ON INCREASING THE OBSERVABILITY OF MODERN MICROPROCESSORS

Hector Arteaga and Hussain Al-Asaad
Department of Electrical & Computer Engineering
University of California
Davis, CA, U.S.A.

ABSTRACT

Microprocessors are becoming increasingly complex and difficult to debug. Researchers are constantly looking for new methods to increase the observability and controllability of microprocessors. This paper introduces a new method to improve the observability of modern microprocessors and thus simplifying the task of debugging them. The method revolves around an observation circuit that provides access to important internal signals without interrupting the microprocessor execution. The output of the observation circuit is ported to the output of the microprocessor in order to easily detect various physical faults and design errors. Experimental results show that physical faults and design errors are detected faster using our method. Moreover, several errors are detected by the observation circuit without being detected by the microprocessor outputs.

Keywords: Post-silicon verification, observability improvements, microprocessors, design for debug.

1 INTRODUCTION

As microprocessors evolve, the number of transistors they contain grows exponentially, significantly increasing the complexity and burying signals deeper and deeper within. This in turn makes microprocessors progressively difficult to debug [1]. Researchers are constantly battling with this issue, looking for ways to increase the observability and controllability of microprocessors to make them more testable [2]. Methods, such as sample-on-the fly and JTAG [2][3], have been introduced to make it easier to retrieve data within the microprocessor. However, these methods interrupt the execution flow in order to observe the data. As an alternative we provide a way to check the validity of internal signals without interrupting microprocessor execution. This is achieved by incorporating a small observation circuit that is fed by the signals desired to be monitored and porting the output of the observation circuit to the output of the microprocessor.

In order for our method to be feasible, we must be selective when it comes to choosing the signals to be monitored because the number of signals grows exponentially with complexity. Tapping into all of the signals in the microprocessor would be unrealistic and would significantly increase the total area of the microprocessor or slow it down dramatically if at speed testing is desired. In our experiments we focus on monitoring control signals since they govern how the data will be moved about and regulate the execution of the microprocessor. Nevertheless, our method can be applied to any set of signals in the microprocessor.

In the next section, we introduce the microprocessor used in our experiments, detailing the instruction set architecture and the various hardware components. We then discuss our observation circuit in Section 3 followed by our experimental results in Section 4. Finally, we conclude the paper in Section 5.

2 MICROPROCESSOR USED

For our experiments, we used the “JAM” microprocessor [4] and modified it to include a simple monitoring circuit driven by the more influential control signals in the microprocessor. JAM is a 32-bit 5-stage pipelined microprocessor with a RISC architecture that supports precise interrupts. The pipeline stages are the generic instruction fetch (IF), decode (ID), execute (EX), memory access (MEM), and write back (WB). The JAM microprocessor uses a split memory

architecture (separate instruction and data cache) arranged in 64-bit wide words even though the instructions and data are only 32-bits wide.

2.1 Instruction Set Architecture

JAM is a 32-bit microprocessor that incorporates a wide range of different instructions and several permutations of the same instruction. It supports 22 different types of instructions and allows them to be combined with up to 3 different immediate formats (immediate, extended immediate, and displaced) in which the 16-bit immediate value is extended to 32-bits in different fashions. For example, there are separate ADD instructions depending whether you want the second operand to be the contents of another register, the immediate value sign extended to 32-bits (immediate), the 32-bit value in which the immediate value consumes the 16 most significant bits and the lower bits set to 0 (immediate extended), or the immediate value multiplied by 4 and sign extended to 32-bits (displaced). Not every instruction has all 4 subtypes available. Overall, JAM supports 47 different instructions requiring the opcode to be 6-bits long (not every combination is a valid opcode). Figure 1 breaks down the immediate and register instruction formats into the various components and Table 1 shows the supported instructions by the JAM microprocessor.

Stalling. Multiply and load instructions are the only instructions that stall the pipeline. The multiply instructions take two 32-bit operands and computes a 64-bit result in 33 clock cycles using Booth's encoding algorithm (the first cycle is a setup cycle). Since JAM is a 32-bit microprocessor, there are separate multiply instructions to allow you to retain the most significant or the least significant 32-bits of the result as well as immediate and register formats of the instruction. While the multiply is computing the result, the pipeline is stalled at the execute stage to prevent new instructions from entering the pipeline.

The other instruction that stalls the pipeline is the store instruction. The pipeline is stalled for one clock cycle to avoid malicious writes to memory due to the way the memory access unit and the memory is implemented.

Multiply and store instructions automatically stall the pipeline, but the pipeline can also be stalled when certain hazards are detected. Stalling due to hazards is minimized by the forwarding logic implemented in the JAM's datapath. However, the forwarding logic does not guarantee that data will be available in every situation. Load word (LW) hazards, which occur when an arbitrary instruction immediately following an LW instruction uses the results of the LW instruction, stall the pipeline for one cycle. Moreover, branch instructions stall the pipeline when they require unavailable data after they enter the ID stage (where they are resolved). All other hazards are

Table 1 JAM's supported instructions.

Arithmetic	Logic	Control	Other
ADD	AND	CMP	GET
ADDV	OR	JUMP	PUT
MUL-Lo	XOR	SET	TRAP
MUL-Hi		RESET	LW
SUB		BEQ	SW
SUBV		BNE	SHS, SHZ

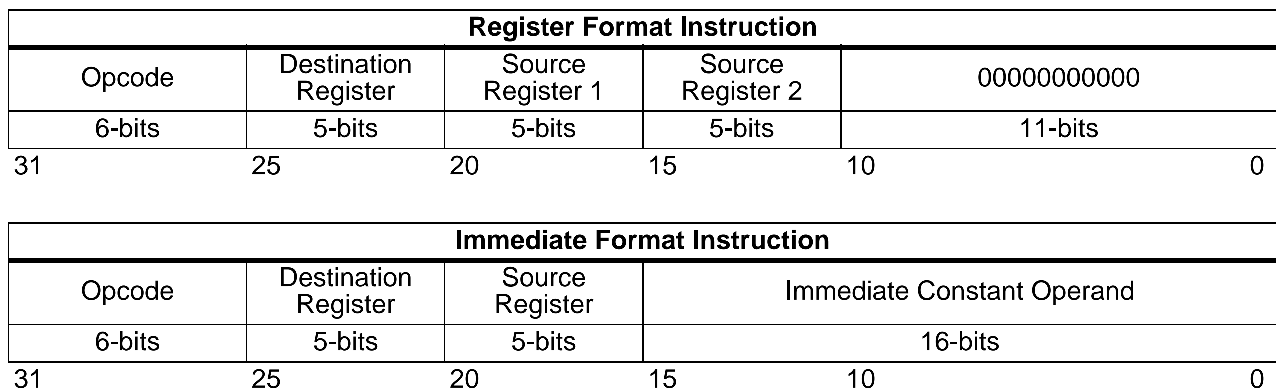

Figure 1 Formats for register and immediate instructions in the JAM microprocessor.

avoided through the use of forwarding logic.

The JAM microprocessor also support traps and interrupts, which also stall the pipeline, but they were not used in our design and hence they are not covered in this paper.

2.2 HARDWARE

The following is a list of various hardware units in the JAM microprocessor.

- *SRAM*: The memory architecture used in the JAM microprocessor is somewhat uncommon in that the words are 64-bits long even though JAM is a 32-bit microprocessor. The memory is composed of two SRAM arrays with each array composed of eight 512K x 8-bit integrated circuits. The two arrays are connected to form two 512K x 64-bit units, which are used as the separate instruction and data memory. The ability to enable or disable each of the 8 ICs within the SRAM arrays is maintained, allowing access to half a memory line at a time (32-bits). This feature also allows the reading of any combination of 4 of the 8 ICs, but the JAM microprocessor supports reading/writing to aligned words only. The two least significant bits of the address are always assumed to be 0.
- *Memory Access Unit*: The memory access unit (MAU) controls access to and from memory. There is one for each of the memory units (Instruction and Data). It receives a 32-bit address buss, 32-bit data buss, and three 1-bit control signals: *read*, *write* and *reset*. These signals are then converted to the 19-bit address buss, eight chip select bits and the output enable and write enable signals required by the SRAM arrays. When reading data, the full 64-bit memory word line is presented to the MAU, but only 32-bits are forwarded to the microprocessor depending on the 0th bit of the 32-bit address buss presented to the MAU by the microprocessor. Similarly, the 0th bit is used to decide which half of the word line is to be written to on a write operation.
- *Immediate Extension Unit and Register File*: The immediate extension unit is a simple unit that serves the purpose of extending the 16-bit immediate value to 32-bits in the different formats supported. The lower 2-bits of the opcode decide which format will continue on to see if it will be used in the execute stage. If the immediate value is not used, then the second operand will be one of the 32 register values in the register file. The only special register is R0 which always contains the 0 value. All other registers can hold any 32-bit value.
- *Control Unit*: The control unit resides in the ID stage. It takes in the 6-bit opcode and outputs the 24 signals used to control the behavior of the microprocessor. Many of the signals used by our monitoring circuit are produced by the control unit.
- *Integer Unit/ALU*: The integer unit is the main engine of the execute stage, implemented as a state machine. It contains the ALU as well as the control logic needed to produce and forward the appropriate result to the next stage. Since the multiply instruction is implemented using Booth's algorithm, the integer unit is responsible for stalling the pipeline for the 33-cycles needed to complete the multiply operation and is responsible for controlling the data flow between iterations of the multiply algorithm. A single cycle multiply unit was not implemented due to area overhead considerations. The ALU has the ability to perform only the more basic operations such as ADD, SUB, shift, AND, OR, etc. Moreover, it is also used to calculate the intermediate results of the multiply operation.

3 OBSERVATION CIRCUIT

In order not to contribute too much area overhead to the design, we decided to keep our observation circuit as simple as possible. To do this while maintaining a high probability that an error will be propagated to the output, we decided to implement our observation circuit as an XOR tree. This design assures than any single error on one of the monitored signals is propagated through the XOR tree and can be seen at the XOR tree output. However, no guarantees can be made when multiple errors are present at the monitor circuit inputs. Thus, errors on signals that affect numerous signals under observation may not be detected due to aliasing. It is up to the designer to choose the observed signals so that aliasing is minimized.

Another element we took into consideration is the pin-out count. Pin-out count is becoming increasingly more constraint [1], so we restricted the output of our observation circuit to a single bit. This increases the importance of proper signal selection to minimize

aliasing since there is only one output where a fault or design error can be detected. If pin-out is not a factor, having more outputs can reduce aliasing and increase the fault and design error coverage by the observation circuit. It also shortens the depth of an XOR tree, allowing it to compute its output signature faster.

For our experiments, we selected 50 different signals we deemed necessary to observe, which accumulated to a total of 87 bits. These can be reduced to a single bit using a seven stage XOR tree. Actually, a seven stage XOR tree can reduce up to 128 bits down to 1. So, we can expand our XOR tree without adding delay if we find more signals to observe. However, the lower signal count allows us to save some area as fewer gates are required to do the job. Our monitor circuit uses 89 2-input XOR gates total as apposed to the 127 needed if we were to fully utilize the seven stage XOR tree.

After implementing our XOR tree and verifying its correct functionality, we duplicated the design and treated one as the specification and the other as the implementation. We then proceeded to inject single stuck-line errors [6] into our implementation circuit and observed the output of both circuits to determine how soon the errors are detected using only our XOR observation signal compared to when the error is detected using the 212-bits of the microprocessor output (data-buss, address-buss, and memory control signals). We did this using two different elementary programs, one consisting of mostly multiply operations and the other consisting of the more basic ALU operations such as adding, shifting, and logic operations.

We further experimented with error detection by checking both the XOR observation signal and the 212-bits of the microprocessor output. We compared the obtained results to detecting errors using the microprocessor outputs alone (without the XOR observation signal). In this case, 500 clock cycles are simulated using a random instruction sequence.

4 Experimental Results

The experimental results for the multiply program can be seen in Table 2 and that of the ALU program in Table 3. For every fault/error in the tables, we report the first clock cycle where the error is detected using the monitoring XOR tree alone (XOR) and the microprocessor outputs alone (CPU). If the fault/error is not detected, an “ND” is reported in the tables. Signals with no detected errors by either the XOR or the CPU are not reported in the tables.

The simulation results of the multiply and addition programs demonstrate that our observation circuit detected every error that was detected by observing the 212-bits of the microprocessor output, usually detecting it sooner. Our observation circuit also detected errors not detected by the microprocessor output. In the 220 different simulation runs performed, only seven cases were noted where the error propagated to the microprocessor output sooner than propagating to the monitoring XOR tree output.

Table 2 First clock cycle where errors on control signals are detected by the multiply program which ran for 200 clock cycles.

Signal	Stuck-at-0		Stuck-at-1	
	XOR	CPU	XOR	CPU
cid_cmp	1	84	6	44
cid_bsel	1	ND	6	ND
ex_wb_dest_buf(0)	89	ND	1	ND
ex_wb_dest_buf(4)	ND	ND	1	ND
ex_wb_valid_buf	89	ND	1	2
wb_rw(0)	6	40	1	40
wb_rw(4)	ND	ND	1	40
mem_jump_trap	ND	ND	7	3
id_rb(0)	40	43	1	ND
id_rb(4)	ND	ND	1	43
if_zero	ND	ND	2	1
wb_rw(0)	6	40	1	40
wb_rw(4)	ND	ND	1	40
idex_in.cex_bsel	3	6	1	43
idex_in.cex_regsel(0)	ND	ND	1	ND
idex_in.cex_regsel(1)	1	ND	ND	ND
idex_in.cex_aluop(0)	1	6	5	43
idex_in.cex_aluop(2)	ND	ND	1	43
cex_valid_res	1	43	6	ND
cex_valid_reg	ND	ND	1	ND
idex_in.cm_write	6	41	1	3
idex_in.cm_read	86	122	1	5
cm_valid_mem	1	40	6	ND
cm_valid_reg	ND	ND	1	ND
idex_in.cwb_sel	ND	ND	1	40
idex_in.cwb_enable	1	40	6	80
cid_beq	ND	ND	6	3
exmem_reg.cm_write	41	1	1	1

Table 3 First clock cycle where errors on control signals are detected by the addition program which ran for 200 clock cycles.

Signal	Stuck-at-0		Stuck-at-1	
	XOR	CPU	XOR	CPU
cid_cmp	1	ND	5	12
ex_mc_finished	ND	ND	1	ND
wb_rw(0)	10	6	1	7
ex_mc_finished	ND	ND	1	ND
id_rb(0)	3	11	1	10
id_rb(4)	3	ND	1	11
if_zero	ND	ND	2	1
wb_rw(0)	10	6	1	7
idex_in.cex_bsel	2	4	1	ND
idex_in.cex_regsel(1)	1	ND	4	8
idex_in.cex_multop	ND	ND	1	ND
cex_valid_res	1	11	4	ND
cex_valid_reg	4	ND	1	ND
idex_in.cex_psw_enable	4	ND	1	3
cm_valid_mem	1	12	4	14
cm_valid_reg	4	ND	1	ND
idex_in.cwb_sel	4	ND	1	6
cid_bsel	1	ND	10	ND

Table 4 First clock cycle where errors in the datapath are detected by the multiply program which ran for 200 clock cycles.

Signal	Stuck-at-0		Stuck-at-1	
	XOR	CPU	XOR	CPU
--ALU SIGNALS--				
ALU_NOP(0)	ND	ND	6	6
ALU_NOP(2)	ND	ND	ND	40
ALU_ADD(0)	6	6	ND	ND
ALU_ADD(2)	ND	ND	6	6
ALU_SUB(0)	ND	ND	80	40
ALU_SUB(2)	ND	ND	80	40
cin	80	40	80	40
b_in(0)	80	40	85	40
b_in(31)	80	46	80	46
--IMM_EXT SIGNALS--				
mode(0)	44	5	ND	ND
mode(1)	45	44	ND	5
imm(15)	44	44	80	5
im(0)	ND	40	80	40
im(31)	ND	ND	80	40
r(0)	ND	40	80	40
r(31)	ND	ND	40	40

There were 28 test cases where the monitoring XOR tree output detected the error while the microprocessor output did not and many other cases where the monitoring XOR tree output detected the error significantly sooner.

These results demonstrate a good performance by our observation circuit when many of the important control signals are directly fed into the monitoring XOR tree. However, when errors are injected in the datapath, the performance of our observation circuit decreases. An experiment was performed where the monitoring circuit remain unchanged (monitoring important control signals), but the errors were injected within the ALU and Immediate Extension Unit of the microprocessor. The results for this run can be seen in Table 4. In this run, the monitoring XOR tree output took significantly longer to detect the errors and in four cases failed to detect errors detected by the microprocessor outputs.

If detecting errors in datapath is of a primary interest, then we can feed datapath signals to the monitoring XOR tree of the microprocessor. Another solution is to incorporate other “mini” XOR trees in each of the units in the microprocessor and feeding the outputs of the mini XOR trees to the major monitoring XOR tree, thus feeding datapath signals indirectly. This would allow the individual units of the microprocessor to be observed in a similar fashion to the control unit.

Our final experiment aimed at comparing the error detection by observing the following signals: (i) the microprocessor outputs and the XOR observation signal and (ii) the microprocessor outputs only. We used a random instruction sequence and ran the simulation for 500 clock cycles. The resulting improvement due to the embedding of an observation circuit in the JAM microprocessor is shown in Table 5. The results demonstrate a significant improvement of error detection using an XOR observation circuit. The embedding of an XOR observation circuit in the JAM microprocessor enabled us to detect 40 errors not detectable by the microprocessor outputs. Moreover, errors are detected earlier by using an observation circuit. On average, an error is detected 10 cycles earlier

Table 5 JAM's error detection improvement due to the embedding of an observation circuit.

	The overall processor including control unit	The control unit only
Total number of injected errors	2772	54
Number of detected errors	2054	54
Errors detected at the XOR observation circuit output but not at the microprocessor outputs	40	13
The average number of cycles for an error to be detected at the XOR observation output or microprocessor outputs	76	24
The average number of cycles for an error to be detected at the microprocessor outputs	86	82

if the microprocessor is augmented with an observation circuit. Moreover, if the error is in the control unit, then it is detected 58 cycles earlier.

5 CONCLUSION

The monitoring XOR tree provides an excellent view of the internal signals of the microprocessor. By observing a single signal (the XOR tree output), we can achieve close to the fault and design error coverage obtained by observing the entire microprocessor output signals. However, this requires that all signals we desire to check be directly fed into the observation circuit. This is not feasible for a large number of signals as the size of the monitoring circuit would become too large. By observing only a subset of the signals, we can check for correctness of certain aspects of the microprocessor such as the control logic in our example.

On the other hand, if we observe both the XOR tree output and the traditional microprocessor outputs, we can detect errors that cannot be detected using the microprocessor outputs alone in addition to detecting errors earlier.

It is obvious that the observation circuit imposes additional area to the design, however, the area overhead is often small and there is no need to interrupt the microprocessor execution during the monitoring of the internal signals.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0092867.

REFERENCES

[1] A. Irion, G. Kiefer, H. Vranken, H.J. Wunderlich, "Circuit partitioning for efficient logic BIST synthesis", *Proc. Design Automation and Test in Europe*, 2001, pp. 86-91.

[2] Y. Xu et al., "Advanced topics of DFT technologies in a general purposed CPU chip", *Proc. International Conference on ASIC*, 2003, pp. 1179-82.

[3] M.S. Abadir, T.M. Mak, and Li-C. Wang, "Tutorial 15: Validation and verification of high-performance microprocessors: Common challenges and solutions", *Proc. International Test Conference*, 2003.

[4] J.E. Thelin, A. Lindstrom, and M. Nordseth, *Concert'02 Architecture Specification and Implementation*, March 2002, (http://www.etek.chalmers.se/~e8mn/web/jam/).

[5] H. Al-Asaad and J. P. Hayes, "ESIM: A multimodel design error and fault simulator for logic circuits", *Proc. IEEE VLSI Test Symposium*, 2000, pp. 221-228.

[6] D. Van Campenhout, H. Al-Asaad, J. P. Hayes, T. Mudge, and R. Brown, "High-level design verification of microprocessors via error modeling", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 3, No. 4, pp. 581-599, October 1998.

[7] H. Al-Asaad and J. P. Hayes, "Logic design verification via simulation and automatic test pattern generation", *Journal of Electronic Testing: Theory and Applications*, Vol. 16, No. 6, pp. 575-589, December 2000.