Mixing Symbolic and Ternary Simulation Techniques for the Verification of Processor-Based Systems

Flávio Miana[*1]
miana@cpdee.ufmg.br

Julio Cezar de Melo[*1]
demelo@cpdee.ufmg.br

Claudionor N. Coelho Jr[*2]
coelho@dcc.ufmg.br

Patricia Nattrodt[*2]
patty@dcc.ufmg.br

Antônio O. Fernandes[*2]
otavio@dcc.ufmg.br

Abstract

We present a new technique to support processor validation and verification in absence of information when modeling reactive systems. Current processor validation techniques will not tolerate absence of information for some of its registers. In order to overcome this problem we combine symbolic simulation with ternary logic simulation techniques. We exemplify our technique by simulating an Application Specific Instruction Processor(ASIP) core with its embedded logic.

1. Introduction

One of the most important tasks during the design of a processor is its validation. Processors are becoming so complex that exhaustively verifying its implementation is impractical. Therefore, it is only feasible to test exhaustively a processor when the number of possible states is small. For larger systems, the validation must be confined to some portions of the system. This is usually the reason for known bugs found in industry, such as the *Pentium* bug that was found in the *Floating Point Unit*, demanding Intel to replace millions of processors already in the market[12]; and the bug described by *Fujita et a*l[7], where a network coprocessor presented anomalous behavior after the chip was manufactured.

One of the techniques used to verify a processor behavior is symbolic simulation[2][8]. In symbolic simulation, we uniquely represent the sets of values any variable may take. Examples of verification tools using symbolic simulation can be found in Murφ[6], SMV[8] and COSMOS[11], and more recently in conjunction with process algebras[13][14] such as Circal[15] and CSP[16]. Because this technique has a prohibitive complexity, commercial tools for processor validation and verification are based on ternary logic instead.

[*1] Electrical Engineering Department - Federal University of Minas Gerais - Brazil
[*2] Computer Science Department - Federal University of Minas Gerais - Brazil

Ternary logic means a third *unknown* or *indeterminate logic value* (usually named X) is added to the binary logic set {0,1}. The third value X can be used to reduce the number of cases of the system to be tested[1] by encapsulating values that are *unknown* or *indeterminate* to the processor. Commercial simulators based on the languages *VHDL*[3] and V*erilog HDL*[4] support ternary logic. Albeit its constant use for processor validation and verification, loosing information for some variables can invalidate completely the simulation. For example, consider the following skeleton for a processor behavioral simulator, where branches are handled based on an external condition.

```
IR = Mem[PC];
  ....
    switch (IR) {
  ....
      case branch :

          if (CC)
            PC = PC +2;
          else
            PC = PC + offset;
     .....
```

Figure 1. Handling *branch* with *Condition Code*

In Figure 1, let us consider CC as a condition code dependent of an external value. Therefore, CC can have one of the three values {0,1,X}, defined in the ternary logic. Assume CC equals to X while the simulator fetches the next instruction. Since X represents an unknown value, the simulator cannot determine the next value for the PC register. As a result, the simulation will become invalid because the simulator will propagate X to the memory and to the all registers of the processor, making the simulation results useless.

This paper presents a mechanism to support processor validation in presence of indeterminate states or input values, as defined in ternary logic. For such models, some registers of the processor cannot tolerate the loss of information, such as the PC in the previous example. These registers will be treated symbolically. As we are going to show later, this method can be very useful for the simulation of reactive systems implemented by processor cores with surrounding logic, where the logic simulation may generate indeterminate states to the processor during the simulation, and the processor must process them appropriately.

Figure 2 presents one of the properties of reactive systems that we want to verify. Given a state **S1**, the system stays in this state until the occurrence of an event **ev,** which changes the system's state to **S2**. For such systems, we are going to show that the simulation converges to a valid state even in absence of information for some time.

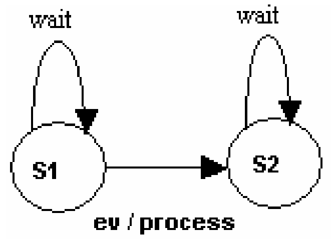


Figure 2. Reactive System Model

This paper is organized as follows. In Section 2, we provide a mathematical background, important to understand the concepts used in the rest of the paper. We introduce a new technique that can be used to verify processor-based systems in Section 3. An example is presented in Section 4 to illustrate the application of the proposed approach. Finally, Section 5 concludes the paper and suggests some future work.

2. Mathematical Background

We present in this section a brief summary on the theory of posets and how they can be used for dealing with absence of information in processor-based systems according to definitions presented in [1][5].

We denote sets by **A** and **B** and individual elements of the sets by *a* and *b*. The *cartesian product* **A** x **B** of the two sets **A** and **B** is the set of all ordered pairs (*a*, *b*), where $a \in$ **A** and $b \in$ **B**. A *binary relation on* a set **B** is any subset of **B** x **B**. Let **R** be a binary relation on **B**, i.e., **R** $\subseteq$ **B** x **B**. We say that **R** is *reflexive* if and only if (iff) *a***R***a* for all $a \in$**B**. Similarly, **R** is *antisymmetric* iff *a***R***b* and *b***R***a* implies $a = b$ for all $a, b \in$ **B**. Finally, **R** is *transitive* iff *a***R***b* and *b***R***c* implies *a***R***c* for all $a, b, c \in$ **B**. A binary relation on **B** which is *reflexive*, *antisymmetric*, and *transitive* is called a *partial ordered* on **B**.

A *poset* (partially ordered-set) is an ordered pair $\langle$**S**$, \sqsubseteq \rangle$, where **S** is a set and $\sqsubseteq$ is a partial order on **S**. Intuitively, we will view a partial order as ordering the values by their “information content”. That is, elements less than others “contain less information”.

If $\langle \mathbf{S}, \sqsubseteq \rangle$ is a *poset*, $\mathbf{A} \sqsubseteq \mathbf{S}$, and $b \in \mathbf{S}$, then b is a *lower bound* of **A** iff $b \sqsubseteq a$ for all $a \in \mathbf{A}$. A *lower bound a* of **A** is called *greatest lower bound* of A, written glb(**A**), iff $b \sqsubseteq a$ for every *lower bound b* of **A**. The concept of *upper bound* and *least upper bound* of **A** , written lub(**A**), are defined dually. If $\mathbf{A} = \{a,b\}$, we will write glb(a,b)(lub(a,b)) rather than glb({a,b})(lub({a,b})). Clearly, if glb(**A**) exists, it is unique, and the same holds for lub(**A**).

Mapping f: $\mathbf{A} \rightarrow \mathbf{B}$ consists of a function f assigning an element b from the codomain **B** to each element a of its domain **A**, written as $b = f(a)$.

Given a *poset* $\langle \mathbf{S}, \sqsubseteq \rangle$ and mapping $f : \mathbf{S} \rightarrow \mathbf{S}$, we say that f is *monotone* iff

$$a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$$

This monotonicity definition is consistent with our use of information content. If a mapping is monotone, we cannot “gain” any information by reducing the information content of the arguments to the function.

We can apply the concept of *partial ordered set* to $\Gamma=\{0,1,X\}$ in order to formalize the concept of *unknown value*. Assuming the partial order $\leq$ on Γ as $a \leq a$ for all $a \in \Gamma$, $X \leq 0$ and $X \leq 1$, we can show in Fig. 3 the Hasse diagram of partial order.

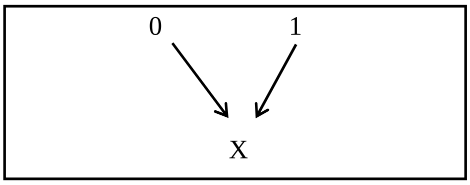


Figure 3. The ≤ partial order

We can extend the theory of ternary logic in digital circuits to word-level systems by the following definitions.

Definition 1 : Assume a word can take the values in the set $\mathbf{A} = \{\alpha_0, \alpha_1, \alpha_2, ... \alpha_{n-1}\}$, where n is the number of the elements on **A**. A partial ordering representing the absence of information can be defined in the following way. For all $\alpha_i \in \mathbf{A}$, $\alpha_i \leq \alpha_i$ and $X \leq \alpha_i$.

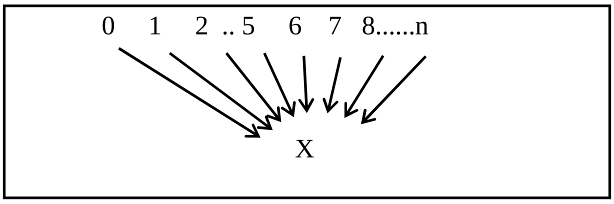


Figure 4. The ≤ partial order on **A**

In Figure 4, we present the partial order for **A** = {0, 1, 2,..., n}. In this figure, we can see that glb(α_i, α_j) = X if i ≠ j, implying that if two different values are possible for a single variable, the variable looses its information content. The accuracy for information loss can be improved by the following definition.

Definition 2 : Consider a set **B** = {β_0, β_1, β_2,... β_{n-1}} where *n* is the number of the elements on **B** and each β_i can be represented by the binary encoding $r_{m,i}r_{m-1,i}...r_{0,i}$, $r_{k,i} \in \{0,1\}$. For each $r_{k,i}$, $r_{k,i} \leq r_{k,i}$ and X ≤$r_{k,i}$, for k ∈ [0,m].

For example, assume we represent each number of the set **B** = {0, 1, 2, 3} with two bits, i.e, β_0 = 00, β_1 = 01, β_2 = 10, β_3 = 11.

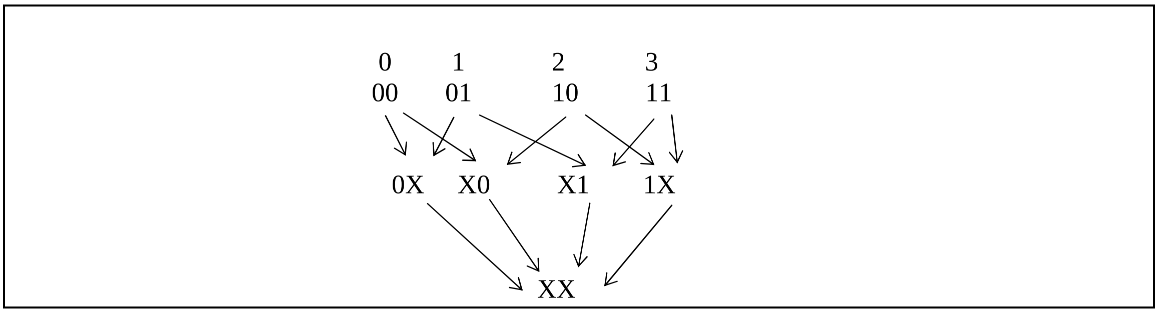


Figure 5. The ≤ partial order on **B**

In Figure 5, we can see that the *greatest lower bound* of **B** is given as following:

$$glb(\beta_i, \beta_j) = glb(r_{mi},r_{mj}).\ glb(r_{mi-1},r_{mj-1})...\ glb(r_{0i},r_{0j})$$

In *Definition 2,* the concept of partial-order applied to elements in set **B** on bit-level generates new elements containing less information than original numbers. We can see that in some elements there are some bits with less information than others. In this case, the "content information" of the elements diminishes gradually.

These definitions are useful when modeling incompletely-specified systems. In the next section, we show an approach in order to allow symbolic simulation of portions of processors in ternary based simulators.

3. Processor Symbolic Simulation with Ternary Logic

In the first section, we presented two known approaches used to simulate processors. The symbolic simulation is very powerful because each symbolic variable represents a set of different conditions to the processor. However, it becomes prohibitive due to its complexity for larger systems.

On the other hand, ternary logic simulation can cover many conditions of the system, though it becomes impractical in some cases because the simulator cannot tolerate loss of information for some of the variables during the simulation. Once there is a loss of information, the system monotonically propagates this loss.

Considering the advantages of these techniques, we developed a simulator for the validation of processor-based systems mixing symbolic simulation with ternary logic. The registers and the memory are simulated according to the ternary logic that makes the simulation feasible by allowing loss of information. The Program Counter (PC) is simulated using symbolic techniques, since it is a critical register and cannot tolerate the loss of information. Thus, the state of the program being executed is dependent on the PC, i.e. we must maintain the current state of the program being simulated in terms of its internal registers and memory for each PC.

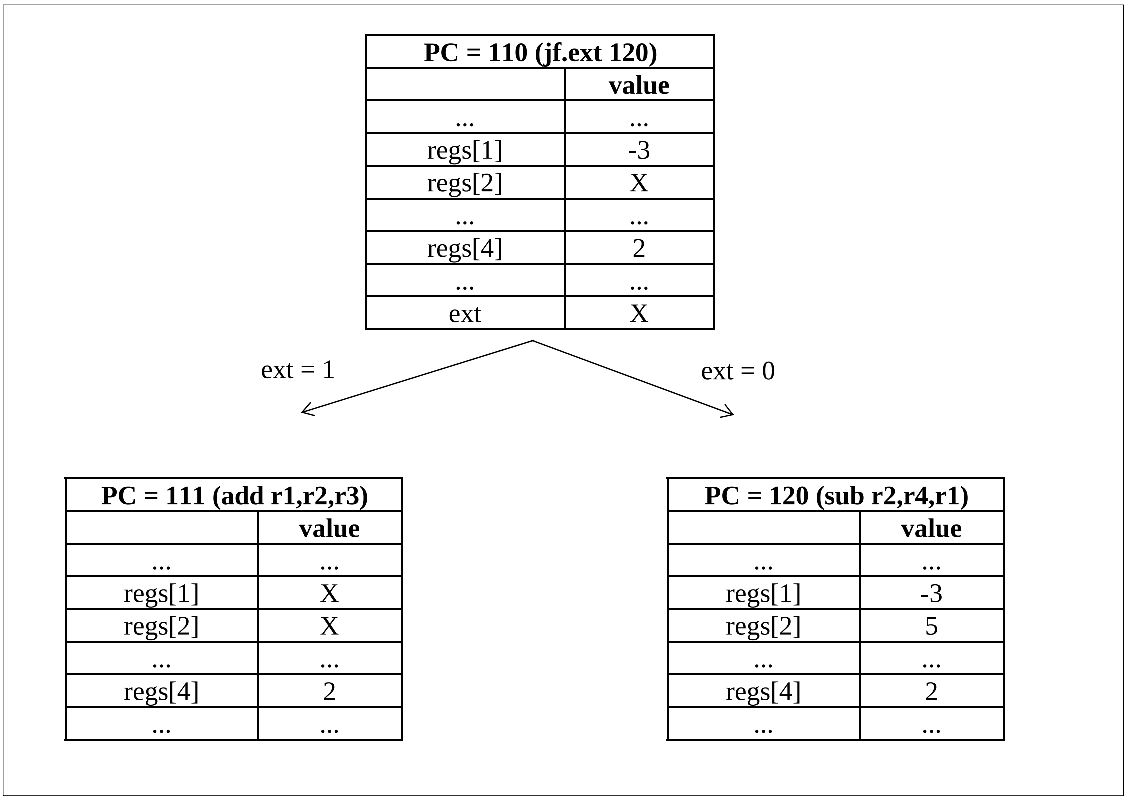
In order to exemplify the simulator behavior, assume the following *assembly code* for some hypothetical RISC machine:

PC	Assembly Code
100	ld r1,0(r2)
...	...
110	jf.ext 120
111	add r1,r2,r3
112	ldi r3,#4
...	....
120	sub r2,r4,r1
...	....

Table 1. Assembly Sample Code

On address 110, the jf.ext represents a branch if an external condition *ext* is false. The other instructions belong to common RISC instruction sets, as given by[10].

When PC =110, we find the branch instruction defined previously. If the condition *ext* is *unknown*, the simulator forks its state into two parts as follows.



PC = 110 (jf.ext 120)	
	value
...	...
regs[1]	-3
regs[2]	X
...	...
regs[4]	2
...	...
ext	X

PC = 111 (add r1,r2,r3)	
	value
...	...
regs[1]	X
regs[2]	X
...	...
regs[4]	2
...	...

PC = 120 (sub r2,r4,r1)	
	value
...	...
regs[1]	-3
regs[2]	5
...	...
regs[4]	2
...	...

Figure 6. Forking States

Figure 6 presents a partial snapshot of the state following the execution of the code when PC = 110. Assume that *regs* represents the set of general-purpose registers of the processor core and that the external input *ext* has an undefined value prior the execution of the branch instruction. In addition, below each box labeled by the PC, we present the partial state after the instruction was executed. In this figure, we see that when instruction labeled by PC = 111 is executed, the values for *regs[1]* and *regs[2]* becomes undefined. Nonetheless, when the instruction labeled by PC = 120 is executed, *regs[1]* and *regs[2]* have defined values.

The simulation continues in these two ways regardless of each other. The forking situation can occur whenever the decision to be made by the processor is indeterminate. The simulation system increases its reliability since more information about each possible state is available. We discuss how this information can affect the behavior of the system in the next two subsections.

3.1Visibility of Symbolic Simulations in Ternary Environments

Symbolic simulations in ternary environments means that the system behaves as a ternary model externally. However, in the core simulator there may be many states being simulated. The Figure 7 illustrates this model.

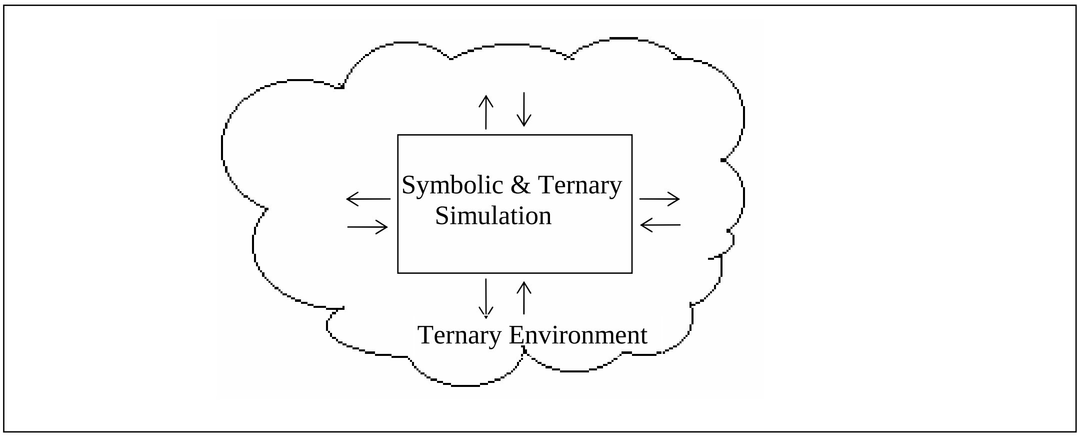
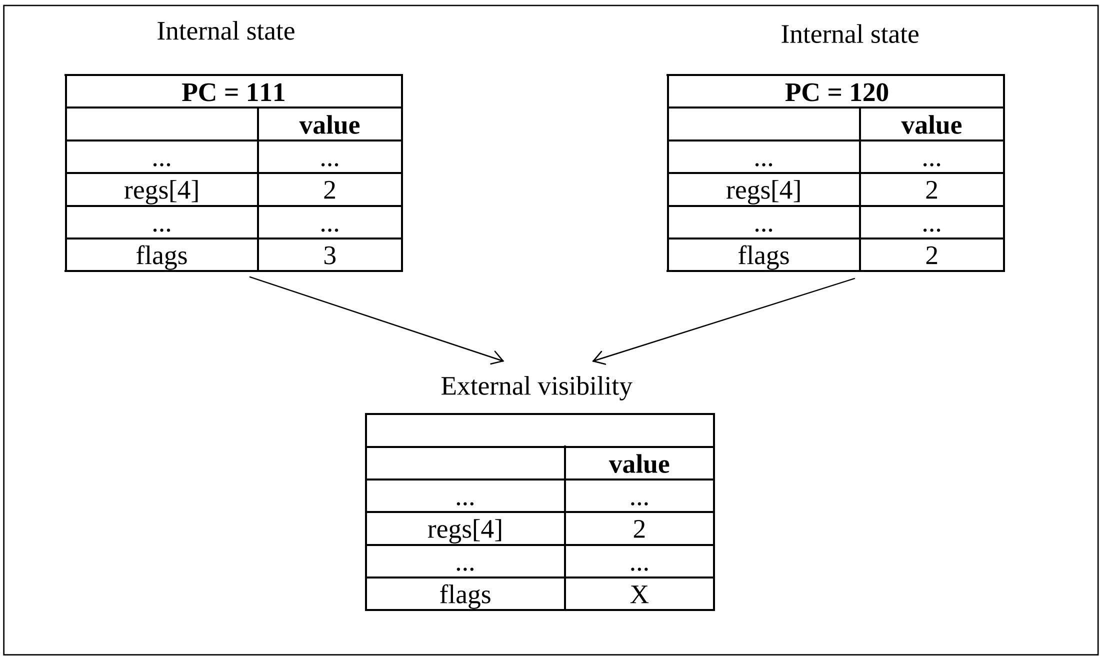


Figure 7. Simulation Environment

In this way, symbolic simulation is transparent to the ternary environment. In the previous example, a snapshot of the processor simulation can be found in Figure 7.

In this simulation, we can maintain the external visibility of each processor state by reducing the information content of the state for all PCs being simulated in any cycle. . This can be achieved if we apply glb(state[PC_1],...,state[PC_n]).

Let us consider Figure 8, for example. The processor *flags* is externally indeterminate since it assumes different values for $flags_{PC=111}$ and $flags_{PC=120}$. Note also that the value for *regs[4]* is determinate, since it has a value 2 regardless of the PC value. From this external visibility, we can obtain information of the state of the system tracing the simulation step by step and verifying its correctness.



Internal state

PC = 111	
	value
...	...
regs[4]	2
...	...
flags	3

Internal state

PC = 120	
	value
...	...
regs[4]	2
...	...
flags	2

External visibility

	value
...	...
regs[4]	2
...	...
flags	X

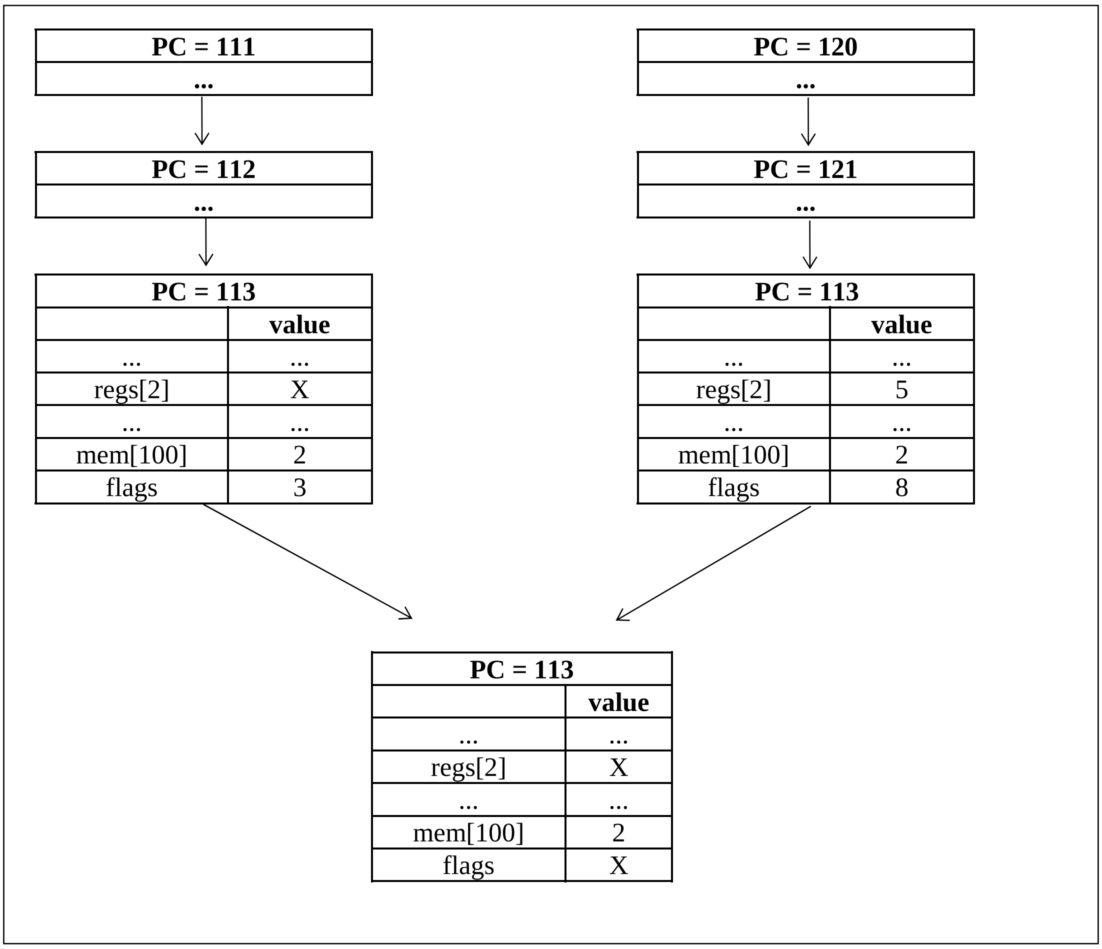
Figure 8. Snapshot and External Visibility

As mentioned before, internally we have a symbolic simulation for the PC. This approach has a constraint when considering the possible number of forks that may be generated by the third unknown logic value. In the next subsection, we show a mechanism used to overcome this problem.

3.2 Limiting the Exponential Complexity of Symbolic Simulations

Symbolic simulation is based on unique coding for variables rather than on actual values for the design under simulation. Thus, it is possible to simulate entire classes of values in a single run. In large systems, this approach may be expensive due to its exponential complexity. In such cases, it would be important to find a way to reduce this complexity.

We can minimize the number of cases to simulate by reducing the information content when different states exist for a single PC. In this case, we use internally the glb function to collapse the states. Figure 9 presents a snapshot where two different states appear for a single PC.



PC = 111
...

PC = 112
...

PC = 113	
	value
...	...
regs[2]	X
...	...
mem[100]	2
flags	3

PC = 120
...

PC = 121
...

PC = 113	
	value
...	...
regs[2]	5
...	...
mem[100]	2
flags	8

PC = 113	
	value
...	...
regs[2]	X
...	...
mem[100]	2
flags	X

Figure 9. Snapshot and Internal Visibility

By restricting the number of cases to be simulated to the different PCs that may be alive at any time, we constrain the simulation time to at most the ROM (code) size on each cycle.

In the next section we provide an example of applications running on a simulator which uses the techniques proposed earlier.

4. Simulation of Application-Specific Instruction Processors (ASIPs)

We developed a prototype using a 16-bit RISC core to test the ideas presented in this paper. We present an ASIP in which a data-acquisition system is embedded into the architecture as follows.

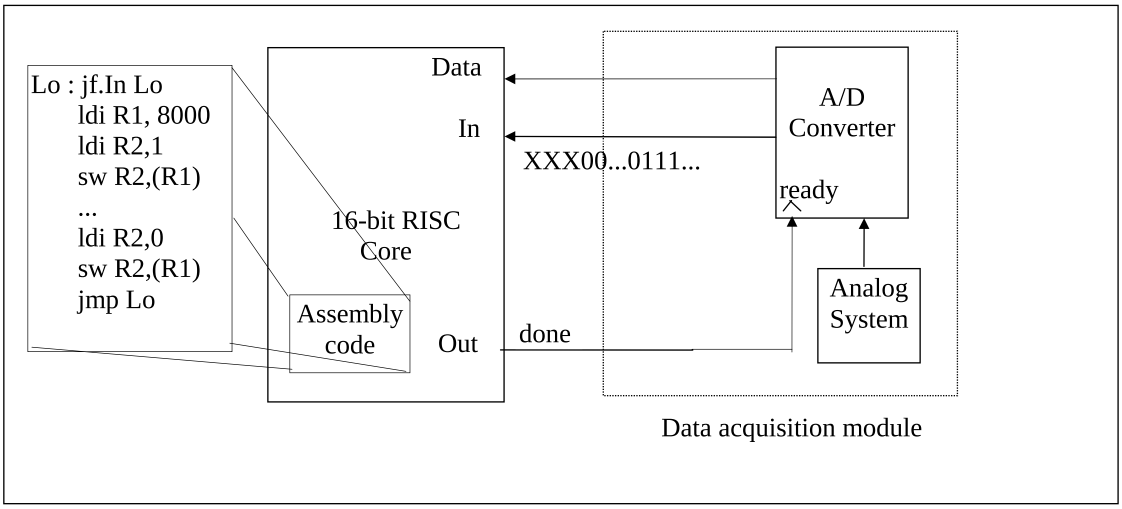


Figure 10. Co-Simulation of HW/SW for ASIP

In Figure 10, we can see that the simulator controls the data-acquisition by setting the ready port in the A/D converter. The core reads and process the value on Data port when In port is set by A/D converter. As a result, the Out port is set or not depending on the data processing performed by the simulator.

Allowing the presence of undefined values, e.g X, we can verify and validate the HW/SW and propagate accordingly X value from the logic simulation into the processor using our technique. For example, assume the RISC assembly sample used to process data from the data acquisition module is given by the code on the left portion of Figure 10, and that PC is set initially to *Lo*. Also, consider the address of Out port as 8000 initially set to 0 and a value X coming into the core via the external input In port for 3 cycles (representing for example that it may take from 1 to 3 cycles to complete a data conversion). Applying symbolic simulation to the PC register according to the ideas presented in Section 3, we can see that the simulation continues normally and after some time, the value on the Out port will have a defined value again. Thus, we can verify and validate the processor in absence of information regardless of the value that is propagated to the PC register in the processor core.

5. Conclusions

Due to its complexity, the validation of processors is a hard task and may demand as much effort as the design itself. Usual techniques for validation are based on symbolic simulation or ternary logic simulation. The former is prohibitive in larger systems due to its complexity. The later easily propagates wrong values in absence of information.

We proposed a new technique useful for validating a processor core architectures where additional logic is added to the processor. This approach takes advantage of mixing symbolic simulation and ternary logic techniques to improve the accuracy of simulation results in absence of information.

For future work, we intend to extend this technique to verify quantified temporal assertions on a specification. Also, we intend to investigate a mechanism to automatically identify variables that must be symbolically simulated.

6. References

[1] C. H. Seger and R. E. Bryant. *Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories*. Technical Report 93-08, Department of Computer Science, University of British Columbia, July 1993.

[2] R. E. Bryant. *Symbolic boolean manipulation with ordered binary-decision diagrams.* ACM Computing Surveys, pages 293-318, September 1992.

[3] R. Lipsett, C. Schaefer and C. Ussery. *VHDL : Hardware Description and Design*. Kluwer Academic Publishers, 1989.

[4] D. E. Thomas and P. R. Moorby. *The Verilog hardware description language*. Kluwer Academic Publishers, 1991.

[5] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order* Cambridge University Press, 1994.

[6] D. L. Dill, A. J. Drexler, A. J. Hu and C. H Yang. *Protocol Verification as a Hardware Design Aid*. ICCD,1992.

[7] Fujita et al. *Bug Identification of a Real Chip Design by Symbolic Model Checking*. EDAC, 1994.

[8] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan and D. L. Dill. *Symbolic Model Checking for Sequential Circuit Verification*. IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13. No. 4, April 1994.

[9] D. L. Beatty. *A Methodology for Formal Hardware Verification with Application to Micropocessors*. PhD thesis, Carnegie-Mellon University, 1993.

[10] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc.,1996.

[11] D. L. Beatty, K. Brace, R. E. Randal, Kyeongsoon Cho, and Lawrence Huang. *User's guide to COSMOS: a compiled simulator for MOS circuits.* Computer Science Department, Carnegie-Mellon University, October, 1987

[12] V. R. Pratt, *Pentium Report # bug1*, Department of Computer Science, Stanford University, 1994.

[13] Mine, G., *Formal Specification and verification of digital systems*, McGraw-Hill, 1994

[14] A. Gupta, *Formal Hardware Verification Methods: A Survey*, Formal Methods in System Design, Vol 1., No 2/3, 1992, pp.151-238

[15] G. J. Milne, *Circal and the representation of communication, concurrency and time*. ACM Trans. on Programming Languages and Systems, 7(2), 1985.

[16] C.A.R. Hoare, *Communicating Sequential Processes,* Prentice Hall International Series in Computer Science, 1985.