Instruction Randomization Self Test For Processor Cores

Ken Batcher Christos Papachristou
Department of Computer Engineering
Case Western Reserve University, Cleveland, OH 44106

ABSTRACT

Access to embedded processor cores for application of test has greatly complicated the testability of large systems on silicon. Scan based testing methods cannot be applied to processor cores which cannot be modified to meet the design requirements for scan insertion. Instruction Randomization Self Test (IRST) achieves stuck-at-fault coverage for an embedded processor core without the need for scan insertion or mux isolation for application of test patterns. This is a new built-in self test method which combines the execution of microprocessor instructions with a small amount of on-chip test hardware which is used to randomize those instructions. IRST is well suited for meeting the challenges of testing ASIC systems which contain embedded processor cores.

1. INTRODUCTION

Embedded processor cores are a popular design component for many systems on silicon. Core based design offers several advantages including design reuse and portability to different ASIC systems. This allows processors to be used in a variety of applications in a cost effective manner. However, designing with processor cores presents new challenges for testing since access to these embedded processor cores becomes further removed from the pins of the ASIC [1].

BIST test methods relying on scan to deliver test patterns are quite effective, but cannot always be used with systems containing embedded processors. Access to the core for scan insertion may be difficult due to the need to isolate the core from the remaining ASIC system. Sometimes processor cores support some form of self test hardware; however, integration into a ASIC system may be difficult or impossible if such a system uses a incompatible BIST method. The processor architecture itself may prohibit the use of scan test methods since such methods require design for test rules. Processor designs containing a mixture of latches and flip-flops, asynchronous logic, internal tri-states, and gated clocks are becoming popular due to the need for low power design [2, 3]. Structures like these create many problems for scan based testing. If a processor core does not support scan, the designer may be forced to use a less efficient test method. These other methods often do not have the advantages associated with BIST[4].

Instruction Randomization Self Test (IRST) is a test methodology which exploits the functional behavior of a microprocessor (e.g. the ability to execute instructions) to achieve stuck-at-fault coverage for the design. This is done by continuously executing a random stream of processor instructions and compressing execution results using internal test hardware. Special software programs, written in the native assembler code of the processor, are designed to exploit the randomization behavior to achieve the observability and controllability needed for good fault coverage. IRST offers an attractive alternative for built-in test for processor core designs where scan BIST cannot be used. The test method is general purpose so that it can be used with various processor architectures that share the following features found in most processor architectures[9]:

1. Processor must fetch instructions from read/write memory.
2. Instructions must perform operations on internal registers and/or memory.
3. A single instruction cycle is required.
4. Processor must be able to execute a branch instruction.

Functional testing of microprocessors has a long history [5,6,7,8]. However, functional testing achieves low fault coverage because it does not consider the RTL structure and it is not based on a fault model (s-a-fault). More recently, some research has been initiated towards functional testing using some RTL information. However, these techniques are not exactly suited to embedded microprocessor cores.

Most testing methods used for fault validation focus on the gate level implementation details of a design. IRST approaches the validation problem from the RTL level by abstracting gate level detail and focusing on the functional, instruction level behavior of a processor. This unique property has several advantages[9]. Instruction level coverage techniques are used along with randomization to make IRST an efficient and effective test method.

Instruction randomization methods have been used previously in the industry for microprocessor testing[10]. IRST differs from this method in that it designed as a at-speed BIST method and relies on algorithms using hardware assisted by software control to achieve fault coverage. An earlier technique using instruction randomization relied on self modifying code[11]. This approach, although requiring almost no test hardware, lacks the advantages of a built-in self test and cannot easily be applied to an embedded processor core. Recent methods rely on automated functional test generation to make self-test programs which are quite efficient at detecting faults[12]. IRST shares some similar advantages, but applies a different method using hardware instruction randomization to create a self-modifying test program. This work expands on IRST ideas previously introduced and further defines IRST properties[13]. Much effort was devoted to fault analysis and structuring the software to achieve better instruction coverage which has been shown to result in better fault coverage.

An example of IRST applied to an embedded processor core is used as a proof of concept for this new test method. Experimental results were obtained using fault simulation analysis to determine the effectiveness of IRST on a real world testing problem. Many examples and illustrations throughout this text use an instruction set for the DLX machine. This is a generic processor description containing common features found in many processors today [14].

2. IRST ARCHITECTURE

Two key components of IRST are the test software and the hardware (Figure 1). Both play important roles for providing the necessary controllability and observability to achieve good stuck-at fault coverage. Cooperation between the IRST hardware and software is a key feature. The cooperation between software and hardware is a unique property of IRST not common in other BIST test methods.

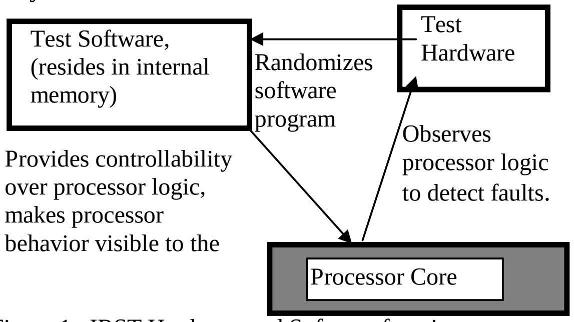
Test Software, (resides in internal memory)
Randomizes software program
Test Hardware
Observes processor logic to detect faults.
Provides controllability over processor logic, makes processor behavior visible to the
Processor Core

Figure 1. IRST Hardware and Software functions

The test hardware provides three functions during the operation of IRST. First it modifies the test software to provide a pseudo random sequence of instructions, thus creating a randomized test program. Second, the test hardware monitors the instruction fetch and R/W activity as the test software executes to determine if the behavior indicates faulty logic. Third, it provides a source of randomized seed data which the test software uses to randomize register operands.

As the test software executes it will exercise a variety of control and data path logic in the processor core and thus provides the necessary controllability over the processor logic. The test software provides observability since the

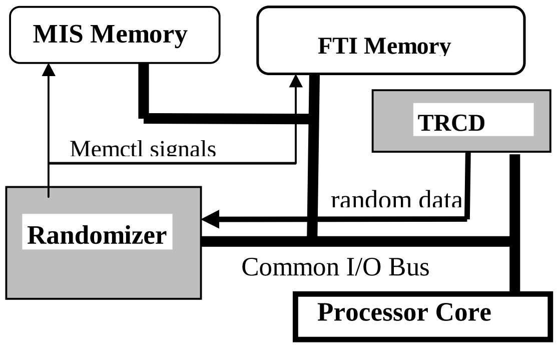

Figure 2. IRST hardware organization.

instruction flow and data operands (kept in registers or memory) are made visible to the test hardware by exposing them on the central bus as the program executes. The software program also manages the test mode operation providing end of test determination.

2.1. Test Hardware

IRST hardware consists of a randomizer, modifiable instruction storage (MIS) memory, fixed test instruction (FTI) memory, and a Test Response Compression Device (TRCD). Figure 2 details the organization of the test hardware when connected to a processor core. The MIS memory is an embedded RAM large enough to hold a small set of instructions (typically 64 to 512 for most designs). FTI memory is read only and is also used to contain the IRST software. IRST is more cost effective if embedded RAM already exists in the ASIC in the form of an instruction or data cache. Since IRST is a BIST test method code store must exist inside the ASIC. Most processor designs contain or support some form of on-chip memory and thus are good candidates for IRST.

Observability of the core is provided by the TRCD which is a LFSR configured as a MISR or CRC generator. It is used to observe and compress key processor signals into a repeatable signature. Typically a parallel realization of the LFSR is necessary since the TRCD must be capable of compressing the entire I/O bus in a single instruction cycle. An important requirement of the TRCD is that the signature result must be readable by the test software at any time during the self test. Typically a central processor bus where most of the important processor activity occurs is selected as the input to the compression device. This central bus may be the source of data R/W activity and/or instruction fetches.

When IRST is enabled, the TRCD can be read and written by accessing memory (section 2.4). As instructions are fetched from MIS memory, they too are compressed into the TRCD signature which provides observability of the program flow during the self test. Even though IRST relies on one bus for observability, various results from other buses and register locations are channeled to the input of the TRCD by software execution.

The randomizer (Figure 3) is a logic block designed for randomization of instructions as they are fetched from the MIS. The randomizer operates by reading each instruction in parallel to the instruction fetch of the processor. It will randomize the instruction and then write it back to the MIS at the same address from which it was fetched. In this way a new randomized instruction is generated 'on-the-fly' which will be executed again the next time the instruction address is fetched.

First a decode is performed on the instruction to generate an *opcode dependent randomization mask.* This mask contains a 1 in every position for each bit in the instruction

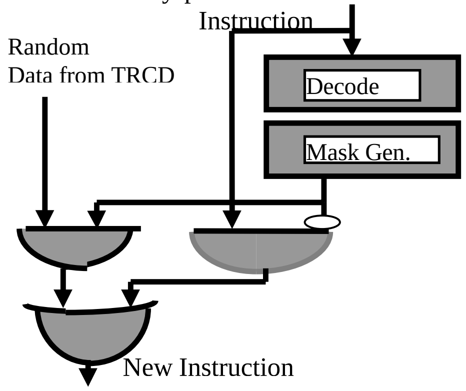

Figure 3. Design of randomizer

field that is to be randomized. Randomization masks are determined in hardware by the opcode type being fetched and will only modify certain instruction fields so that the instruction remains a meaningful and defined permutation of the original opcode. The mask is first used to clear out the randomization fields in the original instruction. Then the mask is used to select which bit positions are to be updated with random data. The new data is then ORed in down stream with the masked instruction to obtain a permutation of the original instruction.

Figure 4 illustrates what an arbitrary mask selection might be for a 16 bit DLX instruction set. For example, with ALU instructions, the operands A and B are randomized where the opcode and type/mode fields are fixed and never change. The randomizer also contains some control logic for managing IRST operations including read/write control for the TRCD, MIS, and FTI memories, enabling and disabling the self test, and determination of pass/fail criteria.

The random data source for the randomizer comes from the current signature of the TRCD, Good controllability is achieved over the processor core since a continuous stream of instructions is created which exercise a variety of instruction opcodes, types and operands when executed. Thus different logic is controlled in the processor each time a randomized instruction executes. The TRCD thus serves both as a compression device, used to compact the circuit response, and as a source of seed data for random test pattern generation. The test patterns are used to randomize the instructions as well as the operands (section 3.3) when the TRCD is read into registers. The dual functionality of the TRCD is similar in concept to the registers in a circular BIST arrangement [15]. The circular self test path in this case is linked by the cooperation between software and hardware during test execution.

Opcode	**Type/ Mode Field**	**Operand Field A**	**Operand Field B**
Alu	0000	1111	1111
Alui	0011	1111	1111
Branch	1111	0011	1111
Load	1111	1111	0000

Figure 4. Opcode dependent randomization masks.

2.2. Test Software

The main function of the software program is to provide a source of random stimulus for controllability over various data and control paths in the processor core as random instructions execute. This is accomplished with the assistance of the hardware randomizer which controls the test software, giving it a stochastic behavior. This behavior is desirable so that enough processor logic can be tested. However some structure must be given to the software since it must also provide these other functions during test mode:

1. Observability of internal registers.
2. Randomization of instruction operands.
3. Determination of test completion.
4. IRST pass/fail criteria indication.

It is important to control the execution of the software so that these objectives can be met as well as allowing the code to be random enough in order to provide good controllability. Therefore, the IRST software is organized into two distinct code regions as shown in Figure 5. All of the randomized test instructions reside in the MIS space. A small portion of the test software, called the fixed test instructions (FTI), reside in a separate memory region which is read only during IRST test mode. The FTI instructions are used to provide added observability, randomization of register operands, test termination and pass/fail criteria.

2.3. Instruction Fetch Operation

While IRST is enabled, the randomizer will force all instruction fetches to come from either the FTI or MIS memory space. Whenever an instruction is fetched from a special fixed address called the *trigger address*, the randomizer will conditionally toggle to the other memory region. When a toggle branch is 'taken' is known as toggle frequency which is adjustable in hardware(section 2.5). When the FTI software has finished processing it simply branches to the trigger address, which forces the randomizer to enable fetches from the MIS space. The next fetch will be code from the MIS region where instructions are fetched, randomized, and written back, all in the same cycle.

The MIS code will toggle back to the FTI code whenever the same trigger address is fetched. When this occurs, the randomizer forces execution back to the FTI code at a fixed location (FTI Start). Toggling from MIS code will occur at pseudo random intervals since the MIS code will execute in a stochastic fashion due to randomized branch operations. Each 'toggle' is defined as a branch to the MIS code region terminating with a branch back to the FTI code region. While fetching from FTI code space, the instruction writeback randomization is disabled therefore the FTI instructions are fixed and are not modified.

2.4. Read/Write/Fetch Cycle Behavior

Most processors execute three different types of cycles across a central bus to external memory which are fetch, read

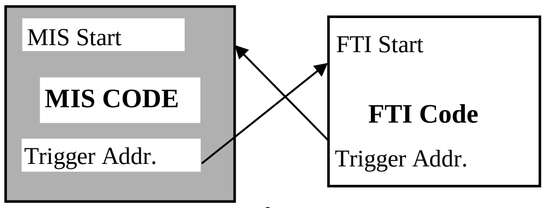

Figure 5. Test software organization.

and write cycles. During IRST execution, the function of these cycles is modified (Figure 6) Since IRST is a built in self-test technique, all read/fetch cycles must be mapped internal to the ASIC device. This is true since no memory exists outside the chip and thus any memory read or fetch activity would not be deterministic.

Operation	MIS Region	FTI Region
Read	read data from TRCD, no compress	read data from TRCD, no compress
Fetch	fetch/modify/write-back all in same cycle	fetch instruction
Write	compress data written into TRCD	compress data written into TRCD

Figure 6. Instruction cycle behavior during IRST.

Fetch cycles as previously explained are modified to fetch from either the internal memory MIS or FTI code regions. Read instructions are mapped to read from the TCRD; instead of reading external memory, the current TRCD value is driven onto the central data bus. In this way, reads will return different pseudo random values each time they are executed which represent the cumulative sum of all processor activity observed so far across the central bus. The data written to memory by write instructions is compressed into the TRCD from the central data bus just as instructions are compressed during a fetch. This provides added observability by making the register contents visible to the TRCD.

2.5. IRST Software Operation

The operation of the IRST software is summarized in Figure 7. After loading the initial test software into the MIS and FTI memory regions, the IRST test hardware is enabled which allows the randomizer and TRCD to run. Execution will begin in the FTI code region where the iteration count is initialized. Store instructions are performed to observe register contents and registers are then weighted. A branch is then made to the trigger address which forces execution from the MIS region which is shown as the shaded region in Figure

7. The randomized MIS instructions will be executed and responses captured in the TRCD for a pseudo random number of instruction cycles.

After toggling from MIS code, the FTI software first improves the observability of the self test by writing selected register contents to the TRCD input where it is compressed. This makes internal register contents observable to the TRCD. After this, the FTI code will increment a counter and compare it against the terminal count. If the terminal count is not yet reached, the register contents are weighted (section 3.3) and a branch is made to the trigger address which toggles into the MIS memory space. Like other forms of self-test weighting pseudo random patterns is crucial for achieving good controllability [16].

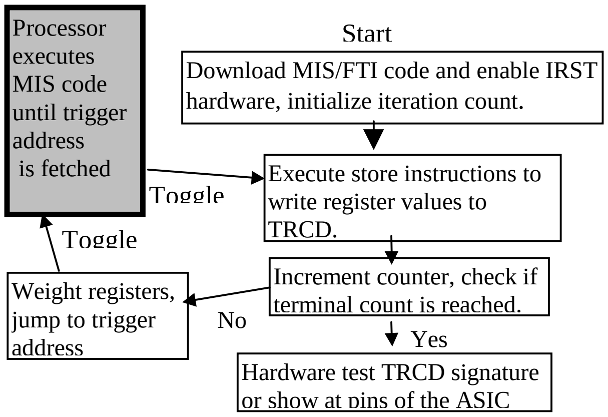
Processor executes MIS code until trigger address is fetched
Start
Download MIS/FTI code and enable IRST hardware, initialize iteration count.
Execute store instructions to write register values to TRCD.
Toggle
Toggle
Increment counter, check if terminal count is reached.
Weight registers, jump to trigger address
No
Yes
Hardware test TRCD signature or show at pins of the ASIC

Figure 7. IRST software operation.

The *toggle frequency* can be adjusted in the randomizer by conditionally allowing the branch to be taken back to the FTI region. For example a toggle frequency of 1/8 can be enforced by allowing the toggle to occur only if the current random TRCD value has the 3 least signficant bits set. If the toggle branch is not taken, the MIS code will simply wrap back to the first MIS instruction.

Execution of FTI code, although necessary for observability of registers and randomization of operands, largely constitutes overhead since these instructions are not randomized. However, too much time spent in the MIS region can adversely affect observability since register contents are sometimes overwritten as described in section 3.4. Furthermore the weighted random patterns created by the FTI code can also become overwritten before they are used as source operands thus reducing controllability. Therefore the designer must adjust the code efficiency such that the time spent in either code region is not excessive.

If the terminal count is reached the signature is then examined for pass/fail criteria. The value in the TRCD will contain a deterministic signature which reflects the program behavior observed on the I/O bus by the TRCD. The TRCD signature can be analyzed to indicate a successful or unsuccessful test. This is suggested rather than allowing the FTI software to test the signature since faulty logic may exist in the compare circuitry of the processor code, which incorrectly indicates a good part[17]. While the MIS and FTI instructions are executed, the TRCD will continue to run compressing every instruction cycle as described in Figure 6. This not only provides observability of the I/O bus on each instruction cycle but also provides fresh random data which can be used by the FTI code to randomize and weight register operands.

The randomization properties IRST guarantee that the MIS code will perpetually execute a sequence of operations and then eventually return to FTI code space. This is true since the execution thread will advance to the trigger address as the code progresses sequentially or as a result of randomized branch operations to the trigger address. Code hazards such as infinite loops are automatically prevented since branch instructions will change in the same cycle each time they are fetched, resulting in a branch with a new destination address or new type of conditional branch which is not taken. Likewise illegal instruction types can be prevented by structuring randomization masks to avoid them.

Since a pseudo random sequencer is used to randomize the test software and to manipulate the operands used by those instructions, the same initial test program will be modified in the same way and behave in the same way on all processor cores. This is true so long as the processor core does not contain faulty logic. Therefore faults can be detected by the software provided they are controlled by the program execution and observed by the TRCD. Comparison of the TRCD to a known good signature (determined by simulation) provides the necessary pass/fail checking for the processor core with fault coverage serving as the degree of certainty that the logic is free of manufacturing defects.

3. SOFTWARE RANDOMIZATION TECHNIQUES

An affective IRST implementation will use a software program highly optimized for detecting faults. Many techniques are available to increase the controllability and observability of the test software. Techniques exist for improving the IRST self test for both the FTI and MIS code. The hardware mask selection is also important since it has a direct affect on the randomization of the software. This is typically done as the software is being written. Many of the techniques in this section were developed based on IRST coverage analysis and from fault simulation.

Initial MIS Fetch	2nd MIS Fetch	3rd MIS Fetch
LB [R2],R3	SW 500(R4),R1	SH (R9),R0
ADD R3,R2,R1	SLL R1,R2,#5	SUB R9,R1,R1
BNEZ R1,x30	BEQZ R7,x2F0	BNEZ R10,x50
ANDI R1,R2,#3	XORI R11,R2,#7	SUB R5,R0,R9
LW R1,30(R2)	LBU R3,[R2}	LH R1,40(R8)

Figure 8. MIS code modified during IRST operation.

3.1. Opcode Mix in MIS Program

Writing the initial test program is important in determining the controllability of the MIS software. This is true since the randomizer is fixed in hardware and uses the opcode field of the instruction to determine how the permutation of the instruction occurs. In most processor architectures opcodes are grouped into different types each having different modes. An example might be data transfers, arithmetic/logical, control, and floating point as is the case with the DLX processor[14]. Therefore an arithmetic instruction in the initial program will always be an arithmetic instruction, but will have different operands and instruction modes each time it is executed (Figure 8).

The designer should choose the right mix of opcodes at compile time to maximize the fault coverage of the processor.

Some knowledge of the processor core is helpful since execution of certain instructions may yield higher fault coverage than others. For example a DSP processor may be very datapath intensive, therefore an opcode mix of heavily weighted with floating point instructions may yield the best coverage. A general purpose CPU may achieve better fault coverage with a more balanced mix of opcodes. Therefore, in writing the initial test program, it is important to include all the different types of instructions and weight the instruction mix to maximize detection of faults.

3.2. Instruction Sequencing of MIS Program

Sequencing of instructions especially in pipelined processors is also important for maximizing the fault coverage of the self test. Certain combinations of instructions may excite different faults as the processor executes. When writing the initial MIS program, the designer can choose an appropriate mixture of instructions which increase the probability of detecting faults dependent on instruction ordering. This usually requires detailed knowledge of the instruction sequencing logic. For example pipeline stalling logic might exist which stalls the instruction pipeline if a conditional branch is followed by certain ALU instructions. By intermixing ALU and branch instructions in the initial MIS program, detection of faults in the stalling logic is possible. If an instruction does not stall when it should, an incorrect branch may be taken which changes the instruction sequencing on the I/O bus. This makes the fault detectable since the TRCD will detect when a different instruction is fetched during compression.

3.3. Operand Weighting

Another challenge of IRST is dealing with random-pattern resistant faults. These are common in data path units such as processor ALUs which require unique input combinations in order to sensitize these rare faults. It is necessary to make the IRST instructions as efficient as possible since test time can still be excessive when it comes to covering random pattern resistant faults in the data path[18]. Weighting of instruction is done by the FTI software in order to improve the random quality of the patterns for MIS data operands. Selection of weights can be done using methods based on the structural analysis of the processor logic similar to other weighted random pattern BIST methods [19].

Register	16 bit value ranges
R8 to R1	xFFFF to x0000, pure random
R10 to R9	Mostly ones
R12 to R11	0x7FFF to x7FF0
R15 to R13	x003F to x0000

Figure 9. Sample operand weighting set.

FTI code is used to create a data set for operand registers which is random, but weighted. For example with the DLX machine, the registers can be manipulated to result in the following ranges shown in Figure 9. These value ranges will provide a data set such that controllability over random pattern resistant faults are more likely to occur. For instance, if an ADD instruction uses R12 as an operand there is a higher probability that the ALU will roll over and exercise a ripple carry or partial ripple carry path compared to using R1 as an operand. To achieve operand weightings with IRST, the FTI code simply reads the TRCD into a register. This makes the register contents pseudo random since every instruction cycle causes the TRCD to compress and thus create new random data. After this, logical ALU operations are performed to manipulate the contents of the registers to achieve the desired weighting. For example a mostly-ones combination is created by reading the TRCD 3 times into 3 different register destinations. Next OR instructions are performed on those registers to result in a register that has a 7/8 probability of a one in each bit position.

Figure 10 summarizes how a randomized instruction achieves fault coverage over the processor elements. Randomization occurs at the instruction level and the operand level. The randomized instruction fields exercise different areas of the control logic and data path logic as shown with the dashed lines in Figure 10. Data operand randomization is achieved from the register contents, randomized during self test execution. The register file is thus used as a source of random patterns which the software delivers along different data paths during self-test execution.

3.4. Observability Enhancing Operations

As the MIS Code executes, the results of many operations are stored into registers. These values are not observable on the central bus and therefore do not get compressed into the TRCD. It is necessary therefore to make register result values visible to the test hardware so that potential faults can be observed. This is done by simply executing store instructions to expose register contents. As described in Figure 6, the TRCD will compress values written to the central bus by register-to-memory store instructions. Store instructions in the MIS Test Code are not sufficient to observe enough register contents since those operations occur randomly. Thus FTI test instructions execute several store operations to dump register contents to the central bus each time a toggle is made to the FTI region.

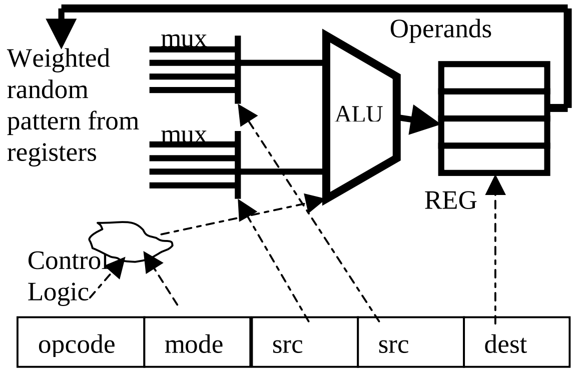

Figure 10. Random instructions covering processor logic.

Due to the stochastic nature of the MIS code, sometimes register contents are overwritten. For example in the following code segment the SUB instruction overwrites the result of the ADD instruction:

```
ADD    R1, R11,R12  ; Add R11 and R12, store into R1
SUB    R1,R4,R16    ; Add R4 and R16, overwrite R1 !!!
```

To prevent this from happening, MIS code can be written and opcode masks selected to structure the code in this way:

```
ADD    R1,R11,R12   ; Add R11 and R12, store into R1
SUB    R2,R4,R16  ; Add R4 and R16, store into R2
```

In this example, the destination fields of the ALU opcodes are not randomized and thus fixed at compile time, but the other two operand fields are. Therefore this code will not overwrite the result of previous operations Many variations of this technique are possible which involves both the initial software program and randomization mask selection.

3.5. Operand Chaining

Operand chaining is a technique used to efficiently increase the observability of instructions results that are stored into registers. As described in section 2 many operations are stored into registers and not observable on the central bus. By using store instructions in the FTI software, it is possible to expose register contents to the TRCD for compression into the self-test signature. This principle is know as *latent observability* since results of instruction execution which contribute to fault coverage are not immediately observable by the TRCD. The result of such events are observed later as subsequent instructions execute.

If a processor contains 32 different registers it would necessary to execute 32 separate store instructions in the FTI code so that all register contents can be made visible. This requires a good deal of processing overhead which does not contribute much to the overall fault coverage. By using operand chaining, the results of many operations can be observed with a single store instruction. For example, a MIS Code sequence written in this way requires 3 store instructions to observe the register results R1 R2, and R3:

```
ADD   R1, R11,R12 : Add R11 and R12, store into R1
SUB   R2,R4,R16   ; Subtract R4 and R16, store into R2
XOR   R3,R5,R7    : XOR R5 and R7, store into R3.
```

By chaining operands, intermediate results are used as the operand of a subsequent instruction. This enables a single store instruction for the R3 register to observe the results of all 3 instructions:

ADD **R1**, R11,R12 : Add R11 and R12, store into R1
SUB **R2**,**R1**,R16 ; Add R16 and R1, store into R2
XOR R3,**R2**,R7 : XOR R2 and R7, store into R3.

By reducing overhead in the FTI code the software will become more efficient at detecting faults and thus higher fault coverage can be achieved with less overall test time.

4. EXPERIMENTATION WITH IRST

The IRST methodology was applied to an ASIC design containing an embedded processor core. Many of the techniques and principles described in this work were utilized during the design of the IRST hardware and software.

The processor core was a 16-bit machine with a RISC-style architecture similar to the DLX machine. Logic blocks inside the core included an ALU for logical and integer math, memory controller for read/write operations, branch logic, conditional test logic, two stages of instruction decode, and a central register bank. The processor used a 3 deep instruction pipeline and 128 instructions of on chip memory for the MIS/FTI code space. The instruction set contained of 3 basic types of instructions:

1. ALU with register and immediate data operands.
2. Memory R/W with register and immediate data address.
3. Conditional and unconditional branches.

4.1. Test Implementation

The test software contained 64 MIS instructions and 62 FTI instructions. A small driver program of 40 instructions was used to download the test program and enable the IRST logic. The TRCD was designed as a CRC generator that can compress 32-bits in a single clock cycle. It was developed as a parallel realization of a LSFR using the primitive irreducible polynomial $X^{28} + X^{27} + X^1 + 1$. Two 16-bit busses were compressed by the TRCD to provide observability of core operations. One bus was used as the source of data R/W and instruction fetch operation. The other bus was an internal bus which was used for instruction operand traffic.

Synthesis of the Randomizer and TRCD resulted in a gate count of 760 gates. Other logic overhead needed for multiplexing controls to the memory storage accounted for about 122 more gates resulting in a total test overhead of 3.1% when compared to the processor core gate count of 27,860. The memory was not considered test overhead since it already existed in the ASIC. The total number of faults present in the test logic was 774 compared to 43,927 in the processor core.

4.2. IRST Design Process

Testing began with the creation of the TRCD and Randomizer. Next an initial MIS program was created along with the FTI test code. A small driver program was used to load the memories with this IRST test code and then enable IRST. The IRST hardware and software were then functionally debugged in simulation. After a working model was created, a few toggle node simulations were performed to get a feel for the logic controlled by the IRST. Modifications were made to the Randomizer, masks, MIS code and FTI code to boost the toggle node coverage. Stuck-at fault simulation was then performed on the core logic using IRST as the stimulus for the core.

4.3. Test Results

Three fault simulation trials were made using different mixtures of instruction opcodes and register weights. Each one was executed for 50,000 instruction cycles (Figure 13). Adjustments were made in the MIS opcode mix (Figure 12), toggle frequency, and register weighting schemes to improve the fault coverage on each run. Using missed fault information helped to identify areas of logic which needed further testing. With these changes, a fault coverage of 92.5 % was eventually achieved for the third trial. This trial was repeated for 220,000 cycles to see what fault coverage was possible. The fault coverage did improve with a slow rate of improvement up to 94.8% (Figure 14).

5. CONCLUSION

This paper introduces various concepts associated with IRST and illustrates some of its advantages. IRST is a new built-in self test method which can be applied to microprocessor cores. The example shown demonstrates that a fairly high level of fault coverage can be achieved with a reasonable amount of test hardware overhead on this particular design. There are several advantages IRST offers as a test methodology especially when dealing with processor cores.

Flexibility is a key advantage that IRST offers for testing processor cores. The MIS and FTI test software can be changed at any time to develop different test programs. Thus IRST is not totally fixed after device fabrication unlike other forms of BIST. With on-chip memory, IRST provides many advantages associated with other BIST methods including at-speed fault validation and field diagnostics which can be enabled at any time.

A unique property of IRST is that the test logic is totally non-intrusive to the processor core. For this reason there is no need to redesign the core for insertion of scan chains, as is needed with scan methodologies. The IRST self test is naturally isolated from the processor core and other functional blocks which might reside on a silicon system. The test hardware for IRST is relatively low compared to other BIST methods. There is no need to store test patterns on chip since they are automatically generated. Furthermore the core suffers no loss in performance or increase in size due to the scan insertion.

Many processor cores today are designed with deep sub-micron technology in order to enable high speed operation. With such designs, at-speed testing has become extremely important since some speed-sensitive faults cannot be detected with low frequency test vectors [20,21]. IRST

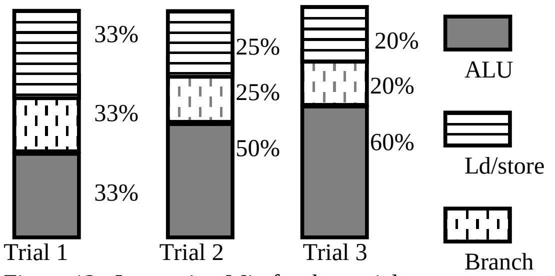

Figure 12. Instruction Mix for three trials.

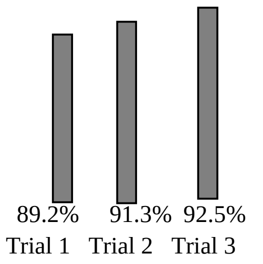

Figure 13. Fault coverage results for three trials.

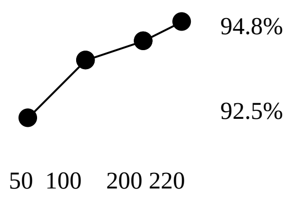

Figure 14. Fault coverage for trial 3. 1000s of instr. cycles.

offers a true at-speed test since valid instruction paths and combinations are exercised during the test execution.

While IRST executes, the core is totally unaware that the IRST test is enabled. It merely fetches and executes instructions as they are presented to the core from the randomizer. Thus while IRST is enabled, the processor will execute instructions on every clock cycle in a 'functional' mode just as if it where executing real software programs. This differs from scan test methods since several clocks are needed to shift in serial patterns before a functional clock is provided. The amount of time spent by IRST in a functional mode is much higher than that scan BIST. It is believed that this feature combined with the fast execution of IRST might be a key advantage of IRST which results in potentially higher fault detection efficiency compared to scan based methods.

The IRST methodology is still experimental at this time. More effort will be used to further develop IRST in hopes of making it a popular BIST method for processor cores.

6. References

1. Murray and J Hayes."Testing ICs, getting to the core of the problem". IEEE Design and Test of Computers.Vol.29, no.11.
2. T.V. Woods, et. al "Amulet 1: An Asynchronous ARM Microprocessor". Transactions of IEEE. Vol 46, no.4. 1997.
3. B. Tuck. "Power Pushed to Prominence by System on Chip and Portable Products". Computer Design. January 1998.
4. R.E. Massara. Design and Test Techniques for VLSI and WSI Circuits. Institute of Electrical Engineers. U.K. 1989.
5. D. Brahme and J. Abraham. "Functional Testing of Microprocessors", IEEE Trans. on Computers, C-33, 6, 1984.
6. A. van de Goor et al. "Functional Testing of Current Microprocessors", International Test Conference, Sept. 1992.
7. J. Lee and J. Patel. "An Instruction Sequence Assembling Methodology for Testing Microprocessors", International Test Conference Sept. 1992.
8. S. Thatte and J. Abraham. "Test Generation for Microprocessors", IEEE Trans. on Computers Vol. C-29, 1980
9. U. Bieker, et.al. "Star-Dust: Hierarchical Test of Embedded Processors by Self-Test Programs". Department of Computer Science, University of Dortmund, Germany.
10. H. Klug. "Microprocessor testing by Instruction Sequences Derived from Random Patterns". ", International Test Conference, 1988.
11. K. Batcher. "Microprocessor Self-Test Apparatus and Method". U.S. Patent # 5668947. Sept. 1997. Filed Apr. 1996.
12. J. Shen, J Abraham. "Native Mode Functional Test Generation for Processors with applications to Self Test and Design Validation". International Test Conference, Oct. 1998.
13. K. Batcher, C. Papachristou. "Instruction Randomization Self Test for Validation of Processor Cores". ITC Workshop for Core Based Testing, November 1997
14. J. Hennessy and D. Patterson. Computer Architecture, A Quantitative Approach, 1996. Morgan Kaufmann Publishers Inc.
15. J. Carlettta, C. Papachristou. "A Markov Chain Model for Conventional and Circular BIST Analysis". ICCD 1995.
16. J. Savir, P. Bardell. "Built-in Self-Test Milestones and Challenges". VLSI Design, 1993. Volume 1.
17. M. Miller and Y. Tanaka. "High Fault Coverage BIST Strategy for State Machine BasedDesigns". 1993 Compass Design Automation Users' Conference.
18. A. Majumdar. "On Evaluation and Optimal Weights for Weighted Random Pattern Testing" IEEE Transactions on Computers. Vol 45, Number 8. August 1996.
19. J. Waicukauski, E. Lindbloom, E. Eichelberger, O. Forlenza. " A Method for Generating Weighted Random Test Patterns". IBM J. Res. Develop. Vol 33, No 2 March 1989.
20. B. Nadeau, D. Burek, et al. "Scan-Bist, A Multifrequency Scan-Based BIST Method." IEEE Design and Test of Computers. Vol. 11, Number 1. Spring 1994.
21. M. Hamad. "A Statistical Methodology for Modeling and Analysis of Path Delay Faults in VLSI Circuits". Computers and Electrical Engineering. Vol. 23, Number 5. September 1997.