THE UCLA MIRROR PROCESSOR: CONTROL IMPLEMENTATION AND TESTING CONSIDERATIONS FOR A VLSI RISC WITH MICRO ROLLBACK

Titus Lai

January 1991
CSD-910001
The UCLA Mirror Processor:  
Control Implementation and Testing Considerations  
for a VLSI RISC with Micro Rollback  

*Titus Lai*  

Computer Science Department  
University of California  
Los Angeles, California 90024-1596  
U.S.A.  

January 1991
Abstract

Many methods for achieving fault tolerance are based on the use of concurrent error detection — the outputs of every component are checked before being used by any other component. In order to minimize the performance degradation usually associated with concurrent error detection, checking can be done in parallel with the transfer of information between components. Since module state may be modified before an error is flagged, system modules support micro rollback — the ability to undo recent state changes and roll back the system to its state prior to the clock cycle in which the error originated.

The UCLA Mirror Processor is a RISC microprocessor capable of micro rollback. Concurrent error detection is accomplished by running two Mirror Processor chips in lock-step synchronization and comparing their external outputs and a compressed signature of their internal states.

This report details the function, timing, and implementation of the Mirror Processor controller, emphasizing the error detection and recovery mechanisms. The area overhead of the error detection and rollback logic is shown to be quite large: 67% of the controller area is dedicated to micro rollback. The performance overhead is found to be minimal. Functional testing, in which a processor chip is tested in software through the use of its normal instruction set, is shown to be adaptable to the Mirror Processor. Special instructions added for testing the error detection and rollback logic are presented, along with example test procedures using these instructions.
**Table of Contents**

Chapter 1: Introduction ........................................................................................................... 1
  1.1. Micro Rollback ............................................................................................................ 1
  1.2. Overview of Mirror Processor .................................................................................... 2
  1.3. Main Processor ........................................................................................................... 3
  1.4. Error Detection .......................................................................................................... 4
  1.5. Error Recovery .......................................................................................................... 4

Chapter 2: The Mirror Processor Controller .................................................................... 5
  2.1. Controller Design Considerations ............................................................................. 5
  2.2. Description of the Controller .................................................................................... 6
  2.3. Controller Timing ....................................................................................................... 8
    2.3.1. Processor Timing ................................................................................................ 8
    2.3.2. Memory Interface Timing .................................................................................. 8
    2.3.3. Normal Instruction Cycle .................................................................................. 11
    2.3.4. Interrupts and Traps .......................................................................................... 13
    2.3.5. Rollbacks ........................................................................................................... 14
  2.4. Details of Controller Implementation ........................................................................ 14
    2.4.1. busIR ................................................................................................................. 14
    2.4.2. busIR Latch and Driver ..................................................................................... 15
    2.4.3. Next State Logic ............................................................................................... 15
    2.4.4. PLA's ................................................................................................................ 16
    2.4.5. Condition Code Logic ....................................................................................... 17
    2.4.6. Post-FSM Logic and Valid Bit Logic ................................................................. 18
    2.4.7. Memory Control Logic ....................................................................................... 19
    2.4.8. Memory Enable Signals ...................................................................................... 19
    2.4.9. Interrupt Logic ................................................................................................... 20
    2.4.10. Comparator Logic ............................................................................................. 21
    2.4.11. Controller Rollback Memory .......................................................................... 23
    2.4.12. Critical Paths ................................................................................................... 24
  2.5. Alternate Wait State Timing ....................................................................................... 25
  2.6. Clock Generation ....................................................................................................... 26

Chapter 3: Error Detection and Recovery ........................................................................ 32
  3.1. Rollback Timing ........................................................................................................ 32
    3.1.1. Internal Rollback .............................................................................................. 32
    3.1.2. External Rollback .............................................................................................. 32
3.1.3. Absolute and Relative Rollbacks .......................................................... 32
3.2. Error Detection Timing ............................................................................. 33
3.2.1. Register File Parity Errors .................................................................... 33
3.2.2. busIN Parity Error ................................................................................. 33
3.2.3. Comparison Errors ................................................................................ 33
3.2.4. busOUT Parity Error .............................................................................. 33
3.2.5. Multiple Rollbacks ................................................................................ 34
3.3. Rollback Control Logic and Timing ............................................................. 36
3.3.1. Overview .................................................................................................. 36
3.3.2. External Rollback Signals ....................................................................... 37
3.3.3. Input Latches .......................................................................................... 37
3.3.4. Internal Error Signals ............................................................................. 37
3.3.5. Post-rollback Counter ............................................................................ 38
3.3.6. Rollback Enable Bit and Counter .......................................................... 38
3.3.7. Internal Rollback Logic ......................................................................... 38
3.3.8. Valid Bits ................................................................................................ 39
3.3.9. Selection Logic ....................................................................................... 39
3.3.10. Rollback Amount Encoder ..................................................................... 40
3.3.11. Rollback Amount Arbitration ............................................................... 40
3.3.12. Frame Counter and Rollback Counter .................................................. 41
3.3.13. Shutdown Logic .................................................................................... 41
3.3.14. Pad Enabling ........................................................................................ 42
3.3.15. Rollback Bit .......................................................................................... 42
3.3.16. Rollback Procedure .............................................................................. 42
3.3.17. Critical Path .......................................................................................... 43
3.4. State Repair ............................................................................................... 43

Chapter 4: Controller Implementation ............................................................... 48
4.1. Layout ........................................................................................................ 48
4.2. Overhead .................................................................................................... 48
4.2.1. Area Overhead ....................................................................................... 48
4.2.2. Transistor Overhead ............................................................................. 51
4.2.3. Examples ................................................................................................ 52
4.2.3.1. CMP .................................................................................................. 52
4.2.3.2. RB ..................................................................................................... 54
4.2.3.3. STATE .............................................................................................. 55

Chapter 5: Simulation and Debugging ................................................................. 57
5.1. Architecture Level Simulation ................................................................. 57
5.2. Switch Level Simulation ............................................................................ 58

Chapter 6: Testing ............................................................................................. 59
6.1. Testing Considerations .............................................................................. 59
List of Figures

Figure 1.1: Prototypical Register with Delayed Write Buffer .................................................. 2
Figure 1.2: Mirror Processor ................................................................................................. 3
Figure 2.1: Controller ............................................................................................................. 7
Figure 2.2: Memory Interface Timing .................................................................................... 10
Figure 2.3: Control Timing ................................................................................................... 12
Figure 2.4: busIR Latch and Driver ....................................................................................... 15
Figure 2.5: PLA Bit Slice ($\phi_{1b}$ PLA) ............................................................................... 18
Figure 2.6: Memory Address Enable Bit ............................................................................... 20
Figure 2.7: Comparator Logic Block ..................................................................................... 22
Figure 2.8: Clock Generator ................................................................................................ 27
Figure 2.9: d4 Delay Inverter ............................................................................................... 28
Figure 2.10: Slave Clock Generation Timing ....................................................................... 28
Figure 2.11: Clock Generation Timing .................................................................................. 29
Figure 2.12: $\phi_2$ Generation ............................................................................................. 30
Figure 2.13: $\phi_2$ Clock Generation Timing ........................................................................ 31
Figure 3.1: Rollback Control ............................................................................................... 36
Figure 3.2: Rollback Amount Arbitration Logic ................................................................... 41
Figure 3.3: Rollback Procedure ............................................................................................ 44
Figure 3.4: Rollback Control Timing .................................................................................... 45
Figure 3.5: Rollback Amount Arbitration ............................................................................ 46
Figure 3.6: State Repair Procedure ...................................................................................... 47
Figure 4.1: Controller Floorplan .......................................................................................... 48
List of Tables

Table 2.1: Data Size Line Encoding ................................................................. 9
Table 2.2: Interrupt Vectors ............................................................................ 21
Table 2.3: Comparison Logic Error Signals ..................................................... 22
Table 4.1: Blocks in Controller Layout ............................................................ 49
Table 4.2: Controller Implementation Statistics ............................................. 50
Table 4.3: Area Overhead, Dimensions .......................................................... 50
Table 4.4: Area Overhead, Percentage of Total Area ....................................... 51
Table 4.5: Transistor Overhead, Number Used .............................................. 52
Table 4.6: Transistor Overhead, Percentage of Total Number ....................... 53
Table 6.1: Instruction Argument Definitions ............................................... 64
1. INTRODUCTION

In a fault-tolerant computing system, the ability to detect errors and recover quickly from them is of utmost importance. In many systems, checkers are placed at the outputs of each module in the system in order to confine the error to that module and prevent it from causing damage elsewhere. However, because module outputs must be checked before being sent to other modules, the system can experience significant delays while the outputs are being checked.

One way to get around this problem is to do the checking at the same time the outputs are being sent to other modules. In this case, the results of the checker may not be ready until several cycles later, at which point an error may have propagated to other modules. Thus, it will be necessary to undo the results of the last few cycles and restore the state of the system to a known error-free state.

1.1. Micro Rollback

A method of backing up a CPU several cycles, known as micro rollback, is described in [8]. It basically involves the placing of an \( N \)-entry delayed write buffer (DWB) at each register that holds information across cycles (e.g., program counter, status register, instruction register, etc.). Figure 1.1 shows a circuit diagram of a typical DWB. When a register is written, it is actually written into the first entry of the DWB. At each subsequent cycle, the entire DWB is shifted by one entry, and the last entry is written into the actual register. Along with each entry in the DWB, there is a bit indicating whether that entry is valid or not. The bit is normally set when the register is written; to cancel the results of the last \( M \) cycles, the valid bits of the \( M \) most recent entries are cleared (where \( M \leq N \)). When the register is read, a selection circuit picks out the most recent entry with a set valid bit. In addition, when the DWB shifts, the actual register is written only if the valid bit is set. The same idea is also used in the register file, except that the DWB entry includes the address of the register being written each cycle.

The Mirror Processor is a VLSI processor employing hardware support for micro rollback, based largely on the Berkeley RISC II processor[4]. One of its intended goals is to evaluate the area and
performance overhead of micro rollback. This report describes the implementation of the controller of the Mirror Processor, details its rollback and recovery schemes, and presents some issues related to the testability of the processor.

1.2. Overview of Mirror Processor

The Mirror Processor (Figure 1.2) is an implementation of a fault-tolerant RISC processor employing hardware support for micro rollback. It is designed to detect and recover from all single transient errors and, in some cases, multiple transient errors without the need for extensive error-
correction code circuitry and without the sacrifice in speed suffered by checking data before its use.

![Diagram of Mirror Processor]

**Figure 1.2: Mirror Processor**

1.3. **Main Processor**

At its core, the processor is a 2 μm CMOS implementation of the Berkeley RISC II processor (MOSIS SCMOS design rules with λ=1 μm). Under basic operation it is, with a few exceptions, binary-compatible with RISC II. Built on top of that are various error-checking blocks and delayed write buffers which are transparent to the user. In addition, several new instructions have been added to the instruction set for the purposes of testing the rollback logic.
1.4. Error Detection

The processor is designed for duplex operation: a master chip functions as the main processor in the computer system, while a slave chip follows along in lock-step synchronization. The slave performs the same operations as the master, but instead of writing its outputs to the external bus, it compares them with the values written by the master. An error is flagged if any differences are found.

In addition, parity checkers are present in each processor to detect errors in values read out of the register file and in values read from the external input data bus (busIN). Each processor does its own local parity checking and flags an error if a parity error is detected.

1.5. Error Recovery

An external rollback signal is made available to the entire processor system (CPU, memory manager, instruction cache, etc.). Whenever either processor detects an error, it pulls the rollback line and forces the entire system to roll back a certain number of cycles. If the rollback occurred because of a comparison error or because of a parity error on the external input data bus, it is assumed that a transient fault caused the error and that simply rolling back should remove the fault. However, if the error was a parity error on a register file read, the faulty value may have been present in the register file for a long time prior to its being read, so simply rolling back may not remove the fault. In this case, the processor with the good value must send that value to the other processor. This procedure is known as state repair, and it takes place after any rollback caused by a parity error.
2. THE MIRROR PROCESSOR CONTROLLER

This chapter presents the Mirror Processor controller design in detail. The data path design is presented in [5].

2.1. Controller Design Considerations

There were many possible ways that the controller for the Mirror Processor could have been implemented. As the processor is a RISC processor, there are relatively few instructions, and all of them are simple instructions in the sense that they only perform one operation. Because of the simplicity of the instruction set, it is not necessary to build a microprogrammed controller; it is possible to efficiently hardwire the controller, which can then run significantly faster than a ROM with a microprogram.

The original RISC II has a single generalized decoder to decode the opcode and generate 39 control bits which are then AND’ed with the various clock signals to produce 100 control signals (including multiple copies, clock signals, and control bits with no clock qualification). Unfortunately, such a simple scheme could not be used in the Mirror Processor controller. Many of the control signals are dependent not only on the opcode but also on the various rollback and repair signals. In addition, the Mirror Processor has about double the number of control bits as RISC II, due to the additional logic required to implement error detection and micro rollback. In particular, the controller takes 69 inputs and generates 134 control signals, 68 of which are internal to the controller and 66 of which are sent to the data path. In addition, it generates 17 external signals and four clock signals. Note that unlike in RISC II, there is no distinction between "control bits" and "control signals"; every signal already has some sort of clock qualification as it is generated.

A simpler way to implement the control would be to use one or two PLA’s (Programmable Logic Arrays) or ROM’s to generate all of the control signals. As noted before, because the Mirror Processor has a short cycle time and relatively few control signals, a ROM would require too much time to operate and take up too much area to be a practical solution.
PLA's, on the other hand, present the best compromise between speed/area and design/layout effort. Thus, it was decided that as many of the control signals as possible would be generated directly by several PLA's. However, several of the inputs to the controller become valid much later than the instruction. In order to allow the PLA's enough time to evaluate, it was decided not to have them wait for the late signals; instead, those control signals depending on the late inputs would be generated by random static logic. In addition, some signals have to become valid as soon as possible after the arrival of the instruction. These signals are also generated by random static logic.

2.2. Description of the Controller

The controller of the Mirror Processor is shown in Figure 2.1. Instructions come in from the pads on bus\textsubscript{IN} and are latched onto bus\textsubscript{IR}. A four-state finite state machine keeps track of whether the processor is executing a normal instruction, executing the second cycle of a two-cycle instruction, or performing state repair. The Next State Logic block computes the next state based on the incoming opcode and on the repair signals generated by the Rollback Logic (Chapter 3). The next state, plus the opcode and the repair signals, are sent to three PLA's that decode the opcode and generate the majority of the control signals used by the data path.

Of the signals that aren't generated by the PLA's, most of the rest of them are generated by the Post-FSM Logic and Valid Bit Logic blocks. They produce signals that are dependent on values coming from the Rollback Logic, Interrupt Logic, and Condition Code Logic blocks, all of which produce their results well after the instruction arrives. The Valid Bit Logic block generates the valid bits used by all of the DWBs, while the Post-FSM Logic block generates the rest of these "late signals." There is, in addition, a separate Condition Code Logic block used to analyze the condition codes from the PSW and generate a condition bit used to control jump instructions; it is separate from the PLA's because it too depends on signals that arrive long after the instruction. Finally, a separate Memory Control Logic block generates the signals used to initiate a memory access; they are not generated by one of the PLA's because they are needed as soon as possible after the arrival of the opcode.
The remaining blocks handle inputs from external sources other than the incoming instruction. The Interrupt Logic Block monitors the external interrupt, reset, and shutdown lines as well as signals from the data path indicating a trap condition. It decides if the current cycle will be interrupted and gates both the calli instruction onto busIN and the proper interrupt vector onto busOUT. The Rollback Logic block serves double duty: it monitors the internal parity and comparison error signals and signals a rollback to the rest of the system if an error is detected, and at the same time it monitors the external rollback and rollback amount lines and signals the rest of the processor that a rollback is taking place if the line is pulled. Finally, a DWB saves the contents of busIR and one of the state bits each cycle and restores them on a rollback.

Figure 2.1: Controller

The generation of the four clock phases is done on-chip in the Clock Generator block. However, in
order to allow more control over the clock signals when testing the processor, separate input pads for each of the four signals is provided as well.

The implementation details of each of the controller blocks are given in Section 2.4 and summarized in Section 4.

2.3. Controller Timing

This section details the timing of the Mirror Processor from the point of view of the controller. Timing as it relates to the data path is covered in greater detail in [5].

2.3.1. Processor Timing

The processor runs on a four phase clock with a total cycle time of 100 ns. The first and third phases ($\phi_1$ and $\phi_3$) are 30 ns each (25 ns high, 5 ns low), and $\phi_2$ and $\phi_4$ are 20 ns each (15 ns high, 5 ns low).

2.3.2. Memory Interface Timing

Figure 2.2 shows the timing used by the processor to communicate with the memory system. The address of the word, halfword, or byte being accessed is gated onto the address/data lines, the data size lines are set (the encoding is the same as in RISC II; see Table 2.1), and the read/write line is set to indicate the type of access ($0 =$ read, $1 =$ write), after which the address enable line is asserted. It is assumed that the memory does not run on the same clock as the processor, so the address enable line is present to tell the memory when to begin the access. On the leading edge of the address enable, the memory can assume that the address and the read/write signal are stable and latch them in. If the memory cannot retrieve the requested data (on a read) immediately, it has approximately 25-30 ns to assert the wait line (45 ns minus time to go through pads minus delay from rising edge of $\phi_1$ to rising edge of address enable), as the processor must latch the wait signal at the end of $\phi_2$. If the memory can respond immediately and the access is a read, it should begin gating the data onto the address/data lines.
immediately after the falling edge of address enable and continue to do so until the falling edge of the
data enable signal. If the access is a write, the memory should wait until the rising edge of data enable,
at which time the data to be stored should be stable on the address/data lines and can be latched in.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>out.size&lt;1&gt;</th>
<th>out.size&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>halfword</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.1: Data Size Line Encoding

Addresses are gated onto the pads during \( \phi_4 \), and the address enable line is asserted shortly after the
trailing edge of \( \phi_4 \). The data on a read is latched in by the processor on \( \phi_3 \), while the data on a write is
gated onto the pads beginning \( \phi_2 \), with the data enable signal asserted shortly after the trailing edge of \( \phi_2 \).

The address enable line rises on the falling edge of \( \phi_4 \) and falls on the falling edge of \( \phi_1 \); the
memory must latch the address within that time (30 ns). It then has until the falling edge of data enable to
gate the data onto the address/data lines; this will occur 50 ns after the falling edge of address enable
(i.e., on the falling edge of \( \phi_3 \)). If the memory cannot respond in that time, it must signal a wait state.
Data enable is asserted only on cycles that are not wait cycles, so if a wait state is signaled, the memory
will then have one cycle plus 50 ns to respond. Note that the address enable line will not be reasserted
during the wait cycle. In the case of a write, the data to be written will be gated onto the address/data
lines during \( \phi_2 \) regardless of whether the cycle is a wait cycle or not; however, the data enable line will
not be asserted on a wait state, just as in a read. Once data enable is enabled, the memory will have 30 ns
to latch the data in.

The short length of time allowed for the memory to raise the wait signal is a result of certain
implementation constraints. Section 2.5 describes these constraints and presents a possible solution to
this problem.
Figure 2.2: Memory Interface Timing
2.3.3. Normal Instruction Cycle

Figure 2.3 shows the timing of the various blocks in the controller during a normal instruction cycle. The Mirror Processor has a pipelined instruction fetch/execution cycle: as one instruction is being executed, the next instruction is being fetched. The instruction fetch begins on $\phi_4$, when the address of the instruction to be fetched is gated onto $busOUT$ and through the pads. If the memory cannot respond with the requested data that cycle, it raises the $wait$ line, which the controller latches on $\phi_2$. If the wait line is high, the $state.wait$ bit is set, and the processor does nothing except shift the DWBs (with valid bit $= 0$) until the next $\phi_2$, when the $wait$ line is latched again. If the memory can respond, the incoming instruction is gated onto $busIN$ and passed straight through onto $busIR$ during $\phi_3$. As the opcode is arriving on $busIR$, the Next State Logic computes the next state, and its output is latched on the falling edge of $\phi_3$. While all this is happening during $\phi_3$, the $\phi_4$ PLA begins precharging. It evaluates and latches its outputs on $\phi_4$, which is when the $\phi_1$ PLA's precharge. These PLA's evaluate and latch their outputs on $\phi_1$. Meanwhile, the Memory Control Logic also evaluates on $\phi_4$, just in time generate the signals needed for the next instruction fetch. The Condition Code Logic PLA precharges $\phi_2$ and evaluates $\phi_3$, in time for the condition bit to determine which address to gate out on $\phi_4$ for conditional jumps. Note that every PLA evaluates every cycle, regardless of whether or not it is a wait cycle.

The sequence for a two-cycle instruction (load, store) is only slightly different. Because of the instruction fetch pipeline, the instruction following the load/store arrives on $busIN$ during $\phi_3$ of the first cycle of the load/store. However, it is not decoded until $\phi_3$ of the second cycle, so it must be stored somewhere. At the same time, the opcode of the load/store is re-decoded during the first cycle, so it must remain on $busIR$ across two cycles. Thus, during the first cycle of the load/store, the next instruction is latched into the $IRlatch$ but is not gated onto $busIR$. During the second cycle, when $busIN$ will have data during $\phi_3$, the $IRlatch$ is not reloaded, but its value is gated onto $busIR$.

The instruction execution timing during $\phi_4$ is constrained by certain implementation details and impacts greatly on the controller's critical path (Section 2.4.12). Section 2.5 describes possible
Figure 2.3: Control Timing
alterations to the Mirror Processor's design that could improve the critical path and also allow the memory a longer time to assert the \textit{wait} signal.

2.3.4. Interrupts and Traps

The interrupt logic monitors the external Interrupt Request (IRR), \textit{shutdown}, and \textit{reset} lines, as well as various lines from the data path indicating a trap condition. All are latched on $\phi_2$, at which time the internal interrupt signals are sent to the rest of the processor. If the interrupt or trap is taken, a hardwired \texttt{call} instruction is gated onto \texttt{busIN} during $\phi_3$, in place of whatever instruction would normally be gated on from the pads, and an interrupt vector corresponding to the type of interrupt or trap is gated onto \texttt{busOUT} during $\phi_4$. The instruction that is actually interrupted is allowed to finish, but an invalid bit is shifted into the register file DWB, effectively canceling the results of the instruction.

The handling of interrupts after this point is exactly the same as in RISC II; all of the caveats that apply to RISC II apply to the Mirror processor as well. For instance, the first instruction of every interrupt handler must be a \texttt{getipc}, typically to \texttt{r24}, in order to save the address of the instruction being fetched during the execution of the interrupted instruction. In addition, all interrupt handlers must end with the following sequence:

\begin{verbatim}
  jmpx abw,0(r25)
  reti abw,0(r24)
\end{verbatim}

in order to restore the sequence of memory accesses prior to the interrupt. Finally, the interrupt handler must be careful to save its own state before re-enabling interrupts and make sure there are more free register windows before making a subroutine call [4].

There are, however, two cases where the interrupt sequence differs from the RISC II scheme. Two of the trap signals, "bad shift amount," and "address misalignment," cannot be generated by the data path before the end of $\phi_2$. In these cases, they are latched $\phi_3$, and the trap is taken in the following cycle. However, because the point at which the trap is taken is one cycle after the instruction that caused the trap, more needs to be done in order for the trap handler routine to determine the instruction that caused
the trap. So, in addition to latching the signals $\phi_3$, invalid bits are written into every DWB on $\phi_4$, when they load their valid bits. Thus, there will be no state change after the trap is detected, and the trap handler will be able to find the correct instruction.

Reset and shutdown are handled in exactly the same manner as other interrupts, except that they have higher priorities than any other type of interrupt or trap.

2.3.5. Rollbacks

The rollback memory attached to $busIR$ operates in exactly the same way as the Instruction Register in the data path [5]. It loads from $busIR$ on $\phi_4$ and updates on $\phi_2$. When a rollback occurs during $\phi_3$, the valid bits are cleared (set to 0), and the most recent valid entry is selected and gated onto $busIR$ all on $\phi_3$.

2.4. Details of Controller Implementation

This section presents detailed descriptions of each of the blocks in the controller except the rollback logic, which is described in Chapter 3.

2.4.1. $busIR$

$busIR$ stores 14 bits of $busIN$: the opcode (seven bits), the Set Condition Codes (SCC) bit, the Data Immediate (IMM) bit, the condition field (four bits), and the five-bit destination register (Rd) field, four bits of which coincide with the condition field. All of the other blocks in the controller read off of this bus, which serves to buffer the controller from $busIN$ during loads, stores, and state repairs. Loads and stores are two-cycle instructions; the opcode must remain available for two cycles because the PLA's decode the opcode every cycle, regardless of what else is going on that cycle (e.g., state repair, rollback, interrupt, etc.). In addition, the destination register for a load must be available during the second cycle for the incoming data. During a rollback, the value of $busIR$ is restored by the controller's rollback memory. If a state repair takes place after a rollback, $busIR$ keeps the instruction available while the data word being repaired is read in on $busIN$. 
2.4.2. *busIR* Latch and Driver

Normally, the interface between *busIN* and *busIR* acts as a transparent latch, while during loads and stores it holds the next instruction over one cycle as described in Section 2.3.3. The driver is similar in design to the *busIN* driver[5], with the addition of a latch between *busIN* and the driver; see Figure 2.4. SPICE runs show that it takes 8.0 ns to charge *busIR* from zero to one.

![Figure 2.4: busIR Latch and Driver](image)

2.4.3. Next State Logic

The state of the controller logic is determined by a four-state finite state machine, with each state representing the type of cycle that is currently executing. Most cycles are represented by the normal state, that is, a normal one-cycle instruction. The second cycle of a load or store instruction is represented by the suspend state, so called because the instruction fetch that would normally take place during that cycle is replaced by the load/store access instead, so the instruction fetch pipeline is temporarily suspended. The other two states represent the first and second state repair cycles (*repair1* and *repair2* respectively).

Although four states can be represented by two bits normally, three bits must be used in this case, one to distinguish between normal and suspend states (bit 0), and two more to indicate repair1, repair2, or no repair. The normal/suspend bit is saved each cycle in the controller's rollback memory and is restored on a rollback. If a state repair takes place after the rollback, the bit must be remembered across
both cycles of the repair since it will not be restored again after the repair, so it is carried along unchanged through both cycles. Bit 1 is set for repair1, while bit 2 is set for repair2; neither bit is set in the absence of state repair.

The next state is computed during $\phi_3$ based on the current state, the repair signals set during $\phi_2$, and the opcode of the instruction being executed in the current cycle. In normal operation, the only time the state needs to change from normal to suspend is when the next cycle will be the second cycle of a load or store. Only one opcode bit (bit 5) is needed to distinguish between load/store instructions and all others, so the suspend state will be true only if the current state is normal, and bit 5 of the opcode that arrived in the previous $\phi_3$ is set. Thus, opcode bit 5 ($op5$) is latched every $\phi_4$ to be used in the following $\phi_3$. The actual next state computation is implemented in random static logic, with its outputs latched $\phi_3$ and re-latched $\phi_2$ before being fed back around as inputs (see Figure 2.1). Note that the re-latching during $\phi_2$ could just as easily have been done during $\phi_1$ or $\phi_4$; the phase in which to do the latching was chosen at random.

The suspend and $op5$ bits are latched one more time during $\phi_3$ before being sent to the Controller Rollback Memory (Section 2.4.11). Since the Rollback Memory is loaded during $\phi_4$ and the next $op5$ bit is also latched during $\phi_4$, this latch prevents the $op5$ bit from being overwritten before it is written into the Rollback Memory. It was later noticed that the latch is not needed for the suspend bit, and in addition, it would not be needed for the $op5$ bit if it were latched during $\phi_1$ or $\phi_2$ instead. However, there was insufficient time to make the necessary design modifications.

2.4.4. PLA’s

Most of the control signals used by the data path are generated by three domino logic PLA’s. The $\phi_4$ PLA, the smallest of the three, generates the signals that must be valid during $\phi_1$. It precharges during $\phi_3$ and evaluates during $\phi_4$, thus requiring its inputs (including the three bits of next state) be stable before the rising edge of $\phi_4$. The $\phi_{18}$ and $\phi_{16}$ PLA’s generate the signals that are used in the other phases.
Because there are a large number of such signals, they are divided up between two PLA's in order to avoid having a very large PLA with a long evaluation time. The two PLA's precharge during $\phi_3$ and evaluate during $\phi_1$. Most of their outputs are also latched during $\phi_1$; however, several of the $\phi_{1b}$ PLA's outputs are not latched until $\phi_2$ because their previous values are being used $\phi_1$. Specifically, the PSW update and the register file write both take place $\phi_1$, so signals pertaining to these blocks are delayed until $\phi_2$.

The PLA's were generated by feeding the output equations to eqntout [6] and then passing the output to espresso [6] with the "output phase optimization" option to minimize the number of product terms. magic layout was generated with mpla[6] using a template originating from Caltech but modified heavily. Figure 2.5 shows a "worst case" bit slice of the $\phi_{1b}$ PLA. It combines the input with the largest gate load, the NAND term with the greatest number of series transistors, the NOR-plane input with the largest gate load, and the NOR term with the largest drain load. SPICE simulations of the worst cases of each PLA show evaluation times of 13.0 ns, 16.5 ns, and 18.5 ns for the $\phi_4$, $\phi_{1a}$, and $\phi_{1b}$ PLA's respectively. The setup time for the $\phi_4$ PLA is 7 ns (i.e., the input must be stable 7 ns prior to the rising edge of the evaluate clock, which is $\phi_4$ in this case; see Section 2.4.12); the setup times for the $\phi_1$ PLA's were not calculated because their inputs must already meet the $\phi_4$ PLA setup time.

2.4.5. Condition Code Logic

This block of logic takes as input the four condition code bits from the PSW and the four bits from the condition field in the instruction and produces as output one bit indicating whether or not the condition codes satisfy the indicated condition (in the case of a conditional branch). The output bit is AND'ed with some of the PLA outputs to produce signals required at the beginning of $\phi_4$ (these signals determine which address is gated onto busOUT for the next instruction fetch), so the output of this block must be ready by the middle of $\phi_3$. As we have limited ourselves to static gates of no more than four inputs, a static NAND/NOR tree implementation of the logic would require five levels of large, four-input
NAND and NOR gates, which would be slow and which would require a lot of area. However, all of the inputs can be ready by the end of \( \phi_2 \), so dynamic logic can be used instead. The logic has been implemented as a PLA with eight inputs, one output, and 18 product terms, generated in the same manner as the other two PLAs. The PLA precharges \( \phi_2 \) while latching its inputs at the same time, and it evaluates and latches its outputs during \( \phi_3 \). SPICE runs show the output is ready 7.75 ns after the rising edge of \( \phi_3 \).

2.4.6. Post-FSM Logic and Valid Bit Logic

Some of the control signals used by the data path are dependent on the condition bit, the rollback signal, the interrupt signal, and the wait signal, all of which become valid long after the PLA's evaluate. In order to allow the PLA's to evaluate without waiting for them, these control signals are produced outside of the PLA's by the Post-FSM Logic and Valid Bit Logic blocks. The valid bits shifted into the DWBs each cycle are produced by the Valid Bit Logic block, while the rest of them are produced by the
Post-FSM Logic. In either case, every signal is produced by AND'ing the required condition bit(s) with the decoded opcode signals coming from the PLA's. Static random logic is used since the inputs arrive at different times.

2.4.7. Memory Control Logic

Five of the bits that control the memory access (read/write, instruction/data, data size (2 bits), address/data pad enable) must be valid as soon into $\phi_4$ as possible, since the instruction fetch begins that phase. They cannot be included in the $\phi_4$ PLA because that PLA requires most of $\phi_4$ to evaluate, so instead they are generated by the random logic in the Memory Control Logic block. Both dynamic and static implementations were considered. It was found that because all of the inputs are ready before the end of $\phi_3$, the static version was faster than the dynamic version (the delay being only the delay through a latch, rather than the time to discharge the output line plus the delay through the latch), so the static version was used.

2.4.8. Memory Enable Signals

The two memory enable bits, address enable and data enable, are also generated in the Memory Control Logic block. The memory interface scheme requires that the enable signals not go up until the address or data is stable on the address/data lines, so the enable signals cannot simply be tied to the rising edges of $\phi_4$ and $\phi_2$; they must be delayed until it is certain that the memory sees the proper address/data value on the lines. At the same time, the enable signal should rise as soon as possible after the address/data value is ready in order to give the memory as much time as possible to respond to the request. One way to do this would be to self-time the delay, i.e., use the worst case delay on busOUT to trigger the enable signal (assuming the delay through the pads is equivalent for all pads). However, none of the busOUT lines can be used to directly trigger the signal because it cannot be guaranteed that at least one of the lines will have a transition, so a delay circuit featuring an exact replica of one bit of busOUT must be designed in order to implement this scheme. This would take up far too much area in the
controller, so the self-timing approach was dropped.

The current implementation has the address enable line being enabled from the falling edge of \( \phi_4 \) to the falling edge of \( \phi_1 \). SPICE runs have shown that this gives an adequate delay, although it doesn't give the memory quite as much time to respond. The circuit diagram is shown in Figure 2.6. The falling edge detectors use delay inverters [5] (see Section 2.6) to delay the complement of the clock signal long enough to generate a pulse on the falling edge. The data enable circuit is similar, except that it is valid from the falling edge of \( \phi_2 \) to the falling edge of \( \phi_3 \).

![Figure 2.6: Memory Address Enable Bit](image)

2.4.9. Interrupt Logic

This block, implemented in static random logic, takes all the various interrupt and trap condition signals and sets the internal interrupt bit, \textit{state.int}. In the case of a reset or a shutdown, the internal reset or shutdown bit is set instead (\textit{state.reset} and \textit{state.shutdown} respectively). The Interrupt Logic also determines the interrupt vector to gate onto \textit{busOUT}. Since all of the vectors differ only in three bits, only those three bits need to be generated and sent to the \textit{busOUT} driver in the data path. A list of all possible interrupt and trap conditions and their respective interrupt vectors is shown in Table 2.2. It should be noted that the interrupt vectors shown differ from the RISC II vectors in that they are placed in the lower half of the memory space (locations 0x0 to 0x40), while in RISC II they were placed in the upper half (0x80000000 to 0x80000030). This was done in order to simplify the logic gating the interrupt
vector onto busOUT. It should also be noted that only reset and shutdown traps are non-maskable; all
others can be masked out by the interrupt enable bit from the PSW.

<table>
<thead>
<tr>
<th>Cause</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset Pin pulled</td>
<td>0x00000000</td>
</tr>
<tr>
<td>illegal opcode</td>
<td></td>
</tr>
<tr>
<td>privileged opcode</td>
<td></td>
</tr>
<tr>
<td>address misalignment</td>
<td></td>
</tr>
<tr>
<td>illegal shift amount</td>
<td></td>
</tr>
<tr>
<td>Interrupt Request Pin pulled</td>
<td>0x00000010</td>
</tr>
<tr>
<td>Register File window overflow</td>
<td>0x00000020</td>
</tr>
<tr>
<td>Register File window underflow</td>
<td>0x00000030</td>
</tr>
<tr>
<td>Shutdown Pin pulled</td>
<td>0x00000040</td>
</tr>
</tbody>
</table>

Table 2.2: Interrupt Vectors

2.4.10. Comparator Logic

Each output pad has associated with it a small amount of comparison logic. When the processor is
operating in slave mode, each output signal value is compared with the value present on the pad (which is
the value written by the master), and if they differ, an error signal corresponding to that pad is set. Most
of the comparison error signals are routed directly to the Comparator Logic block (Figure 2.7 and
Table 2.3), where they are OR'ed together to create a global comparison error signal (error_CMP). The
exceptions are the multi-bit vectors: data size (2 bits), state compression (4 bits), and address/data pads
(33 bits). In each case, the error signals from the individual pads are first OR'ed together at the pads, and
only a single error signal representing the entire vector is sent to the Comparator Logic block (except for
the address/data lines, where three signals are sent; see below). A more detailed description of the
comparison logic at the pads is presented in [5].

Because of the large number of address/data pad error signals, there was not enough room at the
pads to OR all of them together into a single error signal. Instead, they are OR'ed down to three signals
(one representing bits 0 to 15, one representing bits 16 to 32, and the error signal from the parity bit pad)
which are then sent to the Comparator Logic block. The three signals are OR'ed together here into the
Figure 2.7: Comparator Logic Block

<table>
<thead>
<tr>
<th>Error Signal</th>
<th>Signal(s) Being Compared</th>
</tr>
</thead>
<tbody>
<tr>
<td>error.AD</td>
<td>address</td>
</tr>
<tr>
<td>error.enb.addr</td>
<td>address enable bit</td>
</tr>
<tr>
<td>error.enb.data</td>
<td>data enable bit</td>
</tr>
<tr>
<td>error.id</td>
<td>instruction</td>
</tr>
<tr>
<td>error.ira</td>
<td>interrupt acknowledge signal</td>
</tr>
<tr>
<td>error.rw</td>
<td>read</td>
</tr>
<tr>
<td>error.size</td>
<td>data size</td>
</tr>
<tr>
<td>error.state</td>
<td>state compression bits</td>
</tr>
<tr>
<td>error.sysmode</td>
<td>system mode bit</td>
</tr>
<tr>
<td>error.busA</td>
<td>busA parity error</td>
</tr>
<tr>
<td>error.busB</td>
<td>busB parity error</td>
</tr>
<tr>
<td>error.busOUT</td>
<td>busOUT parity error</td>
</tr>
</tbody>
</table>

Table 2.3: Comparison Logic Error Signals

The final error signal (error.padAD).

Most of the error signals are OR'ed together in a static OR tree and latched $\phi_1$. After the internal rollback and shutdown lines have been set during $\phi_2$, the comparison error signal is AND'ed with them and latched again on $\phi_3$, after which it is sent directly to the Rollback Logic (Chapter 3). The state compression and system mode bit error signals cannot be ready by the end of $\phi_1$, so they are OR'ed in with the $\phi_1$ error signal after the $\phi_1$ latch. In addition, because the state compression errors are ignored during shutdowns, the state compression error bit (error.state) needs to be AND'ed with state.shutdown. However, because state.shutdown changes during $\phi_2$, and error.state does not become valid until
sometime into \( \phi_2 \), \texttt{state.shutdown} needs to be saved. \texttt{save.shutdown} saves \texttt{state.shutdown} during \( \phi_1 \) and is then AND’ed with \texttt{error.state} when it arrives. Finally, because the address/data lines are used twice per cycle, the error signal from those lines (\texttt{error.padAD}) is latched during both \( \phi_1 \) and \( \phi_3 \) (for address and data errors, respectively), conditioned on whether or not a value is being gated out during that phase. \texttt{gate.busOUT.padAD4} is asserted during \( \phi_4 \) and \( \phi_1 \) when an address is being gated out, and \texttt{gate.busOUT.padAD2} is asserted during \( \phi_2 \) and \( \phi_3 \) when data is being stored.

In addition to detecting comparison errors on external signals, the Comparison Logic block also sets the error signals for parity errors on \texttt{busA}, \texttt{busB}, and \texttt{busOUT}. The parity bits read out of the register file are sent to the Comparison Logic block along with the new parity bits generated by the data path after the register read and compared. They are XOR’ed together, and the appropriate error signal is set if they differ. The parity bit calculated for \texttt{busOUT} during the second cycle of a state repair is XOR’ed with the newly-calculated parity bit of the value read from the register file the previous cycle; the \texttt{busOUT} error flag is set if the bits are different. Section 3.2.4 describes the \texttt{busOUT} parity error in more detail.

2.4.11. Controller Rollback Memory

The fourteen bits of \texttt{busIR}, plus the \texttt{normal/suspend} bit and opcode bit 5 of the previous cycle, are saved in a DWB each cycle and restored on a rollback. The controller DWB behaves in exactly the same way as the IR, loading on \( \phi_4 \) and updating on \( \phi_2 \).

During normal one-cycle instructions, the controller DWB contains the same information as the IR except for the state bit, since \texttt{busIR} is loaded directly from \texttt{busIN}. Thus, in those cases the controller DWB is completely redundant and not needed. However, during the first cycle of a load or store instruction, \texttt{busIR} and \texttt{busIN} will have different values: \texttt{busIR} will hold the load/store instruction across to the second cycle, while the next instruction arrives on \texttt{busIN}. When rolling back to that second cycle, both the load/store instruction and the next instruction need to be restored. In this case, the controller DWB is needed to restore the load/store instruction while the IR restores the next instruction.
2.4.12. Critical Paths

The main critical path for the controller occurs during $\phi_3$, when the instruction arrives on $busIN$. Because the PLA's are dynamic, the inputs cannot change during the evaluate phase. In the case of the $\phi_4$ PLA, it evaluates during $\phi_4$, so its inputs cannot change after the rising edge of $\phi_4$. In addition, there is a setup time associated with the inputs, such that if an input changes after the minimum setup time, the PLA may not function properly. For the PLA's used in the Mirror Processor, the worst-case scenario happens if the input closest to the precharge transistor in the AND plane changes to 1 after the precharge clock has been de-asserted. If the node on the other side of the AND plane transistor for this input (node 24 in Figure 2.5) was set to 0 in the previous cycle, and all the other inputs are 0, then the precharged node (node 25 in Figure 2.5) may incorrectly discharge. Thus, the minimum setup time is the latest time prior to the rise of the evaluate clock that the input can change without causing the precharged node to discharge. For the $\phi_4$ PLA, the input must be stable at most 23 ns into $\phi_3$. This results in the following critical path: the instruction arriving on the address/data pads during $\phi_3$ must go from the pads to $busIN$ to $busIR$ to the $\phi_4$ PLA and be stable by 23 ns into $\phi_3$.

The critical path during $\phi_4$ is set by the evaluation time of the $\phi_4$ PLA, while the time required to perform rollback amount arbitration sets the constraints for $\phi_1$ and $\phi_2$ (see Section 3.3.17).

Another critical path during $\phi_2$ resulted in the removal of the conditional return instructions. Because the PSW updates its condition codes during $\phi_1$ and the interrupt logic sets state.int during $\phi_2$, it would be necessary to evaluate the condition bit, increment the Current Window Pointer in the PSW, detect a register window overflow, and cause a trap all within $\phi_2$. Since this was clearly not possible given a 15 ns $\phi_2$, it was decided to make returns unconditional. Thus, the window pointer could be incremented without waiting for the condition bit to be evaluated, and an overflow condition could be detected much sooner in $\phi_2$. In addition, the evaluation of the condition bit could be moved to $\phi_3$, thereby allowing it to be implemented as a dynamic PLA.
2.5. Alternate Wait State Timing

Because the memory is not given very much time to assert the wait signal, it would be preferable to latch the wait line one phase later, during \( \phi_3 \). However, this is not possible with the current controller implementation. Because the PLA’s evaluate every cycle regardless of whether or not it is a wait cycle (Section 2.3.3), busIR must not be reloaded during \( \phi_3 \) of a wait cycle; otherwise, the PLA’s will receive garbage input. Since it is necessary to know whether or not the current cycle is a wait cycle before \( \phi_3 \), the wait line must be latched during \( \phi_2 \).

One possible way to allow the wait line to be latched during \( \phi_3 \) would be to load busIR regardless of the value of the wait line and then disable the latching of the PLA, Memory Logic, and Condition Code Logic outputs during a wait cycle. The outputs would then remain the same as in the previous cycle, as in the case of the current implementation. In addition, the Register File Translator in the data path [5] would have to disable the latching of the destination register number off of busIN during a wait cycle. Since nothing else is dependent on the wait signal being set during \( \phi_3 \), the latching the signal can be delayed until then.

Another way to allow latching of the wait signal during \( \phi_3 \) would be to eliminate the need to use busIR during \( \phi_4 \). Then, the loading of busIR could be delayed until \( \phi_4 \), allowing the wait signal to be latched during \( \phi_3 \). However, the modifications to the controller design to allow this would be more than trivial.

First, there are two blocks that read the opcode of the instruction that arrives during \( \phi_3 \) off of busIR during \( \phi_4 \): the \( \phi_4 \) PLA and the Memory Control Logic. Under normal operation, the Memory Control Logic outputs the same values every cycle except during the second cycle of a load or store instruction. Thus, the only time the opcode is important in the Memory Control Logic is during the second cycle of a load or store, during which time busIR is not re-loaded anyway.

Of the twelve \( \phi_4 \) PLA outputs, ten of them are dependent on the opcode of the just-arrived instruction. One of them (\textit{gate.busOUT\_padAD2}) differs only during the second cycle of a store, so the
arguments for the Memory Control Logic signals apply to this signal as well. The remaining nine signals are needed by the data path at or near the beginning of $\phi_1$. If they were to be implemented in static random logic, the opcode inputs would not need to be ready until the time the signals need to be ready minus the evaluation time. Since the signals are not needed at exactly the beginning of $\phi_1$, the evaluation time can be pushed into $\phi_1$. If the evaluation time is made short enough and pushed far enough into $\phi_1$, the loading of busIR can be delayed until after the wait line has been latched in $\phi_3$. In order to minimize the evaluation time, the opcode decoding must use as few gates as possible. Of the nine signals that need to be generated, six of them control the inputs to the shifter, requiring that the logic detect an ALU instruction, a shift instruction, or a ldhi instruction. Two of the other three signals control the window pointer in the PSW, requiring the detection of calls and returns, and the last signal controls the selection of either the PC or LSTPC (see [5] for full descriptions of these blocks), requiring the detection of the get1pc instruction. Because of the great number of instructions that need to be detected, it would probably require more than “a few” gates to implement all the signals; however, the evaluation time could probably still be shortened enough to allow the loading of busIR during $\phi_4$.

Although the second scheme would involve much more work to achieve, another benefit would be to remove busIR loading from the controller’s critical path during $\phi_3$ (Section 2.4.12). It would, however, probably increase the critical path in $\phi_4$, as the logic would not be given very much time to evaluate.

2.6. Clock Generation

The Clock Generator block generates the four clock phase signals used by the rest of the processor. The block can generate all four signals from an external master clock signal, or all four signals can be supplied externally, in which case the Clock Generator block passes them straight through. A single clock mode bit (in.csel) determines which mode the Clock Generator will operate in.

When operating in on-chip generation mode, the Clock Generator takes two inputs: a fast clock ($\phi_f$), which has a cycle time of 25 ns and is used to generate each phase signal, and a slow
clock ($\phi_s$), which has a cycle time of 100 ns and is used to indicate the start of $\phi_1$. Both clocks are guaranteed to have a 50% duty cycle, and the rising edge of the slow clock is guaranteed to occur a setup time before the rising edge of the fast clock (although the falling edge of $\phi_s$ is not guaranteed to occur before the rising edge of $\phi_f$). Each clock is fed into an edge detector whose outputs serve as inputs to an SR flip-flop. The flip-flop is set on the rising edge of $\phi_s$ and is cleared on the the falling edge of $\phi_f$. The output of the flip-flop is then sent into a four-stage shift register, where each stage represents one of $\phi_1$ through $\phi_4$. The shift register is clocked by $\phi_f$; since the output of the flip-flop is set to 1 only during one cycle of $\phi_f$ per cycle of $\phi_s$, the first stage of the shift register will be loaded with a 1 only every fourth cycle. Thus, on each successive cycle of $\phi_f$, only one stage of the shift register will have a 1, and the stage with the 1 drives the signal for that particular phase.

![Figure 2.8: Clock Generator](image)

Figure 2.8 shows the edge detectors, flip-flop, and shift register. Each stage in the shift register is composed of a master-slave flip-flop. The master stage is clocked by $\phi_f$; the slave stage is clocked by a signal generated from $\phi_f$ ($c_s$). Figure 2.8 also shows the logic that generates two non-overlapping clocks from a single clock. In the figure, inverters marked with a $d$ are delay inverters [5]; the number following the $d$ indicates the number of series P-transistors. A "d4" delay inverter is shown in Figure 2.9. Timing
diagrams for the slave clock and the shift register output are shown in Figures 2.10 and 2.11.

Figure 2.10: Slave Clock Generation Timing

Because \( \phi_f \) has a 25 ns cycle time, each shift register stage will hold its value for exactly 25 ns. Since both \( \phi_1 \) and \( \phi_3 \) are high for 25 ns, the output of the \( \phi_1 \) and \( \phi_3 \) shift register stages can be sent directly to the clock drivers. However, both \( \phi_2 \) and \( \phi_4 \) are high for only 15 ns, and they must each rise 5 ns after the falling edge of the previous phase, so the shift register stage outputs cannot be used directly.

Figure 2.12 shows how \( \phi_2 \) is generated; \( \phi_4 \) is generated in the same way. The shift register stage output
Figure 2.11: Clock Generation Timing
(state.phi2) is AND'ed with \( \phi_f \); however, the rising edge of state.phi2 is delayed 4 ns, and the falling edge of \( \phi_f \) is delayed 6.5 ns to create a signal that rises 5 ns after the rise of \( \phi_f \) (there is a 1 ns propagation delay) and falls 15 ns later. Because the falling edge of state.phi2 is also delayed 4 ns while the rising edge of \( \phi_f \) is not, a straightforward AND'ing of the delayed signals would result in a spike at the output at the beginning of the next \( \phi_f \) cycle. In order to prevent this from happening, the last NAND gate is modified slightly to include the non-delayed state.phi2 as well. Thus, the \( \phi_2 \) output will be high only when state.phi2 and the delayed state.phi2 are high, along with the delayed \( \phi_f \). A timing diagram of the \( \phi_2 \) clock generation is shown in Figure 2.13.

![Figure 2.12: \( \phi_2 \) Generation](image-url)
Figure 2.13: $\phi_2$ Clock Generation Timing
3. ERROR DETECTION AND RECOVERY

The Mirror Processor includes two types of error detection: parity checking of the register file and input bus, and comparison of output pins. Error recovery is accomplished by micro rollback and state repair. This chapter details the timing of the error detection and recovery schemes, as well as the implementation of the rollback control logic. The implementation of the error detection schemes is detailed in [5].

3.1. Rollback Timing

3.1.1. Internal Rollback

The internal comparison and parity error signals are latched by the Rollback Logic on $\phi_3$ and $\phi_4$ respectively. If any error condition is true, the external rollback line is pulled on $\phi_1$, and the proper rollback amount is sent out at the same time. The rollback and rollback amount lines are held until $\phi_4$.

3.1.2. External Rollback

The rollback logic monitors the external rollback and rollback amount lines and latches them on $\phi_2$. If the rollback line is pulled, the internal rollback signal (state.rb) is set and sent to the rest of the processor. On $\phi_3$, all the DWBs invalidate the indicated number of valid bits, and in the case of the controller, busIR is restored from its DWB. If the rollback amount is greater than four, then the external shutdown signal is pulled on $\phi_1$. Note that because the rollback amount is latched at the end of $\phi_2$, any rollback amount arbitration must be completed before then.

3.1.3. Absolute and Relative Rollbacks

In most cases, rollbacks will be absolute. An absolute rollback of $n$ cycles means that the processor will roll back exactly $n$ cycles, regardless of whether any of those cycles are valid or not (see Section 3.3.8). Occasionally, a relative rollback will be required. A relative rollback of $n$ cycles results in rolling back the $n$ most recent valid cycles. The number of cycles actually rolled back will depend on
the number of invalid cycles in the recent past.

3.2. Error Detection Timing

3.2.1. Register File Parity Errors

The values of busA and busB are parity checked when they are read out of the register file during \( \Phi_1 \). The error signal is latched by the rollback logic on \( \Phi_4 \), and in the following cycle a rollback of one cycle takes place. State repair is then initiated (Section 3.4).

3.2.2. busIN Parity Error

A value arrives on busIN during \( \Phi_3 \) of every cycle except a wait, and its value is parity checked on the following \( \Phi_1 \). The error signal is latched the following \( \Phi_3 \), after which a rollback of two cycles is signaled. No state repair is needed.

3.2.3. Comparison Errors

The comparison error signal is latched each \( \Phi_3 \) by the Comparator Logic (Section 2.4.10). If an error is indicated, a rollback of two cycles will be signaled. Again, no state repair is needed.

3.2.4. busOUT Parity Error

Normally, the value on busOUT is checked by the output pad comparison logic. However, during state repair, the comparison is disabled because each processor will have a different value on busOUT (the one doing the restoring will have the value to be restored, while the other one will have whatever value was left there from the previous cycle). Because the value being restored is left on busOUT between the first and second repair cycles, any errors occurring to that value will not be caught by the normal comparison mechanism. Thus, the parity bit that is generated for busOUT is compared against the parity bit of the value that was read out of the register file during the first repair cycle, and an error signal is latched \( \Phi_3 \) if a mismatch occurs, followed by a rollback of one cycle to restart the state repair.
3.2.5. Multiple Rollbacks

There are several situations in which errors occurring during a rollback will cause further rollbacks. How multiple rollbacks are handled is dependent on the number of previous rollbacks and, to a certain extent, on the type of error that caused the most recent rollback.

If a rollback occurs and an erroneous value is restored from a rollback memory, that value will be detected either one or two cycles after the rollback (two cycles in the case of the PC, since it doesn’t get put on the output pads until one cycle after the rollback; two cycles in the case of the controller if the flipped bit(s) is in the destination register field, since its state compression is not exported until the cycle after the rollback; one cycle in all other cases). In either case, the normal two-cycle rollback that would occur due to a comparison error will not invalidate the DWB entry with the bad value. In order to recover in this situation, it is necessary to roll back to the cycle before the one that was rolled back to previously, thus requiring a relative rollback [5] of two or three cycles (i.e., canceling the two or three most recent valid cycles [see Section 3.3.8]). The rollback logic has associated with it a counter that keeps track of the number of cycles since the last rollback; if an error is indicated within two cycles of a previous rollback and the normal amount to rollback is the same as the counter value (i.e., the rollback will be to the same cycle that was rolled back to before), then it is assumed that the error was due to a faulty rollback memory value, and a relative rollback to the cycle before the erroneous one is performed instead. Note that if the second rollback is caused by a parity error, the relative rollback one cycle more will result in the wrong register being repaired, since it is the instruction rolled back to that is used to determine which register to repair. However, this will do no harm, since the correct value will be copied from one processor to another.

It may be the case, however, that the proper cycle to roll back to is greater than four cycles previous to the current one. Since it is impossible to roll back more than four cycles, the rollback amount will be capped at four cycles, thus effectively causing a rollback to the cycle rolled back to before. In this scenario, because the offending instruction is never invalidated, it will repeatedly cause comparison errors
and rollbacks. Since there is no way to recover from this situation, it is necessary to detect when repeated rollbacks are occurring. The rollback logic has another counter that keeps track of the number of rollbacks that have occurred and which is cleared every 16 cycles. If four rollbacks occur within 16 cycles, it is assumed that the processor is stuck in an infinite rollback loop, and a shutdown is signaled.

Another problem occurs if it happens that the repeated rollbacks are caused by an erroneous value in the PSW rollback memory. When a shutdown trap is taken, all the other DWBs are loaded with fixed values associated with the calli instruction or are not read from. The IR and the controller DWB are loaded with the calli instruction, and the PC and MAR are loaded with the address of the shutdown handler. The SDR is not read from during the shutdown trap, and the next access to the SDR will be a write since the only time the SDR is read from is during the second cycle of a load or store, which will not happen again until after it is loaded during the first cycle of the next load or store. The PSW, on the other hand, only changes the Interrupt Enable and System Mode bits while keeping the previous values of the other fields. If one of those other fields is the one that caused the repeated rollbacks, it will still be incorrect after the shutdown, causing even more rollbacks. To correct this situation, all error checking is disabled for three cycles following a shutdown, enough time for the shutdown handling routine to save the PSW and store in a known good value.

Error checking is also disabled for three cycles after a reset in order to prevent random values on the error signals from causing an unwanted rollback.

If, on a register read, both processors detect a parity error in the same register, then it will not be possible to do a state repair, since neither processor has the correct value. However, to allow for the possibility that one of the processors actually has the correct value and that a second transient fault may have caused it to erroneously report an error (e.g., bit flip on busA itself and not the register, flipped parity error signal, flipped repair signal), no state repair will follow the one-cycle rollback. Thus, the instruction reading the registers with corrupted values will be executed again, and if the transient fault case were true, then only one processor will signal a parity error the second time through. However, if
both processors really do have bad values in the same register, then they will both signal a parity error on the same bus again. This will again cause a one-cycle rollback without state repair, and so on until the fourth rollback, after which a shutdown will occur.

3.3. Rollback Control Logic and Timing

This section presents a detailed description of the Rollback Logic in the controller, as well as detailed timing of the rollback procedure. A block diagram of the Rollback Logic is shown in Figure 3.1.

3.3.1. Overview

The Rollback Logic consists of two parts. The first part monitors the error signals generated by the parity checkers and comparators and pulls the external rollback line if an error is detected. It also determines the amount to roll back and recognizes shutdown conditions. The other part monitors the external rollback, rollback amount, and repair lines and sets the internal rollback signal (state.rb). Each part operates independently of each other, with the second part signaling a rollback to the whole system.
and the first part reacting to the fact that something in the system has signaled a rollback.

3.3.2. External Rollback Signals

There are eight external rollback signals shared by all the modules in the system. The rollback line is asserted to tell the system that a rollback is going to occur. Whenever the rollback line is asserted, every module in the system must initiate a rollback in the current cycle. The amount to roll back is given on the rollback amount lines and is represented as a three-bit binary number. The four repair lines indicate which bus (busA or busB) on which processor needs to repaired in the event of a parity error. For a particular bus, the processor that detected the error asserts its out.repairX line, where X is either A or B for busA or busB respectively, which is then demultiplexed to assert either the repairXm or repairXs signal, where m and s correspond to master and slave respectively. Each processor reads in all four signals directly and can thus determine if it needs to be repaired or if it needs to repair the other processor.

All of the external rollback signals are active-low wired-NOR lines held at one by 750Ω pullup resistors. The out line of each pad is connected to ground; when any module in the system wants to assert a signal, it raises that pad’s enable line, thus forcing a zero onto the line. The 750Ω value was chosen to give a slightly faster fall time than rise time (the higher the resistance, the faster the fall time and the slower the rise time) but to have a reasonably fast rise time nonetheless.

3.3.3. Input Latches

The eight external rollback signals are latched during φ2. state.rb is set if the rollback line has been pulled and the reset and shutdown lines haven’t (they have precedence over rollbacks). The latch outputs are sent directly to the rest of the data path and controller.

3.3.4. Internal Error Signals

The busA and busB parity error signals are latched during φ4, while the signals indicating busIN
parity errors, busOUT parity errors, and output pad comparison errors are latched by the controller's Comparator block during \( \phi_3 \). All five signals are sent directly to the Internal Rollback block (Section 3.3.7). In addition, the latched busA and busB parity error signals are sent to the output pads to become the external repair signals (out.repairA and out.repairB respectively; see Section 3.3.2).

3.3.5. Post-rollback Counter

This two-bit counter keeps track of the number of cycles since the last rollback. It is set to 1 during a rollback cycle and increments by one each subsequent cycle until it rolls over to zero, after which it stays at zero. If the Internal Rollback Logic (Section 3.3.7) detects an error when the counter is non-zero, and the normal absolute rollback amount for that type of error is the same as the counter value minus one, then a relative rollback is signaled instead (see Section 3.1.3).

3.3.6. Rollback Enable Bit and Counter

When a reset or shutdown occurs, internal parity and comparison errors should be ignored for a few cycles afterwards (see Section 3.2.5). The three-bit Rollback Enable Counter is set to four when a reset or shutdown trap is taken and decrements by one each subsequent cycle until it reaches zero, after which it remains at zero. Whenever the counter is greater than zero, the Rollback Enable Bit is cleared. When the Rollback Enable Bit is cleared, internal rollbacks are disabled.

Both the Rollback Enable Counter and the Post-rollback Counter are implemented in a domino logic PLA similar to the ones used elsewhere in the controller. The inputs are latched during \( \phi_2 \) at the same the PLA is precharging, and the outputs are evaluated and latched during \( \phi_3 \).

3.3.7. Internal Rollback Logic

This block of static random logic determines if the next cycle is going to be a rollback cycle and, if so, the number of cycles to roll back. The block evaluates when any of the internal error signals change, and if the signals indicate a rollback is to occur, the external rollback line is pulled beginning \( \phi_1 \). The
amount to rollback is determined by the type of error (one cycle for busA/busB parity errors, two cycles for other errors) and the value of the Post-rollback Counter (Section 3.3.5). The amount is signaled on one of four lines, indicating "roll back 1," "roll back 2," "roll back relative 2," or "roll back relative 3." The first two (indicating absolute rollbacks) are sent directly to the Rollback Amount Encoder (Section 3.3.10), while the other two latter two (indicating relative rollbacks) are sent to the Selection Logic (Section 3.3.9) for translation into an absolute rollback amount. All four lines are mutually exclusive, so that it is not possible to signal more than one rollback amount.

3.3.8. Valid Bits

A set of valid bits like those used in the DWBs is used to keep track of which cycles are valid when performing a relative rollback. It is a four-bit shift register, updated each cycle, where each bit represents one of the past four cycles. When a rollback is performed, each cycle that is rolled back across has its corresponding bit set to zero. See [5] for implementation details. The first stage of each cell is loaded during $\phi_3$, and the second stage is updated during $\phi_4$. Invalidation takes place during $\phi_3$ of a rollback cycle.

A valid cycle is one in which its corresponding valid bit here is set to 1. This will include any cycle that wasn’t a rollback cycle, repair cycle, or wait cycle, and any cycle that wasn’t invalidated by a previous rollback.

3.3.9. Selection Logic

The Selection Logic translates a relative rollback amount into an absolute rollback amount, using the valid bits to select the second or third most recent valid instruction. It is the same as the three-level selection logic in the PC[5], except that the logic to select the most recent valid cycle has been removed. The logic will select the second or third cycle depending on whether a 1 comes in on the "roll back relative 2" line or the "roll back relative 3" line (they are supposed to be mutually exclusive; there should never be a case where both lines carry a 1) and will output a 1 on one of six lines corresponding to
cycles 2 through 7. The logic is purely static, so the proper line will be set shortly after either the
selection lines change or the valid bits shift, both of which happen in $\phi_4$.

3.3.10. Rollback Amount Encoder

This block of static random logic, labeled “Encode” in Figure 3.1, encodes the eight rollback
amount lines, two from the Internal Rollback block (“roll back one,” “roll back two”) and six from the
Selection Logic, into a three-bit binary number representing the actual rollback amount. If the Selection
Logic gives a value greater than four, it is encoded as four (i.e., the processor will never cause a rollback
greater than four cycles).

3.3.11. Rollback Amount Arbitration

It is possible that more than one module in the system will signal a rollback in a given cycle. If this
is the case, and the two modules gate different rollback amounts onto the rollback amount lines, the
greater amount should be the taken. Each module requesting a rollback needs to decide if it is requesting
the greatest amount and, if not, remove its value from the rollback amount line.

The procedure used by the Mirror Processor is a straightforward implementation of the Futurebus
arbitration protocol [9]. The rollback amount being gated onto the external rollback amount lines by the
processor is compared bit by bit with the amount read in from the same lines, starting with the most
significant bit. To begin with, every bit is enabled and gates its value out on line. If any bit gates out a 0
and reads in a 1, it knows some other module is gating out a greater value starting with that bit position,
so it disables all the lower significant bits. Once a bit is disabled, it stops gating its value on the line. If
the bit gating a 0 out reads a 0 in, then it re-enables the lower bits. Note that the most significant bit is
never disabled.

Figure 3.2 shows the implementation of the arbitration logic. In the figure, the $out.RBx$ lines are the
rollback amount lines output by the selection logic, while the $in.RBx$ lines are the external rollback
amount lines (in this figure, they are assumed to be active high). The $enb.pad.RBx$ lines are the actual
rollback amount output pad enable signals (they enable the pads connected to the external rollback amount lines; see Section 3.3.2), and `enb.pad.rb` enables the external rollback line pad. Note that the rollback amount pads will not be enabled unless the rollback signal is asserted as well.

### 3.3.12. Frame Counter and Rollback Counter

The Frame counter is a four-bit counter that increments each cycle and repeatedly counts from zero to 15. It is set to zero whenever a reset or shutdown occurs, or when a Clear Rollback Bit instruction (Section 6.3.1) is executed. The Rollback Counter is a two-bit counter that is incremented whenever a rollback occurs and is reset whenever the Frame Counter goes to zero. It indicates the number of rollbacks that have occurred in the present frame, and is used to detect repeated rollbacks. Both of these counters are implemented in a domino logic PLA that latches its inputs $\phi_1$, precharges $\phi_3$, and evaluates $\phi_4$.

### 3.3.13. Shutdown Logic

This block of static random logic determines when to shut down. A shutdown condition exists if the external rollback amount is greater than four or if a rollback occurs and the Rollback Counter already
has a value of three, indicating that four rollbacks have occurred within the last 16 cycles, which is a sign that the processor is probably in an infinite rollback loop. If a shutdown condition is flagged, the shutdown line is pulled \( \phi_1 \).

3.3.14. Pad Enabling

To assert the external shutdown and rollback signals, the output pad for that signal must be enabled to force a zero onto that line (Section 3.3.2). The Enable Pads block enables one of these two output pads whenever it is decided that that particular signal is to be pulled. Each pad is enabled on \( \phi_1 \) if needed and disabled on \( \phi_4 \), allowing the input latches to latch the signals during \( \phi_2 \).

3.3.15. Rollback Bit

The Rollback Bit is initially cleared on a reset and is set on \( \phi_2 \) whenever a rollback occurs. Once it is set, it remains until cleared by a \texttt{clirrbm} or \texttt{clirrbs} instruction (Section 6.3.1) or a reset.

3.3.16. Rollback Procedure

The entire rollback procedure is shown in Figure 3.3, with the timing of the various Rollback Logic blocks shown in Figure 3.4. The procedure begins when comparison error signals are latched \( \phi_3 \) and parity error signals are latched \( \phi_4 \). The Internal Rollback block determines the amount to roll back as soon as the error signals are stable in \( \phi_4 \) and at the same time checks the Post-rollback Counter to see if a relative rollback is needed. If so, the proper selection signal is sent to the Selection Logic which then sends its output to the Rollback Amount Encoder, all during \( \phi_4 \) and \( \phi_1 \). The external rollback and rollback amount lines are pulled on the rising edge of \( \phi_1 \), at which time rollback amount arbitration takes place. This has until the end of \( \phi_2 \) to complete, at which time the input latches will have latched the external signals and set the state.rb bit. All the DWBs in the processor are invalidated during \( \phi_3 \), and at the same time the Post-rollback counter is set to one. The Rollback Counter is checked during \( \phi_4 \) to see if a shutdown should occur, and then it is incremented. If the shutdown condition is met, the shutdown line
is pulled \( \phi_1 \), and the trap is taken on \( \phi_2 \).

3.3.17. Critical Path

The Rollback Logic's critical path occurs during rollback amount arbitration. In a system where only the processor and its slave can request a rollback, the worst case arbitration scenario will be if one processor requests a rollback of four (binary 100) and the other processor requests a rollback of three (binary 011), since bits one and zero of the second processor must first be enabled and then disabled, and the line value must be stable at the input latch output by the end of \( \phi_2 \). SPICE runs show that with a 20 pf load on the external line and a 750Ω pullup resistance, bit zero can enable and disable itself with plenty of time to spare if each processor's rollback amount has been determined before the rising edge of \( \phi_1 \) (see Figure 3.5).

3.4. State Repair

Figure 3.6 shows the state repair procedure. During the first repair cycle (labeled repair 1), both processors read the values out of the registers indicated by the instruction just rolled back to (i.e., the normal \( \phi_1 \) register file read). The register containing the value to repair is then gated onto \( busD \) during \( \phi_2 \). During the second repair cycle (repair 2), the processor with the correct value assumes master mode and gates \( busD \) onto \( busOUT \) and the address/data lines during \( \phi_2 \), while the other processor assumes slave mode and reads the correct data in from the address/data lines and onto \( busIN \) and the shifter during \( \phi_3 \) and from the shifter to the register file during \( \phi_4 \). Both processors then restore their respective master/slave modes and resume with the instruction that was rolled back to.
Figure 3.3: Rollback Procedure
Figure 3.4: Rollback Control Timing
Figure 3.5: Rollback Amount Arbitration
Figure 3.6: State Repair Procedure
4. CONTROLLER IMPLEMENTATION

The details of the controller layout are presented in this chapter. The Mirror Processor was implemented in 2 μm CMOS using the MOSIS SCMOS design rules with λ=1 μm. The layout tool used was magic [6].

4.1. Layout

Figure 4.1 shows the floorplan of the completed controller layout, minus the routing between blocks. Table 4.1 lists the names of each block in the layout and their function. Note that the state compression logic is not strictly a part of the controller, but it was included in the controller layout in order not to increase the stride of the data path. A summary of the statistics on each module is shown in Table 4.2. The number of hand-drawn transistors does not include transistors generated by PLA tools and transistors in copies of replicated cells. Also, the number of inputs does not include the clock signals or the power lines. A complete accounting of every signal in the controller is given in Appendix A.

Figure 4.1: Controller Floorplan

4.2. Overhead

4.2.1. Area Overhead

Table 4.3 shows the dimensions of the active area of each of the top-level modules in the controller and an estimation of the dimension of each module if there was no support for micro rollback, state repair, or error detection. Estimates are given for the cases where one of the three features is missing and for the case were none are present. Active area in this case is defined as the area of the smallest bounding
<table>
<thead>
<tr>
<th>Block</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK</td>
<td>Clock generator</td>
</tr>
<tr>
<td>CMP</td>
<td>Comparator Logic: generates global comparison error signal (error.CMP)</td>
</tr>
<tr>
<td>COND</td>
<td>Condition Code Logic: generates cond signal</td>
</tr>
<tr>
<td>CUDATA</td>
<td>Controller Rollback Memory</td>
</tr>
<tr>
<td>EXT</td>
<td>External Signal Latches: latches external wait, reset, and shutdown signals</td>
</tr>
<tr>
<td>INT</td>
<td>Interrupt Logic: generates global interrupt signal (state.int) and interrupt vector</td>
</tr>
<tr>
<td>IRLATCH</td>
<td>bus/R Latch and Driver</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory Control Logic: generates all signals for memory accesses</td>
</tr>
<tr>
<td>NS</td>
<td>Next State Logic, plus all associated latches, drivers, multiplexors, and feedback paths</td>
</tr>
<tr>
<td>PADENB</td>
<td>Generates enable signal for address/data output pads</td>
</tr>
<tr>
<td>PHI1A</td>
<td>$\phi_1$ PLA</td>
</tr>
<tr>
<td>PHI1B</td>
<td>$\phi_2$ PLA</td>
</tr>
<tr>
<td>PHI4</td>
<td>$\phi_4$ PLA</td>
</tr>
<tr>
<td>POST</td>
<td>Post-FSM Logic</td>
</tr>
<tr>
<td>RB</td>
<td>Rollback Logic</td>
</tr>
<tr>
<td>STATE</td>
<td>State Compression Logic: generates four bits of exported state based on eight bits of state from the data path and four bits of state from the controller</td>
</tr>
<tr>
<td>VALID</td>
<td>Valid Bit Logic: generates valid bits for every DWB</td>
</tr>
<tr>
<td>CU</td>
<td>Entire controller</td>
</tr>
</tbody>
</table>

Table 4.1: Blocks in Controller Layout

box of a module that does not include power lines or routing to other modules.

For each module, the dimensions were arrived at by determining which parts of the module were not needed for that particular feature (rollback, state repair, or error detection) and estimating the change in the height and width of the module if those parts were to be removed. This was done by looking at the parts of the module that determine the overall height and width. In most cases, the module is laid out in parallel rows of gates, with power rails running between each row. Usually, a single row (the "critical row") determines the width of the module, and the number of rows determines the height. If any of the parts to be removed were part of the "critical row," then the row was shrunk by the amount taken up by the removed parts, and the new width of the module was determined to be the new width of the "critical row" or, if it became smaller than another row, the width of the new "critical row." If an entire row could be removed, or if enough of one row could be removed that all of another row could be moved into the vacant spots, then the new height of the module was determined to be the old height minus the height.
<table>
<thead>
<tr>
<th>Block</th>
<th>Dimensions (λ)</th>
<th>Number of Transistors</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Product Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Hand-drawn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOCK</td>
<td>404x620</td>
<td>389</td>
<td>239</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>151x417</td>
<td>180</td>
<td>170</td>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>COND</td>
<td>420x287</td>
<td>248</td>
<td>0</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>CUDATA</td>
<td>591x963</td>
<td>1328</td>
<td>149</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>EXT</td>
<td>176x96</td>
<td>56</td>
<td>24</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>INT</td>
<td>329x200</td>
<td>196</td>
<td>196</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>IRLATCH</td>
<td>376x404</td>
<td>366</td>
<td>80</td>
<td>19</td>
<td>14</td>
</tr>
<tr>
<td>MEM</td>
<td>212x482</td>
<td>248</td>
<td>114</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>NS</td>
<td>250x396</td>
<td>210</td>
<td>162</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>PADENB</td>
<td>124x91</td>
<td>44</td>
<td>44</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>PHI1A</td>
<td>997x680</td>
<td>948</td>
<td>6</td>
<td>13</td>
<td>28</td>
</tr>
<tr>
<td>PHI1B</td>
<td>1031x690</td>
<td>1044</td>
<td>14</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>PHI4</td>
<td>669x746</td>
<td>610</td>
<td>30</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>POST</td>
<td>308x241</td>
<td>218</td>
<td>146</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>RB</td>
<td>713x1371</td>
<td>1131</td>
<td>356</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>STATE</td>
<td>167x521</td>
<td>228</td>
<td>68</td>
<td>17</td>
<td>4</td>
</tr>
<tr>
<td>VALID</td>
<td>293x164</td>
<td>166</td>
<td>118</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>CU</td>
<td>1553x7451</td>
<td>7636</td>
<td>1824</td>
<td>69</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 4.2: Controller Implementation Statistics

<table>
<thead>
<tr>
<th>Block</th>
<th>Dimensions (λ)</th>
<th>Dimensions without: rollback</th>
<th>Dimensions without: repair</th>
<th>Dimensions without: detection</th>
<th>Dimensions without: all</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK</td>
<td>390x620</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>CMP</td>
<td>116x417</td>
<td>116x397</td>
<td>116x309</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>COND</td>
<td>420x287</td>
<td>420x287</td>
<td>420x287</td>
<td>420x287</td>
<td>420x287</td>
</tr>
<tr>
<td>CUDATA</td>
<td>560x905</td>
<td>0x0</td>
<td>560x905</td>
<td>560x905</td>
<td>0x0</td>
</tr>
<tr>
<td>EXT</td>
<td>176x86</td>
<td>176x86</td>
<td>176x86</td>
<td>176x86</td>
<td>176x86</td>
</tr>
<tr>
<td>INT</td>
<td>167x316</td>
<td>167x299</td>
<td>167x299</td>
<td>167x316</td>
<td>167x299</td>
</tr>
<tr>
<td>IRLATCH</td>
<td>365x362</td>
<td>0x0</td>
<td>365x362</td>
<td>365x362</td>
<td>0x0</td>
</tr>
<tr>
<td>MEM</td>
<td>189x471</td>
<td>189x471</td>
<td>189x477</td>
<td>189x471</td>
<td>189x377</td>
</tr>
<tr>
<td>NS</td>
<td>189x371</td>
<td>189x256</td>
<td>60x301</td>
<td>189x371</td>
<td>60x198</td>
</tr>
<tr>
<td>PADENB</td>
<td>124x74</td>
<td>124x74</td>
<td>107x74</td>
<td>124x74</td>
<td>107x74</td>
</tr>
<tr>
<td>PHI1A</td>
<td>994x679</td>
<td>868x638</td>
<td>810x601</td>
<td>994x679</td>
<td>700x569</td>
</tr>
<tr>
<td>PHI1B</td>
<td>1028x682</td>
<td>886x587</td>
<td>692x446</td>
<td>668x542</td>
<td>600x419</td>
</tr>
<tr>
<td>PHI4</td>
<td>593x674</td>
<td>593x674</td>
<td>543x608</td>
<td>593x674</td>
<td>543x608</td>
</tr>
<tr>
<td>POST</td>
<td>308x230</td>
<td>244x230</td>
<td>268x230</td>
<td>188x230</td>
<td>180x230</td>
</tr>
<tr>
<td>RB</td>
<td>690x1371</td>
<td>0x0</td>
<td>690x1319</td>
<td>572x382</td>
<td>0x0</td>
</tr>
<tr>
<td>STATE</td>
<td>132x500</td>
<td>132x467</td>
<td>132x467</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>VALID</td>
<td>293x151</td>
<td>0x0</td>
<td>277x151</td>
<td>293x151</td>
<td>0x0</td>
</tr>
</tbody>
</table>

Table 4.3: Area Overhead, Dimensions
of the removed row. In many instances, a part of a module is used for more than one feature; in that case, the part is not considered a candidate for removal except in the case where no features are present. For the PLA’s, the new area was calculated by removing all input and output signals used solely by the feature in question and regenerating the PLA with espresso, eqnott, and mpla [6].

Table 4.4 gives the percentage of area that each module would be reduced by in the absence of each feature; the values were calculated directly from Table 4.3. Support for all three features takes up 67% of the total active area of the controller. Although it is not listed in the tables, it should also be noted that bustbR would not be needed in the absence of micro rollback and state repair.

<table>
<thead>
<tr>
<th>Block</th>
<th>Area Overhead, Percent</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rollback</td>
<td>repair</td>
<td>detection</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>CLOCK</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>4.80</td>
<td>25.90</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>COND</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>CUDATA</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>EXT</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>5.38</td>
<td>5.38</td>
<td>0.00</td>
<td>5.38</td>
<td></td>
</tr>
<tr>
<td>IRLATCH</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>0.00</td>
<td>19.96</td>
<td>0.00</td>
<td>19.96</td>
<td></td>
</tr>
<tr>
<td>NS</td>
<td>31.00</td>
<td>74.24</td>
<td>0.00</td>
<td>83.06</td>
<td></td>
</tr>
<tr>
<td>PADENB</td>
<td>0.00</td>
<td>13.71</td>
<td>0.00</td>
<td>13.71</td>
<td></td>
</tr>
<tr>
<td>PHI1A</td>
<td>17.95</td>
<td>27.87</td>
<td>0.00</td>
<td>40.99</td>
<td></td>
</tr>
<tr>
<td>PHI1B</td>
<td>25.82</td>
<td>55.98</td>
<td>48.36</td>
<td>64.14</td>
<td></td>
</tr>
<tr>
<td>PHI4</td>
<td>0.00</td>
<td>17.40</td>
<td>0.00</td>
<td>17.40</td>
<td></td>
</tr>
<tr>
<td>POST</td>
<td>20.78</td>
<td>12.99</td>
<td>38.96</td>
<td>41.56</td>
<td></td>
</tr>
<tr>
<td>RB</td>
<td>100.00</td>
<td>3.79</td>
<td>76.90</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>STATE</td>
<td>6.60</td>
<td>6.60</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>VALID</td>
<td>100.00</td>
<td>5.46</td>
<td>0.00</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>50.10</td>
<td>19.98</td>
<td>30.62</td>
<td>67.12</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Area Overhead, Percentage of Total Area

4.2.2. Transistor Overhead

Table 4.5 shows the number of transistors that could be removed from each block if one or all of the three features is missing. As in the area overhead calculations, the parts of each module that are not needed for each feature were identified, and the number of transistors in each part was subtracted from the
total. For the PLA’s, transistor counts were taken of the new PLA’s generated for the area overhead calculations, and the resulting counts were subtracted from the transistor count of the original PLA. Table 4.6 translates Table 4.5 into percentages; it gives for each block the percentage of the total number of transistors that the block could be reduced by in the absence of each feature. Over 62% of the transistors in the controller are used in the support of micro rollback, state repair, and error detection.

<table>
<thead>
<tr>
<th>Block</th>
<th>Transistors</th>
<th>Transistors Used For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>rollback</td>
</tr>
<tr>
<td>CLOCK</td>
<td>389</td>
<td>0</td>
</tr>
<tr>
<td>CMP</td>
<td>180</td>
<td>0</td>
</tr>
<tr>
<td>COND</td>
<td>248</td>
<td>0</td>
</tr>
<tr>
<td>CODATA</td>
<td>1328</td>
<td>1328</td>
</tr>
<tr>
<td>EXT</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>INT</td>
<td>196</td>
<td>8</td>
</tr>
<tr>
<td>IRLATCH</td>
<td>366</td>
<td>366</td>
</tr>
<tr>
<td>MEM</td>
<td>248</td>
<td>0</td>
</tr>
<tr>
<td>NS</td>
<td>210</td>
<td>30</td>
</tr>
<tr>
<td>P adenB</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>PHI1A</td>
<td>948</td>
<td>155</td>
</tr>
<tr>
<td>PHI1B</td>
<td>1044</td>
<td>279</td>
</tr>
<tr>
<td>PHI4</td>
<td>610</td>
<td>0</td>
</tr>
<tr>
<td>POST</td>
<td>218</td>
<td>22</td>
</tr>
<tr>
<td>RB</td>
<td>1131</td>
<td>1131</td>
</tr>
<tr>
<td>STATE</td>
<td>228</td>
<td>44</td>
</tr>
<tr>
<td>VALID</td>
<td>166</td>
<td>166</td>
</tr>
<tr>
<td>total</td>
<td>7636</td>
<td>3535</td>
</tr>
</tbody>
</table>

Table 4.5: Transistor Overhead, Number Used

4.2.3. Examples

As examples, the overhead calculations for the CMP, RB, and STATE blocks will be described in detail.

4.2.3.1. CMP

The only signal associated with rollback is \textit{state.rb}. It is used as an input to an inverter and a three-input NOR, and the inverted signals is used as an input to a three-input NAND. The inverter and the NOR are part of the "critical row," so removing the inverter and one input of the NOR would shrink the
<table>
<thead>
<tr>
<th>Block</th>
<th>Transistor Overhead, Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rollback</td>
</tr>
<tr>
<td>CLOCK</td>
<td>0.00</td>
</tr>
<tr>
<td>CMP</td>
<td>3.33</td>
</tr>
<tr>
<td>COND</td>
<td>0.00</td>
</tr>
<tr>
<td>CUDA</td>
<td>100.00</td>
</tr>
<tr>
<td>EXT</td>
<td>0.00</td>
</tr>
<tr>
<td>INT</td>
<td>4.08</td>
</tr>
<tr>
<td>IRLATCH</td>
<td>100.00</td>
</tr>
<tr>
<td>MEM</td>
<td>0.00</td>
</tr>
<tr>
<td>NS</td>
<td>14.29</td>
</tr>
<tr>
<td>PADENB</td>
<td>0.00</td>
</tr>
<tr>
<td>PHI1A</td>
<td>16.35</td>
</tr>
<tr>
<td>PHI1B</td>
<td>26.72</td>
</tr>
<tr>
<td>PHI4</td>
<td>0.00</td>
</tr>
<tr>
<td>POST</td>
<td>10.09</td>
</tr>
<tr>
<td>RB</td>
<td>100.00</td>
</tr>
<tr>
<td>STATE</td>
<td>19.30</td>
</tr>
<tr>
<td>VALID</td>
<td>100.00</td>
</tr>
<tr>
<td>total</td>
<td>46.29</td>
</tr>
</tbody>
</table>

Table 4.6: Transistor Overhead, Percentage of Total Number

width by 20λ. The height would not change, as the number of rows remains the same. The active area would then shrink from 48372 λ² to 46052 λ², so the area overhead for rollback is 4.80% of the total area. The number of transistors in the inverter is two, and two transistors are used per input in the NOR and NAND gates, so the total number of transistors used for rollback is six. Six transistors out of a total of 180 gives a 3.33% transistor overhead for rollback.

Without state repair, error.busOUT would not be needed since busOUT is only checked during the second repair cycle. The logic associated with this signal involves an XOR, a NAND, and a pass gate in the "critical row." Removal of these items would shrink the "critical row" by 108λ, making it shorter than the next widest row. However, this next widest row would shrink by 102λ, making it shorter than the new width of the "critical row," so the overall width will continue to be determined by the "critical row," and it will be 108λ shorter. The height will remain the same since no complete row can be removed. The overall active area will shrink from 48372 λ² to 46052 λ², for a 25.90% area overhead for state repair. The gates involved in generating error.busOUT are two XOR's, three NAND's, three pass
gates, and five inverters. The total number of transistors making up these gates is 46 out of a total of 180, resulting in a 25.56% transistor overhead for state repair.

Since the only purpose of the Comparator Logic is to generate a signal when an error is detected, there would be no need for this block if error detection were not being performed. Thus, both the area and the transistor overhead is 100% for error detection.

As the Comparator Logic block would not be needed for error detection, it would also be absent if all three features were removed, so the overall area and transistor overhead is 100%.

4.2.3.2. RB

Without support for micro rollback, there would be no need for the Rollback Logic block, so the area and transistor overhead for rollback is 100%.

In the absence of state repair, the following things would be removed: logic input latches, and output drivers for repairA, repairB, repairA_, repairB_, and ctrl.norepair, the logic to hold the repair signals constant through both state repair cycles (the block labeled ‘REPAIR’ in the layout), the logic and output drivers for out.repairA_ and out.repairB_, and the logic for generating an internal rollback on a busOUT parity error (which is only checked during a state repair; see Section 3.2.4). The width of the module is set by the width of the input latches block (RBINPUTS). Since five of the latches would be removed, making it shorter than the selection logic block (RBSELECT), and RBSELECT would not shrink, the new width of RB would be set by the right edge of the rollback amount encoding sub-block (ENCODE), thus making the overall width 52λ less. The overall height would remain the same since no complete row could be removed, and the overall area would shrink from 945990 λ² to 910110 λ², resulting in a 3.79% area overhead. The number of transistors used by the logic listed above is 86 out of a total of 1131, giving a 7.60% transistor overhead for state repair.

Without error detection, the Rollback Logic block would not need to generate internal rollbacks, and it would only need to respond to external rollbacks. The only parts of the Rollback block that would
need to remain are the Frame Counter/Rollback Counter PLA (RBSHUTDOWN; see Section 3.3.12), the shutdown logic (SHUTDOWN), the rollback bit (RBBIT), and the input latches minus the busA/busB parity error latches. Since this is substantially less than what is presently there, the entire block could be rearranged so that the blocks are stacked on top of each other. The widest block is RBSHUTDOWN, so the new width would be 382λ. Arranging the stacking so that SHUTDOWN and RBBIT are in the same row, the new height would be 572λ. Thus, the new overall area would be 218504 λ², as compared with the original 945990 λ² area, resulting in a 76.90% area overhead. A transistor count of the missing blocks, plus the 12 transistors that would be taken out of RBINPUTS, totals 607, resulting in a 53.67% transistor overhead for error detection.

When all three features are removed, the entire block can be removed, since micro rollback would no longer be supported. Thus, the overall area and transistor overhead is 100%.

4.2.3.3. STATE

The State Compression Logic takes eight bits of compressed signature from the data path, the three state bits from the Next State Logic, and the previous cycle’s opcode bit 5, and XOR’s them all down into a four-bit signature. All of the data path signals are masked out during a rollback cycle, so all the masking logic would be removed in the absence of micro rollback. In addition, the latch that saves state.rb across ϕ₂ would also not be needed, as well as the input inverter for state.rb. This results in a total of eight NAND gates, one NOR gate, one pass gate, and three inverters. The width of the module is determined by the width of the XOR’s plus the width of a latch plus the width of two input inverters. The width of the XOR’s will not decrease, but one of the latches would be removed, allowing room for the four input inverters on the right to be put in its place. Thus, the overall width of the module would decrease by 33λ, while the height would not change at all, bringing the overall area down from 66000 λ² to 61644 λ², resulting in an area overhead of 6.60% for micro rollback. The number of transistors in the logic that would be removed is 44 out of a total of 228, giving a 19.30% transistor overhead.

The case in which support for state repair is removed is similar to the micro rollback case. The
busD state compression bits are masked out during the first repair cycle, so the masking logic would need to be removed. There is also a latch that saves state.repair1 across φ2 as well as an input inverter; these too would be removed. Finally, neither state.repair1 nor state.repair2 would be generated by the Next State Logic, so neither bit needs to be XOR’ed into the final result. Thus, two XOR’s could be removed. The total width of the XOR’s would shrink by 72λ; however, the width of the output drivers would remain the same. Therefore, the overall decrease in width would be accomplished by removing the state.repair1 latch and the four input inverters that handle the repair signals. As in the rollback case, the decrease in width is 33λ and the decrease in height is zero, for a total area overhead of 6.60%. The number of transistors in the unneeded logic totals 26 out of 228, for a transistor overhead of 11.40% for state repair.

Without error detection, there would be no need for exportation of internal state. Thus, the entire State Compression block could be removed, resulting in a 100% area and transistor overhead.

When all three features are removed, the need for state exportation is also removed, since error detection is no longer supported. Thus, the overall area and transistor overhead is 100%.
5. SIMULATION AND DEBUGGING

This chapter presents the methods used in simulating and debugging the Mirror Processor design at the architecture and switch levels.

5.1. Architecture Level Simulation

The entire processor was simulated at the architecture level using a commercial hardware simulation system called ENDOT[3]. The processor was specified in ISP', a hardware description language based on ISPS[1], and two of them were simulated running as a master/slave pair. Simulation input was produced by assembling several small program fragments with a modified RISC II assembler and converting the resulting a.out file into an ENDOT memory. The actual simulations were run by manually controlling each clock phase and then manually checking the values of the lines and registers that were relevant to the instruction being simulated. For example, after an ALU instruction had been executed, the register file DWB would be checked to make sure that the correct result had been stored in the correct register, and the PSW would be checked to make sure the the proper condition codes had been set. For a jump or call instruction, the PC and MAR would be checked to make sure that the proper address had been calculated. Different parts of the data path and controller would be checked similarly for other types of instructions.

The program fragments used as simulation input (see Appendix B) were designed to cover every instruction in as many modes as was practical. Some instructions, such as the jump instructions, are tested for only a few modes (i.e., not every possible combination of condition codes and branch condition was tested) that are believed to cover all the logic in the data path.

The error detection and rollback logic was tested by manually injecting faults into various places and checking that they were properly detected and corrected (if possible). A list was drawn up containing a representative list of transient faults and their expected error recovery procedures, and test sequences were written for each entry in the list. See Appendix C for the full list.
In each case above, the set of test cases was arrived at through ad-hoc means. There is no guarantee that every possible error case has been tested, or that every design error has been uncovered.

The choice of ENDOT as the hardware simulator was not without its problems. To begin with, the simulator has no notion of a capacitive bus (one that holds its value, even if nothing is driving it), and both the controller and the data path make heavy use of capacitive busses, so some very ugly hacks had to be put in the ISP description to emulate them. Another problem occurred with when blocks. A when block conditioned on several inputs usually evaluates whenever one of the inputs changes. However, once the block starts evaluating, the inputs are not checked again until the entire block has evaluated. Thus, inputs that change while the block is still evaluating do not force the block to reevaluate, and there is the possibility that behavior will mask errors in the logic. This is especially important since the capacitive bus emulation makes use of when blocks.

5.2. Switch Level Simulation

Switch level simulation of the extracted magic layout was done with bdsim[7]. Each block in the controller was simulated separately in order to verify that the magic layout matched the ENDOT specification. Bdsim test scripts were generated as follows: for each block, a C program was written containing a translation of the ENDOT description of that block. The C programs cycled through every possible input for the block and calculated the expected output based on the translated ENDOT description. Bdsim commands to generate each input vector and verify each output vector were then output. Several test scripts were also generated to verify the connections between each of the blocks and their interaction behavior. Switch level simulations of the entire data path and controller are described in [5].
6. TESTING

This chapter describes the methods that can be used to test the processor chip once it has been received from the manufacturer and details the additional hardware added to implement these testing functions.

6.1. Testing Considerations

There are several points that had to be kept in mind when designing testability features for the Mirror Processor. In addition to testing the vanilla RISC portion of the processor, all of the error detection and rollback logic must also be tested. In order to do that, schemes had to be developed to force parity and comparison errors and to verify that rollbacks had occurred. It was also desired that all this be accomplished with minimal extra hardware, and that the processor be able to periodically test itself while already in operation.

Because of the minimal hardware requirement, scan latches as used in RISC II were not used. They have a high area overhead and require separate clocks, and, as pointed out in [4], they don’t really add much to either the controllability or the observability of the chip. Instead, to keep the hardware to a minimum and at the same time allow for the self-testing requirement, it was decided to rely on functional testing, where test programs use normal instructions to test that the functionality of the processor is correct. A few new test instructions were added in order to exercise the parity checkers and comparators.

6.2. Testing Vanilla RISC

6.2.1. Functional Testing

*Functional testing*, in which the functionality of a microprocessor is tested using only its normal instruction set, is presented in [10] and [2]. Briefly, a processor is broken down into four functional units, and fault models are developed for each one. The functional units defined by the authors are *instruction sequencing*, *register decoding*, *data transfer* (busses), and *data manipulation* (ALU, shifter, etc.). Tests
for each functional unit are then developed, with the assumption that there can be any number of faults in a single functional unit, but that only one functional unit can contain faults. This is because the other functional units are used in verifying the one under test, so they must remain fault-free in order not to mask faults in the one being tested. Fault models and test procedures are presented for each functional unit except for data manipulation, since each block’s fault model and test procedure is specific to its implementation.

The instruction sequencing fault model and test procedure is particularly important, as the instructions are required to be fault free when testing the other functional units. The fault model presented in [2] states that an instruction is composed of a series of microinstructions executed in sequence, where each microinstruction is composed of a series of microorders executed in parallel. The term instruction is used in a broad sense to indicate a general type of action (for instance, READ a value from a register, ADD two numbers together, BRANCH to a certain location) that is independent of the specific microprocessor in question. Thus, an instruction may refer to a single machine instruction or to an entire sequence of machine instructions. The sequencing for a particular instruction is considered faulty if one or more microinstructions or microorders that are part of the instruction are not executed, or if one or more microinstructions or microorders are executed in addition to the proper ones for the instruction. Since microinstructions are composed of microorders, missing or additional microinstructions will manifest themselves as missing or additional microorders.

A simple fault is a fault in which there is at most one additional microorder (there may be any number of missing microorders). The fault model allows for any number of simple faults. It is shown in [2] that other types of faults (linked, coupled, and cyclic) can be decomposed into a series of simple faults and that detection of simple faults is sufficient to detect the other types of faults. In order to detect simple faults, a set of codewords is associated with all the registers in the processor except the PC. The codewords have the property that any simple fault operating on a codeword will produce a non-codeword. In addition, if a fault occurs such that a register ends up with a codeword associated with another register,
that other register will contain a non-codeword. Methods of choosing an appropriate set of codewords and mapping them to the register set are given in [2]. Therefore, in order to detect missing or additional microorders in the sequencing of a particular instruction, every register is loaded with its codeword, the instruction is executed, and the values of every register are read and compared against their expected values. Any simple fault will be detected by the presence of a non-codeword in a register.

A sequence of four instructions is used to verify the values of each of the registers. The instructions are: READ register $i$ into memory location $a$, LOAD (MOV) register $i$ from memory location $a$, compare (CMP) two registers and set the Z bit if they are equal, and branch if the Z bit is set (BEQ). The most important of these is the READ instruction, which corresponds to a store instruction in the Mirror Processor. Note that no instruction, including the verification instructions, can be tested for additional microorders until the READ instruction has been verified for every register. However, the READ instruction cannot be completely verified until the other three verification instructions have been verified. On the other hand, it is possible to test for missing microorders in the three instructions without their being completely verified, as shown in Procedure 1 in [2], so the first step in the testing process is to perform this test. It is shown in [2] that having no missing microorders in the three instructions is sufficient to verify the rest of the instructions.

Once this test has been done, the READ instruction can be verified for every register, after which every other instruction can be verified. The procedure for verifying the READ instruction is as follows: Each register is first loaded with its codeword. A register is chosen and its value is read out and verified using the other three verification instructions. Then, for every other register, the other register’s value is read out $M \times K + 1$ times, where $K$ is the maximum duration (number of cycles) of the instruction and $M$ is the maximum number microorders in any instruction (i.e., the maximum number of parallel operations supported by the processor). This forces any linked faults to show up. Then the original register is read and verified again, after which the other register is read and verified again. The whole loop is then repeated for every register. This procedure detects simple faults that operate on one register. Other
procedures to detect simple faults involving transfer of data between registers and simple faults involving arithmetic operations follow this first one; they involve alternately reading greater numbers of registers.

6.2.2. Application of Functional Testing

The test procedures outlined in [10] and [2] can be applied to the Mirror Processor for testing the vanilla RISC portion of the processor. It should be noted, however, that the processor model used in the above papers does not take into account some features which are specific to the Mirror Processor. These features should be taken into account when designing test procedures. Some suggestions for incorporating these features into the testing procedure are given below; however, it is by no means a comprehensive list.

First, the test procedures to verify the READ instruction for each register should be modified to include reading out of every entry in the register's DWB. This can be done by loading a register with its codeword and then immediately reading its value back out four times in succession. Since the DWB will shift every cycle, it will not be possible to run the verification instructions on each value as it is read out. Instead, the four values should be stored into other registers that have already been verified, and each of those registers should then be verified to be holding the same value stored into the original register. Note that it will be necessary to verify reading from the actual registers first before verifying reading from the DWB's, since there must be four already-verified registers to store the values read out of a DWB.

Secondly, the maximum duration $K$ of an instruction should include the time it takes for a value to go all the way through the DWB and into the actual register. When verifying all of the instructions apart from the READ instruction, and in particular the LOAD instruction (in this context, LOAD means "load a value into a register," but not necessarily from memory), it will be necessary to ensure that no additional microorders are executed as a value goes through a DWB (note that the READ test checks that no additional microorders are caused by the READ instruction only). Recall that additional microorders can be detected by the presence of a non-codeword in the register. Thus, each LOAD instruction should
be followed by (at least) four NOP instructions in order force a value to go all the way to the actual register before its value is read back out.

An important point to keep in mind is that great care should be taken in placing a machine instruction other than a nop in the delayed slot after a branch, particularly in the case of the verification instructions. The algorithm given in [2] to check for inactive microinstructions and microorders (Procedure 1) is a series of move (MOV) instructions followed by branches on the results of the MOV’s and on the results of compare instructions (CMP). Each CMP instruction is followed by a branch on zero (BEQ) instruction which is followed by either another branch or another CMP; in either case, the intention is that the following instruction be executed only if the BEQ had not taken place, so neither one can be placed in the delay slot (since it would be executed regardless of whether the branch took place or not). Each BEQ is in turn directly dependent on the CMP immediately before it, so the CMP cannot be placed in the delay slot, and each CMP is dependent on the MOV’s that took place prior to the CMP, so the MOV’s cannot be placed in the delay slot. Thus, for this algorithm, nothing can be placed in a delay slot. Since this algorithm uses the four verification instructions exclusively, it follows that the other tests that make use of the verification instructions will have similar compare-branch sequences, and any instruction placed in a delay slot would have to be independent of the verification sequence.

Finally, the instruction sequencing tests assume that any register can be read at any time without changing the user-visible state of the processor (other than the PC). While this is true for the global registers (R0 through R9), a particular local register cannot be directly accessed without first setting the Current Window Pointer in the PSW. Thus, when reading a local register, the PSW must also be saved and restored.

Appendix D shows the sequence of Mirror Processor instructions that make up the four verification instructions. The above points were taken into account when creating the sequences.

Apart from the error detection and rollback logic, the fault models presented in [10] and [2] also do not cover the interrupt handling logic. While trap conditions can be set up in the test programs, and
shutdowns can be forced with repeated rollbacks, there is no way the processor can initiate an external interrupt (i.e., pull the IRR line). Thus, part of the interrupt logic cannot be tested while in self-test mode. However, that part can be tested if an external tester is available to pull the IRR line.

6.3. Testing Error Detection and Rollback Logic

6.3.1. Test Instructions

 Normally, the error detection and rollback schemes are transparent to the user, so there is no way to force a parity or comparison error using the regular RISC II instructions. Thus, several new instructions were added to explicitly force parity and comparison errors and to verify that a rollback has occurred. The instructions are as follows; the argument types are listed in Table 6.1:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs1</td>
<td>Source register 1: register number, instruction bits 14 to 18; content of register is used as operand.</td>
</tr>
<tr>
<td>Rs2</td>
<td>Source register 2: register number, instruction bits 0 to 4 (with IMM bit [13] off); content of register is used as operand.</td>
</tr>
<tr>
<td>Rd</td>
<td>Destination register: register number, instruction bits 19 to 23; result of operation will be placed in this register; for store instructions, content of register is used as operand.</td>
</tr>
<tr>
<td>S2</td>
<td>Source 2: either a register number or a 13-bit immediate. If S2 is a register number, it is the same as Rs2; if it is an immediate, it corresponds to instruction bits 0 to 12 (with IMM bit [13] on).</td>
</tr>
<tr>
<td>X</td>
<td>32-bit immediate: the assembler will turn this into a 19-bit offset from the PC corresponding to instruction bits 0 to 18; offset is added to PC at runtime to produce address of memory access.</td>
</tr>
</tbody>
</table>

Table 6.1: Instruction Argument Definitions

Clear Rollback Bit
clrrbm clrhrs

Add with Bad Parity
addbpm Rs1,S2,Rd addbps Rs1,S2,Rd

Clears the rollback bit in the Rollback Control Logic (Section 3.3.15). clrrbm clears the rollback bit on the master, while clrhrs clears the bit on the slave.

Functions as a normal add instruction, except that an incorrect parity bit is stored in the destination register of one of the processors. addbpm stores the bad parity in the master, while addbps stores the bad parity in the slave. This instruction is used to force parity errors on busA and busB.
Jump if Rollback Bit Is Set
jmprbm Rs1,Rs2,Rd
jmprbm Rs1,Rs2,Rd

If the rollback bit is set, a PC-relative jump is taken, using the contents of Rs2 as the offset, and Rs1 is stored in the destination register. If the rollback bit is not set, the branch is not taken, and either Rs1 or Rs2 is gated onto busD and stored in Rd, depending on the instruction and mode: for jmprbm, the master stores Rs1, and the slave stores Rs2; for jmprbs, the slave stores Rs1, and the master stores Rs2. Since the branch offset can have only a few limited values, there are two separate instructions to allow both the master and the slave to gate any value onto busD in order to exercise the busD state compression logic. This instruction is used to force state compression comparison errors as well as to verify that a rollback has occurred.

Store Bad Data
strbdm Rd,X
strbds Rd,X

Similar to a normal PC-relative store instruction. If the rollback bit is set, both processors will store the contents of Rs into location X; if it is cleared, one of the processors will store the contents of the MAR instead. For strbdm, the master will store the bad data, i.e., the contents of the MAR. For strbds, the slave will do so. This instruction is used to force a comparison error on the data portion of a memory write.

Load with Bad Parity
ldrbpm X,Rd
ldrbps X,Rd

Similar to a normal PC-relative load instruction. If the rollback bit is set, both processors will load Rd with the contents of location X; if it is cleared, one of the processors will gate the loaded data onto busIN with an incorrect parity bit. For ldrbpm, the master will load a bad parity bit, while for ldrbps, the slave will do so. This instruction is used to force a parity error on busIN.

6.3.2. Error Recovery Test Procedures

The procedures used to test the various error detection and rollback schemes are detailed below. A detailed program fragment showing each procedure is shown in the files test10.ras and test11.ras in Appendix B.

6.3.2.1. State Compression Comparison Error

The rollback bit is cleared on both processors (clrrbm and clrrbs), and then a Jump if Rollback (jmprbm or jmprbs) is executed. Since the rollback bits are cleared, the branch will not be taken, but each processor will gate a different value onto busD, causing a comparison error to be detected in the delay slot after the branch. A rollback of two cycles back to the jmprbm or jmprbs instruction will take place, during which the rollback bit will be set. The second time through, the branch should be taken to a continuation point in the test procedure.
To verify that the rollback had actually taken place, a jump to an error routine should follow the Jump if Rollback instruction. If the test arrives at the error routine, then either the rollback did not occur, or the rollback bit was not set during the rollback. In either case, it will be known that there is a fault in the rollback logic. If the test arrives at the continuation point, then either the rollback must have taken place and set the rollback bit correctly, or the rollback bit was never cleared in the first place. To verify the that rollback bits are not stuck at one, they should be cleared again, after which a Jump if Rollback to the error routine should be executed, with the constraint that the values gated onto busD are the same for both processors if the rollback bits are cleared. Thus, if the rollback bits are stuck, the test will arrive at the error routine, while if they are not, the test will continue on. To prevent different values from being gated onto busD, Rs1 and Rs2 should be the same register $R$. This forces the contents of register $R$ to be put onto busD regardless of the master/slave mode of each processor.

6.3.2.2. Register File Parity Error

Add with Bad Parity (addbpm or addbps) is used to store a bad parity bit on one of the processors, after which the rollback bits are cleared, and an instruction that reads the register with the incorrect parity is executed. A rollback of one cycle should occur, followed by a state repair. The instruction should be executed again with no error, after which there should be a jmprbm or jmprbps instruction. If the rollback and repair executed correctly, the rollback bit will have been set, and the branch should be taken to the continuation point. As described in Section 6.3.2.1, the Jump if Rollback instruction should specify the same register in both the Rs1 and Rs2 fields. This will prevent unwanted comparison errors since, even if the branch is not taken, the same value will be gated onto busD by both processors. Note that the parity bit written into the register file is not included in the exported state, so there will be no state compression comparison error when the non-matching bits are read out.
6.3.2.3. Memory Access, Address Comparison Error

The rollback bit is cleared on one processor but not the other (both bits can be set with a rollback forced by one of the previous tests), after which a Jump if Rollback instruction is executed. Since only of the processors has a set rollback bit, only that one will take the branch, resulting in each processor generating a different address for the instruction fetch in the following cycle. Hence, a comparison error will be detected the cycle following the execution cycle of the Jump if Rollback instruction. A rollback of two cycles back to the Jump if Rollback instruction will take place in the cycle after that, during which the rollback bit will be set on both processors. The second time through, both processors will take the branch to the continuation point. A branch to the error routine should follow the Jump if Rollback instruction, so that if the comparison logic does not detect the error, at least one of the processors will end up in the error routine. Here again, unwanted comparison errors can be prevented by ensuring that the same register is specified in both the Rs1 and Rs2 fields of the Jump if Rollback instruction (see Section 6.3.2.1).

6.3.2.4. Memory Access, Data Comparison Error

The rollback bit is cleared on both processors, and then a Store Bad Data (strbdm or strbds) instruction is executed. One of the processors will store the contents of Rd during the second cycle of the instruction, while the other one will store the contents of the MAR. The comparison error will be detected the following cycle, after which a rollback of two cycles back to the second cycle of the store will take place (and setting the rollback bits in the process). The second time through, both processors will store Rd. Following the store, there should be a Jump if Rollback instruction to verify that the rollback bit had been set. As in the Address Comparison Error case, a branch to the error routine should follow the Jump if Rollback instruction so that if the data mismatch goes undetected, at least one of the processors will end up in the error routine. And again, the same register should be specified in the Rs1 and Rs2 fields of the Jump if Rollback instruction in order to prevent a state compression comparison error (Section 6.3.2.1).
6.3.2.5. Input Bus (busIN) Parity Error

The rollback bit is cleared on both processors, and then a Load with Bad Parity (ldrbpm or ldrbps) instruction is executed. One of the processors will load a bad parity bit during the second cycle of the instruction, and that will be detected in the following cycle. A rollback of two cycles back to the second cycle of the load will take place after that, setting the rollback bits at the same time. The second time through, both processors will load the correct parity. Following the load, there should be a Jump if Rollback instruction to verify that the rollback bit had been set. As in the previous cases, both the Rs1 and Rs2 fields of the Jump if Rollback instruction should specify the same register so that both processors will gate the same value onto busD and prevent a state compression comparison error.
7. CONCLUSION

As shown in the preceding chapters, it is quite possible to build an efficient controller for the Mirror Processor. However, the area overhead for error detection and micro rollback support is rather large. Where the RISC II required only a simple instruction decoder, the Mirror Processor has three large PLA's plus a host of random logic to generate all the required control signals. The Rollback Control Logic block itself is equivalent in complexity (if not in area) to the rest of the controller and required a substantial amount of design effort. Extra area was also expended in order to meet the strict timing requirements. Based on this implementation, the area overhead for supporting micro rollback and duplex mode operation is approximately 67% of the active area in the controller.

The design time for this processor certainly was not minimal. A large amount of time was spent simulating various error conditions and verifying the correctness of the error detection and recovery schemes. Part of the problem was that the controller implementation began long before the data path design was stable and long before several major design features had been fully specified. In many cases, large portions of the controller had to be redone as new design decisions were made contradicting older specifications. In particular, the Rollback Logic was redesigned and laid out from scratch twice, once to correct a design error and improve the routing, and once to add in support for the new test instructions and logic for the detection and handling of multiple rollbacks. In addition, the Next State Logic was completely redesigned from scratch once to eliminate the dependence on the current instruction's opcode and remove the block from the $\phi_3$ critical path (although some of the layout from the initial design was salvaged). Also, the opcode encoding (i.e., grouping the opcode of each instruction with as many illegal opcodes as possible in order to minimize the number of bits needed to decode the instruction) was redone several times as new test instructions were specified. Finally, a last-minute decision to reassign the PLA outputs to different PLA's resulted in the creation of a third PLA. The original design called for two PLA's that were approximately the same size, one to evaluate during $\phi_4$ and the other to evaluate during $\phi_1$. However, in order to minimize the evaluation time of the $\phi_4$ PLA, all the outputs were moved
to the φ₁ PLA except for the eleven signals that were required at the beginning of φ₁. The resulting φ₁ PLA would have been much too large in terms of both area and speed, so it was split into two separate PLA's. Other, less major design changes, while not involving restarting from scratch, still required a large amount of time to accomplish, as each time a module was changed, the ENDOT description had to be reverified, the circuit design had to be re-SPICE'ed, and the bdsim test scripts had to be regenerated and rerun.

There are two things that would have greatly reduced the time spent on the design of the controller. First, as mentioned above, the controller should be based on stable specifications, *not* ones that are constantly changing. This alone would eliminate a large number of design iterations. Second, some sort of automated test procedure should be developed in order to reduce the time spent on manual simulations. Although simulating the vanilla RISC portion of the processor manually does not take up too much time, it is quite easy to come up with all sorts of strange, new error conditions for which new test sequences must be made, and after a while the number of tests that are needed to be performed becomes too large to do manually.

Given all the above complaints, however, it still appears that one can successfully detect and recover from most single-event transient errors using micro rollback, without significant time overhead.
A. CONTROLLER SIGNALS

This appendix lists every signal in the controller. The list is broken down by block, and for each block its inputs and outputs are listed. For each input, the block that supplies the input is also listed, and for each output, the blocks that the output goes to are listed.

<table>
<thead>
<tr>
<th>CLOCK</th>
<th>Input from</th>
</tr>
</thead>
<tbody>
<tr>
<td>in.csel</td>
<td>pads</td>
</tr>
<tr>
<td>in.phi</td>
<td>pads</td>
</tr>
<tr>
<td>in.phi1</td>
<td>pads</td>
</tr>
<tr>
<td>in.phi2</td>
<td>pads</td>
</tr>
<tr>
<td>in.phi3</td>
<td>pads</td>
</tr>
<tr>
<td>in.phi4</td>
<td>pads</td>
</tr>
<tr>
<td>in.sync</td>
<td>pads</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs (4)</th>
<th>Output to</th>
</tr>
</thead>
<tbody>
<tr>
<td>phi1</td>
<td>everywhere</td>
</tr>
<tr>
<td>phi2</td>
<td>everywhere</td>
</tr>
<tr>
<td>phi3</td>
<td>everywhere</td>
</tr>
<tr>
<td>phi4</td>
<td>everywhere</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CMP</th>
<th>Input from</th>
</tr>
</thead>
<tbody>
<tr>
<td>busB.par</td>
<td>RF</td>
</tr>
<tr>
<td>busD.par</td>
<td>RF</td>
</tr>
<tr>
<td>busOUT.par</td>
<td>BUSOUT</td>
</tr>
<tr>
<td>error.AD0_</td>
<td>pads</td>
</tr>
<tr>
<td>error.AD1_</td>
<td>pads</td>
</tr>
<tr>
<td>error.AdP_</td>
<td>pads</td>
</tr>
<tr>
<td>error.enb.addr_</td>
<td>pads</td>
</tr>
<tr>
<td>error.enb.data_</td>
<td>pads</td>
</tr>
<tr>
<td>error.id_</td>
<td>pads</td>
</tr>
<tr>
<td>error.ira</td>
<td>pads</td>
</tr>
<tr>
<td>error.rw</td>
<td>pads</td>
</tr>
<tr>
<td>error.size_</td>
<td>pads</td>
</tr>
<tr>
<td>error.state&lt;3:0&gt;_</td>
<td>pads</td>
</tr>
<tr>
<td>error.sysmode_</td>
<td>pads</td>
</tr>
<tr>
<td>gate.busOUT_padAD2</td>
<td>PHI4</td>
</tr>
<tr>
<td>gate.busOUT_padAD4</td>
<td>MEM</td>
</tr>
<tr>
<td>load.PAR_busA</td>
<td>POST</td>
</tr>
<tr>
<td>load.PAR_busB</td>
<td>POST</td>
</tr>
<tr>
<td>load.PAR_busOUTA</td>
<td>POST</td>
</tr>
<tr>
<td>load.PAR_busOUTB</td>
<td>POST</td>
</tr>
<tr>
<td>rfA.par</td>
<td>RF</td>
</tr>
<tr>
<td>rfB.par</td>
<td>RF</td>
</tr>
<tr>
<td>state.rb</td>
<td>RB</td>
</tr>
<tr>
<td>state.shutdown</td>
<td>EXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs (4)</th>
<th>Output to</th>
</tr>
</thead>
<tbody>
<tr>
<td>error.CMP</td>
<td>RB</td>
</tr>
</tbody>
</table>
error.busA                     RB
error.busB                     RB
error.busOUT_                  RB

COND
Inputs (8)          Input from
busIR<4:1>          IRLATCH
state.cc<3:0>       PSW

Outputs (1)          Output to
cond                  POST

CUDATA
Inputs (21)          Input from
CUdata.in<1:0>       NS
busIR<13:0>          IRLATCH
ctrl.CU_write        VALID
state.rb             RB
state.rb<2:0>        RB

Outputs (2)          Output to
busCU3<1:0>          NS

EXT
Inputs (4)          Input from
in.reset             pads
in.shutdown          pads
in.wait              pads
state.rb             RB

Outputs (3)          Output to
state.reset          INT, PSW, RB, VALID
state.shutdown       CMP, INT, POST, RB, VALID
state.wait           MEM, NS, POST, IRLATCH, VALID

INT
Inputs (17)          Input from
busBAR<1:0>          BAR
ctrl.badop           PHI1A
ctrl.badshift        SDEC
ctrl.int_enb         PSW
ctrl.over_under      PHI1A
ctrl.privop          PHI1A
in.irr               pads
out.size<1:0>        MEM
out.sysmode          PSW
state.PSW_overflow   PSW
state.rb             RB
state.repair1        NS
state.repair2        NS
state.reset          EXT
state.shutdown       EXT

Outputs (7)          Output to
_ivec<6:4>  IVEC
out.ira   pads
state.10  VALID
state.int CALLI, IRLATCH, NS, POST, PSW, RFTRAN, VALID
state.int3 VALID

IRLATCH
Inputs (19)  Input from
  IR.write   PHI1A
  busIN<31:19> BUSIN
  busIN13 BUSIN
  gate.IR  PHI1A
  state.int INT
  state.rb  RB
  state.wait EXT
Outputs (14)  Output to
  busIR13 CUDATA, PHI1A
  busIR12 CUDATA, MEM, NS, PHI1B, PHI4
  busIR11 CUDATA, MEM, PHI1B, PHI4
  busIR10 CUDATA, MEM, PHI1A, PHI1B, PHI4
  busIR9  CUDATA, MEM, PHI1A, PHI1B, PHI4
  busIR8  CUDATA, PHI1A, PHI1B, PHI4
  busIR7  CUDATA, PHI1A, PHI1B, PHI4
  busIR6 CUDATA, load.PSW_busCC
  busIR5 CUDATA, RFTRAN
  busIR4  COND, CUDATA, RFTRAN
  busIR3  COND, CUDATA, RFTRAN
  busIR2  COND, CUDATA, RFTRAN
  busIR1  COND, CUDATA, RFTRAN
  busIR0 CUDATA, PHI1B, PHI4

MEM
Inputs (10)  Input from
  busIR<12:9> IRLATCH
  st<2:0> NS
  state.repair1 NS
  state.repair2 NS
  state.wait EXT
Outputs (7)  Output to
  gate.busOUT_padAD4 CMP, PADENB
  out.enb.addr pads
  out.enb.data pads
  out.id pads
  out.rw pads
  out.size<1:0> INT, IMM, pads

NS
Inputs (9)  Input from
  busCU3<1:0> CUDATA
  busIR12 IRLATCH
ctrl.norepair       RB
repairA            RB
repairB            RB
state.int          INT
state.rb           RB
state.wait         EXT

**Outputs (9)**
- CUdata.in<1:0>    CUDATA
- st<2:0>           MEM, PHI1A, PHI1B, PHI4
- state.op_prev_    STATE
- state.repair1     INT, MEM, RB, STATE, VALID
- state.repair2     INT, MEM, POST, RB, STATE, VALID
- state.suspend     STATE

**PADENB**

**Inputs (5)**
- ctrl.override_master_ PHI4
- ctrl.override_slave_  PHI4
- gate.busOUT_padAD2    PHI4
- gate.busOUT_padAD4    MEM
- in.ms                 pads

**Outputs (1)**
- enb.AD_              pads

**PHI1A**

**Inputs (13)**
- busIR<13:7>         IRLATCH
- repairA             RB
- repairB             RB
- st<2:0>             NS
- state.rb_bit        RB

**Outputs (28)**
- ALU.carry.car       POST
- ALU.carry_val       POST
- ALU.gateOUT         POST
- ALU.gateOUT_cond    POST
- INC.gate            POST
- INC.gate_cond       POST
- IR.write             IRLATCH, POST, VALID
- PC.write             VALID
- RF.write             VALID
- ctrl.ALU_op<4:0>     ALU
- ctrl.DIMM_sxt       IMM
- ctrl.badop          INT, POST
- ctrl.over_under      INT
- ctrl.privop          INT
- gate.ALU_busD       ALU
- gate.DIMM_busT      IMM
- gate.IR             IRLATCH
- gate.PC_busD2       PC
gate.PC_busD4  PC
gate.PSW_busD  PSW
load.BAR_busDR  BAR
load.RFaddr_RA  RFTRAN
load.RFaddr_RB  RFTRAN
load.RFaddr_RD  RFTRAN

PHIIB

Inputs (16)  Input from
busIR<12:7>  IRLATCH
busIR0  IRLATCH
in.ms  pads
repairA  RB
repairA_  RB
repairB  RB
repairB_  RB
st<2:0>  NS
state.rb_bit  RB

Outputs (24)  Output to
PAR.busA  POST
PAR.busB  POST
PAR.busIN  POST
PAR.busOUTA  POST
PAR.busOUTB  POST
ctrl.PSW_reti  PSW
ctrl.RFpar_invert  RF
ctrl.SHIFT_sxtS  SHIFT
ctrl.busINpar_invert  BUSIN
ctrl.clrbb  RB
gate.MAR_busOUT2  MAR
gate.MAR_busOUT4  ALU, IMM, MAR
gate.SDR_busOUT  SDR
gate.SHIFT_busL  SHIFT
gate.SHIFT_busR  SHIFT
gate.busA_busD2  RF
gate.busA_busD4  RF
load.PSW_busD  PSW
load.SHIFT_busL  SHIFT
load.SHIFT_busR  SHIFT
load.SHIFT_busT  SHIFT
load.SHAm_BAR  SDEC
load.SHAm_IMM  SDEC
load.SHAm_busB  SDEC

PHI4

Inputs (14)  Input from
busIR<12:6>  IRLATCH
repairA  RB
repairA_  RB
repairB  RB
repairB_  
st<2:0>  
RB  
NS  

**Outputs (12)**

ctrl.CWP_inc<1:0>  
PSW, RFTRAN  
ctrl.PC_select  
PC  
ctrl.SHIFT_sxtT  
SHIFT  
ctrl.override_master_  
PADENB  
ctrl.override_slave_  
PADENB  
gate.IMM13_busT  
IMM  
gate.IMM19_busT  
IMM  
gate.busA_busS  
RF  
gate.busOUT_padAD2  
CMP, PADENB  
load.SHIFT_IMM  
SHIFT  
load.SHam_0  
SDEC  

**POST**

**Inputs (20)**

ALU.carry_car  
PHI1A  
ALU.carry_val  
PHI1A  
ALU.gateOUT  
PHI1A  
ALU.gateOUT_cond  
PHI1A  
INC.gate  
PHI1A  
INC.gate_cond  
PHI1A  
IR.write  
PHI1A  
PAR.busA  
PHI1B  
PAR.busB  
PHI1B  
PAR.busIN  
PHI1B  
PAR.busOUTA  
PHI1B  
PAR.busOUTB  
PHI1B  
cond  
COND  
ctrl.badop  
PHI1A  
state.cc0  
PSW  
state.int  
INT  
state.rb  
RB  
state.repair2  
NS  
state.shutdown  
EXT  
state.wait  
EXT  

**Output (10)**

ctrl.ALU_c  
ALU  
gate.ALU_busOUT  
ALU  
gate.INC_busOUT  
PC  
gate.padAD_busIN  
BUSIN  
load.PAR_busA  
CMP  
load.PAR_busB  
CMP  
load.PAR_busIN  
RB  
load.PAR_busOUTA  
CMP  
load.PAR_busOUTB  
CMP  
load.RFTRAN_busIN  
RFTRAN  

RB
### Inputs (20)

<table>
<thead>
<tr>
<th>Input</th>
<th>Input from</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctrl.RB_write</td>
<td>VALID</td>
</tr>
<tr>
<td>ctrl.clrrb</td>
<td>PHI1B</td>
</tr>
<tr>
<td>error.CMP_</td>
<td>CMP</td>
</tr>
<tr>
<td>error.busA</td>
<td>CMP</td>
</tr>
<tr>
<td>error.busB</td>
<td>CMP</td>
</tr>
<tr>
<td>error.busIN1</td>
<td>PARIN</td>
</tr>
<tr>
<td>error.busOUT_</td>
<td>CMP</td>
</tr>
<tr>
<td>in.RB&lt;2:0&gt;</td>
<td>pads</td>
</tr>
<tr>
<td>in.rb</td>
<td>pads</td>
</tr>
<tr>
<td>in.repairAm</td>
<td>pads</td>
</tr>
<tr>
<td>in.repairAs</td>
<td>pads</td>
</tr>
<tr>
<td>in.repairBm</td>
<td>pads</td>
</tr>
<tr>
<td>in.repairBs</td>
<td>pads</td>
</tr>
<tr>
<td>load.PAR_busIN</td>
<td>POST</td>
</tr>
<tr>
<td>state.repair1</td>
<td>NS</td>
</tr>
<tr>
<td>state.repair2</td>
<td>NS</td>
</tr>
<tr>
<td>state.reset</td>
<td>EXT</td>
</tr>
<tr>
<td>state.shutdown</td>
<td>EXT</td>
</tr>
</tbody>
</table>

### Outputs (17)

<table>
<thead>
<tr>
<th>Output</th>
<th>Output to</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctrl.norepair</td>
<td>NS</td>
</tr>
<tr>
<td>enb.pad.RB&lt;2:0&gt;</td>
<td>pads</td>
</tr>
<tr>
<td>enb.pad.rb</td>
<td>pads</td>
</tr>
<tr>
<td>enb.pad.shutdown</td>
<td>pads</td>
</tr>
<tr>
<td>out.repairA_</td>
<td>pads</td>
</tr>
<tr>
<td>out.repairB_</td>
<td>pads</td>
</tr>
<tr>
<td>repairA</td>
<td>NS, PHI1A, PHI1B, PHI4</td>
</tr>
<tr>
<td>repairA_</td>
<td>PHI1B, PHI4</td>
</tr>
<tr>
<td>repairB</td>
<td>NS, PHI1A, PHI1B, PHI4</td>
</tr>
<tr>
<td>repairB_</td>
<td>PHI1B, PHI4</td>
</tr>
<tr>
<td>state.rb</td>
<td>BAR, CUDA, EXT, INT, IR, IRLATCH, MAR, MEM, NS, PC, POST, PSW, RF, RFTRAN, SDR, STATE, VALID</td>
</tr>
<tr>
<td>state.rb&lt;2:0&gt;</td>
<td>CUDA, IR, MAR, PC, PSW, RF, SDR</td>
</tr>
<tr>
<td>state.rbbit</td>
<td>PHI1A, PHI1B</td>
</tr>
</tbody>
</table>

### STATE

### Inputs (17)

<table>
<thead>
<tr>
<th>Input</th>
<th>Input from</th>
</tr>
</thead>
<tbody>
<tr>
<td>state.PSW&lt;3:0&gt;</td>
<td>PSW</td>
</tr>
<tr>
<td>state.RFaddr&lt;3:0&gt;</td>
<td>RF</td>
</tr>
<tr>
<td>state.busD&lt;3:0&gt;</td>
<td>RF</td>
</tr>
<tr>
<td>state.op_prev_</td>
<td>NS</td>
</tr>
<tr>
<td>state.rb</td>
<td>RB</td>
</tr>
<tr>
<td>state.repair1</td>
<td>NS</td>
</tr>
<tr>
<td>state.repair2</td>
<td>NS</td>
</tr>
<tr>
<td>state.suspend</td>
<td>NS</td>
</tr>
</tbody>
</table>

### Outputs (4)

<table>
<thead>
<tr>
<th>Output</th>
<th>Output to</th>
</tr>
</thead>
<tbody>
<tr>
<td>out.state&lt;3:0&gt;</td>
<td>pads</td>
</tr>
</tbody>
</table>
VALID

Inputs (12) Input from
IR.write PHI1A
PC.write PHI1A
RF.write PHI1A
state.IO INT
state.int INT
state.int3 INT
state.rb RB
state.repair1 NS
state.repair2 NS
state.reset EXT
state.shutdown EXT
state.wait EXT

Outputs (8) Output to
ctrl.CU_write CUDATA
ctrl.IMM_write IMM
ctrl.IR_write IR
ctrl.MAR_write MAR
ctrl.PC_write PC
ctrl.RB_write RB
ctrl.RF_write RF
ctrl.SDR_write SDR
B. TEST PROGRAM LISTINGS

test1.ras

# test1.ras
# Used to test add, sub, unconditional jumps.
#
     .imreg  r5

.start:  add     $3,r0,r1    # test load immediate into register
        add     $7,r0,r2
        add     r1,r2,r3    # test add register to register
        jmp     alw,foo     # test unconditional relative jump
        add     r1,r2,r2

# # Use the following to test unconditional absolute jump:
# #
# #start: add     $3,r0,r1
# #        add     $7,r0,r2
# #foo:    add     r1,r2,r3
# #        add     $foo,r0,r4
# #        jmp     alw,0(r4)    #<-- this tests absolute jump
# #        add     r1,r2,r2
# #
# # Other tests:
# # - add immediate to register other than r0
# # - absolute jump with non-zero offset (i.e. jmp alw,$const,(r4))
# # - substitute sub for add
test2.ras
#
# test2.ras
# Used to test setting of condition codes and conditional jumps
#
  .imreg r5

.start:  add  r0,0x01e40,r1    # Enable interrupts
         putpsw 0(r1)
         add  $0xffff0fff,r0,r1  # substitute constants with appropriate
         add  $0x0000f002,r0,r2  # values to set desired codes
         add  r2,r1,r3,[c]
         jmp r uge,foo
         add  r0,$0,r0  # no-op for delayed branch
         sub  r0,$1,r4
         jmp r alw,end
         add  r0,$0,r0
        foo:  add  $1,r0,r4
       end:   jmp r alw,end  # when done, r4 == 1 if condition is
         add  r0,$0,r0  # true, r4 == -1 if not
#
# Some convenient values:
# - 0xffff0fff + 0x0000f002 sets C
# - 0xffffffff + 1 sets N and V
# - 0 sets Z (duh...)
# - 1 - 1 sets N and C
#
# Note: brain-damaged assembler doesn't like c and nc conditions (must use
#       uge and ult respectively)
test3.ras

# test3.ras
# used to test addc, subc, and, or, xor
#
    .imreg  r5

    .text

    start:       add    r0,50x1e40,r1          # Enable interrupts
               putpsw  0(r1)
               add    r0,50xffff0fff,r1
               add    r1,50x0000f003,r2, {c}  # this sets C; use 0xf000 to not set it
               clrbbm
               clrrbs
               addc   r1,r2,r3
               subc   r1,r2,r4, {c}
               and    r1,r2,r6
               or     r1,r2,r7
               xor    r1,r2,r8

               .space 4

    shutdown:
               getpsw  r1
               putpsw  0(r0)
               add    r0,r0,r0
               add    r0,r0,r0
               add    r0,r0,r0

    # Expected values are:
    #
    # C-bit set:
    #  addc:  r3 == 0x0ffff1002
    #  subc:  r4 == 0x0fffff0fdd
    #  and:   r6 == 0x100000002
    #  or:    r7 == 0x1ffff0fff
    #  xor:   r8 == 0x0ffff0fdd
    #
    # C-bit not set:
    #  addc:  r3 == 0x1ffff0ffe
    #  subc:  r4 == 0x1ffff0fff
test4.ras

# test4.ras
# Tests shift instructions (sll, sra, srl)
#
# .imreg  r5
#
.text
start:  add     r0,$0x1e40,r1    # Enable interrupts
        putpsw   0(r1)
        add     r0,$0x00000000f,r1   # Use 0xf0000000 to test right shifts.
        sll     r1,$1,r2,{c}       # Shift 1
        sll     r1,$2,r3,{c}       # Shift 2
        sll     r1,$8,r4,{c}       # Shift 8
        sll     r1,$16,r5,{c}      # Shift 16
        sll     r1,$24,r6,{c}      # Shift 24
        add     r0,r0,r0           # Empty out DWB
        add     r0,r0,r0
        add     r0,r0,r0
        add     r0,r0,r0
        add     r0,r0,r0
#
# Use the following to test shifting by a register:
#
# start: add     r0,$0x1e40,r1    # Enable interrupts
#        putpsw   0(r1)
#        add     r0,$0x00000000f,r1   # Use 0xf0000000 to test right shifts.
#        add     r0,r1,r2
#        add     r0,r2,r3
#        add     r0,r8,r4
#        add     r0,r16,r5
#        add     r0,r24,r6
#        sll     r1,r2,r2,{c}       # Shift 1
#        sll     r1,r3,r3,{c}       # Shift 2
#        sll     r1,r4,r4,{c}       # Shift 8
#        sll     r1,r5,r5,{c}       # Shift 16
#        sll     r1,r6,r6,{c}       # Shift 24
#        add     r0,r0,r0           # Empty out DWB
#        add     r0,r0,r0
#        add     r0,r0,r0
#        add     r0,r0,r0
#        add     r0,r0,r0
#
# Other tests:
# Replace sll with srl and sra
# Shift by bad value (should cause trap)
test5.ras

# test5.ras
# Tests getlpc, getpsw, putpsw

.imreg r6

.start:
  add r0,$0x1e40,r1          # Enable interrupts
  putpsw 0(r1)
  add r0,$-1,r4              # dummy statement to waste time
  getlpc r3,(c)              # r3 should get 0x0000000c
  add r0,$0,r4,(c)           # sets Z bit
  getpsw r3,(c)              # r3 should get 0x00000648, all CC's cleared
  and r3,$0xfffffffff7,r4    # clear Z bit and change window pointer
  or r4,$0x42,r5              # set I and V bits
  putpsw r0(r5)
  #add r0,$1,r16              # r16, window 3 <-- 1 (NO LONGER VALID)
  add r0,$1,r16              # r16, window 2 <-- 1

# Other tests:
# - putpsw offset by register other than r0
# - putpsw offset by immediate
# - getlpc/putpsw with S-bit disabled should cause privileged opcode trap;
#   putpsw the following line to set the S-bit:
#   add r0,$0x1e70,r1          # Enable interrupts, set S-bit
test6.ras

# test6.ras
# Tests subroutine calls.

        .imreg r5

        .text

main:  add    r0,$0x1e40,r1   # Enable interrupts
       putpsw  0(r1)
       add    r0,$-1,r26      # r26, window 3 <- -1
       add    r0,$325,r11     # r11, window 3 <- 325 (0x145)
       add    r0,$-7,r1       # r1 <- -7
       callr  foo,r26        # r26, window 2 (r10, window 3) <- 0x00000018
       add    r0,$1,r11       # r11, window 2 <- -1
       add    r0,r26,r2       # r2 <- r26, window 3 (should be -1)
       add    r0,r1,r3        # r3 <- r1 (r11, window 2; should be 1)
       add    r0,r15,r4       # r4 <- r15, window 3 (r31, window 2; should
                              #    be -99)
       .space  4

foo:   add    r11,r0,r1      # r1 <- r11, window 2 (should be 1)
       add    r0,$21,r11      # r11, window 2 <- 21
       add    r0,$-99,r31,[c] # r31, window 2 <- -99
       ret    alw,8(r26)      # return to main
       add    r0,r11,r2       # r2 <- r11, window 3 (should be 325)

# # Use the following to test absolute calls:
# #
# #main:  add    r0,$0x1e40,r1   # Enable interrupts
# putpsw  0(r1)
#       add    r0,$-1,r26      # r26, window 3 <- -1
#       add    r0,$325,r11     # r11, window 3 <- 325 (0x145)
#       add    r0,$-7,r1       # r1 <- -7
#       add    r0,$100,r7     # call  0(r7),r26      # r26, window 2 (r10, window 3) <- 0x00000020
#       add    r0,$1,r11      # r11, window 2 <- -1
#       add    r0,r26,r2       # r2 <- r26, window 3 (should be -1)
#       add    r0,r1,r3        # r3 <- r1 (r11, window 2; should be 1)
#       add    r0,r15,r4       # r4 <- r15, window 3 (r31, window 2; should
#                              #    be -99)
#       .space  4
# #
# #foo:   add    r11,r0,r1      # r1 <- r11, window 2 (should be 1)
#       add    r0,$21,r11      # r11, window 2 <- 21
#       add    r0,$-99,r31,[c] # r31, window 2 <- -99
#       ret    alw,8(r26)      # return to main
#       add    r0,r11,r2       # r2 <- r11, window 3 (should be 325)
# # Other tests:
# # Absolute call with non-zero offset (call const(r7),r26)
# # Call to bad address
# # Return to bad address
test7.ras

# test7.ras
# Tests store instructions.

.imreg r5

.text

start:  add  r0,$0x1e40,r1       # Enable interrupts
        putpsw  0(r1)
        add  r0,$325,r1
        add  r0,$101,r3
        str  r1,foo,(c)     # foo <-- 325
        #strb  r0,foo,(c)    # for store byte test
        add  r1,$1,r4
        str  r3,foo+4       # foo+4 <-- 101
        #strb  r0,foo+5      # for store byte test
        #strb  r0,foo+10     # "  "  "  "
        #strb  r0,foo+15     # "  "  "  "

# Use the following for absolute stores:
#
# start:  add  r0,$0x1e40,r1       # Enable interrupts
#        putpsw  0(r1)
#        add  r0,$325,r1
#        add  r0,$foo,r2
#        add  r0,$101,r3
#        st  r1,0(r2),{c}   # foo <-- 325
#        #stb  r0,0(r2),{c}  # for store byte test
#        add  r0,$1,r4
#        st  r3,4(r2)       # foo+4 <-- 101
#        #stb  r0,5(r2)      # for store byte test
#        #stb  r0,10(r2)     # "  "  "  "
#        #stb  r0,15(r2)     # "  "  "  "
#
.space 4

foo:    .word 0xffffffff
        .word 0xffffffff
        .word 0xffffffff
        .word 0xffffffff

# Other tests:
# Store halfwords (stb) to foo and foo+6.
# Store bytes (stb) to foo, foo+5, foo+10, and foo+15 (zero is a good value
# to store).
# Store to bad location (cause address misalignment trap).
test8.ras

# test8.ras
# Tests load instructions.

.imreg r5

.text

start: add r0, $0x1e40, r1       # Enable interrupts
putpsw 0(r1)

ldr foo, r1, {c}     # r1 <= 0x81828384
ldr foo+4, r2, {c}  # r2 <= 0x01020304
ldrh foo, r3, {c}     # r3 <= 0xfffff8384
ldrh foo+6, r4, {c}  # r4 <= 0x00000102
ldrhu foo, r5, {c}     # r5 <= 0x00008384
ldrhu foo+2, r6, {c}  # r6 <= 0x00008182
ldrb foo, r7, {c}     # r7 <= 0xfffff84
ldrb foo+5, r8, {c}  # r8 <= 0x00000003
ldrb foo+2, r9, {c}  # r9 <= 0xfffff82
ldrb foo+7, r10, {c}  # r10 <= 0x00000001 (RFlocal[42])
ldrbu foo, r11, {c}     # r11 <= 0x000000084 (RFlocal[43])
ldrbu foo+1, r12, {c}  # r12 <= 0x00000083 (RFlocal[44])
ldrbu foo+2, r13, {c}  # r13 <= 0x00000082 (RFlocal[45])
ldrbu foo+3, r14, {c}  # r14 <= 0x00000081 (RFlocal[46])
add r0, r0, r0  # clear DWB
add r0, r0, r0
add r0, r0, r0
add r0, r0, r0

# Use the following for absolute loads:
#
#start: add r0, $0x1e40, r1       # Enable interrupts
# putpsw 0(r1)
# add r0, 0/foo, r31
# ld 0(r31), r1, {c}     # r1 <= 0x81828384
# ld 4(r31), r2, {c}     # r2 <= 0x01020304
# ldh 0(r31), r3, {c}     # r3 <= 0xfffff8384
# ldh 6(r31), r4, {c}     # r4 <= 0x00000102
# ldhu 0(r31), r5, {c}     # r5 <= 0x00008384
# ldhu 2(r31), r6, {c}     # r6 <= 0x00008182
# ldb 0(r31), r7, {c}     # r7 <= 0xfffff84
# ldb 5(r31), r8, {c}     # r8 <= 0x00000003
# ldb 2(r31), r9, {c}     # r9 <= 0xfffff82
# ldb 7(r31), r10, {c}     # r10 <= 0x00000001 (RFglobals[42])
# ldbus 0(r31), r11, {c}     # r11 <= 0x000000084 (RFglobals[43])
# ldbus 1(r31), r12, {c}     # r12 <= 0x00000083 (RFglobals[44])
# ldbus 2(r31), r13, {c}     # r13 <= 0x00000082 (RFglobals[45])
# ldbus 3(r31), r14, {c}     # r14 <= 0x00000081 (RFglobals[46])
# add r0, r0, r0  # clear DWB
# add r0, r0, r0
# add r0, r0, r0
# add r0, r0, r0
#

.space 4

foo: .word 0x84838281       # .word 0x81828384 in reality
    .word 0x04030201       # .word 0x01020304 in reality
#
# Other tests:
#  Load from bad location (cause address misalignment trap).
test9.ras

# test9.ras
# Provides interrupt handlers (of a sort) for testing interrupts and traps.

.text

.reset:
  jmp   alw, reset_handler
  add   r0, $0x670, r1  # CWP=11, SWP=00, I=1, S=1, P=1, CC=0000

.space 8

.int:
  getlpc r24  # save NXTPC (calli has already saved PC)
  jmp   alw, int_handler
  add   r0, $1, r23

.space 4

.over:
  getlpc r24
  jmp   alw, over_handler
  getpsw r23

.space 4

.under:
  getlpc r24
  jmp   alw, under_handler
  add   r0, $3, r23

.space 4

.reset_handler:
  putpsw 0(r1)
  jmp   alw, start
  add   r0, r0, r0

.space 4

.int_handler:
  jmp   alw, 0(r25)  # restore PC
  reti  alw, 0(r24)  # restore NXTPC
  add   r0, r0, r0

.space 4

.over_handler:
  str   r1, r1save
  callr over1, r1  # decrement CWP
  str   r10, foo  # save window to foo

.over1:
  str   r11, foo+4
  ret   alw, 20(r1)  # increment CWP
  sll   r23, $1, r22  # decrement SWP
  srl   r22, $8, r22
  sub   r22, $1, r22
  sll   r22, $8, r22
  srl   r22, $1, r22
  and   r22, $0x180, r22
  and   r23, $0xfffffe7f, r21
  or    r22, r21, r21
  putpsw 0(r21)
  ldr   r1save, r1
  jmp   alw, 0(r25)
  reti  alw, 0(r24)
  add   r0, r0, r0
.space 4

under_handler:
    jmp   alw,0(r25)
    reti  alw,0(r24)
    add   r0,r0,r0

.space 4

start:  add   r0,$1,r1
        add   r0,$325,r10
        str   r10,rlsave
        add   r0,$101,r11

loop:  callr  loop,r2  # will cause reg file overflow after 2 loops
        add   r0,r0,r0

# Use the following to cause reg file underflow
#
#start:  add   r0,$start,r1
#        ret   alw,0(r1)
#
# Use the following for interrupting subroutines:
#
#start:  add   r0,$1,r1
#        callr  subl,r27  # r27, window 2 <-- 0x000000c0
#        add   r0,$325,r10  # r10, window 2 <-- 325
#        add   r0,$1,r11  # r11, window 3 <-- 1
#
# .space 4

#subl:  add   r0,$-1,r10  # r10, window 2 <-- -1
#        ret   alw,8(r27)
#        add   r0,$-2,r11  # r11, window 3 <-- -2
#
# .space 4

rlsave: .word 0xffffffff
foo: .word 0xffffffff
    .word 0xffffffff

# Other tests:
#   Cause external interrupt (including both cycles of load and store
#   instructions)
#   - both cycles of load and store instructions
#   - call instruction
#   - instruction after call instruction
#   - first instruction of subroutine
#   - instruction after return
#   - instruction after returning from subroutine
# test10.ras

# Tests test instructions and rollback logic

.rimreg r5

.text

start:    add  r0, $0x1e40, r1  # Enable interrupts
          jmp r alw, start0
          putpsw 0(r1)

foo:      .word 0

# Register file parity error, master

start0:   add  $1, r0, r1    # r1 < 1
          add  $16, r0, r4   # r4 < 16
          addbpm  $2, r0, r2 # r2 < 2, bad parity on master
          clrrbm
          clrrbs
          add  r1, r2, r3    # r3 < 3, cause parity error on master
          jmprbm r4, r4, r0  # if rollback occurs, jump to cont0
          add  r0, r0, r0    # (12)
error0:   jmp  alw, error0   # 0x00000034
          add  r0, r0, r0

# Verify clearing of rollback bits

cont0:    clrrbm            # clear rollback bits
          clrrbs
          jmprbm r4, r4, r0  # if still set, jump to error1
          add  r0, r0, r0
          jmpr alw, cont1
          add  r0, r0, r0    # (6)
error1:   jmp  alw, error1  # 0x00000054
          add  r0, r0, r0

# Register file parity error, slave

cont1:    add  $2, r0, r1    # r1 < 2
          addbpm  $2, r0, r2 # r2 < 2, bad parity on slave
          clrrbm
          clrrbs
          add  r1, r2, r3    # r3 < 4, cause parity error on slave
          jmprbs r4, r4, r0  # if rollback occurs, jump to cont2
          add  r0, r0, r0    # (11)
error2:   jmp  alw, error2  # 0x00000078
          add  r0, r0, r0

# State compression comparison error

cont2:    add  r0, r0, r0    # reset rollback counter
          add  r0, r0, r0
          add  r0, r0, r0
          add  r0, r0, r0
          add  r0, r0, r0
          add  r0, r0, r0
          add  r0, r0, r0
          add  r0, r0, r0
          add  $3, r0, r1    # r1 < 3
          add  $16, r0, r2   # r2 < 16
          add  $7, r0, r3    # r3 < 7
          add  $16, r0, r4   # r4 < 16
clrrbm
clrrbs
jmrprbm r1,r2,r6  # r6 <= 3 on master, r6 <= 16 on slave
   add r0,r0,r0  # (20)
error3: jmpr alw,error3  # 0x000000C4
   add r0,r0,r0

cont3: clrrbm
clrrbs
jmrprbs r3,r4,r7  # r7 <= 16 on master, r7 <= 7 on slave
   add r0,r0,r0  # (7)
error4: jmpr alw,error4  # 0x000000DC
   add r0,r0,r0

# Memory access, address comparison error
cont4: add r0,r0,r0  # clear rollback counter
   add r0,r0,r0
   add r0,r0,r0
   add r0,r0,r0
   add r0,r0,r0
   add r0,r0,r0
   add r0,r0,r0
   add r0,r0,r0
   add r0,r0,r0
   add $20,r0,r2  # r2 <= 20
clrrbm
jmrprbm r2,r2,r0  # slave takes branch, master doesn’t
   add r0,r0,r0  # (17)
error5: jmpr alw,error5  # 0x00000120
   add r0,r0,r0

cont5: clrrbs
jmrprbs r4,r4,r0  # master takes branch, slave doesn’t
   add r0,r0,r0  # (6)
error6: jmpr alw,error6  # 0x00000134
   add r0,r0,r0
   add r0,r0,r0
cont6: jmpr alw,cont6  # 0x00000140
   add r0,r0,r0

# Other tests:
# - choose values for jmprb that have same state compression (e.g., 1 & 16)
# - test instructions with S-bit disabled should cause privileged opcode trap;
#   putpsw the following line to set the S-bit:
# add r0,$0x1e70,r1  # Enable interrupts, set S-bit
test11.ras

# test11.ras
# Tests test instructions and rollback logic (continuation of test10.ras)

.reg r5

.text

start:  add r0,$0x1e40,r1  # Enable interrupts
        jmp r0,start1
        putpsw 0(r1)

foo:   .word 0xffffffff

# Memory access, data comparison error

start1:  add r16,r0,r4
        clr r16
        clrrb
        clrrb

        strbdm r4,foo
        add r0,r0,r0
        jmp rbm r4,r4,r0
        add r0,r0,r0

        error7: jmp alw,error7
                add r0,r0,r0

        cont7: clr r16
        clrrb
        clrrb

        strbds r4,foo
        add r0,r0,r0
        jmp rbs r4,r4,r0
        add r0,r0,r0

        error8: jmp alw,error8
                add r0,r0,r0

# bus parity error

cont8:  add r0,r0,r0  # reset rollback counter

        add r0,r0,r0
        add r0,r0,r0
        add r0,r0,r0
        add r0,r0,r0
        add r0,r0,r0
        add r0,r0,r0
        clr r16
        clrrb
        clrrb

        ldrbpm foo,r1  # r1 <- 16, pad parity on master
        add r0,r0,r0
        jmp rbm r4,r4,r0
        add r0,r0,r0

        error9: jmp alw,error9
                add r0,r0,r0

        cont9: clr r16
        clrrb
        clrrb

        ldrbpm foo,r1  # r1 <- 16, pad parity on slave
        add r0,r0,r0
        jmp rbs r4,r4,r0
        add r0,r0,r0

        err10: jmp alw,err10  # 0x000000B0
                add r0,r0,r0

        cont10: jmp alw,cont10  # 0x000000B8
                add r0,r0,r0
# Other tests:
#   - test instructions with S-bit disabled should cause privileged opcode
#     trap;
#     putpsw the following line to set the S-bit:
#     add   r0,$0x1e70,r1    # Enable interrupts, set S-bit
C. ERROR DETECTION TESTS

Error on busA: roll back 1, repair state
Error on busB: roll back 1, repair state
Error on busIN: roll back 2
Error in CMP (comparator): roll back 2
   pads
   busOUT
   exported state

Repeat above during load/store (relative and absolute)
   Error on busIR: roll back 2
   Error on address: roll back 1
   Error on outgoing data (store): roll back 1
   Error on incoming data (load): roll back 1

Multiple errors (two)
   Bad value in same register on both chips: retry rollback/repair
   Unrelated error in same chip
      both in registers: roll back 1, repair busA, re-discover other fault, roll back 1, repair busB
      one in registers, one in pad/state export/busIN: roll back 2, repair wrong busA/busB, re-execute
         instruction with bad register, roll back 1, repair state
   Unrelated error in other chip: same as in same chip

Error occurring during state repair
   Good value gets bit flip while being transferred: roll back 1 to cancel state repair, re-execute, roll back
      1, redo repair
   Unrelated error in same chip
      both in registers: repair one at a time (busA first)
      one in registers, one in pad/state export/busIN: shut down
   Unrelated error in other chip: shut down

External rollback right before parity error rollback/state repair

Error occurring during rollback
   Error in rollback memory: roll back "one more" (shut down if original rollback amount > 1)
      value
      IR
      MAR
      PSW
      PC (should fix itself w/out causing rollback)
      CU
      SDR
      valid bit
      as above
   Unrelated error: parity errors are ignored; comparison errors other than busD, PSW, CU: roll back 2 +
      amount of previous rollback + 2 (i.e., shut down)

Make sure state repair (and the rollback that comes before it) takes precedence over interrupt
Make sure shutdown take precedence over everything except reset
External rollback at same time as internal rollback: rollback greater amount
D. VERIFICATION INSTRUCTIONS

This appendix shows the mapping of the four verification instructions referred to in Section 6.2.1 and [2]. There are two different sets of instructions, one for the global registers and one for the local registers.

\[ R_{1}-R_{9} : \]
\[
\text{MOV a,R}_i : \\
\text{ldr a,R}_i \\
\text{add R}_0,R_0,R_0 \quad \text{! NOPs to clear DBW} \\
\text{add R}_0,R_0,R_0 \\
\text{add R}_0,R_0,R_0 \\
\text{add R}_0,R_0,R_0 \\
\text{CMP R}_i,R_j : \\
\text{sub R}_i,R_j,R_0,c \\
\text{BEQ a :} \\
\text{jmp r eq,a} \\
\text{add R}_0,R_0,R_0 \quad \text{! delay slot} \\
\text{READ(R}_i) : \\
\text{str R}_i,a \\
\]

\[ R_{10}-R_{73} : \]
\[
\text{MOV a,R}_i : \\
\text{str R}_1,b \quad \text{! save R}_1 \\
\text{getpsw R}_1 \quad \text{! save PSW} \\
\text{putpsw \it{window}} \quad \text{! set proper window} \\
\text{ldr a,R}_i \quad \text{! R}_i \text{ adjusted for proper window} \\
\text{putpsw \it{R}_1} \quad \text{! restore PSW} \\
\text{ldr b,R}_1 \quad \text{! restore R}_1 \\
\text{add R}_0,R_0,R_0 \quad \text{! NOPs to clear DBW} \\
\text{add R}_0,R_0,R_0 \\
\text{add R}_0,R_0,R_0 \\
\text{add R}_0,R_0,R_0 \\
\text{CMP R}_i,R_j : \\
\text{str R}_1,c \quad \text{! save R}_1 \\
\text{getpsw R}_1 \quad \text{! save PSW} \\
\text{str R}_2,d \quad \text{! save R}_2 \\
\text{str R}_3,e \quad \text{! save R}_3 \\
\text{putpsw \it{window}} \text{~}i \quad \text{! set window for R}_i \\
\text{add R}_i,R_0,R_2 \quad \text{! move R}_i \text{ to R}_2 \\
\text{putpsw \it{window}} \text{~}j \quad \text{! set window for R}_j \\
\text{add R}_j,R_0,R_3 \quad \text{! move R}_j \text{ to R}_3 \\
\text{putpsw R}_1 \quad \text{! restore PSW} \\
\text{sub R}_2,R_3,R_0,c \quad \text{! compare R}_2,R_3 \]
ldr    c,R1    ! restore R1
ldr    d,R2    ! restore R2
ldr    e,R2    ! restore R3
add    R0,R0,R0    ! NOPs to clear DWB
add    R0,R0,R0
add    R0,R0,R0

BEQ a:
    jmpr    eq,a
    add    R0,R0,R0    ! delay slot

READ(Ri):
    str    R1,b    ! save R1
    getpsw    R1    ! save PSW
    putpsw    window    ! set proper window
    str    R1,a    ! read register
    putpsw    R1    ! restore PSW
    ldr    b,R1    ! restore R1
    add    R0,R0,R0    ! NOPs to clear DWB
    add    R0,R0,R0
    add    R0,R0,R0
    add    R0,R0,R0
REFERENCES


4. M. G. H. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI," Report No. UCB/CSD 83/141, Computer Science Division (EECS), University of California, Berkeley (October 1983).


