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

Chien-De Chiang and Juinn-Dar Huang
Department of Electronics Engineering
National Chiao Tung University, Hsinchu, Taiwan
xander@acar.ee.nctu.edu.tw, jdhuang@mail.nctu.edu.tw

***Abstract*—In this paper, we propose a new processor modeling technique that partitions a cycle-accurate model into two layers, an inner functional kernel and an outer timing shell. The kernel is an untimed but high-speed instruction set simulator (ISS) and is suitable for software development; while the timing shell provides additional timing details for cycle-accurate hardware behavior. When a new processor member is added to the family, it demands only a new 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.**

I. 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 I [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.

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.

TABLE I 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.

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 later 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 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.

In this paper, 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. Fig. 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.

The rest of this paper is organized as follows. Section II presents the two-layered architecture for the cycle-accurate model. Section III discusses the implementation concerns and details. The extensive experimental results are reported in Section IV and Section V gives the concluding remarks for the paper.

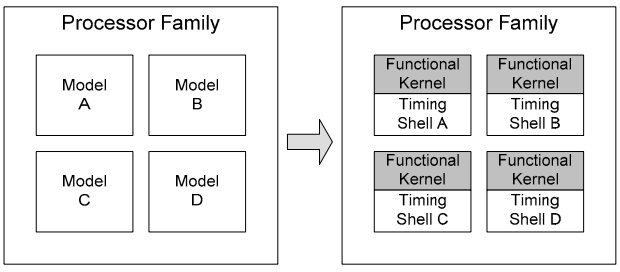

Fig. 1. Models partitioned into a functional kernel and timing shells.

II. TWO-LAYERED MODEL ARCHITECTURE

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 an untimed instruction set simulator (ISS), while the timing shell provides cycle timing details and communicates with the outside world. The proposed two-layered architecture for cycle-accurate model is shown in Fig. 2.

The timing shell samples input signals from the outside of the model, which include clock, reset, interrupts, instruction bus, data input bus, and so on. Then it takes proper actions 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*. 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. Finally, the timing shell updates the output signals, containing user-visible registers, address bus, data output bus, and so on, using the values obtained from the functional kernel at the appropriate cycles.

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 result packet 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 II. 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.

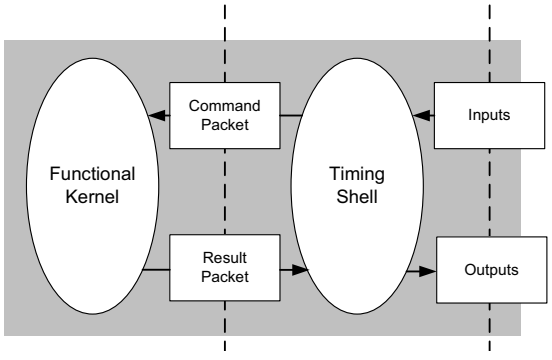

Fig. 2. Two-layered architecture for cycle-accurate model.

TABLE II CONTENTS OF RESULT PACKET

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

III. MODEL IMPLEMENTATION

A. 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 Fig. 3. 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.

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. That is exactly why the timing shell is required for performing proper timing synchronization.

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.

B. 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).

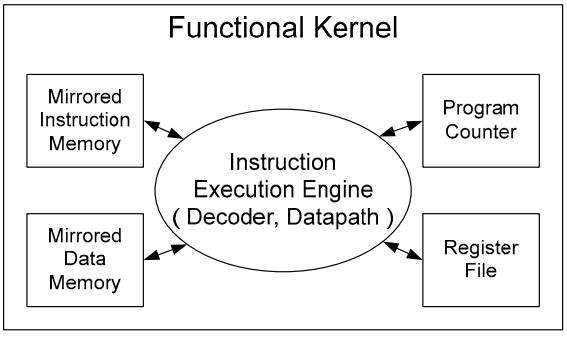

Fig. 3. The architecture of functional kernel.

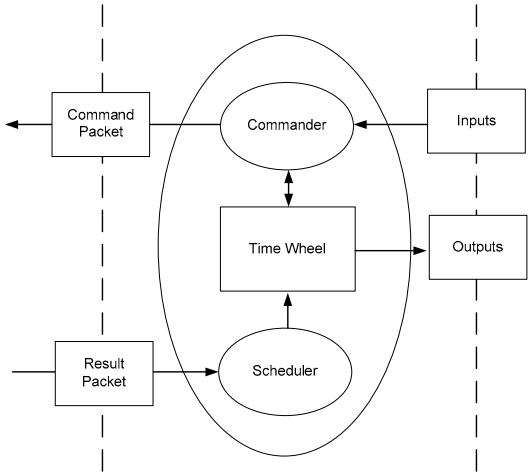

Fig. 4. The architecture of timing shell.

Fig. 4 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.

A *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.

Fig. 5 gives an example which shows how a `branch` instruction (located at address `100` with target address `120`) gets executed by an ARM9TDMI-like 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. As shown in Fig. 5, the `branch` instruction is 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.

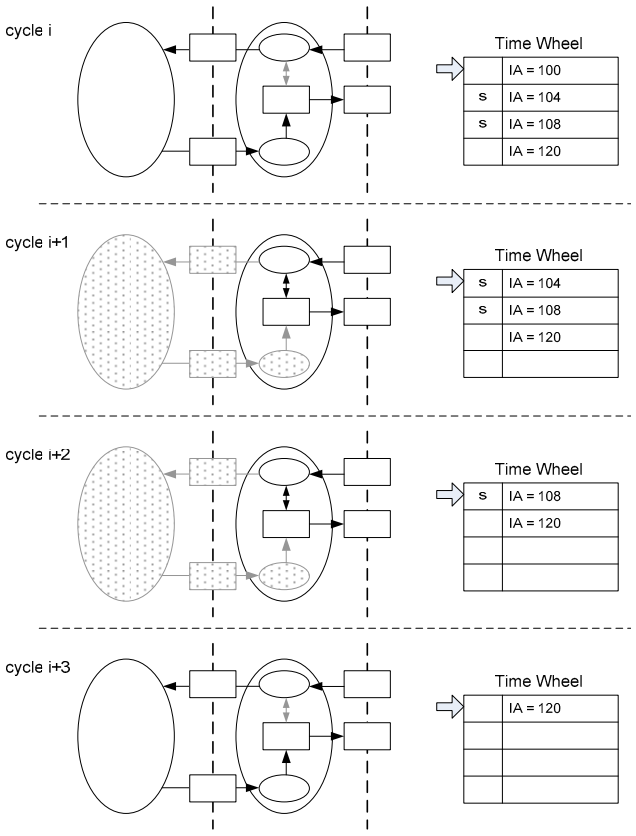

Fig. 5. The execution steps for a `branch` instruction.

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.

By doing so, a scheduler can correctly place each of the update events contained in a result packet returned by the kernel into the right slot (cycle). For example, if an `add` instruction is fetched by an ARM9TDMI core in the current cycle and there are no branches and hazards detected, then the destination register should be updated and presented at the output exactly 4 cycles later though the new value is already known (via the result packet) now. Of course, the scheduling task can be far more complicated than that described in the previous example if interrupts, branches, multi-cycle instructions, and data hazards are all mixed up in a scenario.

IV. 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, 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 III, are selected for extensive analysis and comparisons.

TABLE III BENCHMARK PROGRAMS USED IN EXPERIMENTS

Category	Benchmark Program
Auto/Industrial	bitcount, qsort
Consumer	jpeg
Office	stringsearch
Network	dijkstra
Security	sha
Telecomm.	CRC32

TABLE IV 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 V 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

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 IV and V.

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.

V. CONCLUSION

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 paper, we propose a processor modeling technique that partitions the cycle-accurate model into two layers, the functional kernel and the timing shell, where the functional kernel acts as an untimed ISS (or a PV model) while the timing shell provides detailed timing information. In this way, the functional kernel can be shared within an entire processor family with a same ISA, and only a customized 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.

ACKNOWLEDGEMENT

This work was supported in part by the Ministry of Economic Affairs, Taiwan, R.O.C., under Grant 94-EC-17-A01-S1-038, and the National Science Council of Taiwan under Grant NSC 95-2220-E-009-006.

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.