IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors

Sung-Boem Park, Ted Hong, Subhasish Mitra, *Senior Member, IEEE*

***Abstract*—The objective of IFRA, Instruction Footprint Recording and Analysis, is to overcome the challenges associated with a very expensive step in post-silicon validation of processors – localization of bugs in a system setup. Special recorders are inserted into the processor to record semantic information about data and control flows of instructions passing through various design blocks. The recording is done concurrently during the normal operation of the processor in a post-silicon system validation setup. Upon the detection of a problem, the recorded information is scanned out and analyzed offline for bug localization. Special program analysis techniques, together with the test program binary of the application executed during post-silicon validation, are used for the analysis. IFRA does not require full system-level reproduction of bugs or system-level simulation. Simulation results on a complex super-scalar processor demonstrate that IFRA is effective in accurately localizing electrical bugs with very little impact on overall chip area.**

***Index Terms*—post-silicon validation, silicon debug, design-for-debug, electrical bug, circuit marginality**

I. Introduction

Post-silicon validation involves operating one or more manufactured chips in actual application environment to validate correct behaviors across specified operating conditions. According to recent industry reports, post-silicon validation is becoming significantly expensive. Intel reported a headcount ratio of 3:1 for design vs. post-silicon validation [Patra 07]. [Yeramilli 06] observes that the increasing use of design resources and equipment costs in post-silicon validation makes it prohibitively expensive in the future. According to [Abramovici 06], post-silicon validation may consume 35% of average chip development time.

Post-silicon validation involves three major steps [Josephson 06, Livengood 99, Wagner 06, Sarangi 07]: 1) detecting a problem (e.g., through system crash, segmentation fault or error detection) by applying proper stimulus; 2) localizing and identifying the root cause of the problem; and, 3) fixing or bypassing the problem. For the second step, post-silicon bug localization involves identifying the bug location and an instruction sequence that exposes the bug. The bug localization step often dominates post-silicon validation efforts [Josephson 06] and is the focus of this paper.

Major factors that contribute to the high cost of current post-silicon bug localization approaches (details in Sec. V) are:

1. *Failure reproduction* involves returning the hardware to an error-free state, and re-executing the failure-causing stimulus (including instruction sequences, interrupts, and operating conditions) to reproduce the same failure. In a system environment, it may be very costly to reproduce a failure, especially for *electrical bugs* [Josephson 01], which manifest themselves only under certain operating conditions. Examples of electrical bugs include setup and hold time problems, synchronization problems, noise, and circuit marginalities. The expense of the reproduction is exacerbated by the presence of asynchronous I/Os, and multiple clock domains. Techniques to make failures reproducible [Heath 04, Sarangi 06, Silas 03], are intrusive to system operation and may not expose important bugs.

2. RTL system simulation for obtaining golden responses is several orders of magnitude slower than silicon speed, and also requires expensive external logic analyzers to record all primary I/O signals in cycle accurate fashion [Silas 03].

The objective of *IFRA,* Instruction Footprint Recording and Analysis, is to overcome these post-silicon bug localization challenges. Figure 1.1 shows a post-silicon debug flow using IFRA. During chip design, a processor is augmented with low-cost hardware recorders (Sec. II) for recording *instruction footprints* – semantic information describing data and control flows of dynamic instructions as they pass through various parts of the processor. During post-silicon validation, instruction footprints are recorded concurrently with system operation in a circular fashion to capture the last few thousand cycles of history before a failure manifests. After the failure manifests, the recorded footprints are scanned out through a Boundary-scan JTAG interface. The footprints, together with the test program binary executed during post-silicon validation, are then post-processed using special analysis techniques (Sec. III), in order to identify the bug location and the bug exposing stimulus. The location is provided in terms of microarchitectural blocks, such as control FSMs for various arrays of storage elements, pipeline registers, adders, decoders, etc. The stimulus is provided in terms of a short instruction sequence that enters each microarchitectural block. These

Part of this paper was presented at the Design Automation Conference (DAC) 2008, S.B. Park, S. Mitra, "IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors".

S.B. Park and T. Hong are with the Department of Electrical Engineering, Stanford University, (e-mail: sbpark84@stanford.edu, tedhong@stanford.edu).

S. Mitra is with the Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA (e-mail: subh@stanford.edu).

analysis techniques do not require failure reproducibility or RTL simulation.

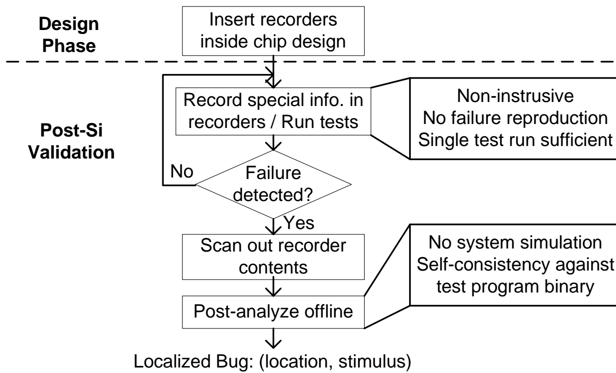

Figure 1.1. Post-silicon debug flow using IFRA.

Once a bug is localized using IFRA, existing circuit-level debug techniques [Caty 05, Josephson 06] can then quickly identify the root cause of bugs, resulting in significant gains in productivity, cost, and time-to-market.

In this paper, we demonstrate the effectiveness of IFRA for an Alpha 21264-like superscalar processor model. This model is sufficiently complex, yet its structured architecture provides opportunities for efficient bug localization. The primary goal of IFRA is to localize electrical bugs, since they require considerable post-silicon validation efforts [Patra 07]. Extensive IFRA simulations demonstrated:

1. For 75% of injected bugs, IFRA pinpointed their exact location-time pairs. For 21% of injected bugs, IFRA correctly identified their location-time pairs together with 2 to 6 other candidates, on average. IFRA completely missed correct location–time pairs for only 4% of injected bugs.
2. IFRA does not require any system-level simulation or failure reproducibility.
3. IFRA hardware introduces a very small area impact of 1% (including 60KBytes of distributed on-chip storage).

Major contributions of this paper are:

1. Introduction of IFRA to bridge a major gap between system-level and circuit-level debug, by allowing quick localization of bugs to a few design blocks from anomalous system-level behaviors.
2. Low cost methodology for recording control and data flows of dynamic instructions in a compact and non-intrusive manner.
3. Off-line program analysis techniques to analyze the recorded information for bug localization without requiring system-level simulation and failure reproduction.
4. Demonstration of the effectiveness of IFRA for a complex super-scalar processor.

Section II describes the hardware support required for IFRA. Section III describes the off-line program analysis techniques performed on the recorded information. Section IV presents simulation results, followed by an overview of related work in Sec. V, and conclusions in Sec. VI.

II. IFRA Hardware Support

We use an Alpha 21264-like superscalar processor model [Alpha 99] to explain the IFRA recording infrastructure. The shaded parts in Fig. 2.1 indicate the three hardware components: an ID assignment unit, a set of distributed recorders with dedicated storage, and a post-trigger generator.

As an instruction is fetched, the ID assignment unit tags it with an ID that will uniquely identify it later during the post-analysis. The ID is under the same control as the instruction it is associated with. For example, the ID receives the same stall and invalidate/flush signals as the instruction does, and when the instruction is stored in a queue, the ID is also stored in a queue receiving the same control signal.

While the tagged instruction flows through each of the pipeline stages, it generates an instruction footprint and stores it into the recorder associated with the pipeline stage. Each footprint consists of:

1. The instruction's identification number that was tagged
2. *Auxiliary information*, which tells us what the instruction did in the microarchitectural blocks contained in that pipeline stage.

The post-trigger generator is responsible for detecting a failure, stopping the recording and draining the footprints out of the recorders through the scan chain. The rest of the section will go through each of the three hardware components in more details.

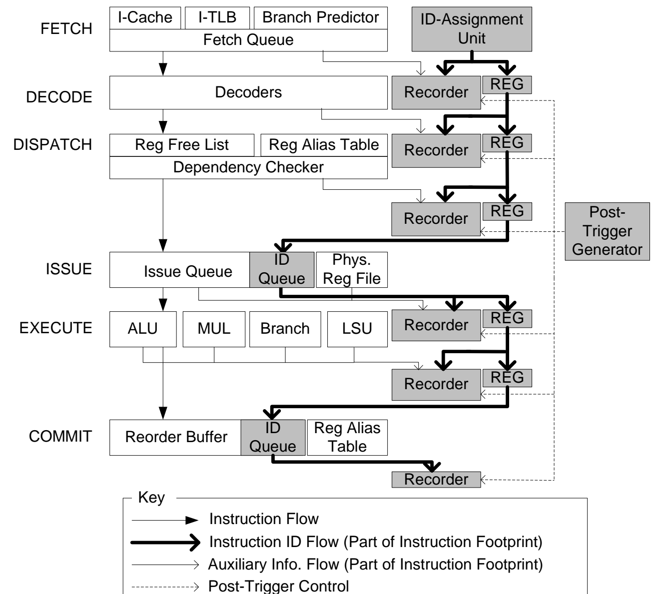

Fig. 2.1. Superscalar processor augmented with recording infrastructure. The figure only shows the flow for a single way for simplicity.

A. ID-assignment Unit

A special ID assignment scheme is used so that all uncommitted instructions can be identified during the post-analysis, and to ensure that no instructions with identical IDs change their relative orders in any of the recorders. The scheme is as follows. For a processor with at most n instructions in-flight, each instruction ID is $\log_2 4n$ bits wide. Instruction IDs are assigned to individual instructions as they

exit the fetch stage and enter the decode stage as shown in Fig. 2.2.If an ID X has been assigned in the previous cycle, and there are k instructions that exit the fetch stage in the current cycle, then k IDs, X+1 (mod 4n), X+2 (mod 4n)…X+k (mod 4n) are assigned to the k instructions. When an instruction with ID Y causes a pipeline flush, ID of Y+2n+1 (mod 4n) is assigned to the first instruction that is fetched after the flush completes.

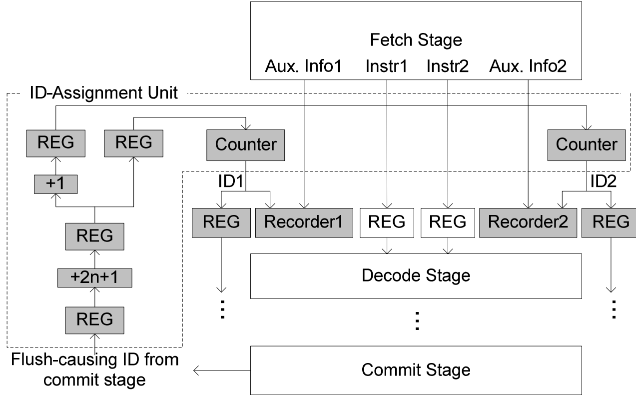

Fig. 2.2. The area enclosed by the dashed line indicates the ID-assignment unit for a 2-way processor. Shaded parts indicate the newly added hardware.

In Appendix A, we formally prove how the specified ID assignment scheme can be used to uniquely identify each instruction during the post-analysis, even in the presence of pipeline flushes, multiple clock domains, and dynamic voltage and frequency scaling in each of the clock domains.

B. *Instruction Footprint Recorder*

Fig. 2.3 shows the internal structure of the recorder. One recorder is associated with one way of a pipeline stage (e.g., four recorder's are associated with a 4-way fetch stage). Each recorder records footprints of instructions as they leave the pipeline stage. The main circular buffer acts as storage for instruction footprints and each buffer entry contains three fields: 1-bit *idle* field, *instruction ID/idle cycle count* field and *auxiliary information* field.

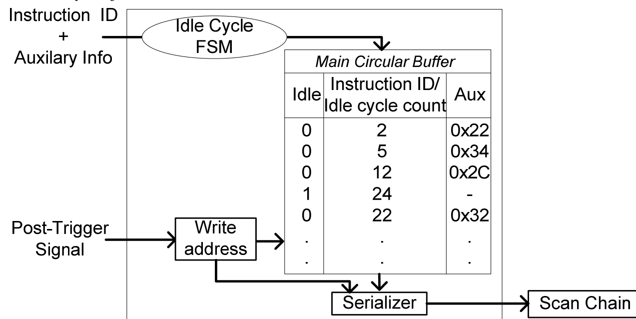

Fig. 2.3. Recorder's internal structure

Idle = 1 indicates that no instruction has passed by (i.e., NOPs detected), and the number of consecutive idle cycles are stored in the second field, with the third field left blank. The Idle cycle FSM is responsible for maintaining idle cycle counts. Idle=0 indicates that an instruction has passed by and left its instruction ID in the second field and its auxiliary information in the third field. For example, in Fig 2.3, the first three entries of the circular buffer correspond to instructions with IDs 2, 5, 12. Next, there are NOPs for 24 cycles, followed by an entry for instruction with ID 22.

The auxiliary information records information specific to the pipeline stage where the recorder is inserted. Table 2.1 shows auxiliary information collected in each pipeline stage of an Alpha 21264-like 4-way superscalar processor (detailed configuration in Sec IV). The third column indicates the number of bits of auxiliary information for each recorder, and the last column indicates the total number of recorder required for each pipeline stage. The commit-stage recorder has a different structure from the rest; it has a single register that records the ID of the youngest committed instruction, rather than having a circular buffer.

TABLE 2.1. AUXILIARY INFORMATION FOR EACH PIPELINE STAGE.

Pipeline stage	Auxiliary information	# bits	Num
Fetch	Program Counter	32	4
Decode	Decoding results	4	4
Dispatch	2-bit residue of reg. name	6	4
Issue	3-bit residue of operands	6	4
ALU, MUL	3-bit residue of result	3	4
Branch	None	0	2
LSU	3-bit residue of result; memory address	35	2
Commit	Exceptions	~0	4

C. *Post-trigger Generators*

In order to ensure that the entire error-to-failure history is captured using reasonably sized recorders, we assume the presence of early failure detection mechanisms, *post-triggers,* for the failure scenarios listed in Table 2.2. Detection of any one of the failure scenarios terminates the recording.

TABLE 2.2. FAILURE SCENARIOS AND POST-TRIGGERS.

Failure Scenario	Post-triggers	
	Soft	Hard
Array error	-	Parity check
Arith. error	-	Residue check
Exceptions	-	In-built exceptions
Deadlock	Short (2 mem loads) instruction retirement gap	Long (2 secs) instruction retirement gap
Segfault	Tlb-miss + Tlb-refill	Segfault from OS; Address equals 0

We assume the presence of parity bits for arrays (e.g. register file, reorder buffer, register rename table, register free list, scheduler, and various queues). We also assume the presence of residue codes for arithmetic units in ALUs and address calculators. Such parity bits and residue codes exist in several commercial processors [Ando 03, Leon 06, Sanda 08]. Unimplemented instruction exceptions and arithmetic exceptions are already present in most processors. This refers to the first three scenarios.

Unlike the first three failure scenarios listed in Table 2.2, the last two failure scenarios may be detected several millions of cycles after an error occurs. In order to prevent the history

recorded in the recorders to be overwritten during this time, we introduce the notion of soft and hard post-triggers. A *hard post-trigger* fires when there is an evident sign of failure, while a *soft post-trigger* fires when there is an early symptom of possible failure. A hard post-trigger causes the recording and the processor operation to terminate. A soft post-trigger causes the recording in all recorders to pause, but allows the processor to keep running. If a hard post-trigger for the failure corresponding to the symptom occurs within a pre-specified amount of time, the processor stops. If a hard post-trigger does not fire within the specified time, the recording resumes assuming that the symptom was false.

For deadlocks, a soft post trigger event fires when no instruction retires within the time required to perform two memory loads. The corresponding hard post trigger event is two additional seconds of no retirement.

For segmentation fault (or segfault), there is a single hard post-trigger to detect null-pointer dereference, and a pair of soft and hard post-triggers to detect illegal reading/writing into unallocated memory or writing into read-only memory. Null-pointer dereference is detected by adding simple hardware to detect whether the memory address equals zero in the Load/Store unit. For other illegal memory accesses, TLB-miss is used as the soft post-trigger. If a segfault is not declared by the OS while servicing the TLB-miss, the recording is resumed on TLB-refill. Since the recording is paused in the event of a soft post-trigger, there may be a period of time that may act as a blind spot during post-silicon validation.

III. POST-ANALYSIS TECHNIQUES

After a hard post-trigger fires, all recorder entries, together with the write addresses of the circular buffers, are scanned out through the JTAG interface. The localization analysis begins by combining recorder contents with the test program binary in a process called *footprint linking* (Sec. III.A). The end result shows where each dynamic instruction was present and what each instruction was doing at each time instance. Next, four high-level post-analysis techniques (Sec. III.B) targeting different parts of the processor are run on the linked footprints. If one or more high-level techniques identify inconsistency in the flow, low-level post-analysis techniques (Sec. III.C) are performed starting from the earliest occurrence of the inconsistency. If no inconsistency is found, the low-level analysis starts from the youngest entries of the recording.

A. Footprint Linking

During system run, a fetched instruction drops multiple footprints across multiple recorders. Footprint linking is a process of relating the multiple footprints back to the single instruction that dropped the footprints. This method works for a processor supporting pipeline flushes, out-of-order execution and multiple clock domains, where each domain can undergo dynamic voltage and frequency scaling. The algorithm completes the following five steps.

1. Within recorders associated with the in-order pipeline stages, identify all the instructions that caused a pipeline flush.
2. Identify all the uncommitted and committed instructions.
3. Select the youngest committed instruction in the dispatch-stage recorder and find its instruction ID. For each of the other recorders, find the youngest footprint with the same ID as the youngest committed instruction. All these footprints correspond to a single dynamic instruction and they should be linked together.
4. Step 3 is repeated for all the committed instructions starting from the youngest to the eldest committed instructions present in the dispatch-stage recorder.
5. Step 3 is repeated on the uncommitted instructions.
6. PCs from the fetch-stage recorder are mapped to the instructions in the binary.

B. High-level Analysis

1) Data Dependency Analysis

Our first post-analysis approach is to verify whether data dependency order is preserved, i.e., if there is a producer-consumer relationship in the serial execution trace, whether the consumer instruction executes after the producer instruction has produced its result. The analysis is performed on instruction issue sequence (obtained from issue-stage recorders) and the serial execution trace (derived from the fetch-stage recorders and commit-stage recorders). Consider the example in Fig. 3.1. Architectural register names (rather than physical register names) are obtained from the assembly code instruction mapping done in step 6 of the linking.

Fig. 3.1 illustrates an example of data dependency analysis. Since instruction with ID 0 shown in the serial trace produces a value on R0, while the instruction with ID 3 consumes a value from R0, data dependency exists between those two instructions. Instruction with ID 0 enters the ALU while the instruction with ID 3 enters the multiplier (shown in the execution-stage recorders). Assume that the two functional units are in different clock domains, and also assume that the ALU has a latency of 3 cycles. Since the two dependent instructions are in different clock domains with a possibility of dynamic frequency scaling, it is not possible to directly check their relative timing. However, we know that the issue-stage recorders must be in a single clock domain, and thus know that the instructions with ID 3 and ID 5 must be issued at the same time (shown in the issue-stage recorders). In this case, the distance between 0x03 and 0x00 is only two cycles, which is shorter than the 3-cycle latency of the ALU. This implies the consumer instruction with ID 3 was issued prematurely, before the producer instruction with ID 0 has completed.

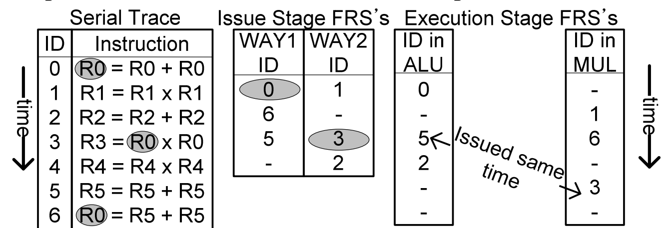

Fig.3.1. Data dependency analysis example.

Any inconsistency identified using this analysis is then

further analyzed by the low-level analysis described in Sec. III.C and Appendix B to identify an error in one of following microarchitectural blocks of the processor: registers at dispatch, issue, execute stages; issue buffer entries and issue buffer's control; forwarding paths among dispatch, issue, execute stages; register files; register renaming of dispatch stage.

2) Program Control Flow Analysis

Our second post-analysis technique verifies program control flow. There are four illegal cases that are checked for:

1. Transition in the absence of control flow transition instruction (instruction that changes the PC, e.g., branch, jump).

2. No transition in the presence of unconditional transition instruction (instruction that always changes PC value)

3. Illegal target in the presence of direct transition (with target address that does not depend on a register value).

4. Illegal target in the presence of indirect transition (with target address that depends on a register value).

Fig. 3.2 shows an example serial execution trace (derived from recorder data from fetch and commit stage) illustrating all four illegal cases. The first two cases can be checked by checking PC. The third case can be easily checked since the instruction itself contains all the necessary information to compute the target address. The fourth case is checked by checking whether the address addition has been done correctly using residue arithmetic.

A violation in the program control flow is further analyzed by the low-level analysis to identify an error in one of the following microarchitectural blocks: address calculator in execution stage; all pipeline registers between fetch and execution stages; forwarding path between execution and fetch stage; speculation recovery; register renaming.

Serial Execution Trace

	PC	INSTRUCTIONS
case 1	0x00	Add instruction
case 2	0x20	Unconditional jump to 0x40
case 3	0x08	Conditional branch to 0x20
	0x30	Assign 8 to register r0
case 4	0x34	UnConditional jump to r0
	0x40	-

Fig. 3.2. Four illegal cases of control flow transitions.

3) Load/Store Analysis

The third post-analysis technique involves verifying that a stored value to a memory address matches the value that is later loaded from that same address. In the absence of DMA activity, which may modify the memory content, any mismatch indicates a bug in the load/store unit or memory system (cache, memory controller, memory, etc) external to the processor core. In order to check for such mismatches, for each load/store instruction, memory addresses and residue of memory contents in Load/Store-unit recorders are recorded. A detailed localization approach can be found in Appendix B. Memory addresses affected by DMA activities may be factored out during post-analysis by recording the instructions sent to DMA engines using external logic analyzer.

4) Decoding Analysis

This technique checks whether all committed instructions are decoded correctly and whether they pass through the correct sequence of modules without disappearing or being distorted in the middle. Recording part of the decoded instruction bits (which functional unit should the instruction go to, how many operands it uses and whether it requires a destination register) verifies the operation of the decoder. Checking that instructions went to the correct functional units ensures that there was no corruption in the decoded opcode field of pipeline registers. Corruption of pointers or states associated with regular array structures is checked by observing whether instructions appear or disappear in the middle of a pipeline. For example, corruption of the empty flag bit of an issue buffer results in sudden disappearance of instructions.

C. *Low-level Analysis*

The low-level analysis mainly involves checking for consistency in residue bits collected by recorders (shown in Table 2.1). Residues of operands used by consumer instructions must match the residues of results produced by producer instructions. Additionally, the residues of physical registers used by consumers must match the residues of physical registers in which the producers placed their results. The localization refinement comes from comparing the right set of residues, in addition to knowing which post-triggers were activated and knowing which high-level post-analysis techniques detected inconsistencies. A brief description of the low-level analysis can be found in Appendix B.

IV. RESULTS

We evaluated IFRA by injecting errors into a micro-architectural simulator augmented with IFRA, as described in Sec. II. Post-analysis techniques described in Sec. III are used for bug localization. We used Simplescalar 3.0 architectural simulator [Simplescalar] with Alpha 21264 configuration (4-way pipeline, 64 maximum instructions in-flight, 2 ALUs, 2 multipliers, 2 load/store units). For this particular configuration, there are 200 different microarchitectural blocks. The number does not include the array-like structures and arithmetic units that are protected by parity and residue respectively. Each block has an average size equivalent of 10K 2-input NAND gates. The microarchitectural blocks can be inferred from low-level analysis shown in Appendix B. Seven benchmarks from SPECint2000 (bzip2, gcc, gap, gzip, mcf, parser, vortex) were chosen as validation test programs. The recorders were designed to collect information according to the setup described in Table 2.1. Each recorder was sized to have 1,024 entries.

All bugs were modeled as single bit-flips to target hard-to-repeat electrical bugs that pose major post-silicon validation challenges. Many electrical bugs affect speed paths [Silas03], and speed paths manifest themselves as incorrect values arriving at flip-flops for certain input combinations and operating conditions.

Errors were injected in one of 1,191 flip-flops (Table 4.1).

Note that, no errors were injected in array-like structures since they have built-in parities for error detection. Errors were injected in input / output registers and various control registers controlling the array structures. Pipeline registers in Table 4.1 include decoded opcode, register specifiers, immediate data, address, offset, etc. Valid bits indicate whether a given instruction is valid or not in a pipeline register.

TABLE 4.1. ERROR INJECTION BITS.

Description	# bits
PC, next PC	128
Memory Address used by Load/Store	128
Input/Output latch of Array Structures	82
Pointers to Array structures	23
Control states of Array Structures	4
Pipeline Registers	800
Valid Bits	26

Upon error injection, the following scenarios are possible:

1. The error has no effect at the system level.

2. The error does not cause any post-trigger mechanism to trigger, but produces incorrect program output.

3. Failure manifestation with short error latency, where recorders successfully capture the history from error injection to failure manifestation (including situations where recording is stopped upon activation of soft post-triggers).

4. Failure manifestation with long error latency, where 1024-entry recorders fail to capture the history from error injection to failure (including soft triggers).

Cases 1 and 2 are related to coverage of validation test programs and post-triggers, and are not the focus of this paper. Hence, error injection runs resulting in these cases are ignored and not reported. Any error injection run which does not result in the activation of any post-trigger within 100K cycles from error injection are included in this category. For errors resulting in cases 3 and 4, we report results in Tables 4.2 and 4.3. For case 4, we pessimistically report that our IFRA approach completely misses correct bug location-time pair (included under "completely missed" category in Tables 4.2). All error injections were performed after a million cycles from the beginning of the program in order to demonstrate that the history between error injection and failure manifestation is sufficient for effective post-silicon bug localization as is the case with IFRA.

Tables 4.2 and 4.3 present results from 800 error injections that resulted in cases 3 and 4. The "*exactly located*" category represents the cases in which the error injection location and time exactly matched with the localized hardware block and the stimulus. The percentage of bugs belonging to this category must be very high for an effective bug localization technique. The "*candidate located*" category represents the cases in which the IFRA produced several localized hardware blocks with stimulus, and at least one of them matched with the error injection location and time. The "*completely missed*" category represents the cases where the error injection location and time did not match any of the localized hardware blocks and stimulus. An effective bug localization technique must have very few "completely missed" cases. **It is clear from Table 4.2 that a large percentage of bugs were uniquely located to correct location-time pair, while very few bugs were completely missed, demonstrating the effectiveness of IFRA.** For "candidate located" cases, Table 4.3 reports statistics on the number of possible candidates. It is clear from Table 4.3 that the number of such candidates is very small.

TABLE 4.2 IFRA BUG LOCALIZATION SUMMARY.

Exactly Localized	75%
Correctly Localized with Candidates	21%
Completely Missed	4%

TABLE 4.3. STATISTICS FOR "CANDIDATE LOCATED" CASES.

Post-analysis technique	Number of candidates			
	Mean	**Min.**	**Max.**	**Std.Dev**
Data dependency	6.3	2	34	7
Control-flow	5.3	2	10	4.2
Load / Store	2	2	2	0
Decoding	2.4	2	3	0.55

Our synthesis result (Synopsys Design Compiler with TSMC 0.13 microns library) shows that the area impact of IFRA infrastructure is 1% on the Illinois Verilog Model [IVM] (an open-source RTL implementation of Alpha-like core) assuming a 2MB on-chip cache, which is typical of the current desktop/server processors. The overhead is largely dominated by the circular buffers present in the recorders, because of the absence of any global at-speed routing and the simplicity of the recorder's control. Total information storage for all recorders adds up to 60 Kbytes, which is a very small fraction of total on-chip storage.

V. RELATED WORK

Related work on post-silicon validation can be broadly classified into six categories: scan dump [Caty 05, Dahlgren 03], check-pointing with deterministic replay [Silas 03, Sarangi 06], embedded trace buffers for hardware debugging [Anis 07], on-chip program and data tracing [MacNamee 00], fault-tolerant computing [Austin 99, Lu 82, Oh 02], and on-line assertion checking [Abramovici 06, Bayazit 05, Chen08].

Debugging techniques using scan dump, checkpointing with deterministic replay, and embedded trace buffers require failures to be reproducible. Moreover, they require simulation for comparison of observed states against golden responses. If easy failure reproduction support is present, it will also help IFRA by allowing recorders to record unlimited lengths of history through repeated recording and dumping.

On-chip storage of program and data traces [MacNamee 00], commonly used in embedded processors (e.g. ARM, Motorola's MPC, Infineon's Tricore), have some similarity with IFRA in that they also store program flow of the software executed on the processor. However, they are fundamentally different because they target software debugging running on correct hardware, thus need to store very different forms of information than IFRA.

The difference between IFRA and traditional fault-tolerant computing is that the latter mainly focuses on error detection and recovery, while IFRA focuses on bug localization. That

FRA does not interfere with the system behavior (no code modification or resource conflicts) is essential.

On-line assertion checking techniques are complementary to IFRA in that such techniques can be efficiently used to generate post-triggers and also for fine-grained bug localization together with the post-analysis techniques supported by IFRA.

VI. Conclusion

IFRA targets the problem of post-silicon bug localization in a system setup, which is a major challenge in processor post-silicon design validation. The major novelty of IFRA is in the introduction of a high-level abstraction for bug localization through new low-cost hardware recorders hat record semantic information about instruction data and control flows concurrently in a system setup, and special analysis techniques that analyze the recorded data for localization after failure detection. These design and analysis techniques enable IFRA to overcome major post-silicon bug localization challenges. 1. It helps bridge a major gap between system-level and circuit-level debug. 2. Failure reproduction is not required. 3. Self-consistency checks associated with the analysis techniques eliminate the need for full system-level simulation.

IFRA raises several interesting research questions that can be explored in the future. 1. Sensitivity analysis and characterization of the inter-relationships between post-analysis techniques, architectural features, error detection mechanisms, FRS sizes and bug types. 2. Wider variety of post-triggers based on assertions, e.g., [Abramovici 06, Bayazit 05], and symptoms [Wang 04]. 3. Applicability of IFRA for homogeneous / heterogeneous multi-core systems, and system-on-chips (SoCs) consisting of non-processor designs. 4. Applicability of IFRA to directed diagnostic test generation and fault diagnosis.

Appendix A: Footprint Linking

During system run, a fetched instruction drops multiple footprints across multiple recorders. Footprint linking is a process of relating the multiple footprints back to the single instruction that dropped the footprints. There are three challenges involved in linking, due to the structure of modern superscalar processors:

1. Instructions may get issued and execute out of program order.
2. With speculative execution, many instructions may get removed in the middle of the pipeline when there is a branch misprediction or an exception.
3. Not all recorders will be placed in the same clock domain, and each clock domain can undergo dynamic voltage and frequency scaling.

We show that by assigning instruction IDs with $\log_2 4n$ bits, where n is the maximum number of instructions-in-flight, to each instruction footprint, we can link all footprints together in the presence of the above three challenges. The proof is split into three; Section A.1 describes how to relate footprints within a pipeline stage and describes a method of creating a basic data structure that is to be used for further analysis. Section A.2 addresses the first challenge and Sec. A.3 addresses the second challenge. While doing so, we do not use any explicit timestamps or global synchronization mechanisms to show that the third challenge is overcome.

The reader must be aware that other shorter ID widths such as $\log_2(2n+2)$ bits or $\log_2 2n$ bits can be used instead of the $\log_2 4n$ bits presented in this paper. However, $\log_2(2n+2)$ bits requires an expensive modulo 2n+2 operation, and also has a more complicated proof. ID width of $\log_2 2n$ requires extra buffers to store flushed instructions and the proof is not as clean as the one for $\log_2 4n$ bits. Thus we have chosen to present the proof for the $\log_2 4n$ bits case. Note that the end results of all three variations are the same: footprints in distributed buffers are all linked together after identifying all the uncommitted instructions.

The reader should also be aware that using program counter value instead of the IDs does not work. The trouble comes when executing a loop, which produces multiple instances of the same instruction with the same PC value. In the presence of out-of-order execution, these multiple instances may execute out-of-order, after which, it would be impossible to tell which footprint corresponds to which instance of the instruction.

We make the following assumptions on the underlying superscalar processor structure [Shen 05]. The last assumption is the only limitation of the current linking method, and requires further investigation.

1. There are four in-order pipeline stages (fetch, decode, dispatch, commit) and two out-of-order pipeline stages (issue, execute), as shown in Fig 2.1. Input to the centralized instruction window takes place in-order (process called dispatch), and output from the instruction window takes place out-of-order (process called issue).
2. n is the number of instructions-in-flight and also the number of reorder buffer entries, and it is greater than 0. The number of centralized instruction window entry is always less than or equal to n, but we will assume it to be equal to n.
3. Mispredicted speculative branch can only cause a pipeline flush once it becomes non-speculative.
4. For the sake of the recorder's simplicity, and to avoid any global at-speed routing, each recorder is a simple circular buffer, without any capability of removing misspeculated instructions once they are placed in.
5. Individual pipeline stages, except the execution stage, are within their own clock domain.
6. Identical functional units are within a single clock domain. For example, if there are three ALUs, they are all in a single clock domain. If there are 2 Load/store units, they are in a single clock domain, but separate from the clock domain that ALUs are in.
7. When an external interrupt occurs, we stop fetching new instructions and allow the instructions that are already in the pipeline to retire. Thus an interrupt never causes a pipeline flush.

8. When an exception is detected, instead of handling it using OS, a hard post-trigger will be activated and halt the processor. Thus, the system operation is stopped before any pipeline flush is initiated by the exception. Consequence is that no instruction will ever cause pipeline flush twice, and thus all instructions causing a pipeline flush will always eventually commit.
9. All branch misprediction will only cause a pipeline flush after execute stage.

We use the following ID assignment scheme:

1. IDs are assigned as instructions exit the fetch stage and enter the decode stage.
2. If an ID X has been assigned in the previous cycle, and there are k instructions that exit the fetch and enter the decode stage in the current cycle, then k IDs, X+1 (mod4n), X+2 (mod4n)…X+k (mod4n), are assigned to the k instructions.
3. When an ID Y causes a pipeline flush, the first instruction, which is fetched after the flush completes, gets the ID Y+2n+1 (mod4n).

A.1 Interpreting recorder data within a pipeline stage

First of all, given the recorder contents, we need to identify the eldest and the youngest entries. The identification is aided by two values that are scanned out with the recorder content after a failure is detected by a hard post-trigger: the write pointer and the *full flag*. The full flag indicates whether the recording has ever been overwritten or not. The full flag is cleared when the recording begins and is set when the write pointer wraps around from 1023 to 0 during code execution. Figure A.1.1 illustrates how to identify the eldest and the youngest entries of a recorder's N-entry circular buffer given the full flag. If the full flag is cleared, 0th entry is the eldest and $(X-1)^{th}$ entry is the youngest entry. If the full flag is set, then a wrap around happened in the circular buffer, where X^{th} entry is the eldest and $(X-1)^{th}$ entry is the youngest. Once the eldest and the youngest entries are identified, the wrapped entries are unwrapped, so that all recorder contents now have their eldest entry at the bottom and the youngest entry at the top. After which, the compacted idle cycles are expanded back. For example, an idle entry with idle cycle count of 5 will expand to occupy 5 entries.

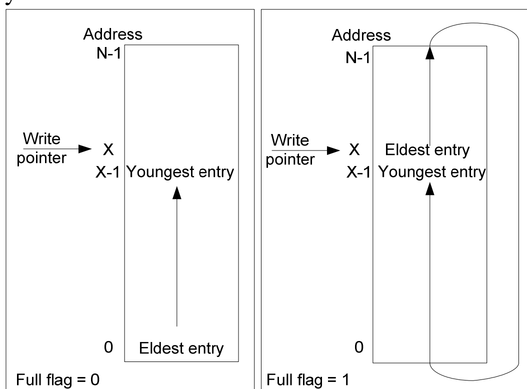

Figure A.1.1: Identification of the eldest and youngest entries of a recorder's N-entry circular buffer given the full flag. At the time of failure, the write pointer was pointing at address X.

Then for each pipeline stage, except the execution stage, we collect all the modified recorder contents associated with the stage, and juxtapose them so that the youngest entries are aligned. For the execution stage, juxtapose the recorder contents associated with each clock domain. Figure A.1.2 shows an example after juxtaposing recorder contents associated with a 4-way pipeline stage. Since the recorders within the same clock domain are all synchronized and their recordings stop simultaneously, aligning the youngest entry is sufficient to obtain the relative timing information among the recorders in the same clock domain, without using any explicit timestamps. After juxtaposing, the footprints in the same row correspond to the same clock cycle.

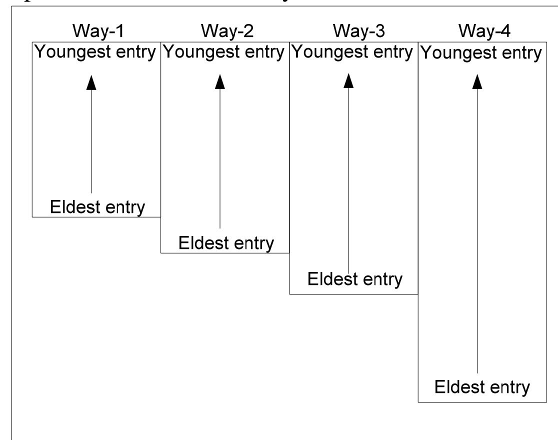

Figure A.1.2: Juxtaposing recorder contents associated with a 4-way pipeline stage

The length of history recorded by each recorder is not necessarily the same; some recorders encountered more idle cycles, and thus have recorded longer history. Given the juxtaposed recorder contents, since the post-analysis relies on each row to be complete, we discard all the incomplete rows. Thus, for a given pipeline stage, except the execution stage, if the recorder with the shortest history has M entries after expansion, then only M youngest entries from each of the recorder in the same pipeline stage will be kept. Figure A.1.3 shows that after discarding incomplete rows, we are left with an M-by-4 matrix of instruction footprints.

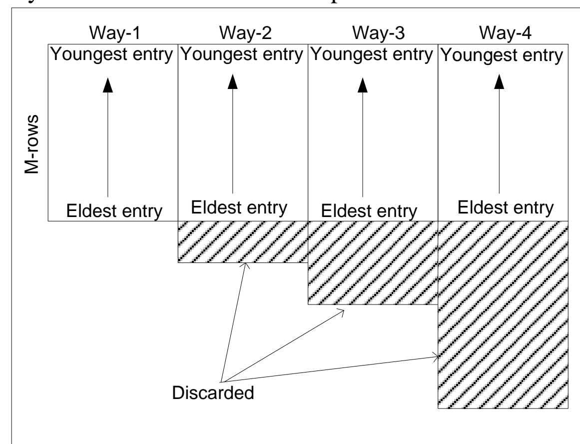

Figure A.1.3: Discarding incomplete rows from the example shown in Fig A.1.2

For the recorders associated with the out-of-order pipeline stages (issue and execute), we keep the footprint matrix as it is, which will be refer to as issue-stage footprint matrix and execute-stage footprint matrix. For the recorders associated with the in-order pipeline stages (fetch, decode, etc), we convert it to a one-dimensional array of footprints, where the (i, j) entry in the matrix corresponds to $(4i+j)^{th}$ entry in the array. We will refer to the array as fetch-stage footprint array, decode-stage footprint array, etc, hereafter.

In summary, we have found the relative timing information among the recorders associated with a single pipeline stage. The next section explains a method of finding relative timing information across multiple pipeline stages that could be in different clock domains.

A.2. Telling apart two instructions with the same ID

Finding the relative timing information across multiple pipeline stages is split into two steps. The first step, described in the current section, provides a theorem that could be used to tell apart any two instructions with the same ID. The theorem proven in the first step is used in the second step, described in Sec. A.3, to identify all the footprints associated with uncommitted instructions and then to link all the footprints together.

<u>Theorem 1</u>: *The relative order in which two instructions with the same ID appear in any footprint array or footprint matrix will never differ from the relative program order of the two instructions. The program order is defined to be the order in which instructions will execute in a single-issue, single-stage pipeline processor.*

In other words, if there are two instructions X and Y that have the same ID, and X is before Y in program order, then we claim that X will always occupy a younger entry than Y in any footprint arrays. In addition, for a footprint matrix, X will always occupy a younger row than Y. For footprint arrays, theorem 1 holds trivially, because instructions enter the in-order pipeline stages in program order. For footprint matrices that are associated with out-of-order pipeline stages, we use lemma 1.1-1.5 to prove the theorem.

<u>Lemma 1.1</u>: *If an reorder buffer entry is occupied by an instruction with ID X, the instruction in the next entry (younger entry) either has the ID X+1 (mod 4n) or X+1+2n (mod 4n), at any given time.*

Proof: There are two cases to consider: whether the instruction with ID X causes a flush or not. If X is a flush-causing instruction, then all the instructions having the ID of X+p (mod 4n), where $1 \le p \le n-1$, will get flushed from the pipeline, and the newly fetched instruction will have the ID of X+2n+1 (mod 4n), as described in the ID-assignment scheme. If X does not cause a pipeline flush, then the next instruction in program order, which is the instruction with ID of X+1 (mod 4n) is placed in the next entry.

<u>Lemma 1.2</u>: *If an reorder buffer entry is occupied by an instruction with ID X, the instruction in the kth younger entry either has the ID X+k (mod 4n) or X+k+2n (mod 4n), at any given time.*

Proof: Lemma 1.2 is an extension of lemma 1.1, and it will be proven using induction.

Base case of k =1 is true since it is precisely lemma 1.1.

Inductive case: Suppose lemma 1.2 is true for k^{th} younger entry, thus the kth younger entry either contains an instruction with ID X+k (mod 4n) or X+k+2n (mod 4n). Let's consider the two cases separately. If the entry has the ID X+k (mod 4n), then according to lemma 1.1, the next entry, the $(k+1)^{th}$ entry should contain the ID of X+k+1 (mod 4n) or X+k+1+2n (mod 4n). If the entry has the ID X+k+2n (mod 4n), then the next entry, the $(k+1)^{th}$ entry should contain the ID of X+k+2n+1 (mod 4n) or X+k+2n+1+2n (mod 4n). The last ID is equivalent to X+k+1 (mod 4n), because 4n (mod 4n) = 0. Thus, lemma 1.2 holds for all k, where k is a positive integer.

<u>Lemma 1.3</u>: *If a reorder buffer entry is occupied by an instruction with ID X, the instruction in the kth younger entry has an ID distinct from X, at any given time, where $1 \le k \le n$.*

Proof: From lemma 1.2, the kth younger entry either has the ID X+k (mod 4n) or X+k+2n (mod 4n). Let's consider the two cases separately.

To prove $X+k \pmod{4n} \neq X \pmod{4n}$, it is sufficient to prove $X < X+k < X+4n$:

$1 \le k \le n$

$\rightarrow X+1 \le X+k \le X+n$

$\rightarrow X < X+k < X+n$, because $X < X+1$ and $X+n < X+4n$ for $n > 0$.

To prove $X+k+2n \pmod{4n} \neq X \pmod{4n}$, it is sufficient to prove $X < X+k+2n < X+4n$:

$1 \le k \le n$

$\rightarrow X + 1 + 2n \le X + k + 2n \le X + 3n$

$\rightarrow X < X+k+2n < X+4n$, because $X < X+1+2n$ and $X+3n < X+4n$ for $n > 0$.

<u>Lemma 1.4</u> *: All instructions in an n-entry reorder buffer have distinct ID at any given time.*

Lemma 1.4 is a corollary of lemma 1.3.

<u>Lemma 1.5</u>: *The relative issue/execution order of two instructions with the same ID will never differ from the relative program order of the two instructions.*

Only the instructions that coexist in the reorder buffer can switch their relative issue/execution order from the program order [Shen 05]. For example, if X is before Y in program order, and if there exists a time when both X and Y coexist in the reorder buffer, then Y has a chance of getting issued/executed earlier than X. However, X will always commit before Y. From corollary 1.4, no two instructions that were assigned the same ID can ever coexist in the reorder buffer in any given time. Thus the lemma 1.5 holds.

Since the issue-stage recorder and execution-stage recorder records instruction footprints as instructions gets issued and finish executing respectively, theorem 1 is a corollary of lemma 1.5. If one records the youngest committed instruction and there were no pipeline flushes, then theorem 1 would be sufficient to link all the instructions in all the footprint arrays and matrices together using the following algorithm.

1. Select the youngest committed instruction in the

dispatch-stage footprint array and find its instruction ID. For each of the footprint array and matrices in other pipeline stages, find the youngest instruction footprint with the same instruction ID as the youngest committed instruction. All these footprints correspond to footprints of a single dynamic instruction and they should all be linked together.

2. Repeat step 1 for all the committed instructions starting from the youngest to the oldest committed instructions present in the dispatch-stage footprint array.

The algorithm relies on linking instructions from the youngest entries, since the relative orders of instructions with the same ID will never swap. However, in the presence of pipeline flushes, which are common in today's processor supporting at least one of exceptions or branch prediction, the algorithm breaks. The algorithm relies on ID of a fetched instruction to always exist at each of the pipeline stages. If an ID of a fetched instruction is not present in one of the pipeline stages, then the linking algorithm will simply link the older instruction with the same ID. However this is a wrong ID to link to. Even if we somehow find out that the instruction was flushed, we do not know, especially in the out-of-order pipeline stages, whether it was flushed before the instruction was dispatched, issued, or executed. The next two sections will present a method and prove that it can identify all the uncommitted instruction in all the pipeline stages, so that the above algorithm can be used to link footprints.

A.3 Identification of uncommitted instructions

The aim of this section is to tell whether each footprint is committed or not, and to tell whether it is flush-causing and/or flushed or neither. Section A.3.1 and A.3.2 provides method for footprints recorded in in-order pipeline stages and out-of-order pipeline stages respectively. For each of the cases, two categories of uncommitted instructions are addressed: uncommitted instructions that were in-flight at the time of hard post-trigger activation and instructions that were removed (aka flushed) from the pipeline on discovery of branch misprediction.

A.3.1 Uncommitted instructions in in-order pipeline stages

Identifying uncommitted instructions in recorders associated with the in-order pipeline stages (fetch, decode, dispatch) are trivial. The first category of uncommitted instructions, which were in-flight at the time of post-trigger activation, is identified by recording the ID of the youngest committed instruction in the commit-stage. Any IDs occurring after the ID of the youngest committed instructions are labeled as uncommitted, as illustrated in Fig. A.3.1.

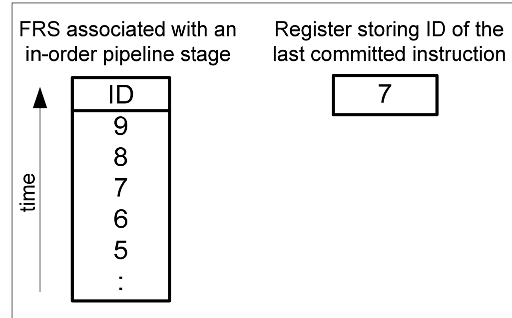

Figure A.3.1: Since ID 7 is the last committed instruction, ID 8 and 9 that occur after 7 are IDs of uncommitted instructions.

The uncommitted instructions of the second category are identified using theorem 2.

<u>*Theorem 2:*</u> *A pipeline flush in a footprint array is characterized by a jump in ID.*

Proof: Let X be the ID of a flush-causing instruction. At the time of pipeline flush, in the recorders associated with in-order pipeline stages, only IDs from X+1 (mod 4n) to X+n-1 (mod 4n) can exist after the ID X. The newly fetched instruction will get ID X+2n+1 (mod 4n), as described in the ID-assignment scheme. We prove that X+2n+1 (mod 4n) – X+k (mod 4n) >1 for all k between 1 and n-1:

X+2n+1 (mod 4n)– X+k (mod 4n) = 2n+1-k (mod 4n)

Since $1<=k<=n-1$, $2n>=2n+1-k>=n+1$

$\rightarrow$4n>2n>=2n+1-k>=n+1>1 for n>0

$\rightarrow$4n>2n+1-k>1

$\rightarrow$2n+1-k (mod 4n) $\neq$ 0 and $\neq$ 1

$\rightarrow$2n+1-k (mod 4n) >1

$\rightarrow$X+2n+1 (mod 4n) – X+k (mod 4n) >1 for all k between 1 and n-1

In order to find the uncommitted instructions using theorem 2, start from the youngest recorder entry, and look for any consecutive ID entries that do not differ by 1. For the example shown in Fig. A.3.2, there is a jump between ID 20 and ID 6. Once found, subtract 2n+1 from the younger of the two ID entries to identify the flush causing instruction, which is ID 3 in the example. All IDs between 3 and 20 are uncommitted instructions.

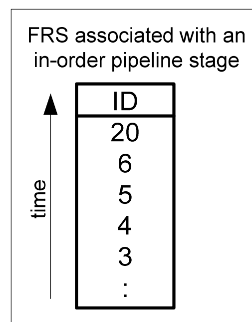

Figure A.3.2: Assumes n=8; Flush causing instruction ID is 3, and ID 4,5,6 are flushed instructions.

A.3.2. Uncommitted instruction in out-of-order pipeline stages

We will first of all identify the flushed instructions and then identify the uncommitted instructions that were in-flight at post-trigger activation. Since the identification method for

issue-stage and execute-stage footprint matrices are the same, we only consider the issue-stage footprint matrix without the loss of generality.

The identification of the flushed instruction must be done in conjunction with linking of the committed instructions. If the youngest committed instruction is flush-causing, we pause linking and immediately search for the flushed instructions, a method of which is described in the following paragraph. If it is not a flush-causing instruction, we can link the IDs corresponding to the youngest committed instruction using the algorithm presented in Sec. A.2, even if there could be as many as n instructions above the top-most committed instruction in the dispatch-stage footprint array. A simple application of lemma 1.4 tells us that none of these uncommitted instructions above the top-most committed instruction have the same ID as the top-most committed instruction. Thus, if the top-most committed instruction has ID Q, the top-most ID Q in the issue-stage footprint matrix corresponds to the youngest committed instruction, and they are linked together. The linking is continued for the committed instructions in the dispatch-stage footprint array, starting from the top-most committed instruction and working your way downwards until you encounter a flush-causing instruction. After which, we have to search for the flushed instructions.

For the rest of this section, let's denote W to be the ID of the flush-causing instruction and denote X to be the ID of an instruction that got flushed by ID W. The relationship between X and W is given by X = W+k (mod 4n), where 1<=k <=n-1. We would like to find out whether the flushed ID X reached the issue stage during the system validation run, and if it did, identify which ID X in the issue-stage footprint matrix corresponds to the flushed instruction.

In order to find whether the flushed instruction reached and left the issue stage, we only need to check for the presence of ID X in certain consecutive rows of the footprint matrix. The rows are bounded by a *younger isolating row* at the top and an *older isolating row* at the bottom. The elder isolating row is always below the row that contains the flushed ID X, if there is any, and always above the row that contains another instance of ID X that is the youngest among the ID Xs dispatched before the flushed ID X. Similarly, the younger isolating row is always above the row that contains the flushed ID X, if there is any, and always below the row that contains another instance of ID X that is the eldest among the ID Xs dispatched after the flushed ID X. If one has the ability to find the elder and the younger isolating row for a particular flushed ID, then it is trivial to find out whether the flushed instruction left the issue stage; if there is an ID X in the rows bounded by the isolating rows, then that is the flushed ID X we were looking for, if there is not any, we can conclude that the flushed instruction did not leave the issue stage. We will later show that the younger isolating row is unnecessary, but let's assume its necessity for now, for easier understanding.

A.3.2.1 Identifying the younger isolating row

We do not need a strict isolator between the flushed ID X and another ID X that is the eldest among the ID Xs dispatched after the flushed ID X. This fact is because of how we do the linking operation: we start from the youngest committed instructions and work our way towards the elder instructions using the algorithm in Sec A.2. .Thus, a one-way isolator is sufficient, where the flushed ID X cannot exist above, but the other X can exist below the isolating row.

<u>*Theorem 3:*</u> *Suppose there is a flush-causing ID W, and a flushed ID X that is flushed by W. Then the younger isolating row for ID X can be found using the following method: in the issue-stage footprint matrix, from the row containing the flush-causing ID W, search upwards until you encounter the first row that contains W+2n+1 (mod 4n).*

Proof: In the ID assignment scheme, W+2n+1 (mod 4n) is assigned to the newly fetched instruction after the flush completes. Thus, by definition, ID W+2n+1 (mod 4n) must have entered the recorders strictly after all the previously flushed instructions.

If the ID W+2n+1 (mod 4n) cannot be found because the recording stopped before fetching a new instruction after the flush, then use the row above the top most row as the younger isolating row.

A.3.2.2 Identifying the elder isolating row

<u>*Theorem 4*</u>*: Suppose there is a flush-causing ID W, and a flushed ID X that is flushed by W. Then the elder isolating row for ID X can be found using the following method: in the issue-stage footprint matrix, from the younger isolating row, search downwards until you encounter the first row that contains either X-n (mod 4n) or X-3n (mod 4n).*

We use lemma 4.1-4.4 to prove the theorem

<u>*Lemma 4.1*</u>*: If an instruction with ID X is occupying a reorder buffer entry, then the instruction in the kth older entry either has the ID X-k (mod 4n) or X-k-2n (mod 4n).*

Proof: It will be done using induction

Base case: Let instruction with ID Z be the instruction occupying the previous entry (1st older entry). If Z is not a flush-causing instruction, then X = Z+1 (mod 4n) or Z = X-1 (mod 4n). If Z is a flush-causing instruction, then X = Z+2n+1 (mod 4n) or Z = X-2n-1 (mod 4n)

Inductive case: Let instruction with ID Z(k) be the instruction occupying k^{th} older entry. Assume Z(k) = X-k (mod 4n) or X-k-2n (mod 4n). There are two cases to consider.

If Z(k) = X-k (mod 4n) and its previous instruction is not a flush-causing instruction, then Z(k+1) = X-k-1 (mod 4n) = X –(k+1) (mod 4n). If it is a flush-causing instruction, then Z(k+1)=X-k-2n-1 (mod 4n) = X-2n – (k+1) (mod 4n).

If Z(k) = X-k-2n (mod 4n) and its previous instruction is not a flush-causing instruction, then Z(k+1) = X-k-2n-1 (mod 4n) = X-2n-(k+1) (mod 4n). If it is a flush-causing instruction, then Z(k+1)=X-k-2n-2n-1 (mod 4n) = X-k-1 (mod 4n) = X-(k+1) (mod 4n).

<u>*Lemma 4.2*</u>*: Before instruction with ID X enters the re-order buffer, an instruction with ID X-n (mod 4n) or X-3n (mod 4n) must have existed and committed.*

Proof: By lemma 4.1, just before ID X enters the re-order

buffer, the youngest entry contains ID X-1 (mod 4n) or X-2n-1 (mod 4n). Also by lemma 4.1, the nth older entry contains ID X-n (mod 4n) or X-3n (mod 4n). Suppose X-n (mod 4n) was the one that was present, but did not commit. That implies that X-2n+j (mod 4n), where 1<=j<=n-1 is a flush-causing instruction that flushed X-n (mod 4n). After the flush, X+j+1 (mod 4n) is the ID of the newly fetched instruction. However, we have skipped the ID W, which is in the range X-k (mod 4n), where 1<=k<=n-1. We have a contradiction. The same argument can be made to say that if X-3n (mod 4n) was present before W, it must have committed.

Lemma 4.3: *The flushed instruction with ID X, if it exists, cannot co-exist with any of ID X-n (mod 4n) and X-3n (mod 4n) in the reorder buffer at any given time. Consequence is that ID X will always be in a row above the row that contains either X-n (mod 4n) or X-3n (mod 4n).*

Proof: Suppose X-n (mod 4n), rather than X-3n (mod 4n) is the committed one that is first encountered below the younger isolating row. Then lemma 1.4 tells us that the following IDs can be present in the reorder buffer at the same time as X-n (mod 4n): X-n+j (mod 4n), X-n+2n+j (mod 4n), where 1<=j<=n-1. However, none of them equals X. Now suppose X-3n (mod 4n) is the one of the two IDs that is first encountered below the younger isolating row. Lemma 1.4 tells us that the following IDs can be present in the reorder buffer at the same time as X-3n (mod 4n): X-3n+j (mod 4n), X-3n+2n+j (mod 4n), where 1<=j<=n-1. Again, none of them equals X.

Lemma 4.4: *Another instance of ID X that is the youngest among the ID Xs dispatched before the flushed ID X, does not coexist with any of X-n (mod 4n) and X-3n (mod 4n) in the reorder buffer. Consequence is that the other instance of ID X will always occur in a row below the row containing either X-n (mod 4n) or X-3n (mod 4n).*

Proof: Suppose X-4n (mod 4n) is in the reorder buffer. Then lemma 1.4 tells us that the following IDs can be present in the reorder buffer at the same time as X-4n (mod 4n): X-4n+j (mod 4n), X-4n+2n+j (mod 4n), where 1<=j<=n-1. However, none of them equal X-n (mod 4n) or X-3n (mod 4n).

A.3.2.3 Uncommitted instructions in-flight at the time of post-trigger activation

After linking all the committed instructions and identifying all the flushed instructions, what is left at the end are the uncommitted instructions in-flight at the time of post-trigger activation, making the identification trivial.

A.4 Summary

Figure A.4.1 shows the overall flow of the linking process. Note that while linking, we did not use any explicit synchronization or timestamps. Furthermore, even if the frequency of each pipeline stages varies dynamically, since we did not rely on any timing information, the linking process still works.

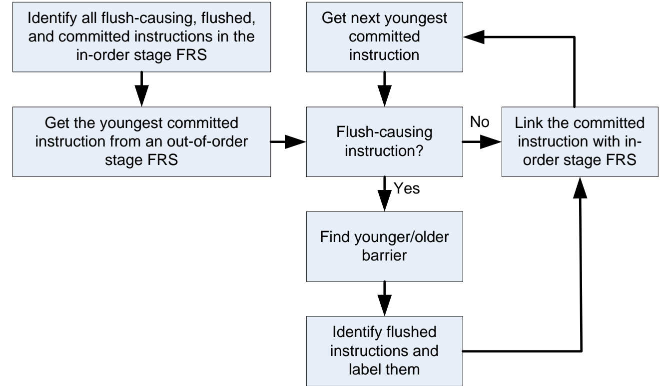

Figure A.4.1: Overall flow of the linking process

APPENDIX B: SUMMARY OF LOW-LEVEL ANALYSIS

1) IF array error (→1.0)
2) IF arithmetic error (→2.0)
3) IF alignment exception (→10.0)
4) IF unimplemented instruction exception (→11.0)
5) IF integer overflow exception (→12.0)
6) IF deadlock (→21.0)
7) IF instruction access segfault (→13.0)
8) IF data access segfault (→14.0)
9) IF control flow analysis violation (→13.0)
10) IF data dependency analysis violation (→20.0)
11) IF load/store analysis violation (→21.0)

1.0 Error in array element OR (→3.0)
2.0 Error in arithmetic unit OR (→3.0)
3.0 Error in exception generation unit
4.0 Error in register value at the output of execution stage (→6.0) OR
IF using arithmetic unit (→2.0)
IF using load/store unit (→21.0)
IF using complex ALU, then error in complex ALU
IF using branch unit (→2.0)
5.0 Error in speculative register alias table (-->16.0) OR (→15.0) OR (→18.0)
IF there exist previous read to the same architectural register name and it mismatches
IF there exist another read or write to the same architectural register name before the previously mentioned one
IF first read/write matches with thir read, but mismatches with second read then error in second read
IF first read/write matches with second read but mismatches with third read
IF there exist a fourth read
IF third read matches fourth read, then storage between second and third read is faulty
IF there exist read/write before third read to hamming-distance-1 address
IF third read equals hamming-distance-1 address's value, then addressing was faulty
ELSE value reading was faulty
ELSE IF fourth read matches second read, then third read faulty
ELSE third read OR storage faulty
IF there exist previous write to the same architectural register name and it mismatches
IF there exist another read afterwards

IF First write matches third read, but mismatches second read, then second read is faulty
ELSE IF second read and third read matches but mismatches with first write, then first write or storage between first write and second read are faulty
ELSE first write, second read and storage in between are all faulty

6.0 Error in register value at the input of execution stage
Displacement selection multiplexor (→7.0)
OR forwarding path (→8.0)

7.0 Displacement selection multiplexor
IF instruction is supposed to take immediate
IF operand residue doesn't match immediate residue
IF repeated inputs don't match output,
THEN error in multiplexor
ELSE Error in opcode or immediate (→17.0)
ELSE
IF immediate has been obtained from non-immediate field of the instruction
THEN opcode or immediate (→17.0)
ELSE physical register file (→5.0)

8.0 Forwarding path
IF data dependency analysis violation
THEN muxes + select signals
ELSE (→4.0) OR (→9.0)

9.0 Error in physical register file
(→4.0) OR wrong physical register name from RAT (→5.0) OR Similar analysis to 5.0 except use register value instead of physical register name and use physical register name instead of architectural register name

10.0 Error in decoder part 1
(→6.0) OR (→3.0) OR
(Size bits flipped between output of decode to input of address generator) OR
IF instruction is unaligned access
(Wrong decode: unaligned decoded to aligned) OR
(Unaligned access bit flipped between output of decode and input of address generator)

11.0 Error in decoder part 2
(→3.0) OR
IF exception at decode stage
Wrong instruction written from icache (parity protection) OR (→13.0) OR
IF fetched instruction doesn't match with instructions in binary
THEN error in fetch queue OR alignment&rotate unit
ELSE instruction word flip between fetch queue and input of decode stage OR wrong opcode decode OR wrong instruction written in fetch queue
ELSE IF exception at execution stage
Do the same check as above but the following is in addition:
Bitflip in doecded opcode field from output of decode stage to input of execute stage

12.0 Integer overflow
(→2.0) OR (→3.0) OR (→4.0)

13.0 Error in PC
IF control flow violation case 1
IF instruction went to branch unit
THEN opcode corruption (→17.0)
ELSE faulty nextPC select mux
IF control flow violation case 2
IF instruction went to non-branch unit
THEN opcode corruption (→17.0)
ELSE faulty nextPC select mux
IF control flow violation case 3,4
THEN (→4.0)

14.0 Error in address generator
(→6.0) OR (→2.0)

15.0 Error in architectural register name
Wrong decode or wrong decoded bits propagation (→16.0)

16.0 Wrong physical register name from register free list

17.0 Error in decoded bit propagation
IF decoded bits differ with re-simulated result THEN error

18.0 Speculation recovery
IF after flush, results of flushed instruction are used rather than results prior to flush THEN incorrectly not initiated recovery
IF latest results are not seen but older results are seen
THEN incorrectly initiated recovery

19.0 Error in architectural register alias table
Similar analysis to 5.0 but physical register name and architectural register names come from output of reorder buffer

20.0 Error in scheduler
Scheduler array OR
IF ID duplication (from decode analysis)
Incorrectly cleared issued bits in the array OR
Queue pointer flip
IF ID disappearance (from decode analysis)
Incorrectly cleared valid bit in array
OR queue pointer flip
IF deadlocked
IF ID disappearance then valid bit flip from issue until execution
ELSE incorrectly setting valid bit in array

21.0 Error in load/store unit (load/store analysis)

ACKNOWLEDGMENT

The authors would like to thank B. Gottlieb, N. Hakim, D. Josephson, P. Patra, J. Stinson from Intel Corporation, O. Mutlu from Microsoft Research, and E. Rentschler from AMD for their discussions and assistance during the course of this research.

REFERENCES

[Abramovici 06] Abramovici, et al., "A Reconfigurable Design-for Debug Infrastructure for SOCs", Proc. Design Automation Conf. , 2006.

[Alpha 99] Alpha 21254 Microprocessor Hardware Reference Manual, July 1999.

[Ando 03] Ando, H., et al., "A 1.3-GHz Fifth-Generation SPARC64 Microprocessor", IEEE JSSC, vol.38, no.11, pp. 1896-1905, Nov 2003.

[Anis 07] Anis, E. and N. Nicolici, "On using lossless compression of debug data in embedded logic analyzers", Proc. Intl. Test Conf., 2007.

[Austin 99] Austin, T.M., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", Proc. Intl. Symp. on Microarchitecture, 1999.

[Bayazit 05] Bayazit, A.A. and S. Malik, "Complementary Use of Runtime Validation and Model Checking", Proc. Intl. Conf. on Computer-aided Design, 2005.

[Caty 05] Caty, O, P. Dahlgren and I. Bayraktaroglu, "Microprocessor Silicon Debug based on Failure Propagation Tracing", Proc. Intl. Test Conf., 2005.

[Chen 08] Chen, K., S. Malik, and P. Patra. "Runtime Validation of Memory Ordering Using Constraint Graph Checking". Proc. Intl. Symp. on High-Performance Computer Architecture, 2008.

[Dahlgren 03] Dahlgren, P., P. Dickinson and I. Parulkar, "Latch Divergence in Microprocessor Failure Analysis", Proc. Intl. Test Conf., 2003.

[Heath 04] Heath M.W., W.P. Burleson and I.G. Harris, "Synchro-Tokens: Eliminating Nondeterminism to Enable Chip-Level Test of

Globally-Asynchronous Locally-Synchronous SoC's", Proc. Conf. on Design, Automation and Test in Europe, pp1532-1546, 2004.

[IVM] http://www.crhc.uiuc.edu/ACS.

[Josephson 01] D. Josephson, S. Poehhnan, V. Govan, "Debug methodology for the McKinley processor", Proc. Intl. Test Conf., pp451-460, 2001.

[Josephson 06] Josephson, D., "The Good, the Bad, and the Ugly of Silicon Debug", Proc. Design Automation Conf.., 2006.

[Leon 06] Leon, A.S., B. Langley, and J.L Shin "The UltraSPARC T1 Processor: CMT Reliability", Proc. Custom Integrated Circuits Conf., 2006.

[Livengood 99] Livengood, R. and D. Medeiros, "Design for (physical) Debug for Silicon Microsurgery and Probing of Flip-chip Packaged Integrated Circuits", Proc.Intl.Test Conf., 1999.

[Lu 82] Lu, D.J. "Watchdog Processors and Structural Integrity Checking", IEEE T COMPUT., pp.681-685, July 1982.

[MacNamee 00] MacNamee, C. and D. Heffernan, "Emerging On-chip Debugging Techniques for Real-time Embedded Systems", Computing & Control Engineering Journal, vol.11, no.6, pp.295-303, Dec 2000.

[Oh 02] Oh, N., S. Mitra and E.J.McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions", IEEE T COMPUT, vol.51, no.2, pp.180-199, Feb 2002.

[Patra 07] Patra, P., "On the Cusp of a Validation Wall", IEEE DES TEST COMPUT, vol.24 no.2, pp.193-196, Mar 2007.

[Parker 03] Parker K.P., The Boundary-Scan Handbook, 3rd ed., Springer, 2003.

[Sanda 08] Sanda P.N. et al., "Soft-error resilience of the IBM POWER6 processor", IBM Journal of Research and Development, vol.52, no.3, 2008.

[Sarangi 06] Sarangi, S.R., B.Greskamp and J.Torrellas, "CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging", Intl. Conf. on Dependable Systems and Networks, 2006.

[Sarangi 07] Sarangi, S.R., et al., "Patching Processor Design Errors with Programmable Hardware", IEEE MICRO, vol.27, no.1, pp.12-25, Jan 2007.

[Silas 03] Silas, I., et al., "System-Level Validation of the Intel Pentium M Processor", Intel Technical Journal, May 2003.

[simplescalar] www.simplescalar.com.

[Trong 07] Trong, S.D., et al., "P6 Binary Floating-Point Unit", Proc. Intl. Symp. on Computer Arithmetic, 2007.

[Wagner 06] Wagner, I., V. Bertacco and T. Austin, "Shielding Against Design Flaws with Field Repairable Control Logic", Proc. Design Automation Conf., 2006.

[Wang 04] Wang, N.J., et al., "Characterizing the effects of Transient Faults on a High Performance Processor Pipeline". Intl. Conf. on Dependable Systems and Networks, 2004.

[Yerramilli 06] Yerramilli, S., "Addressing Post-Silicon Validation Challenge: Leverage Validation & Test Synergy (Invited Address)", Intl. Test Conf., 2006.