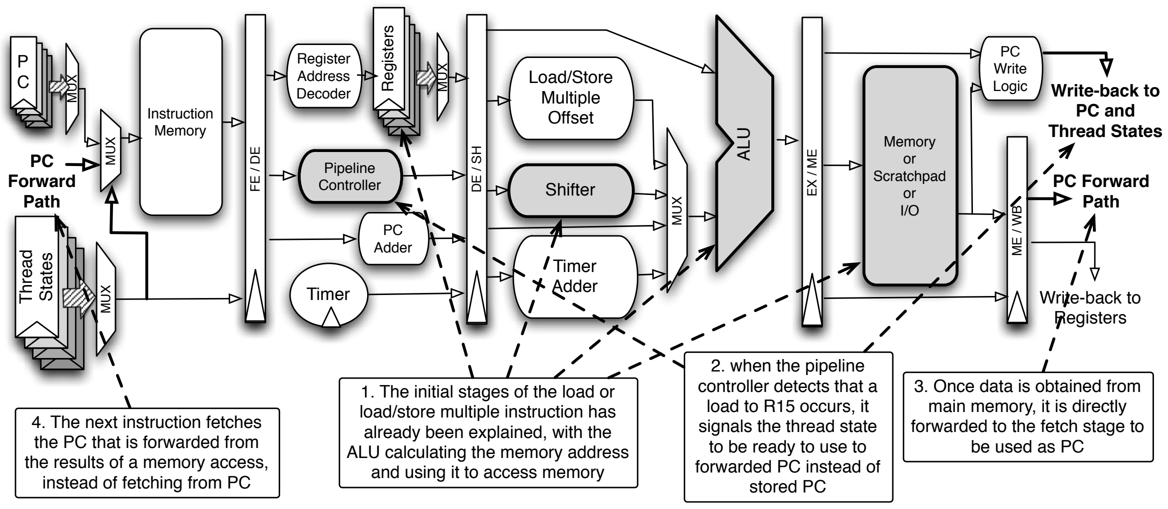

P
C
MUX
PC
Forward
Path
MUX
Instruction
Memory
Thread
States
MUX
FE / DE
Register
Address
Decoder
Registers
MUX
Pipeline
Controller
PC
Adder
Timer
DE / SH
Load/Store
Multiple
Offset
Shifter
Timer
Adder
MUX
ALU
EX / ME
Memory
or
Scratchpad
or
I/O
PC
Write
Logic
Write-back to
PC and
Thread States
ME / WB
PC Forward
Path
Write-back to
Registers
1. The initial stages of the load or load/store multiple instruction has already been explained, with the ALU calculating the memory address and using it to access memory
2. when the pipeline controller detects that a load to R15 occurs, it signals the thread state to be ready to use to forwarded PC instead of stored PC
3. Once data is obtained from main memory, it is directly forwarded to the fetch stage to be used as PC
4. The next instruction fetches the PC that is forwarded from the results of a memory access, instead of fetching from PC

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.

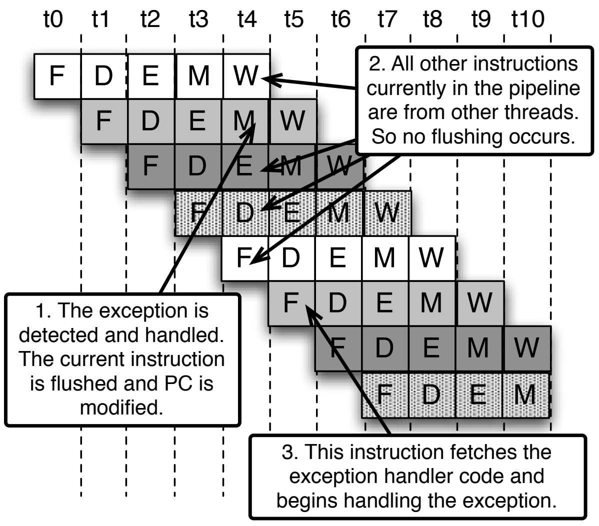

t0
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
F D E M W
2. All other instructions currently in the pipeline are from other threads. So no flushing occurs.
1. The exception is detected and handled. The current instruction is flushed and PC is modified.
3. This instruction fetches the exception handler code and begins handling the exception.

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

Type	Opcode	Functionality

get_time	8	offset = (crm << 32) + crn;
		deadline = *current_time* + *offset*;
		crd = low32(deadline);
		crd+1 = high64(deadline);

delay_until	4	deadline = (crm << 32) + crn;
		if (*current_time* < *deadline*)
		stall_thread();

exception_on_expired	2	offset = (crm << 32) + crn;
		register_exception(offset);

deactivate_exception	3	deactivate_exception();

Table 3.1: List of assembly deadline instructions

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

In section 2.3 we presented various instruction extensions to the ISA to bring timing semantics at the ISA level. We will now present a primitive implementation of those instructions in PTARM. ARM provides extra instruction encoding slots to be used to implement instructions for co-processors attached to the core. In our case, we implement our timing instructions as part of co-processor 13. We have already described the functionality and use case of the different timing instructions, table 3.1 shows a summary of the instructions and their op codes. All instructions have the assembly syntax “***cdp, p13, <opcode> rd, rn, rm, 0***”, with <opcode> differentiating the instruction type.

The timing instructions uses a master clock to obtain and compare deadlines. PTARM implements the clock in the *timer* block that is shown in figure 3.1. Time is currently represented

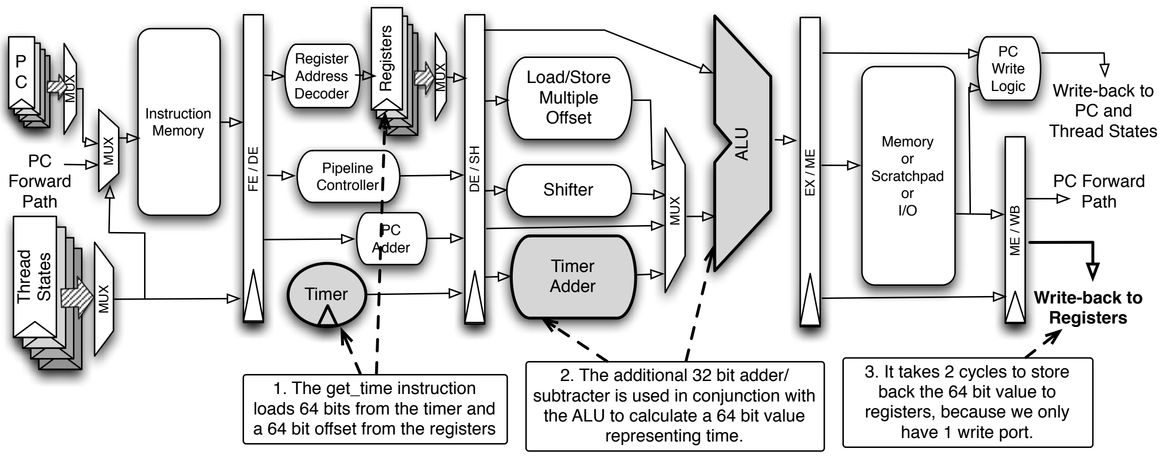

P C
MUX
PC Forward Path
MUX
Thread States
MUX
Instruction Memory
FE / DE
Register Address Decoder
Registers
MUX
Pipeline Controller
PC Adder
Timer
DE / SH
Load/Store Multiple Offset
Shifter
Timer Adder
MUX
ALU
EX / ME
Memory or Scratchpad or I/O
PC Write Logic
Write-back to PC and Thread States
PC Forward Path
ME / WB
Write-back to Registers
1. The get_time instruction loads 64 bits from the timer and a 64 bit offset from the registers
2. The additional 32 bit adder/ subtracter is used in conjunction with the ALU to calculate a 64 bit value representing time.
3. It takes 2 cycles to store back the 64 bit value to registers, because we only have 1 write port.

Figure 3.9: Get_Time Instruction Execution in the PTARM Pipeline

as an unsigned 64 bit value with nanoseconds as its units, and starts at zero when PTARM is reset. Unsigned 64 bits of nanoseconds can represent time up to approximately 584 years. (Todo: discuss timer implementation relative to different clock speeds.) Each of the timing instructions operate on 64 bit values, which is stored in two 32 bit registers. Because the deadlines are stored in general purpose registers, standard arithmetic instructions can be used to manipulate the values. However, PTARM does not current provide 64 bit arithmetic operations, so programmers must handle the overflow in software.

For each thread, the timer value is latched into the pipeline at the decode stage, which is where each thread's reference to time is. Each thread operates on their own private deadlines, and are not affected by the timing instructions from other threads. PTARM contains 4 hardware threads that are interleaved through the pipeline, so each hardware thread can only access the timer once every 4 processor clock cycles, the granularity of time observed by each thread. We will discuss the timing implications of this in section 3.4.1, in this section we merely present how they are implemented in the pipeline.

Get_Time

The *get_time* instruction is used to obtain the current timer value and store it in two general purpose registers. The *get_time* instruction also takes two optional source operands to calculate an offset to the current timer value. This allows the programmer to obtain the desired deadline time without additional arithmetic instructions. Figure 3.9 shows how *get_time* is implemented in the PTARM pipeline.

The timer value and source registers are read in at the decode stage. The *timer adder* adds the lower 32 bits of the timer value and source operand while the ALU computes the upper 32 bits taking into account the carry of the *timer adder*. Once the new deadline has been calculated, it is loaded back into the register file. Because our register file only contains one write port, so this instruction also takes two thread cycles to complete; each cycle writes back 32 bits of the new value. The calculated 64 bit deadline value is written to the destination register rd and rd+1, with rd storing the lower 32 bits and rd+1 storing the higher 32 bits. This instruction will not write to R15 (PC), and it will not cause a branch. If R14 or R15 is specified as rd, causing a potential write to R15,

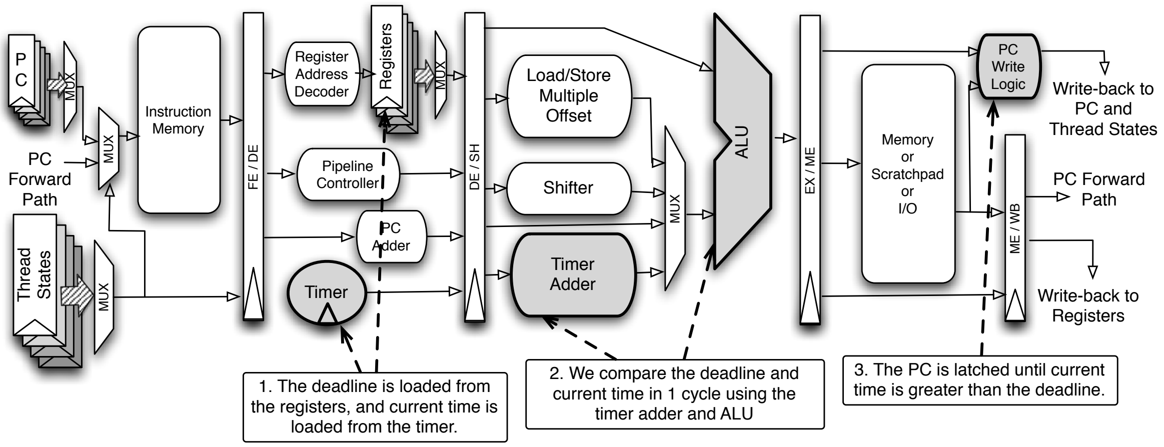

P C
MUX
PC Forward Path
MUX
Instruction Memory
Thread States
MUX
FE / DE
Register Address Decoder
Registers
MUX
Pipeline Controller
PC Adder
Timer
DE / SH
Load/Store Multiple Offset
Shifter
Timer Adder
MUX
ALU
EX / ME
Memory or Scratchpad or I/O
PC Write Logic
Write-back to PC and Thread States
ME / WB
PC Forward Path
Write-back to Registers
1. The deadline is loaded from the registers, and current time is loaded from the timer.
2. We compare the deadline and current time in 1 cycle using the timer adder and ALU
3. The PC is latched until current time is greater than the deadline.

Figure 3.10: Delay_Until Instruction Execution in the PTARM Pipeline

then this instruction will simply act as a NOP.

(Todo: Talk about whether or not the time passed back adjusts for the latency of the instruction.)

Delay_Until

Delay_until is used to delay the thread until the a specified deadline has been reached. It takes in 2 source operands which forms a 64 bit deadline value that is checked against the timer every thread cycle. The 2 source operands are usually the results of a *get_time* instruction. As described in section 2.3, the *delay_until* instruction can be used to specify a lower bound execution time for a code block. This could be useful for synchronization between tasks or communicating with external devices. Figure 3.10 shows the implementation of the *delay_until* instruction in the PTARM pipeline.

The implementation of the *delay_until* instruction is very straightforward, but highlights the reason the *timer adder* is added into the pipeline. The source operands and the timer value is latched at the decode stage, then compared using the *timer adder* and ALU. The next PC is only updated if the timer value is greater than the deadline values passed in. Without the additional *timer adder* in the pipeline, comparing 64 bits using our 32 bit ALU would take two thread cycles. This would decrease the precision of this instruction by a factor of two, because now we can only check the deadline against the timer every two thread cycles. The added *timer adder* allows *delay_until* to check the deadline every thread cycle, to ensure that no additional threads cycles have elapsed right after the deadline is reached.

Exception_on_Expire, Deactivate_Exception

Exception_on_expire and *deactivate_exception* provide a mechanism to actively check for missed code that runs longer than a specified deadline. *Exception_on_expire* is used to register a deadline for immediate miss detection. *Deactivate_exception* is used to deactivate the active checking before the deadline expires. Unlike the *delay_until* instruction, which checks the deadline when the instruction is decoded, the deadlines registered with *exception_on_expire* are checked in hardware in the background. A timer expired exception is triggered in the pipeline when a missed deadline is detected. In section 2.3 we have outlines examples of how the instructions are used.

The actual execution of the *exception_on_expire* and *deactivate_exception* instructions is straightforward. Within the *timer* unit, there is one 64 bit deadline slot for each thread to register an actively checked deadline. With four threads in PTARM, there are four slots in the *timer* hardware. Whenever an *exception_on_expire* instruction is executed, two source operands are read in and stored to the thread's corresponding deadline slot in the *timer*. Every clock cycle of the timer, active deadlines are checked against the current timer value. Once the current timer value surpasses one of the active deadlines, a timer expired exception for the particular thread is raised in the pipeline. We add an entry to the existing ARM exception vector table to create this timer expired exception. The exception is handled the same as any other exception, as discussed in section 3.3.4, and is only handled by the thread that registered the deadline. The other threads in the pipeline remain temporally isolated from this exception. The execution of a *deactivate_exception* instruction simply clears the deadline slot for the specific thread.

If threads need to simultaneously check for multiple deadlines, then the single deadline slot for the thread needs to be managed in software. The software overhead involves managing a list of deadlines and ensuring that the earliest deadline is always being checked in the timer. It is possible to implement more than one deadline slot for each thread in the timer if more precise deadline checking is needed. However, this comes at the cost of additional hardware complexity in the timer, so currently in PTARM we simply have one deadline slot per thread, and use software mechanisms to manage multiple deadlines in a thread.

3.4 Time Analysis of PTARM

3.4.1 Precision of timing instructions

3.5 PTARM VHDL Soft Core

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

Figure 3.11 shows the high level block diagram of the Softcore.

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

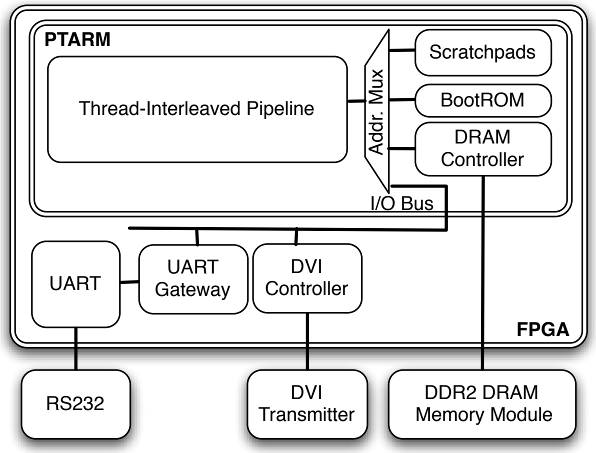


Figure 3.11: PTARM Block Level View

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 [13] 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

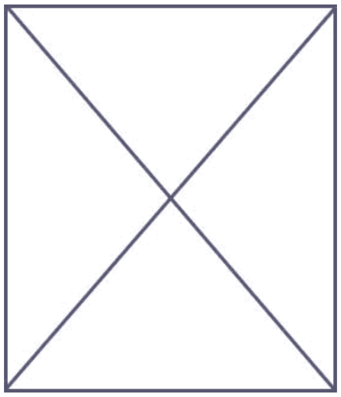


Figure 4.1: Image Placeholder

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 [24] 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 [24]. 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 [10, 14, 51] 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 [7, 38, 2, 12].

In order to not be confused with Kocher's [24] 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 [31, 23], fault injection attacks [8, 15], and many others [51].

In recent years, we have seen a tremendous effort to discover and counteract side-channel attacks on encryption systems [8, 12, 25, 21, 1, 22, 11, 50, 49]. However, it is difficult to be fully assured that all possible vulnerabilities have been discovered. The plethora of research on side-channel exploits [12, 8, 25, 21, 1, 22, 11, 50, 49] 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. [12] discovered two previously unknown time-exploiting attacks on modern x86 processors 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 [24] or blinding signatures [24, 11]. Other techniques such as branch equalization [32, 51] 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 [30] 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 [30], 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 [25], key-card gates [9], set-top boxes [25], and thumbpods [42], which are a good fit for PRET's embedded nature. We demonstrate the effectiveness of our approach by running both the RSA and DSA [34] 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.

5.1.1 Related Work

Kocher outlined a notion of timing attacks [24] on encryption algorithms such as RSA and DSS that require a large number of plaintext-ciphertext pairs and a detailed knowledge of the target implementation. By simulating the target system with predicted keys, and measuring the run time to perform the private key operations, the actual key could be derived one bit at a time.

Kocher also introduced power attacks [31, 23], which use the varying power consumption of the processor to infer the activity of the encryption software over time. These played a large role in stimulating research in side-channel cryptanalysis [33, 22], which also found side-channel attacks against IDEA, RC5 and blowfish [22]. Fault-based attacks [8, 21, 15] were introduced by Bihan et al. [8]. These attacks attempt to extract keys by observing the system behavior to generated faults. For the side-channel attacks that we have missed, Zhou [51] presents a survey on a wide range of side-channel attacks.

Dhem et al. [14] demonstrated a practical implementation of timing attacks on RSA for smart cards and the ability to obtain a 512-bit key in a reasonable amount of time. Several software solutions such as RSA blinding [24, 11], execution time padding [24], and adding random delays [24] have been proposed as possible defenses against this attack. However, these solutions were not widely adopted by the general public until Brumley et al. [10] orchestrated a successful timing attack over the local network on an OpenSSL-based web server. This motivated further research on timing attacks for other encryption algorithms such as ECC [15] and AES [7]. In particular, Bernstien's attack on AES [7] targeted the the run time variance of caches. The introduction of simultaneous multi-threading (SMT) architectures escalated this type of attack on shared hardware components. Percival [38] showed a different caching attack method on SMT, made possible because caches were shared by all processes running on the hardware architecture. Acimez et al. introduced branch predictor attacks [2, 1] that monitor control flow by occupying a shared branched predictor. Compiler and source-to-source transformation techniques [12, 32] have also been developed to thwart side-channel attacks.

Wang et al. [49] identified the causes of the timing attacks to be the underlying hardware. In particular, their work focuses on specialized cache designs, such as Partition-Locked Caches [50] and Random Permutation caches [49] that defend against caching attacks in hardware. Very recently, Coppens [12] discovered two previously unknown attacks on the complex pipeline run time variance of x86 architectures.

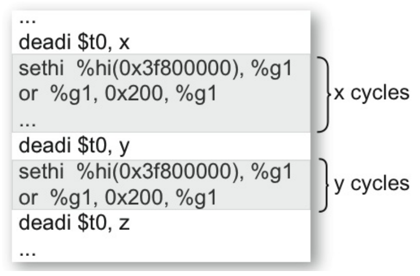
Our work builds upon the experiences of these. Most solutions employ either exclusively hardware or software techniques to defend against attacks. We recognize that a complete solution to control temporal semantics requires a combination of both software and hardware approaches to defend against and prevent future side-channel attacks. Hence, we present an effort that includes timing control instructions to control execution times in software, and a predictable processor architecture to realize the instructions. By doing this, we completely eliminate the source of leaked information used by time-exploiting attacks, rendering the system immune against such attacks.

5.1.2 A Precision Timed Architecture for Embedded Security

Controlling Execution Time in Software

Current instruction-set architectures (ISA) have neglected to bring the temporal semantics of the underlying architecture up to the software level. As a result, it is extremely difficult to control and reason about timing behaviors of the software, even with adequate understanding of the underlying architecture. This proves to be costly in terms of security, because it uncontrollably leaks timing information which can correlate to the secret key.

PRET implements a simple processor extension inspired by Ip and Edwards [18] that adds timing instructions to the ISA. These enable a programmer to have explicit control of execution time in software. To be consistent with the terminology used in [18], we call this instruction the *deadline instruction*. The deadline instruction is used to load values into a special set of registers, called the *deadline registers*. In [18], these registers are decremented each clock cycle in hardware, and act as hardware cycle counters. Whenever the processor executes a deadline instruction, it first checks the corresponding deadline register's contents. If it is zero, then the value specified in the instruction is loaded into the register, and the program continues. If not, the processor replays this instruction until the value reaches zero. Figure 5.1 shows a simple illustration of what the deadline instructions look like. By enclosing instruction blocks within two deadline instructions, we can specify that the enclosing code block should run for x or y cycles. In [18], this instruction was implemented for a single cycle processor. PRET extended this instruction to be used in a multi-threaded pipeline architecture. We will later show a more concrete example of how this instruction is extended for PRET.



```
...
deadi $t0, x
sethi  %hi(0x3f800000), %g1
or  %g1, 0x200, %g1
...
deadi $t0, y
sethi  %hi(0x3f800000), %g1
or  %g1, 0x200, %g1
deadi $t0, z
...
```

Figure 5.1: A method to express timing requirement in software.

Requiring only a few extra registers, the deadline instruction is a powerful extension because it allows for a separation of concern between the functionality and timing behavior of the code. The programmer can implement the correct functionality first, then enclose the source code with deadline instructions to regulate its timing behavior. However, programmers should be cautious because this instruction does not physically increase processor speed. Thus, if the value of the deadline instruction is unreasonably small, then the deadline instruction will have no effect – by the time the second deadline instruction is encountered, the value in the deadline register will already be zero. Then, the deadline register will simply be loaded with the new value, and the program will carry on. It is also possible to throw a hardware exception if the deadline register reaches zero before encountering the second deadline instruction. This means that a block of code ran longer than specified. This allows programmers to register handler functions for code that can potentially lead to missed deadlines. This is not as applicable for cryptosystems, so we do not go into further detail.

The deadline instruction provides a method to control the timing behavior of a program in software. This, however, is only half of the solution. Since the ISA is essentially a contract between hardware and software, it would be meaningless if an instruction is provided, but the underlying architecture does not support it. Therefore, along with the deadline instruction, PRET also provides a *predictable* architecture to ensure that the timing requirements specified in software can actually be met by the architecture.

5.1.3 Pipelining

Complexities of Modern Pipelines

In order to improve instruction throughput and performance, modern processor architectures implement pipelines to execute multiple instructions in parallel. This requires handling of pipeline hazards, which are caused by dependencies in instruction sequences. Conditional branches are the perfect example – the pipeline cannot fetch and begin executing the next instruction without knowing which instruction to fetch. Since a conditional branch usually takes more than one cycle to resolve, the processor is forced to stall until the branch is resolved.

Computer architects use clever speculative techniques to mitigate the effects of pipeline hazards and to substantially improve the average-case performance. For example, branch predictors are used to guess the next instruction needed by the processor for branches [16]. This allows the processor to execute instructions speculatively while rolling back only when needed. While these speculative techniques improve the average-case performance, they introduce several side effects. First, they create *timing variations*. Depending on the outcome of its speculation, the processor might need to discard the wrongly speculated work, and re-execute the correct instructions. Second, these units are *unpredictable*. Since these units are shared by all software processes concurrently running on the processor, the states of speculation units are heavily dependent on the different interleaving of processes. This means that a process can unknowingly be affected by other processes, since the speculation state is shared between them [27]. Because the goal of these speculation techniques is to improve program performance without effort from the programmer, the controls of these speculation units are concealed from the programmer, and cannot be directly accessed or modified in software. Thus, these side effects result in *uncontrollable* timing behaviors in the program. For general purpose applications, these side effects pose insignificant threats, but for security applications, the consequences are uncontrollable sources of side-channel information leakages.

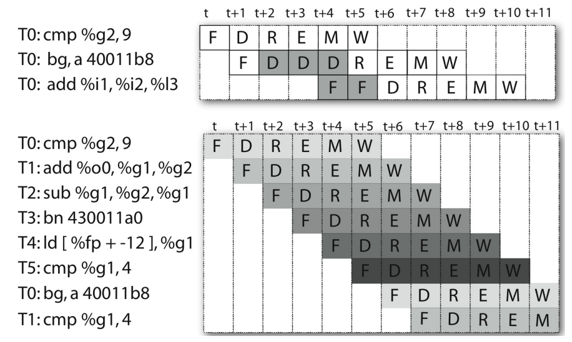
Predictable Pipelines through Thread-Interleaving

An alternative to adding speculation units is to utilize thread-level parallelism by introducing multiple hardware threads in the processor. This allows us to interleave the execution of hardware threads during pipeline stalls. If the thread scheduling policy is implemented in hardware and concealed from the programmer, then any dynamic scheduling scheme inherits the same side effects as mentioned above. This is again because there is no control from the software as to which thread gets to execute on the processor. The class of Simultaneous Multi-Threading (SMT) architectures is an example that shows this. Here, the hardware threads share multiple execution units and execute in parallel depending on a hardware scheduler. Attackers exploit such designs by running a spy thread that executes concurrently with a thread that implements the encryption algorithm. This spy thread probes the components shared with the encryption thread [38, 1] by forcefully occupying the shared units and observing when they are evicted by the encryption thread. The announcement of this vulnerability caused Hyper-Threading, Intel's implementation of SMT, to be disabled by default in some Linux distributions because of its security risks [39]. Thus, in order to bring timing controllability, the hardware thread scheduling policy must be transparent and predictable to the programmer while the states of the hardware threads must be decoupled.

PRET employs a thread-interleaved pipeline [29], which is a multi-threaded architecture where the thread-scheduling policy is a simple round-robin between the hardware threads each cycle. Instructions from each thread are predictably fetched into the pipeline every n cycles, where n is the number of hardware threads. If n is greater than the number of stall cycles needed for data dependency hazards, then we effectively remove those hazards because the data value is available during the next cycle in which the thread is dispatched. For example, if we set n to be the number of stages in the pipeline, then we eliminate the need for any data forwarding/bypassing logic, along with the need for hardware speculation units such as branch predictors.

Figure 5.2 illustrates this idea. The top diagram shows a normal pipeline. The instructions are from the SPARC [43] ISA. *T0* on the left indicates it is an instruction from thread 0. Since this is a single threaded pipeline, multiple instructions are issued from T0.

Cmp (compare) sets a conditional bit if %g2 is equal to 9. *Bg* (Branch greater than) will branch if this bit is set. The *a* means that the instruction following the *Bg* in the branch delay slot is annulled. The *add* is the instruction fetched after the branch is resolved. Notice the branch instruction is stalled at the D (decode) stage of the pipeline, which is where the condition bit is tested. However, this condition bit is not known in the *cmp* instruction until after the execution stage in the pipeline. This is a data dependency hazard, and we need to stall the pipeline even when forwarding is implemented. With a thread-interleaved pipeline, as shown on the bottom, the conditional bit is already set when *T0* needs to be executed again, so no speculation units are needed, and the processor does not need to stall.



	t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8	t+9	t+10	t+11
T0: cmp %g2, 9	F	D	R	E	M	W						
T0: bg, a 40011b8		F	D	D	D	R	E	M	W			
T0: add %i1, %i2, %l3					F	F	D	R	E	M	W	

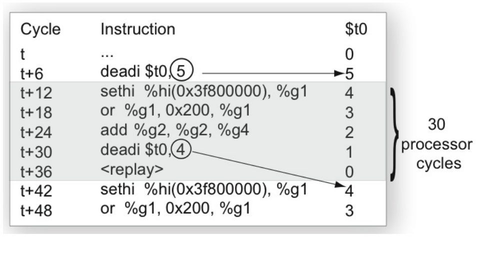
	t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8	t+9	t+10	t+11
T0: cmp %g2, 9	F	D	R	E	M	W						
T1: add %o0, %g1, %g2		F	D	R	E	M	W					
T2: sub %g1, %g2, %g1			F	D	R	E	M	W				
T3: bn 430011a0				F	D	R	E	M	W			
T4: ld [%fp + -12], %g1					F	D	R	E	M	W		
T5: cmp %g1, 4						F	D	R	E	M	W	
T0: bg, a 40011b8							F	D	R	E	M	W
T1: cmp %g1, 4								F	D	R	E	M

Figure 5.2: An example of interleaved threading removing data dependencies

A thread-interleaved architecture is obviously impenetrable to branch predictor attacks [2] or other attacks on the pipeline [12], but more importantly, this gives us a basis for implementing pipelines in a predictable way. We gain in higher instruction throughput without the harmful side effects. Since individual hardware threads maintain their own copy of the processor state (program counter, general purpose registers, stack pointer, deadline registers, etc.), each hardware thread now runs independently of each other with no shared state within the pipeline. Because of the simple and transparent thread-scheduling policy, each hardware thread gets dispatched in a predictable way that is not affected by any other hardware thread. With an understanding of the pipeline, we can now show how deadline instructions are implemented in PRET.

Timing instructions on PRET

Since PRET has multiple hardware threads, each hardware thread contains its own set of deadline registers, which are decremented every time an instruction from that thread is fetched into the pipeline. Specifically, PRET has six hardware threads, so each hardware thread's deadline registers are decremented every six processor cycles. Figure 5.3 shows a concrete example of the execution of one thread on PRET using the deadline instruction. The three instructions enclosed between the deadline instructions take exactly 30 cycles to execute. $\$t0$ shows the contents of deadline register 0. The processor cycle is shown to the left of the instruction. When the first deadline instruction is executed, the instruction will simply load 5 into deadline register 0 and continue because the value of $\$t0$ is currently 0. When the second deadline instruction is executed, since the value of $\$t0$ is not decre-



Cycle	Instruction	$t0
t	...	0
t+6	deadi $t0, 5	5
t+12	sethi %hi(0x3f800000), %g1	4
t+18	or %g1, 0x200, %g1	3
t+24	add %g2, %g2, %g4	2
t+30	deadi $t0, 4	1
t+36	<replay>	0
t+42	sethi %hi(0x3f800000), %g1	4
t+48	or %g1, 0x200, %g1	3

Figure 5.3: A simple example using deadline instructions on PRET

mented to zero yet, this instruction will be replayed. Only at $t+36$ cycles will the value 4 be loaded into deadline register 0. This ensures that the code enclosed will take 30 cycles to execute, as specified in the first deadline instruction. We will show two real-world examples of using deadline instructions with encryption algorithms later in the case study section.

5.1.4 Caches

Caches are one of the main reasons pipelines are so effective in improving performance. The high clock speed of modern processors combined with the high latency to access main memory results in sometimes hundreds of cycles stalled when the processor needs to access the main memory. On-chip fast access memories are used to hide this access latency. Caches are *hardware-controlled* fast-access memories that predict and prefetch data from main memory based on temporal and spatial locality of data accesses from the processor. If the cache control speculation is accurate, then access to data can complete in one cycle, and no stall in the pipeline is required. However, when a misprediction occurs, data needs to be fetched from the main memory, causing a drastic difference in the access time [46]. Unfortunately, caches are often shared in the processor, so just as with the speculation units mentioned previously, this also results in uncontrollable run time variations due to cache interference. Cache attacks target exactly this property of caches, using the timing variation to infer memory access patterns of encryption processes.

Scratchpad memories (SPM) are fast access memories controlled by software. SPMs use less power and occupy less area [4] because no speculation logic is needed. The allocation of data between memory and SPM is done with explicit instructions, either at compile time by the compiler or manually by the programmer. This gives the software control over memory access latencies, and provides a predictable execution time of the program.

Just like caches, the performance of SPMs vary based on the data access patterns of the application, but since the control is in software, it is possible to tune the allocation scheme to achieve even better performance than a generic cache for specific applications. There is abundant ongoing research on allocation schemes and methods for optimizing the performance of SPMs [3, 6, 36, 45, 47]. This may seem to put more work on the compiler or programmer, but it is not uncommon to see software being tuned for each implementation platform. Specifically, high performance parallel algorithms are often fine tuned to work on block sizes depending on the cache size and replacement policy of the platform, and re-tuned when running on different platform. Currently, SPMs can be found in the Cell processor [17], which is used in Sony PlayStation 3 consoles, and NVIDIA's 8800 GPU, which provide 16KB of SPM per thread-bundle [35].

PRET separately provides each hardware thread with 64KB of SPM space. This physical separation is required to decouple the execution time of hardware threads so that there is no scratchpad interference between the hardware threads. However, encryption algorithms also benefit from this. When running an encryption algorithm on its own hardware thread, its scratchpad contents cannot be modified or monitored by spy threads on another hardware thread, making itself immune to shared resource time-exploiting attacks on the fast access memory across hardware threads.

Care must be taken, however, if an encryption algorithm is sharing a hardware thread with other processes, because the SPM space is shared among all software threads utilizing that hardware thread. Since the allocation of SPM is in software, a scheme similar to partition locked caches, which have proven successful against cache attacks [50], can be implemented. The concept is similar to the virtual memory management of operating systems, where a thread supervisor can intercept SPM instructions from processes and create logical partitions of the shared scratchpad space. Another possibility is to run the encryption with the highest priority. If the encryption process cannot be preempted by any other software thread running in the same hardware thread, and other hardware threads cannot affect its behavior, then it is still immune to the attacks.

Clearly, the edge that SPMs give over caches is their controllability in software, which gives the system predictable timing behaviors. This is required for any system where side-channel information leakage may lead to undesirable consequences.

5.1.5 Main Memory Access

For single thread pipelines, there is no contention to access memory. With multiple hardware threads, however, this becomes an issue. Even though a thread-interleaved pipeline decouples the shared states between hardware threads, there is still only one main memory. One approach may attempt to queue up accesses to memory from each hardware thread, but this creates a dependency between hardware threads contending for memory access. This makes the architecture unpredictable again, because a thread has no knowledge of the memory access patterns of other threads. Since the number of requests to memory queued up is unknown, we cannot provide an upper bound on the access time to main memory. Any other dynamic arbitration scheme for memory suggests side effects analogous to a dynamic thread scheduling policy. Clearly, for a predictable architecture, a transparent and predictable scheme must be used.

PRET uses a *Memory Wheel* to implement a time-triggered arbitration scheme to access main memory. Every hardware thread has a window in which it must make its request to the main memory and complete the access. Otherwise, the hardware thread must wait for its window. Such a time-triggered arbitration decouples the time it takes each thread to successfully perform a main memory access, and it provides an upper bound on the time each memory access completes. However, when the hardware thread is waiting for its window, it cannot stall the whole pipeline because this would change the timing behavior of instructions in the pipeline from other hardware threads. Instead, it replays the instruction at the next round when it is dispatched into the pipeline, until the data is received from memory. With the *Memory Wheel* and *Replay Mechanism*, the access to memory is now decoupled between hardware threads in a predictable way.

5.1.6 The PRET Approach

It is important to understand that the foundation of 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 goal of PRET is to bring that *controllability* to the software level, thus eliminating the origin of the attacks. We have explained PRET's software extension to allow timing specification in programs, and PRET's predictable architecture to comply with these specifications, but these two approaches cannot be separated. A predictable architecture by itself would only ease the feasibility of an attack, and software timing specifications are meaningless if they cannot be met by the hardware. By combining both hardware and software solutions, we yield a timing predictable and controllable architecture. Thus, by design, PRET prevents leakage of any timing side-channel information, and eliminates the core vulnerability of time-exploiting attacks.

5.1.7 Case Studies

5.1.8 RSA Vulnerability

The central computation of the RSA algorithm is based primarily on modular exponentiation. This is shown in algorithm 1. Of the inputs, M is the message, N is a publicly known modulus, and d is the secret key. Depending on the value of each bit of d on line 4, the operation on line 5 is either executed or not. This creates variation in the algorithm's execution time that is dependent on the key, as mentioned in [24].

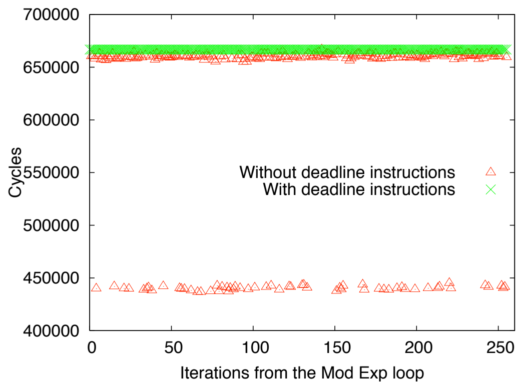
```
Input: M, N, d = $(d_{n-1}d_{n-2}...d_1d_0)$
Output: S = $M^d$ mod N
1 S ← 1
2 for j = n - 1 ... 0 do
3     S ← $S^2$ mod N
4     if $d_j$ = 1 then
5         S ← S · M mod N
6     return S
```

Algorithm 1: RSA Cipher

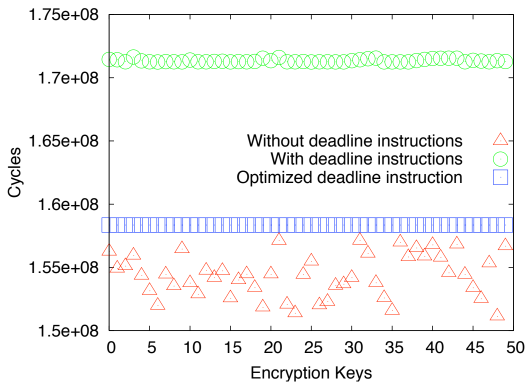
```
Input: M, N, d = $(d_{n-1}d_{n-2}...d_1d_0)$
Output: S = $M^d$ mod N
1 S ← 1
2 for j = n - 1 ... 0 do
3     /* 110000 is 660000÷6 cycles, since deadline registers
      are decremented every 6 cycles.*/
4     dead(110000);
5     S ← $S^2$ mod N
6     if $d_j$ = 1 then
7         S ← S · M mod N
8     dead(0);
9     return S
```

Algorithm 2: RSA Cipher with deadline instructions

When the reference implementation of RSA (RSAREF 2.0) was ported to the PRET architecture, single iterations of the loop varied in execution time almost exclusively due to the value of d_j, which is the j^{th} bit of the key. The triangle points in figure 5.4(a) show the measured run time of each iteration in the for loop (lines 2–6) in algorithm 1. Each iteration took approximately either 440 or 660 kilocycles, with very little deviation from the two means. As a simple illustration, we can fix the execution time of each iteration in software by adding deadline instructions in the body of the loop as shown in algorithm 2. When enclosed with deadline instructions, the execution time of each iteration is uniform, and the bimodality of the execution time is completely eliminated. The x points in figure 5.4(a) show the measured time of each iteration after adding deadline instructions; they are simply a straight line.



(a) Run time of Modular Exponent operation



(b) Run time of RSA operation

Figure 5.4: RSA Algorithm

We observe the large-scale effect of this small change on the whole encryption in figure 5.4(b), where RSA was run fifty times using randomly generated keys. Without the deadline instructions (triangle points), different keys exhibit significant diversity in algorithm execution time. With the deadline instructions added within the modular exponentiation loop (circle points), the fluctuation is dramatically reduced to almost none. The remaining small variations result from code that is outside of the modular exponentiation loop, which is not influenced by the actual key. From figure 5.4(b) we can see that this small variation is not significant enough to correlate the total execution time and the key.

Without explicit control over timing, any attempt to make an algorithm run at constant time in software would involve manual padding of conditional branches. This forces the algorithm to run at the worst-case execution time, similar to what we've showed. As a result, although this makes the encryption algorithm completely secure against time-exploiting attacks, they are not adopted in practice because of this overhead. Nevertheless, with control over execution time, we will show that running encryption algorithms in constant time does not necessarily require it to run at the absolute worst-case execution time.

5.1.9 An Improved Technique of using Deadline Instructions

It is expected that the distribution of RSA run times will be normal over the set of all possible keys [24]. Figure 5.5 shows the run time distribution measured for one thousand randomly generated keys. A curve fitting yields a bell shaped curve formed from the run time distribution of all keys. This means that the execution time of approximately 95% of the keys will be within ±2 standard deviations of the mean, and the worst-case execution time will be an outlier on the far right of this curve. Our previous example fixed the execution time of all keys to be *roughly* at this far right outlier. An improved technique capitalizes on this distribution of run times to improve performance.

First, instead of enclosing the loop iterations of the modular exponentiation operation, we enclose the whole RSA operation with deadline instructions. Now the deadline instructions are used to control the overall execution time of the RSA operation. Note that we could have done this for the previous example as well to fix the execution time to be *exactly* the worst-case, always.

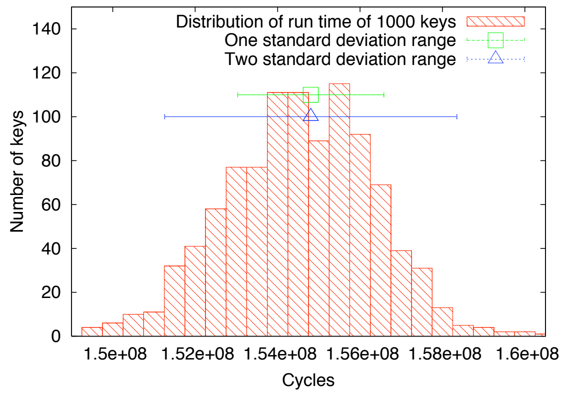
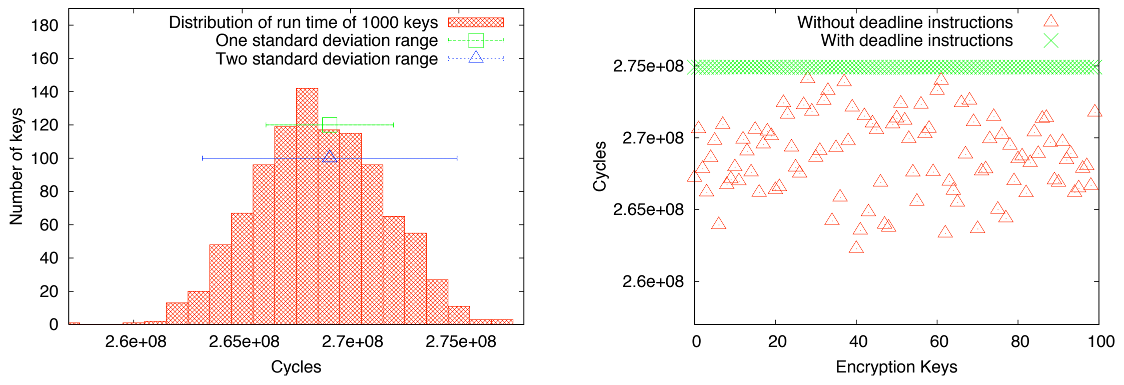


Figure 5.5: Run time distribution of 1000 randomly generated keys for RSA

For RSA, key lengths typically need to be longer than 512 bits to be considered cryptographically strong [40]. This gives roughly 2^{512} possible keys, which is far more than needed for most applications. Suppose we are able reduce the key space the application covers – instead of using 100% of the keys, we refine our encryption system to only assign 97% of all possible keys. Namely, the subset of keys whose RSA execution times fall on the left of the +2 standard deviation line on the curve. Statistically, the keys that lie outside of ±2 standard deviation are the least secure keys anyway, since it is easier for time-exploiting attacks to distinguish those keys. By doing so, we reduce the execution time of the encryption algorithm because we know that keys that are right-side outliers will not be used.

With timing control in software, we can take advantage of this information by simply reducing the value specified in the deadline instructions enclosing the whole RSA operation. The square points in figure 5.4(b) show the results of using deadline instructions in this way. We reran the same fifty keys from the previous section, and enclosed the whole operation with deadline instructions that specified the run time at +2 standard deviations from the bell curve we obtained. We can see that, compared to the previous results that fixed the execution time of each key to take the worst-case time (circle points), we clearly reduced the overhead while still running in constant time. By taking the run time difference between executions with and without deadline instructions, we obtained the overhead introduced for each of the keys with run time below 2 standard deviations (97.9% of keys in our case) within the one thousand key set in our experiment. This calculation reveals that by merely reducing the key space by 3%, running the encryption with optimized deadline instructions only introduced an average overhead of 2.3% over all the keys we measured. All this while still being completely immune to time-exploiting attacks! This is virtually impossible to achieve without explicit timing control, which illustrates the value of decoupling timing control and functional properties of software.



(a) Distribution of 1000 DSA keys (b) Run time of 100 DSA operations

Figure 5.6: Digital Signature Standard Algorithm

Digital Signature Algorithm

Kocher's [24] original paper mentioned that Digital Signature Standard [34] is also susceptible to timing attacks. Thus, to further illustrate our case, we ported the Digital Signature Algorithm from the current OpenSSL library (0.9.8j) onto PRET. We used the same method mentioned above to secure this implementation on PRET. Figure 5.6(a) shows the distribution of DSA run time for one thousand keys. It also shows a normal distribution. Then, we randomly generated another one hundred keys, and measured the run time with and without deadline instructions, which we show in figure 5.6(b). We can see clearly that the run time with deadline instructions is constant, and any time-exploiting attack is not possible.

Currently, we do not know of any work that correlates the key value with run time for different encryption algorithms. However, with the ability to control execution time in software, such a study would be extremely valuable. Figures 5.5 and 5.6(a) show that RSA and DSA follow a normal distribution. Thus, from the algorithm, we postulate that by simply counting the 1 bits in the key should be sufficient to distinguish the 95% of secure keys before assigning. Note that no change to the encryption algorithm itself is needed, but only the key assignment process. Since we can adjust the execution time in software, we can tune the performance of each application based on the application size, key bit length and performance needs. All this can be done while maintaining complete immunity against time-exploiting attacks.

Note that there are several other software techniques specific to encryption algorithms that successfully defend against timing attacks. Our work does not lessen or replace the significance of those findings. Instead, we can use traditional noise injection defenses on PRET as well. For example, if reducing the key space is not possible for some applications running RSA then RSA with blinding can be ran on PRET. By simply running on PRET, the encryption algorithm is also secure against shared hardware resource attacks such as caches, and branch predictors. Other encryption algorithms that do not have software techniques or solutions readily available to counteract timing attacks can easily use the deadline instructions provided by PRET to achieve security against timing attacks.

5.1.10 Conclusion and Future Work

Side-channel attacks are a credible threat to many cryptosystems. They exist not just because of a weakness in an algorithm's mathematical underpinnings, but also from information leaks in the implementation of the algorithm. In particular, this paper targets time-exploiting attacks, and lays out a means of addressing what we consider the root cause of such attacks: the lack of *controllability* over the timing information leaks. As an architecture founded on predictable timing behaviors, PRET provides timing instructions to allow timing specifications in software. In addition, PRET is a predictable architecture that guarantees that timing specifications are honored by the implementation through a thread-interleaved pipeline with separate scratchpad memories for each hardware thread, and a memory wheel to arbitrate access to main memory. This eliminates the shared states in the architecture that create uncontrollable interference leading to some attacks. Through a combination of hardware and software techniques, PRET gives control over the timing properties of programs, which effectively eliminates time-exploiting attacks.

We demonstrate the application of these principles to known-vulnerable implementations of RSA and DSA, and show that PRET successfully defends against time-exploiting attacks with low overhead. Our work does not undermine the significance of any related work, which have mostly been specific to certain attacks. PRET does not target a specific encryption algorithm, because it can be used in combination with these partial solutions on specific encryption algorithms, as well as provide a complete defense for other encryption algorithms which are less researched upon.

Besides time-exploiting attacks, there are other side-channel attacks that are legitimate threats to encryption algorithms such as power, and fault attacks. We plan to continue to investigate PRET's effectiveness in defending against them. We conjecture that the thread-interleaved pipeline used in PRET can potentially help defend against power attacks because the power measured from the processor now includes significant interference from the execution of other hardware threads in the architecture. Currently, PRET is only implemented as a software simulator, but as PRET moves to FPGA implementations, we can further evaluate its effectiveness in defending against power

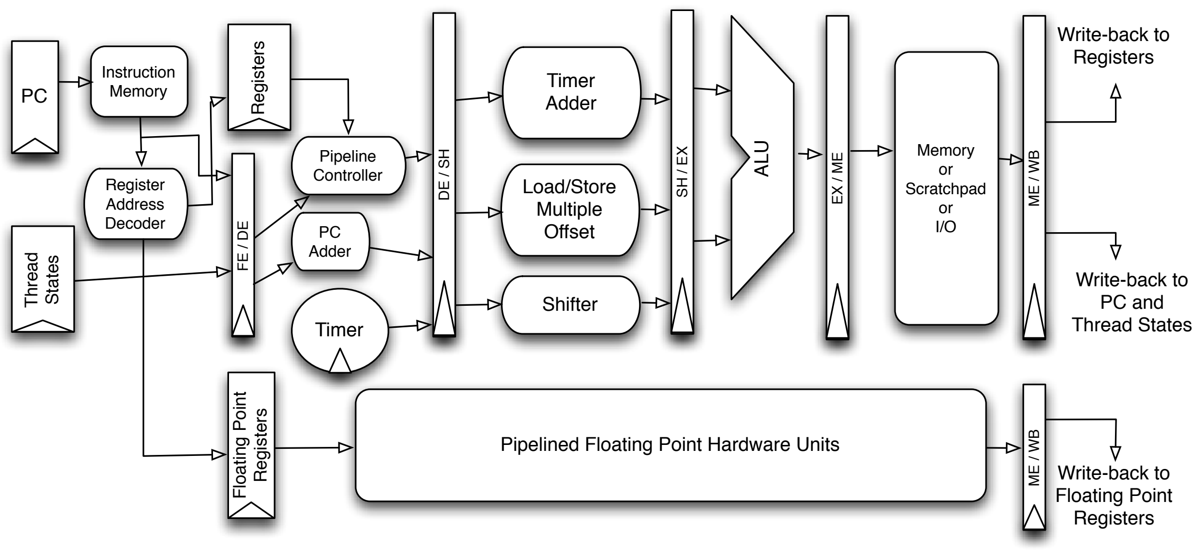


Figure 5.7: Block Level View of the PTARM 6 stage pipeline

attacks. If our conjecture is correct, then PRET with fault tolerant techniques could potentially be a complete solution against several major side-channel attacks.

5.2 Real Time 1D Computational Fluid Dynamics Simulator

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] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 1(1):6–26, 2002.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. *Hardware/Software Co-Design, International Workshop on*, 0:73, 2002.

[5] S. Bandyopadhyay. Automated memory allocation of actor code and data buffer in heterochronous dataflow models to scratchpad memory. Master's thesis, University of California, Berkeley, August 2006.

[6] S. Bandyopadhyay. Automated memory allocation of actor code and data buffer in heterochronous dataflow models to scratchpad memory. Master's thesis, EECS Department, University of California, Berkeley, Aug 2006.

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence Estimation for Speculation Control. In *In 25th Annual International Symposium on Computer Architecture*, pages 122–131, 1998.

[17] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[18] N. J. H. Ip and S. A. Edwards. A processor extension for cycle-accurate real-time software. In *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 4096, pages 449–458, Seoul, Korea, Aug. 2006.

[19] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers, September 2007.

[20] JEDEC. *DDR2 SDRAM SPECIFICATION JESD79-2E.*, 2008.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[27] E. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] J. A. Muir. Techniques of side channel cryptanalysis. Master's thesis, University of Waterloo, 2001.

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

[35] NVIDIA. Technical Breif: NVIDIA GeForce 8800 GPU Architecture Overview. Technical report, NVIDIA, Santa Clara, California, Nov 2006.

[36] H. D. Patel, B. Lickly, B. Burgers, and E. A. Lee. A Timing Requirements-Aware Scratchpad Memory Allocation Scheme for a Precision Timed Architecture. Technical Report UCB/EECS-2008-115, EECS Department, University of California, Berkeley, Sep 2008.

[37] H. D. Patel, B. Lickly, B. Burgers, and E. A. Lee. A timing requirements-aware scratchpad memory allocation scheme for a precision timed architecture. (UCB/EECS-2008-115), Sep 2008.

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

[39] C. Percival. Hyper-threading considered harmful. http://www.daemonology.net/hyperthreading-considered-harmful/, 2005.

[40] Red Hat. Red Hat Certificate System 7.3, Administration guide, B2. Encryption and Decryption.

[41] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS '11: Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 99–108. ACM, October 2011.

[42] 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.

[43] SPARC International Inc. SPARC Standards. Website: http://www.sparc.org.

[44] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Wcet centric data allocation to scratchpad memory. pages 223–232, 2005.

[45] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Wcet centric data allocation to scratchpad memory. *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp.–232, Dec. 2005.

[46] L. Thiele and R. Wilhelm. Design for Timing Predictability. *Real-Time Systems*, 28(2):157–177, 2004.

[47] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratchpad based embedded systems. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, New York, NY, USA, 2003. ACM.

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

[49] 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.

[50] 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.

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