MODELING AND DETECTING CONTROL ERRORS IN MICROPROCESSORS

HUSSAIN AL-ASAAD
Computer Engineering Research Laboratory
Department of Electrical and Computer Engineering
University of California, One Shields Avenue, Davis, CA 95616-5294
E-mail: halasaad@ece.ucdavis.edu

JOHN P. HAYES and TREVOR MUDGE
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122
E-mail: {jhayes,tnm}@eecs.umich.edu

Design validation for microprocessors based on modeling design errors and generating tests for them is discussed. An error model for control errors is introduced and validated experimentally for a small microprocessor. A general validation approach using this model is outlined. Preliminary experimental results suggest that high coverage of control as well as data errors can be achieved using our approach.

1 Introduction

Hardware verification has long been handicapped by the absence of good high-level design error models. To be useful for design validation, error models should satisfy three requirements: (*i*) tests (simulation vectors) that cover the modeled errors should also provide very high coverage of actual design errors; (*ii*) the modeled errors should be amenable to automated test generation; and (*iii*) the number of modeled errors should be small. Several researchers [1][2][3] have proposed high-level error models that satisfy some, but not all, of the above requirements. Al Hayek and Robach [3] have adapted mutation errors from the software testing method called mutation testing, to hardware design verification in the case of small VHDL modules. Mutation testing [4] generates tests that distinguish a program under test from its mutants, where a mutant is created by injecting a small error (mutation) such as changing an add to subtract.

Recently, a set of synthetic error models that attempt to meet all the above-mentioned requirements was defined [5]. They include (*i*) *basic* errors such as a bus-order error that refers to incorrectly ordering the bits in a bus, and (*ii*) *conditional* errors that combine a basic error with a condition over a signal in the design. This paper discusses a related error model that targets control errors in microprocessor-like circuits. The model is defined in Section 2, and a mutation-based validation

approach using it is discussed in Section 3. Section 4 presents a case study based on a small microprocessor, the LC-2 [6].

2 Mutation Control Errors

A *mutation control error (MCE)* denoted (*i*,*c*,*s*,*vc*,*ve*) is a change in the control signal *s* in the cycle *c* of the instruction *i* of the microprocessor from the correct value *vc* to the erroneous value *ve*. For example, in an ADD instruction, the MCE (ADD, execute, load_flags, 1'b1, 1'b0) corresponds to incorrectly maintaining the contents of the flags in the ADD's execute cycle.

MCEs are classified by their detectability as redundant (undetectable), invalid, or testable. Of these, only testable MCEs are targeted for test generation. A *redundant MCE* for instruction *i* does not change the functions performed by *i*. The following conditions typically lead to redundant MCEs:

- *Unchanged visible state*: MCEs which do not affect the processor or memory state are redundant. These include: (*i*) reading a register or memory without storing a new result, (*ii*) loading a register or memory multiple times without reading it until some final value is loaded, and (*iii*) changing registers not visible to the instruction set, which are not used across several instructions.
- *Disabled signals*: MCEs on disabled signals are redundant. For example, an MCE that changes a select signal of a register file with a disabled read port will not affect instruction behavior.

Invalid MCEs violate usage constraints on modules, buses, or the overall microprocessor, for example:

- *Module input constraints*: These prevent inconsistencies such as: (*i*) reading and writing to memory in the same clock cycle, and (*ii*) setting the select bus of a 3-input multiplexer to 11.
- *Bus constraints*: These are bus usage rules such as: (*i*) a bus cannot have multiple drivers at the same time, and (*ii*) a bus cannot be read if it has no data source, e.g. a high-impedance bus.
- *Microprocessor constraints*: These are global operating constraints such as: (*i*) an instruction must be fetched every instruction cycle, and (*ii*) one and only one of the flags must be set.

Testable MCEs change a correct design to one with different functionality that meets all the specified design constraints. Detection of such MCEs requires instruction sequences that distinguish the correct design from erroneous ones. These sequences constitute tests for the modeled errors.

3 Validation Approach

We now outline a microprocessor validation algorithm that generates test sequences for MCEs. The microprocessor's instruction set *IS* is defined by the instruction set architecture (ISA). The design constraints *CT* are derived from the *ISA* and the bus/module usage rules. We assume that a microprocessor implementation *IM* is given that consists of a control unit and a datapath unit; the problem is to verify *IM*. Both the *ISA* and *IM* are specified by a simulatable hardware description language (Verilog in our case).

The proposed verification algorithm is described in Figure 1 in five phases. The first phase identifies all relevant control/data symbols in each instruction. For example, the 16-bit LC-2 instruction ADD DR, SR1, SR2 is represented by a sequence of (name, location, value) symbols as follows:

{(opcode,[15:12],0001), (DR,[11:9],N), (SR1,[8:6],N), (SR2,[2:0],N), (M,[5],0)}

This indicates that bits 15:12 of the instruction specify the opcode which is 0001 for ADD, bits 11:9 specify the destination register DR which is an unsigned integer (N), bits 8:6 and 2:0 specify the source registers which are also unsigned integers, and finally bit 5 is a mode bit *M* which is set to 0. (*M* distinguishes ADD DR, SR1, SR2 from the instruction ADD DR, SR1, imm5, where imm5 is a signed 5-bit constant.) This preprocessing step is based only on the microprocessor's ISA.

The second phase performs symbolic simulation of *IM* and its mutants, where a *mutant* is *IM* with a single injected MCE. For every instruction *i*, we first simulate the control unit cycle by cycle, and evaluate the resulting control signals originating from the control unit. Each such signal has the value undefined, constant, or symbolic; it is undefined if it is never assigned a value in the instruction cycle under consideration. We then simulate the datapath unit to compute the processor state at the end of the instruction cycle, and consequently determine if *IM* violates any specified design constraint. After simulating all cycles of *i*, we compute the final processor state *MSI*. For example, after simulating the ADD instruction described above, we end up with RF[DR] = RF[SR1] + RF[SR2], where RF denotes for the register file.

Next the possible MCEs are injected one at a time and the resulting mutants are simulated for all cycles of *i* to obtain the final processor state *MSM*. By checking the constraints and comparing *MSI* to *MSM*, we can determine whether the current MCE is redundant, invalid, or testable. Redundant and invalid MCEs are dropped at this stage, while testable MCEs are inserted in the error list for later test generation.

The third phase in the verification algorithm is error collapsing to reduce the number of MCEs. Dominance among MCEs in the same instruction can be estab-

Procedure MV(instruction set architecture *ISA*, constraints *CT*, implementation *IM*)

```
Phase 1   1   extract IS from ISA
          2   preprocess every instruction in IS to identify its fields

Phase 2   3   for every instruction i in IS
          4   begin
          5       for every instruction cycle
          6       begin
          7           simulate control and datapath units
          8           if any constraint from CT is violated then
          9               report {erroneous IM} and then stop
          10      end
          11      MSI := processor state in IM after simulating all cycles of i
          12      for every instruction cycle
          13      begin
          14          for every control signal c in IM
          15          begin
          16              ci := value of c in IM
          17              for every possible value cm of c not equal to ci
          18              begin
          19                  inject the MCE (i.e. set c := cm) to form a mutant
          20                  perform complete simulation of the mutant under i
          21                  MSM:= final processor state in mutant
          22                  if any constraint from CT is violated then MCE is INVALID
          23                  else if (MSI == MSM) then MCE is REDUNDANT
          24                  else add the TESTABLE MCE to error list
          25              end
          26          end
          27      end
          28  end

Phase 3   29  collapse the MCE list via dominance relations

Phase 4   30  set overall test sequence S := ϕ
          31  while there are more MCEs in the list
          32  begin
          33      select an MCE m
          34      generate an instruction sequence s to detect m
          35      remove all MCEs that are detected by s
          36      add s to S
          37  end

Phase 5   38  apply S to IM and ISA
          39  if the responses are different then report {erroneous IM}
          40  else report {correct IM}
```

Figure 1 The microprocessor validation algorithm.

lished for this purpose. An error e_1 is *dominated* by an error e_2 if any test for e_1 is also a test for e_2, in which case, e_2 can be dropped from the error list. Normally, some MCEs in cycle i of an instruction dominate others in cycle j ($i \le j$) of the same instruction.

The fourth phase of the algorithm is test generation. Applying the instruction i is generally necessary to activate an MCE affecting i. We then may need instructions that justify the processor state needed to activate the MCE, and other instructions to

propagate error values to the primary outputs of the processor.

The final phase of the algorithm applies the generated instruction sequence to both *IM* and *ISA*. If a difference is detected in the responses, the implementation is erroneous.

4 Case Study

In this section, we validate the MCE model and illustrate our validation approach by an example.

MCE validation: To evaluate the effectiveness of MCEs, an experiment was designed to apply them manually to a microprocessor called Little Computer 2 (LC-2) [6]—a simple computer used for teaching purposes at Michigan. LC-2 has a representative set of 16 instructions that are subset of the instruction sets of most current microprocessors. The LC-2 was implemented as 921 lines of Verilog code comprising a datapath unit consisting of library modules and a few custom modules, and a control unit described as a finite state machine with five states and 27 output control signals. The errors made during the LC-2 design process were systematically recorded using our error collection system [5].

The actual design errors were injected manually one at a time in the final, presumed correct design of LC-2. We then determine whether testing for all MCEs guarantees the detection of the injected design errors. This is done by deriving the detection conditions for every actual error *e* and then determining if an MCE exists that is dominated by *e*.

We applied this process to a complex actual error and we were able to find a dominated MCE for it as shown in Figure 2. The error occurs when the signal R1_temp is assigned a value independent of any condition. However, the correct implementation requires an if-then-else construct to control the signal assignment.

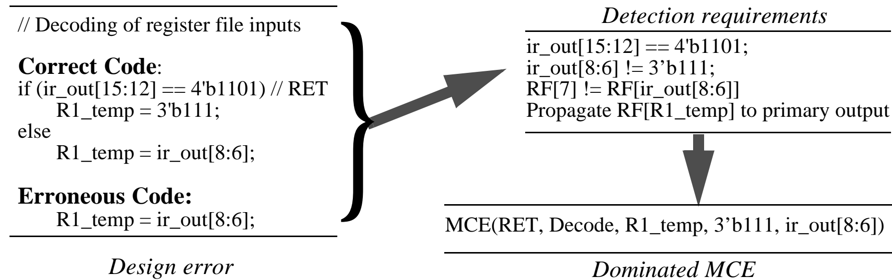

Figure 2 Example of an actual design error, its detection requirements, and the corresponding dominated MCE.

Table 1 Design errors and the number of corresponding dominated MCEs for LC-2.

	Design errors					No. of corresponding dominated MCEs
	Category [5]	**Total**	**Easily detected**	**Unde-tectable**	**Testable**	
Control Unit	Expression error	2	0	0	2	2
	Bit width error	1	1	0	0	0
	Missing assignment(s)	3	0	0	3	3
	Wrong constant(s)	1	0	0	1	1
	Unused signal	1	0	1	0	0
	Always statement	1	1	0	0	0
Datapath Unit	Wrong signal source(s)	3	0	0	3	1
	Bit width error	2	2	0	0	0
	Unused signal	1	0	1	0	0
	Wrong module	1	0	0	1	1
	Total	**16**	**4**	**2**	**10**	**8**

We analyzed manually all design errors in the test implementation of the LC-2 and the results are summarized in Table 1. A total of 16 design errors were found, nine in the control unit and the rest is in the datapath unit. Four of these errors are easily detected by the Verilog simulator, two are redundant, and the rest are testable. We can infer from Table 1 that all testable design errors in the LC-2 control unit are detected after simulation with tests for eight MCEs, and only two testable errors in the datapath unit are not guaranteed to be detected. However, by analyzing their detection requirements, we found that the probability for these two errors being undetected or masked is extremely low.

Approach illustration: To illustrate our validation methodology, we apply it here to the LC-2 instruction ADD DR, SR1, SR2. We define the state of the LC-2 microprocessor as all its storage elements, including the program counter (PC), instruction register (IR), memory-address register (MAR), flags register (FLAGS), register file (RF), and temporary registers (REG1 and REG2). The LC-2's initial state is thus (PC_0, IR_0, MAR_0, $FLAGS_0$, RF_0, $REG1_0$, $REG2_0$). Table 2 shows the control signal values in the implementation for the ADD instruction and the corresponding datapath actions. For every possible MCE *m*, we injected *m* in the implementation to form a mutant that is manually simulated to determine the type of *m*. The ADD instruction has a total of 58 MCEs of which 18 are testable. To reduce the number of testable MCEs, dominance relations among MCEs are used. Of the 18 testable MCEs, only one can be removed by dominance.

In generating a test sequence for the MCEs of an instruction *i*, we first target MCEs in the last cycle of *i* with the hope that other MCEs in earlier cycles of *i* are detected by the generated sequence. The specifications of LC-2 give the starting PC address as 3000H. So, we start our PC value with a number larger than 3000H to give

Table 2 Simulation of the instruction ADD DR, SR1, SR2: control signal values and corresponding datapath actions.

Simulation results	Instruction cycles		
	1: Fetch	**2: Decode**	**3: Execute**
Control signal values	read_mem_bar := 1’b0 write_mem_bar := 1’b1 load_pc_bar := 1’b1 RE1 := 1’b0 RE2 := 1’b0 WE := 1’b0 load_ir_bar := 1’b0 load_flags_bar := 1’b1 load_reg1_bar := 1’b1 load_reg2_bar := 1’b1 reg2_to_bus_bar := 1’b1 sel_ab_mux := 2’b00 R1 := SR1 R2 := SR2 W := DR	read_mem_bar := 1’b1 write_mem_bar := 1’b1 load_pc_bar := 1’b1 RE1 := 1’b1 RE2 := 1’b1 WE := 1’b0 load_ir_bar := 1’b1 load_flags_bar := 1’b1 load_reg1_bar := 1’b0 load_reg2_bar := 1’b0 reg2_to_bus_bar := 1’b1 R1 := SR1 R2 := SR2 W := DR	read_mem_bar := 1’b1 write_mem_bar := 1’b1 load_pc_bar := 1’b0 RE1 := 1’b0 RE2 := 1’b0 WE := 1’b1 load_ir_bar := 1’b1 load_flags_bar := 1’b0 load_reg1_bar := 1’b1 load_reg2_bar := 1’b1 reg2_to_bus_bar := 1’b1 zero_or_sign := 1’b1 sel_alu_mux := 1’b0 sel_rf_mux := 2’b00 sel_pc_mux := 2’b00 S3 := 1’b1 S2 := 1’b0 S1 := 1’b0 S0 := 1’b1 M := 1’b1 R1 := SR1 R2 := SR2 W := DR
Corre-sponding datapath actions	MEM := MEM_0 IR := MEM[PC_0] PC := PC_0 FLAGS := $FLAGS_0$ REG1 := $REG1_0$ REG2 := $REG2_0$ RF := RF_0 MAR := MAR_0	MEM := MEM_0 PC := PC_0 IR := IR_P FLAGS := $FLAGS_0$ REG1 := RF[SR1] REG2 := RF[SR2] RF := RF_0 MAR := MAR_0	MEM := MEM_0 PC := PC_0 + 1 IR := IR_P FLAGS := Detect(REG1 + REG2) REG1 := $REG1_P$ REG2 := $REG2_P$ RF[DR] := REG1 + REG2 MAR := MAR_0

```
307A:    0010 000 100000000          LD R0, 105H
307B:    0010 001 100000001          LD R1, 106H
307C:    0101 001 001 0 00 000       AND R2, R1, R0
307D:    1010 010 100000011          LDI R2, 103H
307E:    0010 000 100000000          LD R0, 100H
307F:    0010 001 100000001          LD R1, 101H
3080:    0001 001 001 0 00 000       ADD R1, R1, R0
3081:    0011 001 100000010          ST R1, 102H
3082:    0001 011 010 0 00 010       ADD R3, R2, R2
3083:    1000 010 100000100          BRZ 104H

3100:    0000 0000 0000 0110         Data = 6
3101:    0000 0000 0000 0101         Data = 5
3102:    xxxx xxxx xxxx xxxx         Storage
3103:    0011 0001 0000 0100         Data = 3104H
3104:    0000 000000000000           NOP
3105:    0000 0000 0000 0001         Data = 1
3106:    0000 0000 0000 0010         Data = 2
```

Figure 3 A test sequence for most MCEs in the ADD DR, SR1, SR2 instruction.

some space for justification of instructions, say 3080H. We generated manually the 10-instruction test sequence shown in Figure 3 to detect all 15 MCEs on control signals having constant values in the ADD instruction.

To get an idea about the total number of MCEs in the LC-2, we analyzed its instruction set and found that 430 MCEs (18.9%) are testable, 763 MCEs (33.5%)

are invalid, and 1085 MCEs (47.6%) are redundant.

5 Discussion

Our initial experimental results suggest that high coverage of data as well as control errors can be obtained by a test set for MCEs. An interesting observation is that most MCEs are either invalid or redundant—only 18.9% of the MCEs in the LC-2 are testable. This can significantly reduce the number of MCEs that need to be targeted by test generation. Moreover, the MCE model proved to be especially useful for detecting errors that involve missing logic—all 'missing assignments(s)' errors in the LC-2 control unit are covered by tests for MCEs.

The MCE error model and validation approach are, at least in principle, expandable to microprocessors with instruction pipelines, multiple instruction issue, etc. The definition of the MCE then needs to be generalized to (I,c,s,vc,ve), where I represents a sequence of one or more instructions. However, the complexity of the MCE model increases rapidly, so the applicability of this approach remains to be seen. Currently, we are working on automating our validation algorithm and extending it to more complex microprocessor types.

Acknowledgments

The research presented in this paper is supported by DARPA under Contract No. DABT63-96-C-0074.

References

[1] I. Pomeranz et al., "Generation of test cases for hardware design verification of a super-scalar fetch processor", *Proc. Int'l Test Conf.*, 1996, pp. 904-913.

[2] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors", *IEEE Transactions on Computers*, Vol. C-29, pp. 429-441, June 1980.

[3] G. Al Hayek and C. Robach, "From specification validation to hardware testing: A unified method", *Proc. IEEE Int'l Test Conf.*, 1996, pp. 885–893.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer", *IEEE Computer*, pp. 34–41, April 1978.

[5] H. Al-Asaad et al., "High-level design verification of microprocessors via error modeling", *Digest of Papers: IEEE Int'l High-Level Design Validation and Test Workshop*, 1997, pp. 194-201.

[6] M. Postiff, *LC-2 Programmer's Reference Manual*, Revision 3.1, University of Michigan, 1996.