

國立交通大學

電子工程學系 電子研究所
碩士論文

應用於具同指令集架構之處理器家族之
高效能雙層式週期精確模型技術

**Efficient Two-Layered Cycle-Accurate
Modeling Technique for Processor Family
with Same Instruction Set Architecture**

研究生：江建德

指導教授：黃俊達 博士

中華民國九十八年二月

應用於具同指令集架構之處理器家族之
高效能雙層式週期精確模型技術

**Efficient Two-Layered Cycle-Accurate
Modeling Technique for Processor Family
with Same Instruction Set Architecture**

研究生：江建德

Student: Chien-De Chiang

指導教授：黃俊達 博士

Advisor: Dr. Juinn-Dar Huang



A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics
College of Electrical & Computer Engineering
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Electronics Engineering & Institute of Electronics

Feb. 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年二月

應用於具同指令集架構之處理器家族之 高效能雙層式週期精確模型技術


研究生：江建德

指導教授：黃俊達 博士

國立交通大學

電子工程學系 電子研究所

摘 要



本論文提出一簡單且快速的處理器模型技術，此技術採用將時序模型分割為功能與時序兩部份的概念，在本論文中，此二部份分別稱為功能核心與時序外殼。功能核心實際上為一無時序、高速的指令集模擬器，適合軟體開發；而時序外殼則提供了額外的時序資訊協助模擬週期精確的硬體行為。當一使用相同指令集的處理器家族加入了新成員，依照提出的模型技術，我們只須要替換時序外殼，沿用跟其他成員相同的功能核心，即可產生此新成員的模型。這技術不僅能確保各模型的功能與規格吻合，並且能有效縮短建立模型花費的時間。在本論文中，我們將此技術應用在使用相同指令集架構的一 ARM7-like 與一 ARM9-like 處理器，並在實驗結果中顯示，本論文提出的雙層式週期精確模型較一般的 RTL 模型模擬速度平均快上約 30 倍。


Efficient Two-Layered Cycle-Accurate Modeling Technique for Processor Family with Same Instruction Set Architecture

Student: Chien-De Chiang

Advisor: Dr. Juinn-Dar Huang

Department of Electronics Engineering &
Institute of Electronics
National Chiao Tung University

ABSTRACT



This thesis proposes a simple and fast processor modeling method utilizing the concept of partitioning a model into functional and cycle-based timing parts, which are named functional kernel and timing shell respectively in this thesis. The kernel is an untimed but high-speed instruction set simulator (ISS) and is suitable for software development; while the timing shell provides additional cycle-based timing details for cycle-accurate hardware behavior. When a new processor member is added to the family, it demands only a new cycle-based timing shell because the kernel is identical to that of its ancestors sharing the same instruction set architecture (ISA). It not only helps ensure functional consistency but significantly reduces the model development time. We take two processors with a same ISA, an ARM7-like one and an ARM9-like one, as our modeling examples to demonstrate the feasibility of the proposed technique. Finally, the experimental results show that, on average our two-layered cycle-accurate model is about 30 times faster than the RTL model in simulation.

誌 謝

首先，我要感謝我的指導教授—黃俊達博士，在碩士兩年當中給我的支持和鼓勵，讓我能有良好的研究環境，在自由的學習風氣之下，培養出獨立研究的能力，又能隨時給予寶貴的意見及指導，對老師的感激之情，並非以簡短的文句可以表達。

當然也要感謝養育我長大的父母對我的栽培，沒有他們，就沒有今日的我。接著要感謝的是所有來參與口試的所有教授們，林永隆教授、和陳宏明教授，百忙當中抽空前來指導我，讓我受益匪淺，也讓我得到了寶貴的經驗，謝謝你們。

我也要感謝實驗室的同學：威毓、瀚蔚、于翔以及亞謙等諸君，跟大家一起修課、做實驗、和討論及分析研究成果更是我人生旅途中一段最值得珍惜的回憶，希望未來在讀書或工作還有機會一起努力。

希望這篇論文能對人類社會有小小的貢獻，如此一來在辛苦也就值得了，再次謝謝大家的幫忙。

Contents

Chinese Abstract	i
English Abstract	ii
Acknowledgment	iii
Contents	iv
List of Tables	vi
List of Figures	vii
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Contribution	2
1.3 Thesis Organization	3
Chapter 2 Preliminaries	4
2.1 SystemC	4
2.2 ACARM7 vs. ACARM9	5
2.3 Instruction Cycle Operations	7
Chapter 3 Model Architecture	8
3.1 Overview	8
3.2 Function of Each Layer	9
3.3 Interfaces between Layers	10
3.4 Modeling Flow	11
Chapter 4 Model Implementation	14
4.1 Functional Kernel	14
4.2 Timing Shell	15
4.2.1 Time Wheel	17

4.2.2 Scheduler	18
4.2.3 More Examples	19
Chapter 5 Experimental Results	24
Chapter 6 Conclusions and Future Works.....	27
References	28



List of Tables

Table 1. Different abstraction views	1
Table 2. Cycle timing for MUL executed by ACARM9.....	7
Table 3. Contents of result packet.....	11
Table 4. Benchmark programs used in experiments and their cycle counts	24
Table 5. Performance of ARM7TDMI-like models.....	25
Table 6. Performance of ARM9TDMI-like models.....	25



List of Figures

Figure 1. Models partitioned into a functional kernel and timing shells.....	3
Figure 2. Models partitioned into a functional kernel and timing shells.....	4
Figure 3. Models partitioned into a functional kernel and timing shells.....	5
Figure 4. Models partitioned into a functional kernel and timing shells.....	6
Figure 5. Two-layered architecture for cycle-accurate model	8
Figure 6. Modeling flow in the timing shell.....	12
Figure 7. An example of modeling flow	13
Figure 8. The components of functional kernel	14
Figure 9. The components of timing shell.....	16
Figure 10. A simple time wheel	18
Figure 11. An example of scheduling	19
Figure 12. The execution steps of a branch instruction.....	20
Figure 13. The execution steps of two instructions existing data hazard.....	22

Chapter 1 Introduction

High-level behavioral models are widely demanded in recent ESL design methodologies [1]. They are usually in different abstraction views for different purposes as shown in Table 1 [2]. In general, a model with higher abstraction view has faster simulation speed and relatively shorter development time so that it is suitable for software development. On the other hand, one with lower abstraction view is slower in simulation and takes more time to be elaborated. But it can provide more accurate timing details about the hardware behaviors, which is extremely useful for system and hardware verification.

Table 1. Different abstraction views

View	Accuracy and Purpose
Functional View (FV)	Event ordering. Functional specification and algorithm development.
Programmer's View (PV)	Bit accurate. Software development and verification.
Architecture View (AV)	Cycle approximate. Architecture exploration and verification.
Verification View (VV)	Cycle accurate. Hardware verification. System level verification.

1.1 Motivation

In modern ESL design flows, software and hardware are under development in parallel so that a set of models in different levels of abstraction views are required throughout the design process. Conventionally, each member of a processor family needs its own complete set of those models though all members actually share a same ISA. Developing all those models for every single processor certainly takes

significant time. Moreover, if those models are developed separately, functional behaviors among different processors in a same family may likely be variant, which requires additional debugging efforts to achieve overall consistency.

However, in general, processors in a same family produce the identical output, in terms of the contents of user-visible registers, output ports and memory, instruction by instruction since they all share a same ISA. The only difference among them is the cycle timing behavior due to their different implementation details. For example, two embedded processors ARM7TDMI and ARM9TDMI both implement the ARMv4T ISA, while the former has a three-stage pipeline and the latter has a five-stage one. This fact suggests a great idea that a cycle-accurate model can be partitioned into two layers, an inner untimed functional kernel and an outer cycle-based timing shell. The functional kernel is merely elaborated once and can then be shared by all processors within a family. Each processor only needs its own specific timing shell. In this way, not only the model development time can be greatly reduced but also the functional consistency among processors is automatically preserved.

1.2 Contribution

In this thesis, we propose an efficient two-layered architecture for cycle-accurate processor modeling, in which the untimed functional kernel is only responsible for generating correct values of user-visible registers, output ports and memory data for given instructions, while the timing shell is in charge of interacting with the external system through the cycle-accurate model interface and updating those user-visible values provided by the functional kernel at the right time (cycle). Hence, when introducing a new processor member, it is no longer necessary to develop its complete model but only its specific timing shell. Figure 1 points out the idea.

According to an existing work [3], the simulation performance of PV and AV models in SystemC are about 500 and 18 times better than that of RTL model. However, the performance of VV model is not addressed in [3]. Using the newly proposed technique, we have successfully created the PV model (i.e., functional kernel) and VV models (i.e., kernel + timing shell) for an ARM7TDMI-like core and an ARM9TDMI-like core in SystemC. The experimental results show that our VV model, which is cycle-accurate, can simulate about 30 times faster than RTL model. That is, our VV model runs even faster than the cycle-approximate AV model in [3]. Meanwhile, our PV model can run about 860 times faster than RTL model.

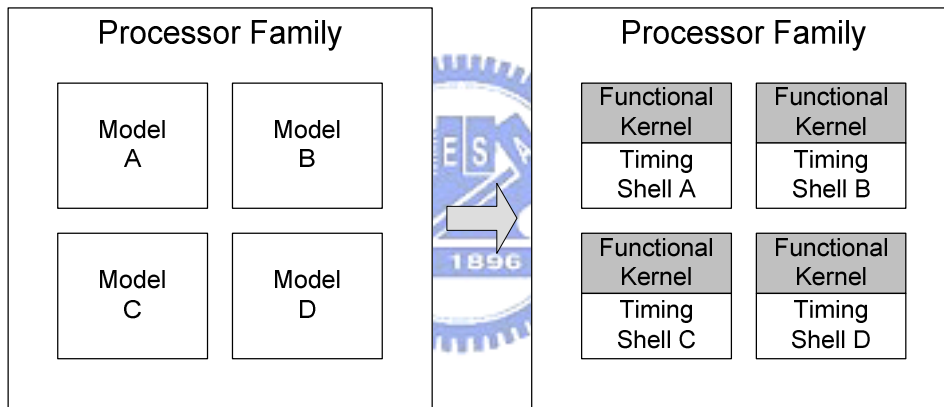


Figure 1. Models partitioned into a functional kernel and timing shells

1.3 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 presents the two-layered architecture for the cycle-accurate model. In Chapter 3, the implementation concerns and details are discussed. The extensive experimental results are reported in Chapter 4. In the end, the concluding remarks for the thesis are presented in Chapter 5.

Chapter 2 Preliminaries

This chapter briefly characterizes the preliminaries of our works. Section 2.1 introduces SystemC, which is used as the modeling language in this thesis. Section 2.2 talks about the timing behaviors of the in-house ARM7TDMI-like and ARM9TDMI-like processors that are modeled for the experiments. In the end, Section 2.3 lists some instruction cycle operations of both processors.

2.1 SystemC

SystemC provides hardware-oriented constructs based on libraries of C++ context and is easy for the designer to build models for analysis or verification [2]. It also supports varieties of different abstraction level especially in the region from PV to VV. In this thesis, we apply SystemC as modeling language and target on the PV model and VV model which are designed for software analysis and hardware verification respectively. Figure 2 shows the basic components of a common SystemC model.

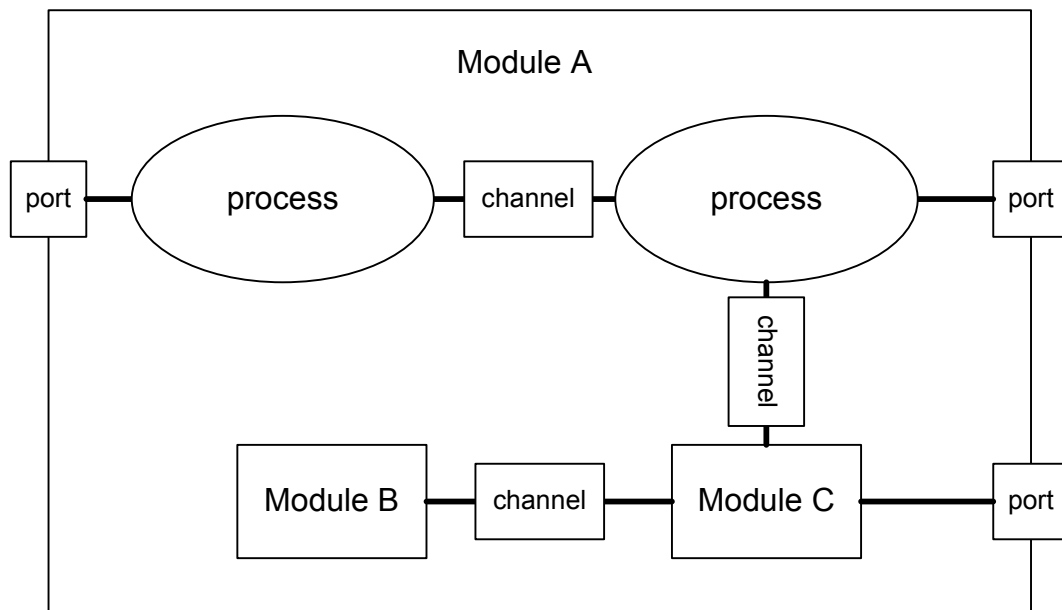


Figure 2. The basic components of SystemC model

- Ports: Ports may be input, output, or inout ports, used for communication with other modules.
- Processes: Processes describe the functionality of module. A module may have one or more processes.
- Modules: A module can contain one or more hierarchical sub-modules.
- Channels: Channels are used for the communication of processes and the sub-modules.

2.2 ACARM7 vs. ACARM9

In this thesis, we build models for an ARM7TDMI-like and an ARM9TDMI-like in-house processor, which are called ACARM7 and ACARM9 respectively, based on the proposed modeling method. They are general-purposed 32-bit RISC processors both apply ARM v4T instruction set architecture but have different pipeline stages, memory system, and timing behaviors. The ARM7TDMI-like processor is 3-stage pipelined while the ARM9TDMI-like processor has a 5-stage pipeline, as shown in Figure 3.

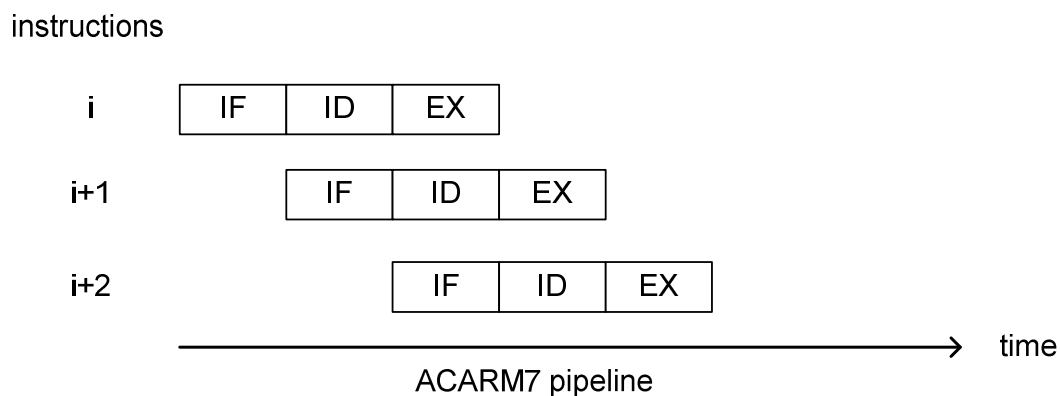


Figure 3a. The pipeline of in house ARM7TDMI-like processor

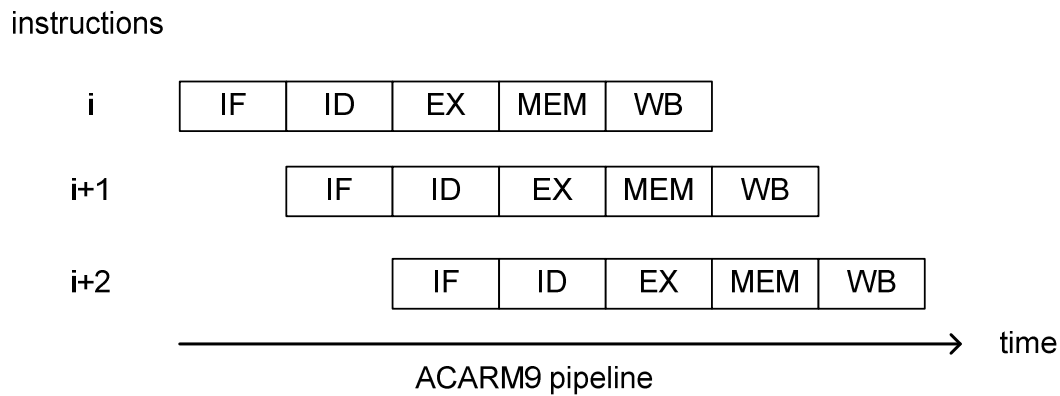


Figure 3b. The pipeline of in house ARM9TDMI-like processor

In the part of memory system, the ACARM7 processor employs the von Neumann memory architecture in which the transferring of instructions and data between processor core and memory actually use the same ports and signals. Due to this, the ACARM7 core cannot access instruction and data in the same clock cycle as shown in Figure 4. On the other hand, the ACARM9 processor has a Harvard memory architecture which has separate sets of ports for instruction and data and can access them at the same time.

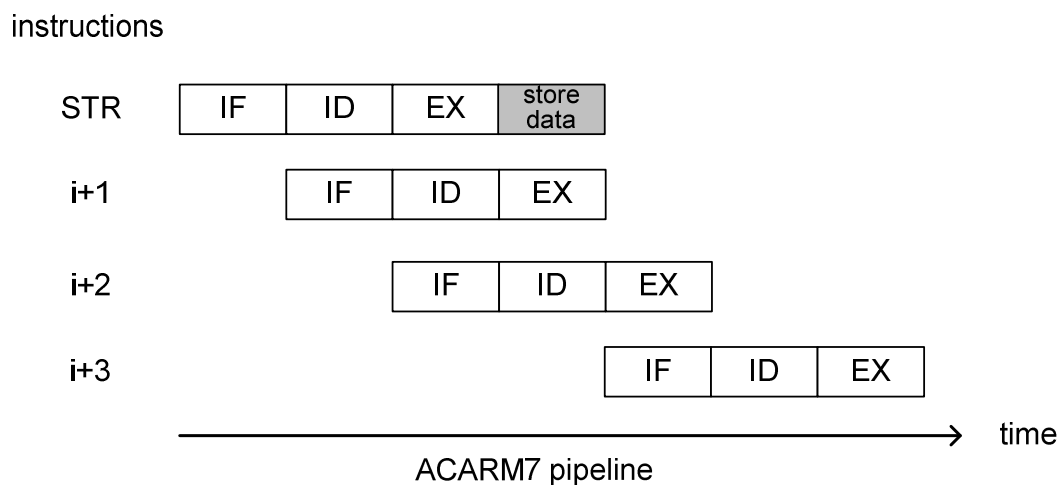


Figure 4. A store instruction on ACARM7 pipeline

Because of the varied pipeline stages, memory architecture and implementation of other functional blocks, the timing behaviors of ACARM7 and ACARM9 processor are quite different although they have the same instruction set architecture.

2.3 Instruction Cycle Operation

Before building a cycle-accurate processor model, we have to know about the cycle operations of each instruction in all condition including special cases such as data hazard or competition for hardware resource, and that those information should be provided by the processor designer. The information has to tell how many cycles the instruction needs to be executed, in which cycle the processor core can fetch the next instruction, and so on, then can we know when the computed results should be updated to register or output signals. Table 2 gives an example which indicates the operations of each execution cycles in normal case and in data hazard case for MUL instruction executed by ACARM9 processor.

Table 2. Cycle timing for MUL executed by ACARM9

Cycle		IA	InMREQ, ISEQ	INSTR
Normal Case	1	pc+12	I cycle	[pc+8]
	2	pc+12	S cycle	
				[pc+12]
Data Hazard	1	pc+12	I cycle	[pc+8]
	2	pc+12	I cycle	
	3	pc+12	S cycle	
				[pc+12]

Chapter 3 Model Architecture

3.1 Overview

To make the proposed layered architecture perfectly work, it is essential to properly describe the role of each layer and clearly define the interface between layers. Here, the functional kernel acts as 1-step untimed instruction set simulator (ISS), while the timing shell provides cycle timing details and communicates with the outside world. These two layers interact with each other by two mechanisms named command packet and result packet. The proposed two-layered architecture for cycle-accurate model is shown in Figure 5.

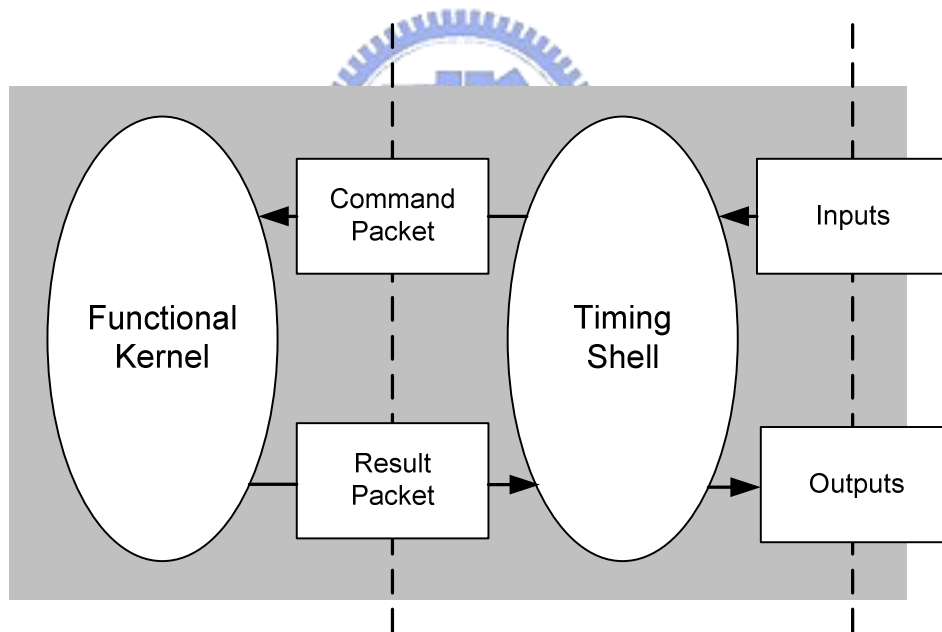


Figure 5. Two-layered architecture for cycle-accurate model

In the rest parts of this chapter, section 3.1 will introduce the functions of layers, the functional kernel and the timing shell; the interfaces between two layers, command packet and result packet as drawn in Figure 5, are described in section 3.2; and the flow of entire model will be presented in section 3.3 finally.

3.2 Function of Each Layer

The functional kernel, which is the inner layer, is actually an untimed instruction set simulator. Without any ideas about input and output ports which might be timing related, the functional kernel just fetch an sequential/non-sequential instruction according to the commands received, and then decodes and executes the instruction to generate correct result values of registers or output ports base on the instruction set architecture completely. Note that the functional kernel is only ISA-dependent since it has nothing to do with the details of pipeline implementation. The new values of registers and output ports and decoded information of instruction generated by the functional kernel need more advanced handling to become timing behaviors of cycle-approximate or cycle-accurate processor model observable to outside environment.

The timing shell, which is the outer layer, takes charge of the communications with the outside environment meaning that it has to sample the input signals and to update the output signals in correct clock cycle to present the pipeline details and model behaviors faithfully. As a result, the timing shell is in responsible in the control of modeling flow, that is, it needs to decide whether to invoke the functional kernel to execute a new instruction or not in the current clock cycle and then schedule in which clock cycle the output signals and registers should be updated caused by the instruction executed according to the result information sent back from the functional kernel. The detail modeling flow of the timing shell will be introduced in the final section of this chapter.

3.3 Interfaces between Layers

As shown in Figure 5, the timing shell and the functional kernel communicate with each other using command packet, which is sent to the functional kernel from the timing shell, and result packet, which is sent in the reverse way. This sector will reveal the contents and function of the packets.

The command packet is sent by the timing shell to request the functional kernel to fetch an instruction and then to execute it. There are two types of command packet, step packet and interrupt packet. Step packets are issued during normal program execution flow. A step packet orders the functional kernel to fetch the next instruction, execute it, update the processor state, and return a result packet. Alternatively, if an external reset or interrupt arises, an interrupt packet is issued instead. An interrupt packet informs the functional kernel to redirect its instruction fetch to the corresponding exception handler, execute it, update the state, and return the results. In every processor non-stall cycle, the timing shell always issues a command packet and the functional kernel always executes the specified instruction and returns the corresponding result packet. The timing shell will not send a command packet during a clock cycle if no instruction fetch is needed due to pipeline stall or a branch and thus the functional kernel won't take any actions including returning a result packet.

The result packet sent from the functional kernel contains a part of the instruction execution outcomes that are necessary for the timing shell to properly update the output signals. It contains three kinds of information which are instruction information, register information, and update information, as listed in Table 3. The instruction information indicates what instruction has just been executed by the functional kernel so that the timing shell knows the exact sequence about when to

update the output values in cycle-by-cycle fashion. The register information contains a list of registers being read and/or written by the executed instruction so that the timing shell can correctly detect all kinds of hazards and take proper forwarding or stall operations. It implies that the timing shell has to know the pipeline details and that is why every processor needs its own timing shell. The update information holds updated values so that the timing shell can properly refresh the related output signals. In brief, under the proposed two-layered architecture, the combination of untimed functional kernel and timing shell can successfully act as a cycle-accurate model for sure.

Table 3. Contents of result packet

Group	Contents
Instruction Information	Types of instruction. For cycle count calculation
Register Information	Registers been read and/or written. For hazard detection.
Update Information	New values For output signals updates.

3.4 Modeling Flow

The modeling flow controlled by the timing shell is shown in Figure 6. In the beginning of every clock cycle, the timing shell samples input signals from the outside of the model, which include reset, interrupts, instruction bus, data input bus, and so on. Then the commander in the timing shell will determine whether to send a command packet to the functional kernel based on the sampled input signals and the current processor states. When the result of an executing instruction is demanded, it queries the functional kernel by issuing a command packet as the path marked “Y” in

the figure. After receiving a command packet, the functional kernel, which is actually an ISS, computes then returns the instruction execution outcomes by sending a result packet back to the timing shell. A mechanism called scheduler, which is the most important part of timing shell, receives the result packet and makes a schedule which points out in which cycle the changes in the processor states, registers, and output signals caused by the instruction should be updated to imitate the timing behaviors of the model. Finally, an updater will updates those changes to the model outputs, containing user-visible registers, address bus, data output bus, and so on, according to the schedule. On the other hand, if no command packet needs be sent, i.e. an stall cycle, the flow just goes to the step of updating as the path marked “N” in the figure.

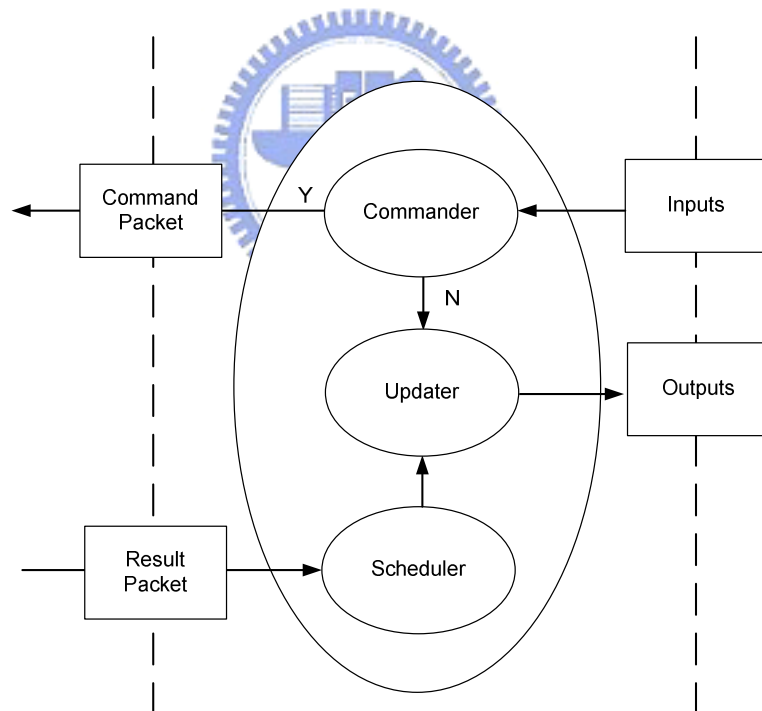


Figure 6. Modeling flow in the timing shell

Note that the updater always updates changes whether in a stall cycle or a non-stall cycle. Since the changes of model outputs caused by an instruction may be in different clock cycle, a stall cycle means not no change happening.

The figure below shows an example of a proposed 3-stage ACARM7 processor model executing three arithmetic instructions without any data hazard. The instructions are computed by the functional kernel as soon as in their pre-fetch stage and the results are scheduled immediately by the timing shell. In each cycle the timing shell will update the model outputs according to the schedule such as a value updating of r1 in cycle i+4 for example.

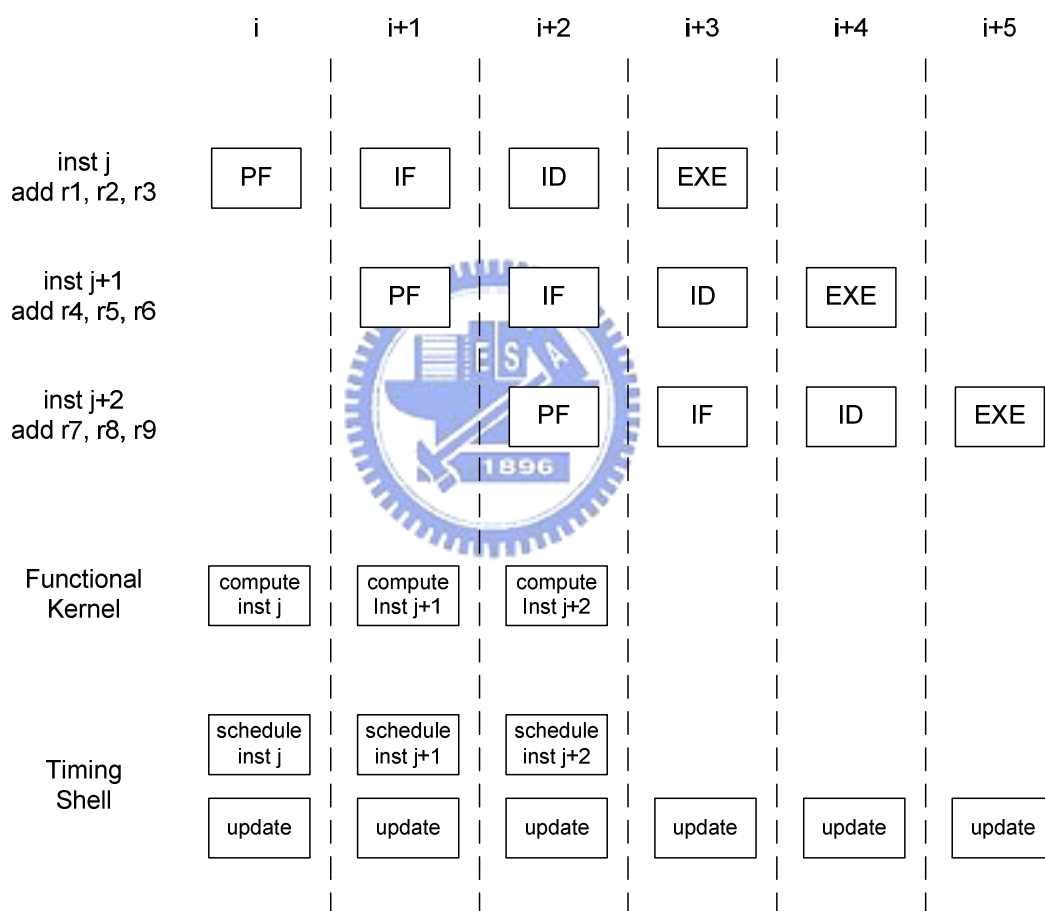


Figure 7. An example of modeling flow

Chapter 4 Model Implementation

This chapter describes the detail implementation of the proposed two-layered model, including the functional kernel and the timing shell. Some examples will also be presented later to show that how the timing shell deals with different pipeline issues.

4.1 Functional Kernel

As mentioned, the functional kernel actually acts as an untimed ISS. Thus the primary components inside are instruction execution engine, program counter, register file, and mirrored instruction/data memory, as depicted in Figure 8. The instruction executing engine is responsible for instruction decoding and datapath operations. The entire functional kernel is built solely based on the ISA specification and absolutely no processor-dependent information can be referred. That is the key reason why the kernel can be safely shared by every processor implementing the same ISA.

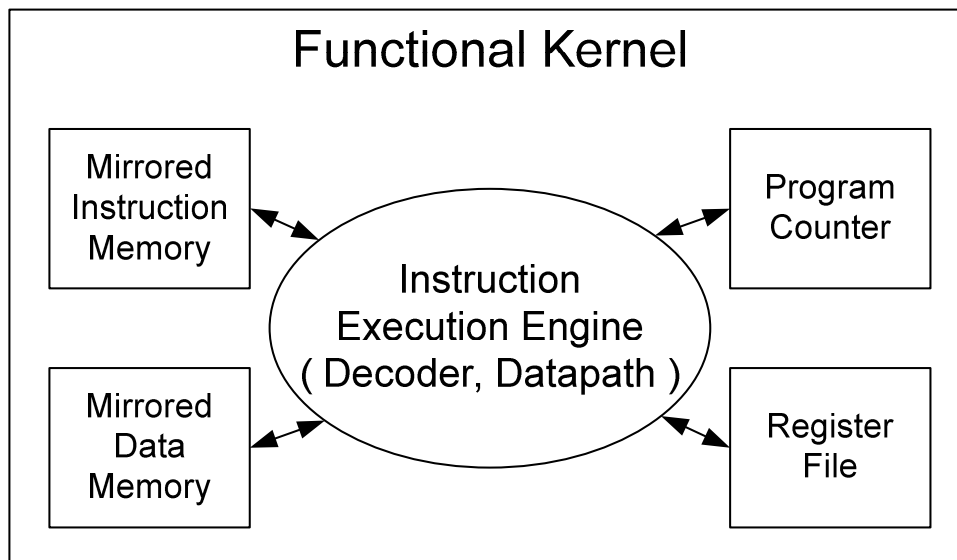


Figure 8. The components of functional kernel

Note that the contents of registers inside the kernel cannot be directly accessed outside the model. Due to the untimed nature, they usually get updated earlier than they should be in the model outside as mentioned in the previous chapter. That is exactly why the timing shell is required for performing proper timing synchronization.

Because the instructions and data from memory outside the model might be timed and not immediately available, the functional kernel, which computes data as soon as an instruction is fetched, might not get correct data from outside. To solve it, we introduce mirrored instruction and data memory for the functional kernel to access. The word “mirrored” means that the two memory have the same initial and final contents with those of the outside instruction and data memory even though there might be some mismatches during simulation due to different timing domains. Before each simulation, the initial values of instruction and data need to be loaded into the mirrored memory.

To maximize the model performance, the functional kernel is implemented in pure C++ without invoking any routines provided by SystemC libraries [4]. It is applicable since the functional kernel is completely untimed. It is also worth mentioning that the kernel itself can be promoted as a PV model or even a standalone ISS just by adding a very simple software wrapper.

4.2 Timing Shell

In contrast to the functional kernel, the timing shell is completely nothing to do with the instruction evaluation. Its job is to properly communicate with the external system under a target abstraction view. The main tasks of the timing shell are: sampling the external inputs to identify incoming instructions and interrupts, querying the functional kernel and getting the execution results, and updating the outputs at the

right time based on the desired abstraction view. It can directly pass the results from the kernel to the model outside just as a PV model does; or it can perform certain scheduling to make the model behave as a cycle-approximate model (AV) or even a cycle-accurate model (VV).

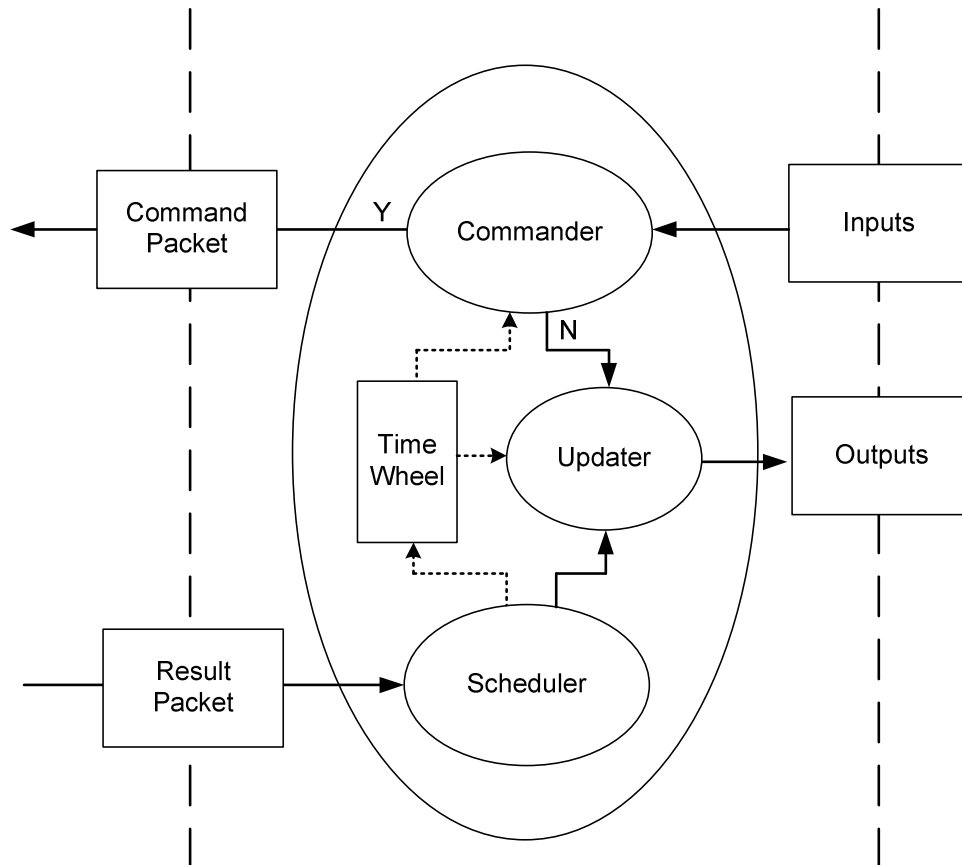


Figure 9. The components of timing shell

Figure 9 presents the architecture overview of the timing shell. At the beginning of a cycle, the timing shell checks the processor state and interrupt inputs to determine whether this cycle should be stalled. For a non-stall cycle, in which the processor needs to execute a new instruction, the shell issues a command packet (step or interrupt) to the kernel and receives a result packet from the kernel. Then a scheduler inside the shell is responsible for correctly scheduling output update events based on

the information carried in result packets. For a stall cycle, on the other hand, there would be no activity between the kernel and the shell.

4.2.1 Time Wheel

In the previous chapter, we have introduced the modeling flow of the timing shell and the mechanism called “scheduler” used to make an updating schedule, which is actually a time wheel table in the timing shell as shown in Figure 9. The time wheel is adopted to record the future update events in coming cycles. Each time slot in a time wheel represents a specific real cycle in the system and carries the necessary information to properly update the outputs in that cycle. In addition, it also records whether the cycle should be stalled possibly due to a branch instruction or a detected hazard. The information mentioned above is written into the time wheel by the scheduler. It is not uncommon that update events carried by a result packet are placed into several slots since virtually all processors nowadays are pipelined. As time advances, the time wheel also moves one cycle ahead, the events recorded in the most recent slot are carried out, and thus the corresponding output signals are updated accordingly.

A time wheel may look like in Figure 10. Each row in the table means a time slot and contains information about whether to invoke the functional kernel, which is indicated by the first element, and information about which model output needs to be updated and their new values. A pointer which tells the current time slot will move down circularly in every clock cycle. The commander and the updater can then access the current time slot in the time wheel for the information they need.

	IA: 0x14	R6: 0x0		
Current Time Slot →	IA: 0x18	R3: 0xFF	R4: 0x8F	
S	DA: 0x20	R1: 0x78		
S	R2: 0x12			
	IA: 0x1C	R0: 0x21		

Figure 10. An example of time wheel

4.2.2 Scheduler

The scheduler is another key component inside the timing shell. It must know all implementation details, such as instruction cycle timing, actions taken in each pipeline stage, control/data dependency among various types of instructions, hazard detection principles, and forwarding/stall mechanism, of the modeled processor. That is, it must have complete knowledge about exactly how in reality the modeled processor schedules all output update events in cycle-accurate fashion.

Taking a simple arithmetic instruction executed on 5-stage ACARM9 processor as example, if the instruction is in the pre-fetch stage during cycle i , then the result should be write back to the register in cycle $i+5$, as shown in Figure 11a.

i	IA: 0x20	← The address of the arithmetic instruction
$i+1$		
$i+2$		
$i+3$		
$i+4$		
$i+5$	R2: 0x57	← The result is written back

Figure 11a. Scheduling result of an arithmetic instruction

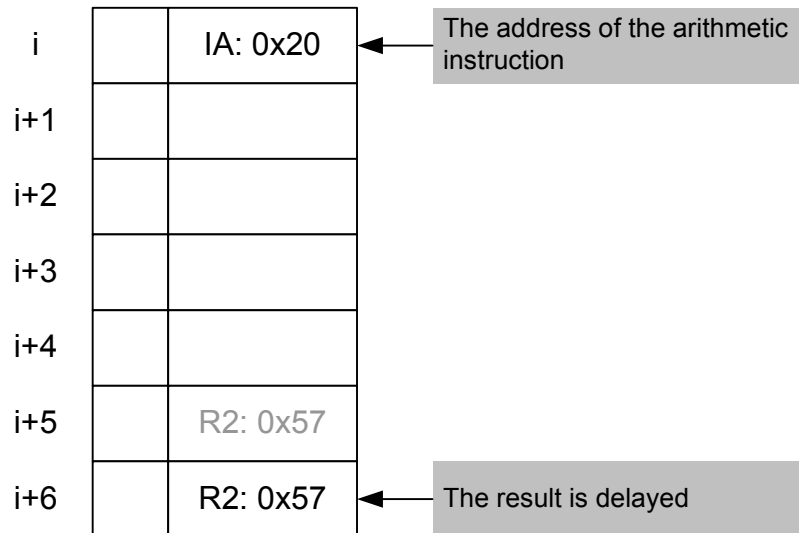


Figure 11b. Scheduling result of an arithmetic instruction with data hazard

In fact, the timing of an instruction may be affected by other instructions in some situations such as data hazard or resource competition. If the previous instruction is a load instruction whose result will be used by the arithmetic instruction, then a data hazard happens and the result of the arithmetic instruction will be delayed 1 cycle as in Figure 11b. To correct compute the effect caused by data hazard and resource competition, the scheduler needs to reserve the information of a number of previous instructions such as instruction type and registers used.

4.2.3 More Examples

Figure 12 gives an example which shows how a branch instruction (located at address 100 with target address 120) gets executed by an ACARM9 cycle-accurate model created by the proposed technique. A branch instruction is not able to stop the processor from fetching the next two consecutive instructions because it cannot change the instruction fetch flow until the third pipeline stage. However, these two fetched instructions are eventually invalidated by the processor, which results in two stall cycles.

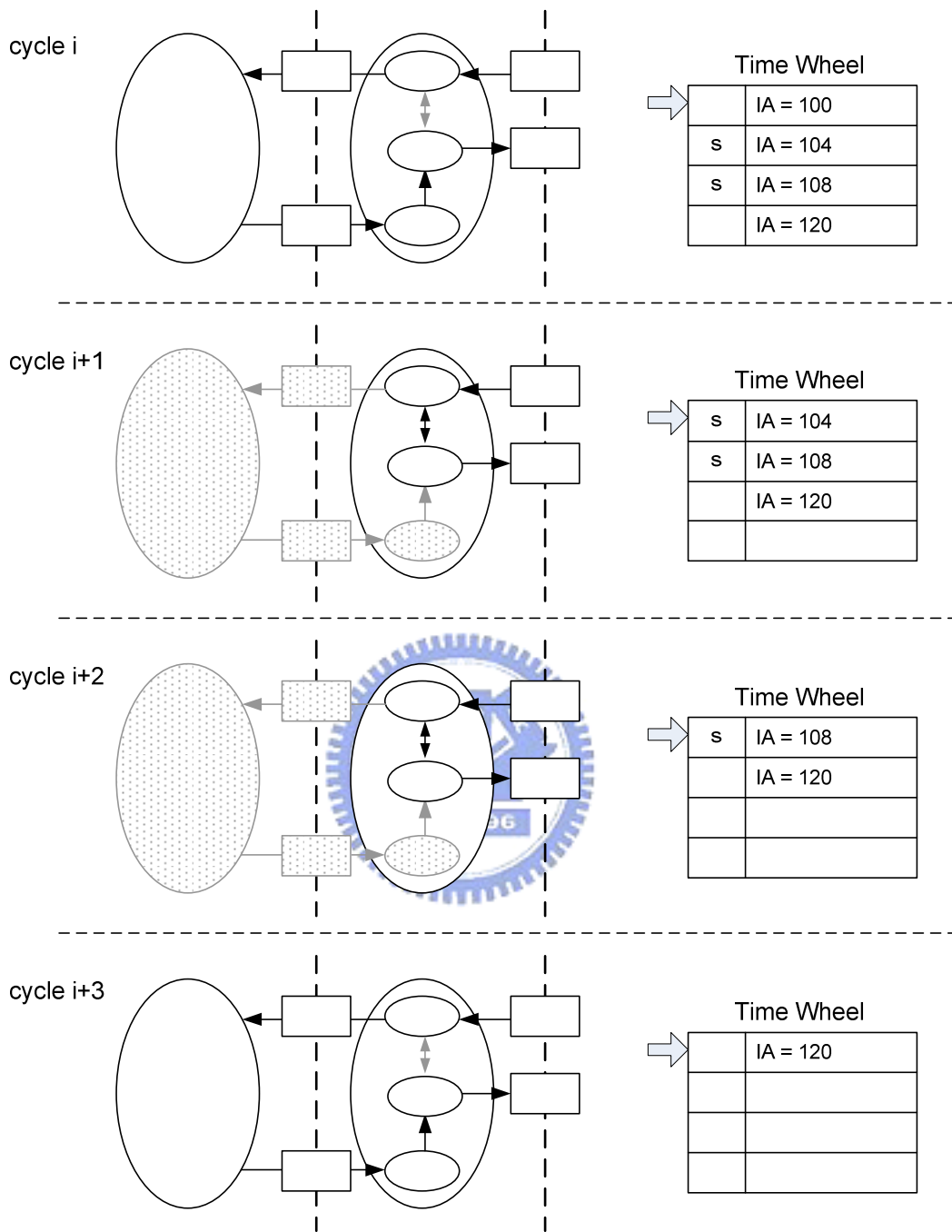


Figure 12. The execution steps of a branch instruction

As shown in Figure 12, the branch instruction is pre-fetched in cycle i. Because it is a valid instruction, the shell issues a command packet and gets a result packet in return. Then three events describing that the values on the instruction address bus (IA)

should be 104/108/120 for cycle $i+1/i+2/i+3$ are inserted into the time wheel. In addition, both cycle $i+1$ and $i+2$ are marked as stall cycles as explained. It indicates that in those two cycles the instruction is still fetched as usual in the first place. But the commander instantly finds it invalid based on the mark recorded in the time slot and therefore no longer sends it to the kernel for execution. Notice that even though a slot is marked as a stall cycle, there could still be some output update events, which are placed in there from earlier cycles. Hence, even in a stall cycle, the shell still has to update those model outputs accordingly though there is no request to the kernel and of course no response from the kernel in that specific cycle.

Figure 13 gives another example of a load instruction, which locates in instruction address 200, leading an arithmetic instruction in address 204 and there exists data hazard between them. The load instruction is pre-fetched in cycle i and its result, which modifies R1 to the value 100, is scheduled in WB stage that is the time slot representing cycle $i+5$. After the load instruction is executed, the scheduler reserves the information about this instruction for the timing computation of the following instructions. In cycle $i+1$, the arithmetic instruction is pre-fetched and the scheduler detects a data hazard due to the previous load instruction according to the information it just reserved and marks an stall cycle for the 2nd ID stage of the arithmetic instruction which is the time slot representing cycle $i+4$. Besides, the result of the arithmetic instruction, which should modifies R2 to the value F in cycle $i+6$, is also delayed by 1 cycle to the time slot representing cycle $i+7$. Then the model keeps go through without stall cycles until cycle $i+4$, in which a stall mark is detected by the commander. In the stall cycle, the functional kernel won't be invoked to fetch any new instruction and there won't be updating of instruction address (IA) as a result.

Finally in cycle $i+5$, the result of the load instruction is updated and because no stall mark is detected in the cycle, the timing shell resumes the execution of the functional kernel to fetch the next instruction.



Chapter 5 Experimental Results

To evaluate the performance of the models created by the proposed technique, we have implemented PV models and VV models using SystemC for an in-house three-stage ARM7TDMI-like processor and an in-house five-stage ARM9TDMI-like processor, which is ACARM7 and ACARM9, and compare these models against their Verilog RTL counterparts. The benchmark programs adopted in our experiments are part of MiBench [5], which is a popular benchmark suite aiming at general embedded applications. Seven programs from varied categories, as shown in Table 4, are selected for extensive analysis and comparisons, and their execution cycle counts for ACARM7 and ACARM9 are also attached.

Table 4. Benchmark programs used in experiments and their cycle counts

Category	Benchmark Program	ACARM7	ACARM9
Auto/Industrial	bitcount	9088644	7996554
	qsort	12075954	12366045
Consumer	jpeg	8796271	7356096
Office	stringsearch	2916981	2503915
Network	dijkstra	12075954	9883181
Security	sha	16815477	13308257
Telecomm.	CRC32	10011396	8101486

Meanwhile, Cadence NC-Verilog (NCV) is chosen as the simulator for RTL models. All PV and VV models are compiled using SystemC 2.2.0 library offered by OSCI [6]. Additionally, when verifying an RTL model, it is highly desirable to have a golden cycle-accurate model that can be co-simulated within the same environment for fast on-the-fly instant result comparisons. Hence, after adding Verilog wrappers,

our VV models are also evaluated under Cadence NCSC co-simulation environment. The experimental results are presented in Table 5 and 6.

Table 5. Performance of ARM7TDMI-like models

	RTL@NCV	VV@NCSC	VV@OSCI	PV@OSCI
bitcount	1	11.43	31.76	773.53
jpeg	1	11.45	31.82	868.97
CRC32	1	10.23	29.07	858.06
dijkstra	1	10.94	31.51	968.57
qsort	1	8.51	23.94	727.08
sha	1	10.53	29.55	808.93
strsearch	1	11.86	31.32	1037.50
Avg.	1	10.71	29.85	863.23

Table 6. Performance of ARM9TDMI-like models

	RTL@NCV	VV@NCSC	VV@OSCI	PV@OSCI
bitcount	1	11.77	31.17	784.85
jpeg	1	12.25	32.80	816.67
CRC32	1	11.92	31.71	893.75
dijkstra	1	11.59	31.49	960.00
qsort	1	10.86	27.64	808.51
sha	1	11.44	30.34	796.43
strsearch	1	10.75	34.82	1075.00
Avg.	1	11.51	31.42	876.46

The performance of each configuration given in the tables is normalized to that of RTL simulation (RTL@NCV). The results suggest that on average the VV model created by the proposed modeling technique is about 30 times faster than the RTL model in a pure SystemC environment (VV@OSCI). Note that it is even faster than the cycle-approximate PV model, which is only 18 times faster, presented in [3]. Moreover, the VV model is about 11 times faster than the RTL model in a hardware co-verification environment (VV@NCSC). It apparently confirms that building a VV model in a higher level language with higher abstract view (SystemC) is a fairly good idea in terms of simulation performance, verification, and model encryption.

Here, we emphasize again that the same functional kernel is actually used for the PV models of both processors and is implemented without invoking any routines provided by SystemC libraries for achieving highest possible performance. The experimental results report that on average the PV model can even simulate almost three orders faster than the RTL model (PV@OSCI). This makes our PV model very attractive in software development and system-level verification.

Chapter 6 Conclusions and Future Works

Models in different abstraction views are widely demanded in current ESL design methodology for analysis, development, and verification of software and/or hardware. It is not uncommon that several models with varied abstraction levels are needed in a project. How to correctly build these models in a short time is becoming a critical issue today.

In this thesis, we propose a processor modeling technique that partitions the cycle-accurate model into two layers, the functional kernel and the cycle-based timing shell, where the functional kernel acts as an untimed ISS (or a PV model) while the timing shell provides detailed cycle-based timing information. In this way, the functional kernel can be shared within an entire processor family with a same ISA, and only a customized cycle-based timing shell is required for a processor. Therefore, not only the model development time can obviously be reduced but also the chances of functional inconsistency among processors can be greatly minimized.

Finally, the experimental results reveal that our VV model is 30/11 times faster than the RTL model in a SystemC/co-simulation environment, respectively. Our cycle-accurate VV model is even faster than the cycle-approximate AV model presented in an existing art. Our PV model can simulate about 860 times faster than the RTL model. These results repeatedly highlight the efficiency of models created by the proposed two-layered modeling technique. For the current version of model, the interrupts and memory aborts, which may need to flush the execution result in earlier stage, can't be simulated by the proposed modeling architecture and they will be our advanced research in the future works.

References

- [1] F. Bacchini, G. Smith, A. Hosseini, A. Parikh, H. T. Chin, P. Urard, E. Girczyc, and S. Bloch, “Building a common ESL design and verification methodology – is it just a dream?” Design Automation Conference, pp. 370–371, Jul. 2006.
- [2] Open SystemC Initiative (OSCI), “The SystemC community,” <http://www.systemc.org/>, 2006.
- [3] Y.-J. Lu, C.-T. Lin, C.-F. Wu, S.-A. Hwang, and Y.-H. Lin, “Microprocessor modeling and simulation with SystemC,” IEEE Int’l Symp. on VLSI Design, Automation, and Test, pp. 1–4, Apr. 2007.
- [4] T. Rissa, A. Donlin, and W. Luk, “Evaluation of SystemC modelling of reconfigurable embedded systems,” Conf. on Design, Automation and Test in Europe, vol. 3, pp. 253–258, Mar. 2005.
- [5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: a free, commercially representative embedded benchmark suite,” IEEE Int’l Workshop on Workload Characterization, pp. 3–14, Dec. 2001.
- [6] Open SystemC Initiative, <http://www.systemc.org>.