

Automatic Simulation and Verification of Pipelined Microcontrollers

ING-JER HUANG AND LI-RONG WANG

*Institute of Computer and Information Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan 804, R.O.C.
E-mail: ijhuang@cie.nsysu.edu.tw*

This paper presents a methodology for automatic simulation and verification of pipelined microcontrollers. Using this methodology, we can generate the simulation for the instruction set architecture (ISA), abstract finite state machine (FSM) and pipelined register transfer level design and compare the simulation results across different levels quickly. We have implemented our method in the simulation and verification of a synthesized microcontroller HT_4 using our behavioral synthesis tool.

Keywords: functional verification, simulation, high level synthesis, microcontrollers, microprocessors

1. INTRODUCTION

Due to time-to-market pressure and the lack of appropriate CAD tools and design methodologies, designers of industrial embedded controllers often miss opportunities to systematically analyze the architecture properties of their designs and explore hardware and software alternatives for future updates. If they want to validate their design, they have to develop a simulator especially for that product. This is time wasting and not economical. Therefore, we have a problem:

- How can we automatically generate simulators for a synthesized microcontroller?

On the other hand, the simulation time increases as the design complexity increases. Therefore, it is critical to verify the functionality of the design and eliminate errors at an early stage in the design process. Correction of functional bugs which are alive until the final gate level simulation requires excessively large amounts of computing time and debugging effort [5]. An efficient verification methodology is needed and would be a great aid to the flow of both synthesis and simulation. We adopt a method that compares the simulation traces of different levels to verify our design. However, the size and information of the simulation results differ for each level. To provide the same benchmark, the ISA (Instruction set architecture) level simulation is instruction-cycle based while the abstract FSM (finite state machine) is state-based and becomes clock-cycle based in RTL simulation. The total execution time in FSM level simulation is 20 times greater than the execution time in the ISA level, and is only half that of RTL simulation. We then have another problem:

- How can we compare and simulate results across different design levels?

Since different simulation levels reveal different degrees of abstraction, we have to sample the right cycles. In this paper, we will present sampling algorithm and implement it in our verifier.

Cooperation with the design team for the HT48100 embedded microcontroller [1] has been established to investigate the adoption of our behavioral level synthesis tool PIPER-II and our design methodology into their design flow. Our synthesized result HT_4 is equivalent to the original design [2]. In parallel with the synthesis flow, the automatic generation of simulation processes by our simulator engine aids in finding synthesis tools or design specification errors in different levels of abstraction. Sometimes, the simulation fails, but the results of cycle-based simulation are too huge and complex for humans to check, so we use our verifier to compare the results between different levels and find out where and why the simulation has failed.

The organization of this paper is as follows. Section 2 briefly introduces our synthesis system and how it can generate simulators for the synthesized microcontroller. Section 3 describes the methodology for the verification of simulation results across different levels. Section 4 presents the experimental results. Section 5 is the conclusion.

2. AUTOMATIC GENERATION OF SIMULATORS FOR DIFFERENT LEVELS

2.1 PIPER-II

PIPER-II is the behavioral domain synthesis tool of ADAS, a full-range design automation system for microprocessors [3]. PIPER-II accepts the abstract specification of an instruction set architecture (ISA) and produces pipelined register transfer level (RTL) designs, consisting of both data and control paths. PIPER-II provides simulators for synthesized designs at the ISA level, the abstract finite state machine (FSM) level, and the pipelined RT level.

Fig. 1 illustrates the conceptual structure of the PIPER-II system. There are two major design flows in PIPER-II. At the left side is the synthesis flow, and at the right side is the simulation and verification flow.

Given the application benchmarks, our simulator engine can generate different level simulators for that microcontroller simultaneously along with simulation results for the benchmarks. We can describe them as follows:

The instruction set architecture specification is an executable specification, which can serve as the ISA simulator. The ISA simulator records the system status after the execution of each instruction in the application benchmark, including the values of the architecture registers, such as the PC (program counter), IR (instruction register) ACC (accumulator), and the memory image.

Fig. 2 shows the abstract FSM specification. We can see that both the data/control path and controller execute one step in each cycle. The control paths, shown in each ellipse, are composed of RTL descriptions; e.g., “move (pc, bus(b1), rom:memAR)” means move data in the PC into the “rom:memAR” register on the ROM. A controller step selects a transition and computes the next state. This controller is given an identifier for each state, such as bc(n), $n=1,2,\dots,N$, N is a positive integer. We can see this on the left hand side of Fig. 2.

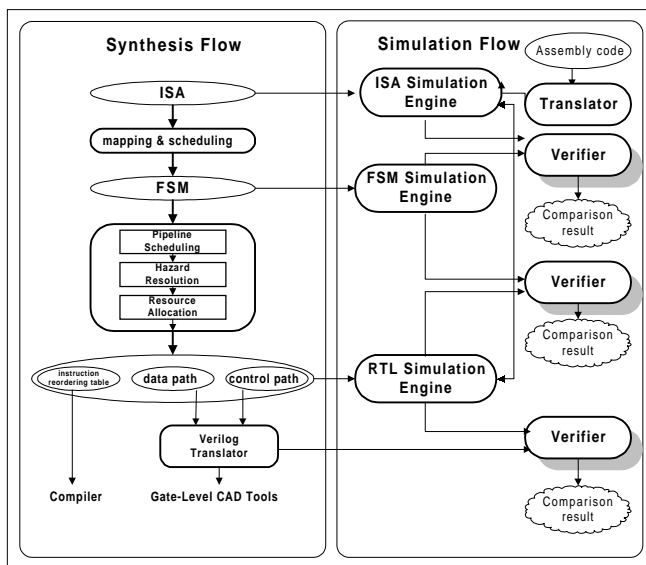


Fig. 1. The structure of the PIPER-II system.

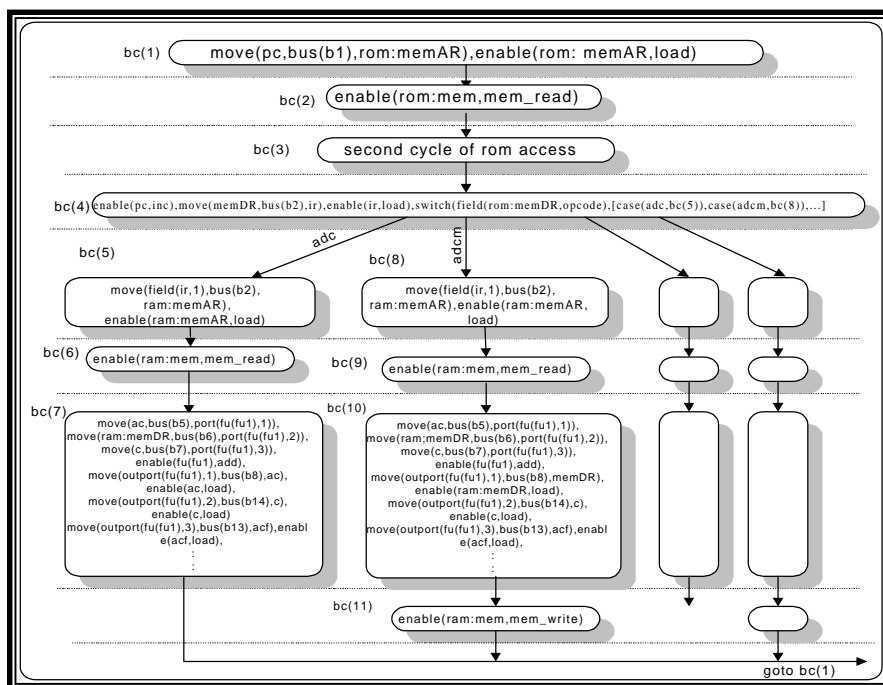


Fig. 2. The abstract FSM specification.

The abstract FSM simulation is a sequential, non-pipelined implementation of the given ISA. The abstract FSM simulator shows the cycle-by-cycle execution of RTL descriptions.

The RTL simulation is designed for the synthesized, pipelined data path and control path. In the RTL simulation, designers can observe:

- execution of state activities in each pipeline stage;
- control and data registers and memory values, and the status of function units' I/O ports, all at the cycle-by-cycle level;
- data transfer at the I/O ports for input/output operation;
- all values of control signals in each cycle.

2.2 Simulator Engine

Fig. 3 shows the structure of the pipelined RTL simulator. The PIPER-II system includes simulation engines for different levels of simulation. Although this tool may produce different synthesized designs, the majority of the data paths for each design are similar. For example, there should be a top controller and a set of functional units such, as an adder and connection network. The top controller acts as a main processor or an ALU, and the functional units act as a co-processor; the connection network includes data transfer between two registers. We simulate the behavior of these primitive functional units and put these simulation rules into the “general library”.

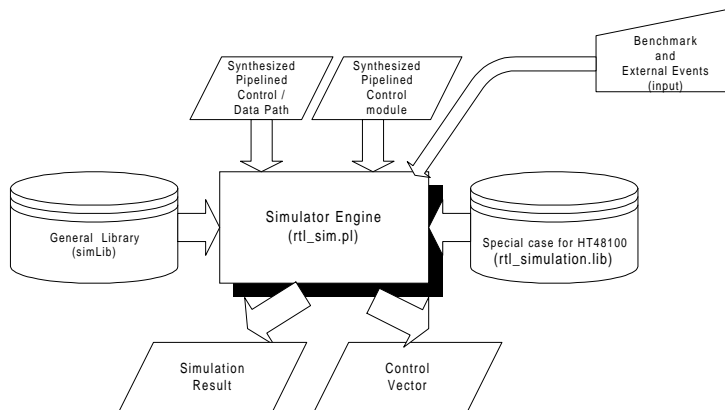


Fig. 3. The pipelined RTL simulator.

Each microcontroller may have special functional units. Some microcontrollers may have the Watch Dog Timer design while others don't. The ALU of some microcontrollers has extra functions for special instructions. To address these application specific functions, the designer has to provide support simulation rules for these special cases and put them in the “special_case” library for the synthesized microcontroller.

Combining the two files mentioned above, the user's application benchmark, and the synthesized data / control path, we can get the RTL simulation for that benchmark using the synthesized design. We can get the ISA level and the abstract FSM level simulation for the same benchmark in the same way.

Then, the designer should provide the user's application benchmark, synthesized data / control path, and user-defined behavior of some special operations for the microcontroller's function units in the "special_case.lib". We can get the ISA level simulation and the abstract FSM level simulation in the same way.

3. VERIFICATION OF SIMULATION RESULTS ACROSS DIFFERENT LEVELS

3.1 Compare System Status Between Two Levels

This verification process is divided into two parts:

1. After execution of an application benchmark, the verifier first compares the final system status of the simulation results across different levels and finds incompatible data if any exist.
2. Then, the verifier compares the system status of the simulation results across different levels after execution of each instruction.

Fig. 4 shows the framework of our verifier. Besides the compared simulation trace from different levels, the verifier needs information from the synthesis design to sample the correct data. We will describe the sampling method in the following.

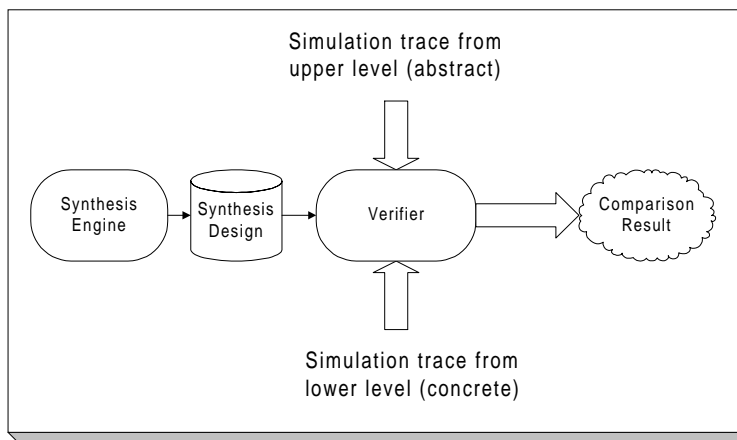


Fig. 4. Compare system status between two levels.

3.2 The Sampling Algorithm

In Fig. 5, in the ISA level simulation (at the top of the Fig. 5), each slot is an instruction execution, but in the abstract FSM levels (in the middle of the Fig. 5), each slot is a state execution in one clock cycle. However, in the pipelined RTL simulation, each slot represents the execution of several states in one cycle, and more than one instruction may execute during one cycle. Since different simulation levels reveal different degrees of abstraction, we have to sample, for each simulation, data in the right slots (cycles). This may sound straightforward. However, internal machine states may be skewed due to the

pipeline hazard resolution mechanism. The skewed states make sampling complicated. Fig. 5 shows an example of skew behavior in which the instruction ‘Inst 3’ is delayed by 4 cycles (one pipeline stage) in the RTL simulation trace, causing sampling of the correct state data of the instruction to be delayed by 4 cycles (as opposed to the normal position).

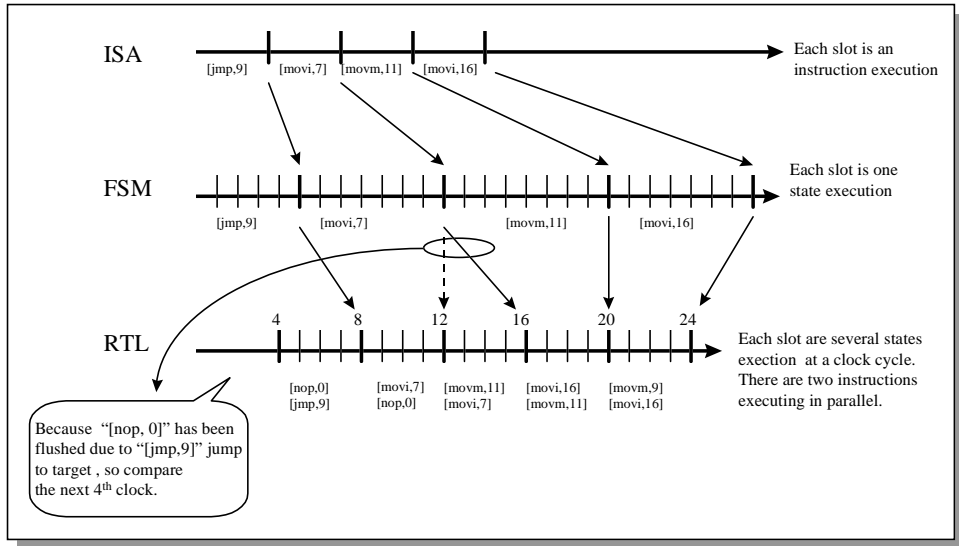


Fig. 5. Comparison across different levels after executing each instruction.

Fig. 6 shows that the algorithm for sampling the correct state data corresponds to the execution of each instruction in the FSM level simulation. Fig. 7 shows that the algorithm for sampling the correct state data corresponds to the execution of each instruction in the pipelined RTL simulation, taking the skew into consideration.

Symbol Definition

- Σ_{ISA} : A set is extracted from the ISA level simulation trace, comprised of the ordered elements $\{I_1, I_2, \dots, I_n\}$, and “n” is the total number of instruction cycles for running a benchmark. For each element, I_k ($1 \leq k \leq n$) records the system status after the k^{th} instruction execution.
- Σ_{FSM} : An input set is extracted from the abstract FSM level simulation trace, comprised of the ordered elements $\{S_1, S_2, \dots, S_m\}$, and “m” is the total numbers of states for executing a benchmark. Each element S_k ($1 \leq k \leq m$) records the system status after the k^{th} state execution.

1. **Initialization:** An input FSM level simulation records set $\Sigma_{FSM} = \{S_1, S_2, \dots, S_m\}$. An output FSM level simulation records set Σ_{FSM} sampled set $O_{FSM} = \phi$.
2. $\Sigma_{FSM} = \{S_k, S_{k+1}, \dots, S_m\}$ and the output set $O_{FSM} = \{F_1, F_2, \dots, F_j\}$
For $i = k$ **to** m
 search the first element S_i , the Next-state of S_i is
 “bc(1)” and append S_i to the output set O_{FSM} as F_{j+1} ,
 $\Sigma_{FSM} = \Sigma_{FSM} - \{S_k, S_{k+1}, \dots, S_i\}$
3. Repeat step 2 until Σ_{FSM} becomes an empty set and output the set O_{FSM} .

Fig. 6. Algorithm for sampling the FSM simulation results.

1. **Initialization:** An input ISA level simulation records set $\Sigma_{ISA} = \{I_1, I_2, \dots, I_n\}$. A pipelined RTL simulation records set $\Sigma_{RTL} = (T_1, T_2, \dots, T_p)$. An output sampled pipelined RTL simulation records set $O_{RTL} = \phi$. A temporary set $A = \phi$. Fire latency is L .
2. Sample the temporary set A :
 - a. $\exists Q, Q > 0$, Q is the minimal integer such that $(Q * L) \geq \text{MAX}$. Append $T_{Q * L}$ to set A , indexed as the first element A_1 .
 - b. **For** $i = 1$ **to** (p/L) , select $T_{(Q+i)*L}$ from Σ_{RTL} and append these elements into A indexed as A_2, A_3, \dots, A_g .
3. Template := I_1
4. Template = I_k , $A = \{A_h, A_{h+1}, \dots, A_g\}$, $O_{RTL} = \{R_1, R_2, \dots, R_j\}$
 In set A ,
 If Template is a branch-taken instruction execution record
 Then lookup this instruction set new PC value at its Z^{th} time period and append A_h to set O_{RTL} as R_{j+1} ,
 $A = A - \{A_h, \dots, A_{h+\lceil(Z-1)/L\rceil}\}$, Template := I_{k+1} .
 Else If Template is a branch-non-taken instruction execution record
 If a “stall” strategy is used for this instruction, lookup this instruction and set the new PC value at its Z^{th} time period,
 Then append A_h to set O_{RTL} as R_{j+1} , $A = A - \{A_h, A_{h+1}, \dots, A_{h+\lceil(Z-Cd)/L\rceil}\}$, Template := I_{k+1}
 If a “flush” strategy is used for this instruction
 Then append A_h to set O_{RTL} as R_{j+1} , $A = A - \{A_h\}$, Template := I_{k+1}
 Else If Template is a non-branch instruction execution record
 Then append A_h to set O_{RTL} as R_{j+1} , $A = A - \{A_h\}$, Template := I_{k+1}
 Else ERROR message, stop sampling!
5. Repeats step 4 until the Template = I_n and output the set O_{RTL} .

Fig. 7. Algorithm for sampling the FSM simulation results.

- Σ_{RTL} : An input set is extracted from the pipelined RTL simulation trace, comprised of the ordered elements $\{T_1, T_2, \dots, T_p\}$, and “p” is the total clock cycles for running a benchmark. Each element T_k ($1 \leq k \leq p$) records the system status, including several instructions which execute several instructions in the k^{th} clock cycle.
- **MAX**: From a synthesized microcontroller design, we can get the maximum number of clock cycles, “MAX”, needed to complete (including fetch, decode, execute and WB stages) an instruction.
- **L**: fire latency, which means that L clock cycles are needed to issue the next instruction.
- C_d : an instruction is decoded in the “ C_d ”th time period ($1 < C_d < \text{MAX}$) for every instruction.
- **Z**: a branch instruction sets its branch target in the “Z”th time period.
- O_{FSM} : an output set of sampled FSM simulation records.
- O_{RTL} : an output set of sampled RTL simulation records.

In PIPER-II, each state in the FSM synthesis results is given an identifier (refer Fig. 2). For each instruction, execution starts at the common initial state “bc(1)”. It is obvious that in the FSM level simulation trace, the execution state before the state bc(1) is the end state of the prior instruction. We can scan the elements from S_1 to S_m in the Σ_{FSM} , select all elements S_i whose next-states are bc(1), and append S_i to the output set O_{FSM} sequentially.

The second algorithm includes a method to select the element at the correct clock cycle in the pipelined RTL simulation, which corresponds to the end of an instruction execution in the ISA level.

Step 2 creates a temporary set A in a subset of Σ_{RTL} . The longest instruction needs MAX cycles to finish, and Σ_{RTL} is an ordered set. First, we select the element T_{Q*L} , where “Q” is the minimal number such that $Q*L \geq \text{MAX}$ and the first instruction must be completed at the $(Q*L)^{\text{th}}$ clock. Then, we can sample at an interval of L clock cycles to ensure that the next instruction will finish during these time slots.

In step 3 and 4 from the first element A_1 , we select an element in set I as a template and find the corresponding element in set A each time. However, control hazards will appear when a branch instruction evaluates the target address when the succeeding instructions have fired. To solve this problem, two hardware strategies, “stall” and “flush”, are often used [4]. If a branch is taken, the fetched instruction has to be discarded and “dump” elements left in set A. After we select the exact element in set A, we also remove these “dump” elements. (E.g., in the Fig. 5, the record which is sampled at clock 12 will be discarded). If a branch is taken, we will discard “ $(Z-1)/L$ ” instructions no matter which strategy is used. This means that we have to remove “ $(Z-1)/L$ ” from set A. If the template is a non-branch instruction record or a branch non-taken instruction and the “flush” strategy is used, we just select the first element in the remaining set A and proceed with the sampling process. If a branch non-taken instruction and a “stall” strategy is used, we insert $(Z-C_d)$ bubbles when we evaluate the target address. Therefore, we have to remove “ $(Z-C_d)/L$ ” elements from set A.

An example is shown in Fig. 8. The fire latency L of this design is 1, MAX = 8. First, we construct the temporary set A. We sample at the 6th clock as A_1 and then sample at every clock. Assume that the “sz, [76]” instruction sets the new PC at its 6th clock. At cc4 ($Z = 6$), the system detects that a control conflict has happened ($C_d = 4$) and stalls the next five instructions. When we select A_1 as the valid sample corresponding to “sz, [76]” we also

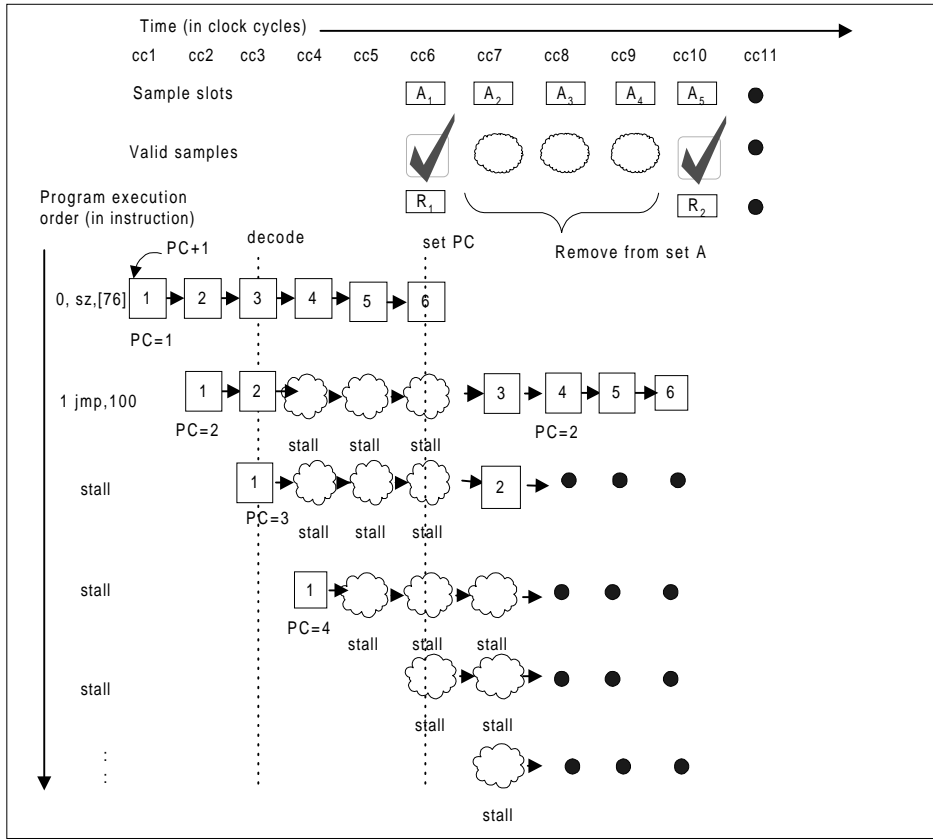


Fig. 8. Example of sampling in the RTL trace when a "stall" strategy is used.

have to remove {A₂, A₃, A₄, A₅, A₆} from set A. The valid sample of the second instruction "movi, 0" is A₇. If "sz, [76]" is not branched, "jmp, 100" will be executed. However, it has been stalled by "sz, [76]" for "Z - C_d = 6 - 3 = 3" cycles. "jmp, 100" will be completed at cc10 instead of cc7. When we select A₁ from set A, we should remove {A₂, A₃, A₄}, too. Thus, we can get a valid sample using A₅.

4. EXPERIMENTED RESULTS

We have applied our techniques to simulate and verify our synthesized design for the HT48100 microcontroller. Our synthesized result HT_4 is equivalent to the original design [2], with a pipeline firing latency of 4. (This means that two instructions are executed in parallel.) It uses the "flush" strategy to eliminate control hazards.

We performed our experiment on a SUN Ultra SPARC workstation with 128 M bytes of RAM.

4.1 Automatic Simulation and Verification of HT_4

We ran several benchmarks for HT_4, including the “LED” program, and we observed the output operations at the I/O port. We show a snapshot of the simulation trace for the benchmark “Sort” in Fig. 9. After the simulation, we used our verifier to compare the simulation results across different levels. We show a comparison of simulation results in Fig. 9, Fig. 10 and Fig. 11. Fig. 10 is a comparison between ISA and the sampled FSM records. The left side is the ISA trace and each record shows the execution order and the system status. On the right side, each data records the system status at the end state of an instruction execution. We also identified the state identifier of this sampled state. It can be seen that the values of the architected registers and memory are the same in each record between the ISA and FSM levels.

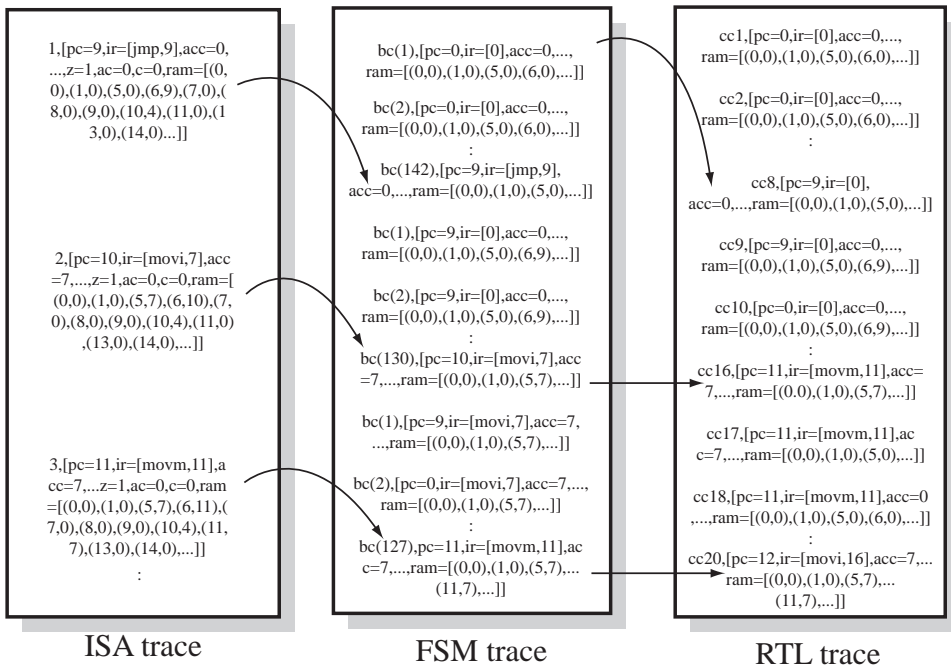


Fig. 9. The simulation results of the benchmark "Sort".

Fig. 11 is a comparison between ISA and the sampled RTL records. The left side is the ISA trace; in the middle, each data records the system status sampled by our algorithm. We also show the clock cycle when each record was sampled. We see that the first compared record was sampled at cc8. The value of IR does not correspond to the value in ISA because data of IR at cc8 was flushed by the “**jmp, 9**” instruction. Then, the verifier removed the record sampled at cc12. The next correct record was sampled at cc16. The PC and IR values at this record are not the same as the record of ISA because it was increased and fetched by the next instruction.

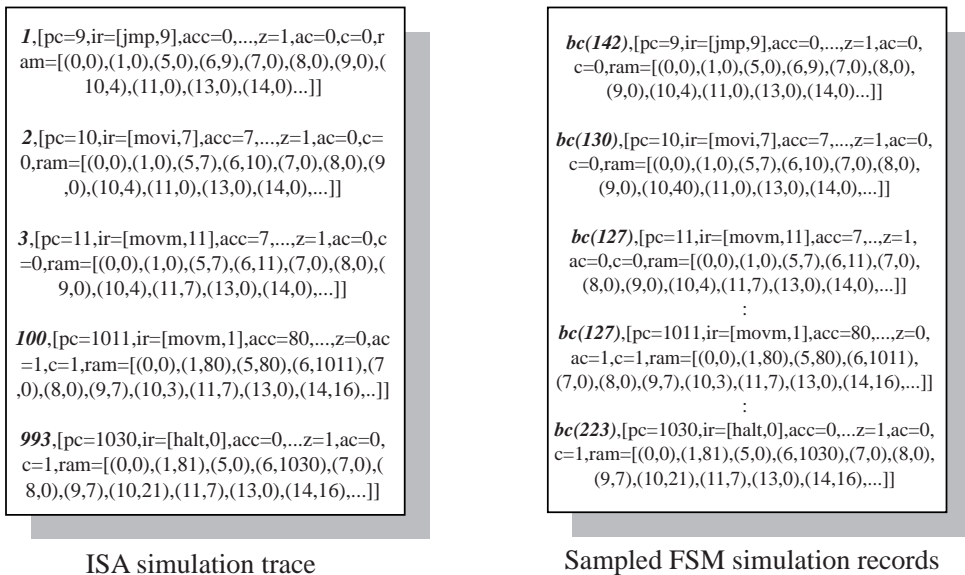


Fig. 10. Comparison of the ISA and FSM simulation results.

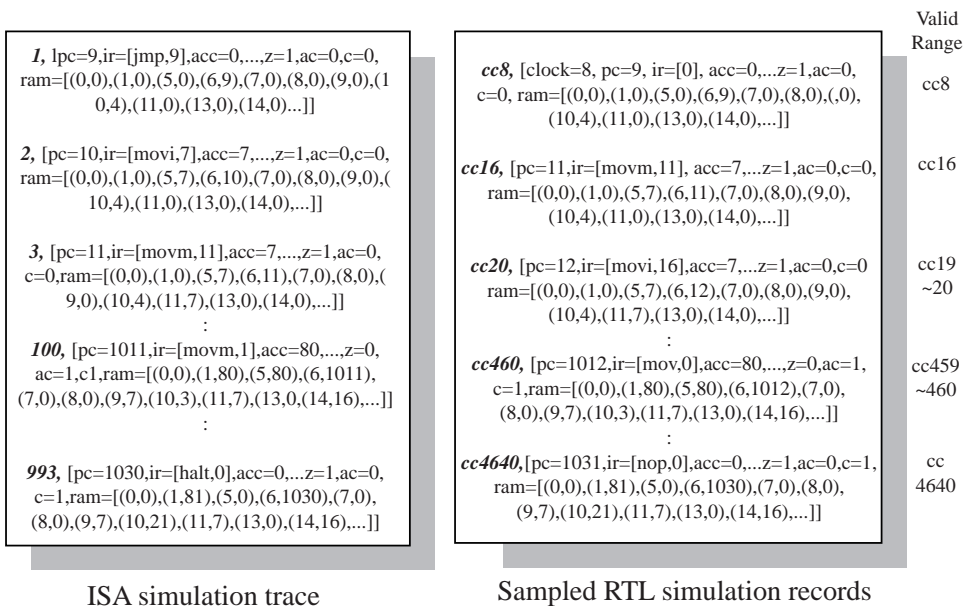


Fig. 11. Comparison between the ISA and RTL simulation results.

On the right side is the “valid range” of the sampled RTL record for each compared instruction. Using our sampling algorithm, we sampled the record every “L” (the fire latency) clock cycles to ensure that that instruction finished in this interval. However, some instruction could be completed before the sampled clock cycle. We provide the “valid range” information to show the user that the result was valid during this period.

4.2 Execution Time of Multi-Level Simulations

The execution times of three levels (behavioral, finite state machine and RTL) of simulation are shown in Fig. 12. The ratio of the execution times for these levels is 1:18:50, showing a significant difference among these levels of simulation. The designer can rely on the upper levels of simulation (behavioral and finite state machine) to quickly navigate

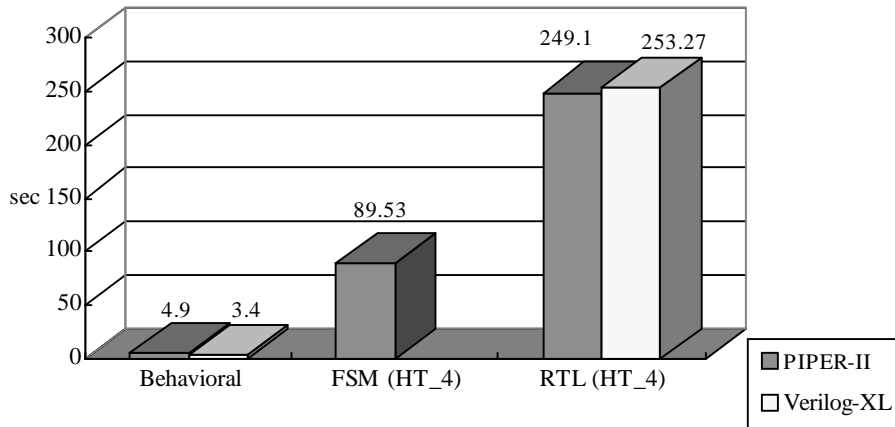


Fig. 12. Execution time of the "Sort" benchmark for different levels of simulation.

through the simulation results and identify possible buggy steps, and can then switch to RTL simulation traces for accurate investigation of the problem with the identified buggy steps.

We also compared the simulation performance of our simulator, written using the logic programming language Prolog, with the Verilog-XL simulator. Both were executed on the same UNIX platform. The Verilog design specifications are available only for the behavioral and RTL levels. At the behavioral level, our simulator is slightly slower than the Verilog-XL simulator. However, this is not a significant problem since behavioral simulation is usually very short. On the other hand, our simulator is slightly faster than the Verilog-XL simulator at the RTL level. The comparison shows that our approach is an effective one, even compared with commercial simulators, such as Verilog-XL. Since Prolog is considered a good programming language for fast prototyping but not in terms of performance, we believe our approach can achieve much better performance than Verilog-XL when the tool is implemented in more efficient programming language, such as C.

5. CONCLUSIONS

In summary, we can automatically generate an ISA level, abstract FSM level and pipelined simulator for the synthesized microcontroller, and we can get different levels of simulation for the same benchmark input. Higher level simulation helps the designer to find bugs in the synthesis results and perform re-synthesis to correct errors in the earlier design stages.

To validate the design, our tool compares the simulation traces across different levels. With the proposed sampling algorithm, we can accurately extract events to be observed at different granularity of time: per instruction, per state, and per clock cycle. The events sampled at different levels are automatically compared with each other to reveal possible design inconsistency. In addition, linkages between these events of different levels are established, so the designer can easily navigate through the simulation traces at different time granularities. We have successfully applied our approach to the verification of an industrial embedded microcontroller and identified many bugs in the original design specification, design tool and design library.

Our design tool has been found to be very helpful in debugging and verifying synthesis designs. The simulator can also be delivered to microcontroller customers to evaluate the microcontroller or integrate the simulator into their system design environments for the systems that they plan to build around the microcontroller.

REFERENCES

1. *HT 48100 Development Data Book*, Holtek Micro-electronics Inc., 1994.
2. I. J. Huang, L. R. Wang and Y. M. Wang, "Synthesis and analysis of an industrial microcontroller" in *Proceedings of Asia and South Pacific Design Automation Conference '97*, 1997, pp. 151-156.
3. I. Pyo, et al., "Application-driven design automation for microprocessor design" in *Proceedings of 29th Design Automation Conference*, 1992, pp. 512-517.
4. David A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, Inc., California, 1994.
5. A. Silburt, I. Perryman and J. Bergeron, "Accelerating concurrent hardware design with behavioral modeling and system simulation" in *Proceedings of 32nd Design Automation Conference*, 1995, pp. 528-533.
6. A. Jemai, P. Kission and A. A. Jerraya, "Embedded architectural simulation with behavioral synthesis environment" in *Proceedings of Asia and South Pacific Design Automation Conference '97*, 1997, pp. 227-232.



Ing-Jer Huang (黃英哲) received the BS degree in electrical engineering from National Taiwan University, Taiwan, R.O.C., in 1986, and the MS and Ph.D. degrees in computer engineering from the University of Southern California, U.S.A., in 1989 and 1994, respectively.

He is currently an associate professor in the Institute of Computer and Information Engineering at National Sun Yat-Sen University, Taiwan, R.O.C. His research interests include computer architecture, hardware/software concurrent engineering and design automation. He is a member of IEEE and ACM.



Li-Rong Wang (王儷蓉) received the BS degree in computer science and engineering from Yuan-Tze Institute of Technology, Taiwan, R.O.C., in 1995, and the MS degree in computer and information engineering from National Sun Yat-Sen University, Taiwan, R.O.C., in 1997. Her research interests include high level synthesis, simulation, and VLSI CAD design. She is currently an engineer in the Custom Design Department of the MXIC Co., Science-Based Industrial Park, Taiwan, R.O.C.