Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

THE UCLA MIRROR PROCESSOR: A BUILDING BLOCK FOR SELF-CHECKING SELF-REPAIRING COMPUTING NODES

Yuval Tamir
Mike Liang
Titus Lai
Marc Tremblay

November 1990
CSD-900040

The UCLA Mirror Processor: A Building Block for Self-Checking Self-Repairing Computing Nodes †

Yuval Tamir, Mike Liang, Titus Lai, and Marc Tremblay
Computer Science Department
4731 Boelter Hall
University of California
Los Angeles, California 90024-1596
U.S.A.
Phone: (213)825-4033 E-mail: tamir@cs.ucla.edu
FAX: (213)825-2273

Abstract

Fault-tolerant systems often rely on self-checking computing nodes. System performance and reliability is increased if the nodes can recover locally from most errors caused by transient faults without requiring system-level recovery. Using a technique called *micro rollback*, it is possible to eliminate most of the performance penalty of concurrent error detection.

We report on the design and implementation of a VLSI RISC microprocessor, called the *UCLA Mirror Processor*, which is capable of micro rollback. In order to achieve concurrent error detection, two Mirror Processor chips operate in lock-step, comparing external signals and a signature of internal signals every clock cycle. If a mismatch is detected, both processors roll back to the beginning of the cycle when the error occurred. In some cases erroneous state is corrected by copying a value from the fault-free processor to the faulty processor. We describe the architecture, microarchitecture, and VLSI implementation of the Mirror Processor, emphasizing its error-detection, error-recovery, and self-diagnosis capabilities.

Index Terms: Micro rollback, self-checking modules, self-repair, VLSI RISC processor.

---

† This research is supported by the SDIO Innovative Science and Technology Office, managed by the Office of Naval Research, contracted to the Jet Propulsion Laboratory under task plan #80-2984; and by Hughes Aircraft Company and the State of California MICRO program.

I. Introduction

The design of computer systems often involves tradeoffs between average performance, real-time performance, and reliability. In order to provide a high probability of meeting real-time constraints, the system is under-utilized most of the time, leading to reduced average performance. In order to meet reliability requirements, fault tolerance is often necessary. In fault-tolerant systems, there are many tradeoffs between real-time and average performance. For example, if error recovery is based on checkpointing and rollback, increasing the frequency of checkpointing typically reduces average performance while allowing the system to meet tighter real-time constraints since less work is lost when recovery is necessary. The goal of our research is to design fault tolerant systems that achieve high average performance as well as a high probability of meeting real-time constraints.

Fault-tolerant systems which include multiple processors often rely on self-checking computing nodes [11, 15]. These nodes detect errors as soon as they occur, thus preventing the spread of erroneous information throughout the system. Such error confinement minimizes the scope of recovery actions. This reduces system unavailability following an error, while recovery is in progress. Since transient faults are more likely to occur than permanent faults [1], system average performance and ability to meet real-time constraints can be increased if the computing nodes recover locally from most errors caused by transient faults without requiring system-level recovery.

In order to implement concurrent error-detection for the self-checking nodes, checkers are usually connected in the communication paths between modules. The checkers reduce performance by requiring either longer clock cycles or additional pipeline stages. As we have previously described [16, 17], this performance overhead can be minimized if the checks are performed in parallel with intermodule communication. Each module processes its inputs immediately when they become available. If the data is erroneous, it is followed, after a delay of a few cycles, by an error indication. If the data processed by the receiver is later flagged as erroneous, any changes to the state of the system due to this information must be undone. Hence, it is necessary to back up processing to the state that existed just before the error first occurred. We call the process of backing up a system several cycles in response to a delayed error signal *micro rollback* [17].

We report on the design and implementation of a VLSI RISC microprocessor, called the *UCLA Mirror Processor* (henceforth *MP*), which can serve as a building block for self-checking self-repairing computing nodes. The Mirror Processor implements the instruction set of the Berkeley RISC II chip [9, 12]. The basic error-detection mechanism of the *MP* is duplication and comparison [4]. Two *MP* chips, a *master* and a *slave*, operate in lock-step, with the slave comparing their results every clock cycle [3, 6]. Local recovery from an error caused by transient faults is accomplished by re-executing the instruction that resulted in the error [2], and, when necessary, repairing the corrupted state in one *MP* chip by copying values from the other *MP* chip. Each *MP* is capable of micro rollback of up to four cycles. Hence, the latency for error detection does not require slowing down normal operation.

The *MP* chip has been laid out using MOSIS scalable CMOS design rules. The chip contains 52,644 transistors and, with 2 μm technology, the size of the chip is approximately 8.4 mm by 6.7 mm.

The operation of the chip has been checked using simulations at the circuit level, switch level, and register-transfer level. Based on these simulations, the chip will operate at a peak execution rate of 10 MIPS.

The basic techniques used in the *MP* have been presented in previous papers. The main contribution of this paper is in presenting how these techniques can be incorporated in a real implementation of a complete microprocessor. The Mirror Processor uses a combination of techniques that, together, result in a practical building block for high-performance fault-tolerant systems. This paper describes the architecture and microarchitecture of the Mirror Processor, emphasizing its error-detection, error-recovery, and self-diagnosis capabilities. We demonstrate the benefits of micro rollback in a real system, show that it can be implemented in a practical VLSI chip, and evaluate the overhead and design issues encountered.

Section II is a brief description of micro rollback and related terminology. The basic microarchitecture and timing of the Mirror Processor are presented in Section III. The datapath, with its special features for micro rollback, error detection, and state repair, are described in Section IV. We have found that much of the added design complexity of the *MP*, relative to a conventional RISC mircroprocessor, was due to the control unit. Section V describes the *MP* control unit. With most self-checking modules, there are difficulties in verifying that the self-checking features are operational once the module is integrated with the rest of the system. Section VI presents special instructions that were added to the *MP* in order to facilitate self-diagnosis. The VLSI implementation of the *MP* is described in Section VII. This includes the floorplan of the chip, the overhead for detection, repair, and micro rollback, as well as a brief description of the design process.

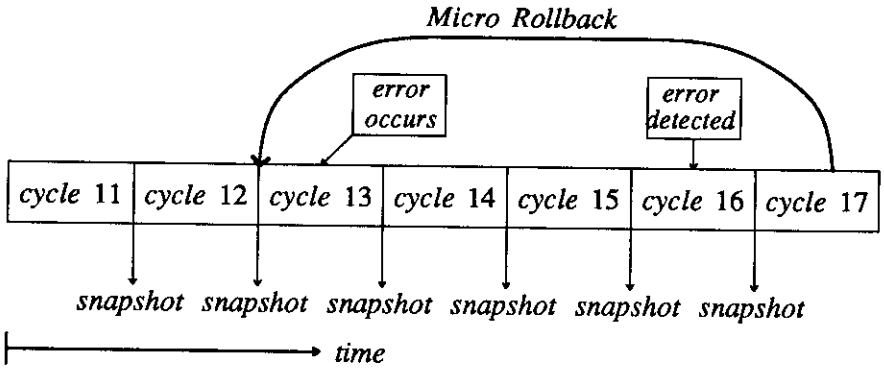

Figure 1: Micro rollback of a module — restoring a saved *snapshot.*

II. Micro Rollback

Micro rollback is based on the idea that a valid system state can be restored by rolling back to a previously saved checkpoint [10]. A micro rollback of a module (subsystem) consists of bringing the module back a few cycles to a *state* that it had reached in the past [17]. In order to be able to perform such an operation, it is necessary to save a "snapshot" of the state of the subsystem (*checkpoint*) at each cycle boundary [5]. Micro rollback restores the state of a subsystem by overwriting the current state with

a "snapshot" taken in the past (Figure 1).

Unlike traditional checkpointing and rollback [10], with micro rollback both checkpointing and rollback are performed entirely in hardware. This allows checkpointing to be performed in parallel with normal operation while recovery is performed in a few clock cycles. Such rapid checkpointing and rollback is essential in systems with real-time constraints, where long delays for recovery are intolerable. In [17] we presented a comparison between micro rollback and traditional instruction retry [2] as well as between micro rollback and schemes used for precise interrupts [14, 5].

The *state* of a module (subsystem) is the contents of all storage elements which carry useful information across cycle boundaries. When a rollback occurs, the number of cycles to be undone must be provided to a rollback controller. This number is called the *rollback distance*. One of the design parameters of a system with support for micro rollback is the *rollback range* — the maximum rollback distance that modules in the system must support. The rollback range is determined by the number of stored snapshots.

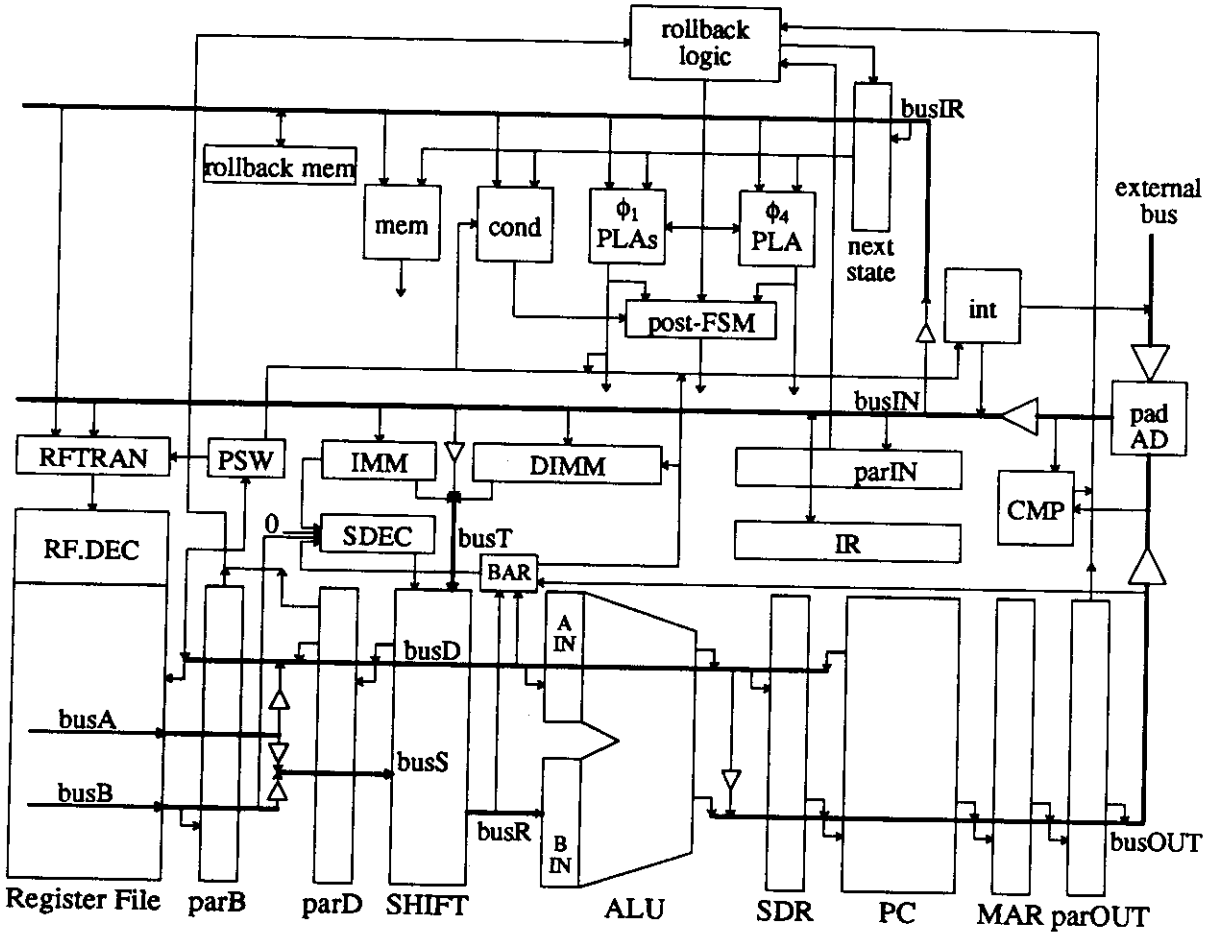

Figure 2: The UCLA Mirror Processor. All the modules below busIN are part of the datapath. The control unit is shown above busIN.

III. The Architecture of the UCLA Mirror Processor

The choice of the basic processor architecture for the *MP* was guided by several considerations: (1) since we were not interested in instruction set design, it was preferable to use an existing instruction set; (2) in order to have a chance of successful implementation in an academic environment, the basic processor architecture had to be simple; (3) in order to be a convincing "proof of concept," the performance of the processor could not be orders of magnitude below the performance of contemporary microprocessors. Based on these considerations, we chose to use the Berkeley RISC II processor [9, 12] as the basis for the *MP*. The RISC II was the basis of the popular Sun SPARC architecture and is similar to other "reduced instruction-set computers." It is a load/store architecture, where only explicit load and store instructions access memory. All ALU and shift operations obtain their operands from the register file and store their results back to the register file. One byte, two byte, and four byte integers are supported. The memory is accessed as a single, flat 4 GByte segment. The register file has multiple register banks to support fast procedure calls [9]. In the *MP*, the register file consists of four banks of sixteen 32-bit registers for procedure stack frames plus ten global general-purpose registers.

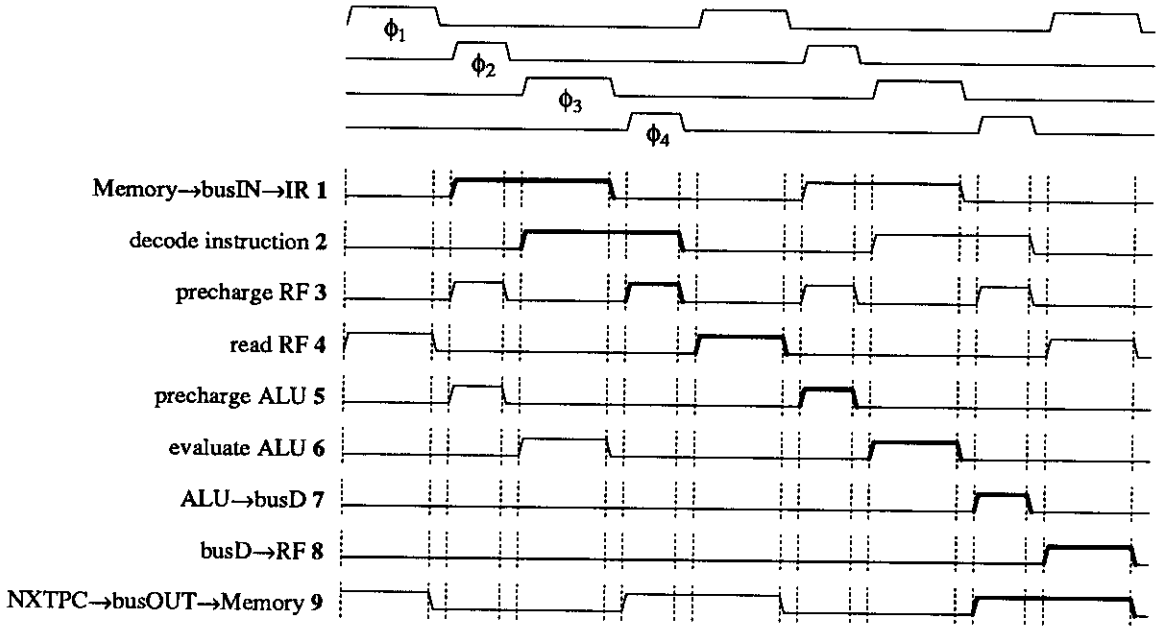

Figure 3: The timing of register-register ALU instructions. The instruction is fetched in one clock cycle and executed in the following clock cycle. The result is stored in the register file in phase 1 of the cycle following the execution cycle.

Figure 2 shows a block diagram of the *MP*. The datapath, shown below the busIN bus, is almost identical to the original RISC II datapath [12]. The differences are related to the fault tolerance features of the *MP* and will be described later. The memory interface is over a multiplexed data/address bus, which is used to reduce the number of pins and thus the cost of fabrication and testing. Register-register instructions are executed at a rate of one instruction per cycle with the bus used every cycle to fetch the next instruction. Due to the multiplexed external bus, two cycles are needed to execute load and store

instructions.

Pipelining is used in order to maintain an execution rate of one instruction per cycle. Four non-overlapping clock phases are used to sequence operations within each cycle. As shown in Figure 3, the multiplexed external bus is used to transfer addresses during ϕ_2 and ϕ_3, and transfer an instruction or data in ϕ_4 and ϕ_1. A typical ALU instruction requires reading two registers from the register file, performing an ALU operation, and writing the result back to the register file. The *delayed write buffer* [17], which is added for micro rollback (Section IV), allows the result to be written to the register file module in the same phase (ϕ_1) that the operands for the next instruction are read.

IV. Fault Tolerance Features in the Mirror Processor Datapath

Duplication and comparison is often used to implement concurrent error detection [4, 15, 6]. The Mirror Processor was designed for use in self-checking nodes where error detection is accomplished by error detecting codes (EDC) in the memory and duplication and comparison for the processor. Two *MP* chips are used in each self-checking node. In order to reduce the board chip count and complexity, a comparator for all the key output pins is implemented in the *MP* chip. Based on the value of a dedicated input pin, the chip operates as a *master* or a *slave* [3, 6]. The slave operates in lock-step with the master, performing the same operation at every clock phase. However, whenever the master produces an output (data or address), the slave, instead, reads the value from the bus and compares it to its own, internally generated value. A mismatch indicates an error.

With the scheme above, there is latency inherent in the error-detection process. Specifically, once the master generates some value, the value must be driven to the output pins, received by the slave, and compared with the slave's internally-generated values. If a mismatch is found, the error signal must be driven from the slave to the output pin and back to the master. Hence, an error can corrupt the processor state before it is detected. The *MP* uses micro rollback to undo such state changes and restore the processor to its state prior to the instruction when the error first occurred.

A. Support for Micro Rollback

Efficient implementation of micro rollback involves delaying commitment of state changes until the new values are known to be correct (checking is complete). For the *MP*, it is sufficient to delay commitment by four cycles since the latency of the error detection mechanisms used is less than four cycles. As described in [17], this is accomplished using *delayed write buffers* (DWBs), as shown in Figure 4. Updates are made by writing to the left stage of the DWB and setting the corresponding *valid* bit. The DWB is shifted right every cycle. After four cycles, the updated value is shifted into the "permanent storage" for the register (the right-most register). Updating of the permanent storage is performed only if the right-most valid bit is set. On a read, the *select* circuitry determines the most recent (the left-most) value for which the valid bit is set. A rollback of n cycles is accomplished in a single phase by clearing the first (left-most) n valid bits so that a subsequent read will obtain an older value.

In [17] we showed that the basic technique described above can be used for the register file. For the register file, the "permanent storage" consists of 74 registers, as described in Section III. In the DWB,

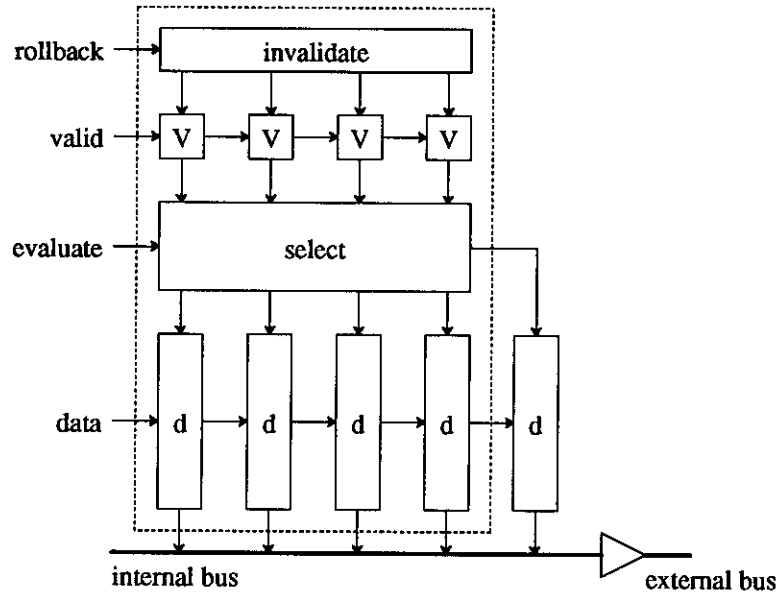

Figure 4: A single register with delayed commitment of updates, using a four stage delayed write buffer (DWB). Writes modify the left-most stage of the DWB. Recent updates are undone by clearing the valid bits. Reads obtain the most recent valid value.

the select circuit includes a *tag* with the register number corresponding to the value stored in the data part. On a write to the register file, the new value and register number are written into the DWB. A read from the register file involves comparing the register number to all the tags to determine if a recent value should be obtained from the DWB. Since the lookup in the DWB is performed in parallel with the permanent register file read, there is little additional delay [17].

As in the Berkeley RISC II processor, there are three registers used for storing the next, current, and last values of the program counter (PC) [7]. In the RISC II design these registers are organized as a small FIFO and the following transfers occur during each cycle:

```
new value → next_PC → PC → last_PC
```

Micro rollback of the PC unit could be supported by treating the three registers as individual state registers (Figure 4). However, this would result in high area overhead of 12 DWB stages. For the *MP* we developed a special, more efficient, mechanism for supporting micro rollback in the PC. As explained below, this mechanism takes advantage of the original FIFO organization of the PC unit.

The basic organization of the *MP* PC unit is shown in Figure 5. As with the single register (Figure 4), there are four valid bits, corresponding to the four DWB stages. A rollback of n cycles is accomplished by clearing the the n left-most valid bits. The key difference between the PC module and the module of a single register, is the selection circuitry. Depending on whether the read access is to the next_PC, PC, or last_PC, pass transistor logic, shown in Figure 5, is used to select the registers corresponding to the first, second, or third valid bits, respectively. Following a rollback, when several of the valid bits are cleared, this selection process may select one of the permanent registers (npc, pc, or lpc)

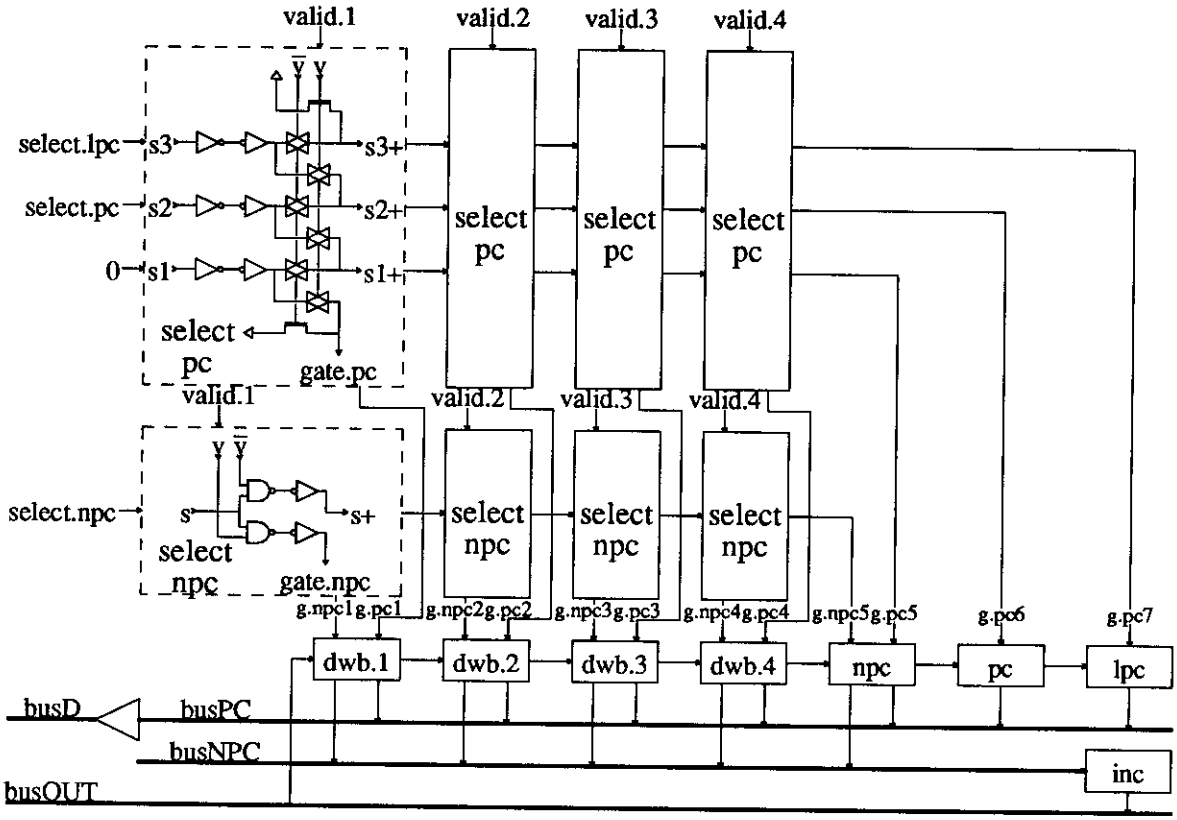

Figure 5: Support for micro rollback in the program counter. Three values are available at any instant: next_PC, PC, and last_PC. A four stage DWB is used to delay commitment of updates to a three stage FIFO queue of permanent storage registers.

instead of a value in the DWB. The next_PC must be read once per cycle since it contains the address to be used for a subsequent instruction fetch. For some instructions, the value of PC or last_PC is also needed (e.g. for PC-relative addressing). If only one value at a time had to be read from the entire PC module, the top part of the select circuitry would be sufficient. However, due to timing considerations, next_PC and PC may be read at the same time (during ϕ_1) so separate selection logic and a separate internal bus (busNPC) was required for next_PC.

Starting with a microarchitecture that does not include support for micro rollback, it is clear that DWBs are needed for registers that hold values across cycle boundaries [17]. Many VLSI implementations depend on the ability to store values for a few phases or cycles using the inherent parasitic capacitance of the circuits for dynamic storage. For example, once a value is asserted on an internal bus, it may be assumed that the value will remain stable for several cycles, even if the driving circuit is disconnected. Whenever such dynamic storage is used to store values across cycle boundaries, support for micro rollback requires the ability to restore the value when rollback is performed.

The problem of rolling back dynamic storage was encountered in the design of the *MP*. One case led to the addition of a *memory address register* (MAR), which was not part of the original microarchitecture. As shown in Figure 3, the instruction fetch involves driving an address to busOUT

and to the external bus during ϕ_4 of a cycle and ϕ_1 of the next cycle. This value is obtained by reading next_PC during ϕ_1 and incrementing it during ϕ_3 of the first cycle. When a rollback is performed to the cycle boundary (between ϕ_4 and ϕ_1), the instruction fetch that was in progress at the time must be re-executed. This requires the address of the instruction to be restored to the external bus. Furthermore, the processor must allow the memory the same amount of time to perform the restored instruction fetch as the time to perform a normal instruction fetch. In order to meet these requirements, a memory address register, with the structure shown in Figure 4, is connected to busOUT. The value on busOUT is written into the MAR during ϕ_1 of every cycle. During ϕ_3 of the rollback cycle, the appropriate number of MAR DWB stages are invalidated and the most recent remaining MAR value is read. This value is driven onto busOUT, thus correctly restarting the instruction fetch operation.

Two other registers with DWBs had to be added to the datapath of the *MP* to support micro rollback. When instructions are fetched, they arrive on the chip during ϕ_3 and are decoded during the latter part of ϕ_3 and the entire ϕ_4. A rollback to the cycle boundary requires restoring the instruction that was fetched during the cycle preceding the boundary and driving it onto busIN, as though it was arriving from the external bus. This task is performed by the *instruction register* (IR). The *store data register* (SDR) is used to restore the data of a store instruction if the rollback is to the boundary between the two cycles of the store. The *processor status word* (PSW) register is, of course, needed even without micro rollback. A DWB had to be connected to the PSW (as in Figure 4) to allow rolling back its values as well. It should be noted that the external memory (or cache) must also include a DWB [17] so that recent stores to the memory can be rolled back to maintain a consistent state with the processor.

B. Support for Error Detection and Error Recovery

As discussed earlier, the *MP* was designed as a building block of self-checking nodes where the primary error-detection mechanism is duplication and comparison. The basic support for this mode of operation is the ability of the chip to operate in *slave* mode, where it does not drive values onto the external bus. Instead, the slave chip compares the values on the external bus, generated by the *master*, to internally-generated values. In addition to the external bus lines, the comparison includes the memory control signals, a mode bit (system/user), and the interrupt acknowledge signal.

Effective detection of errors in the external memory (or cache) can be implemented using error-detecting codes at a lower cost than using duplication and comparison. Hence, the master and slave processors share (read from) the same memory (or cache). The *MP* uses single bit parity on the external bus and memory for error detection. The external bus is thus 33-bits wide, all addresses and data generated by the processor include a parity bit. A ''compressed'' tree of static XOR gates [20] connected to busOUT (parOUT), is used to generate the parity. Data and instructions read from the external memory must include a parity bit. The parity of these values is checked by a similar circuit (parIN).

The two techniques described above are sufficient if the only requirement is error detection. However, one of the goals of the *MP* is to be able to recover from all errors caused by a single transient fault in the processors. When an error is detected, both processors (master and slave) roll back to the

cycle boundary preceding the likely cause of the error. In the *MP*, both of the above errors require a rollback of two cycles. If the error is incorrect parity on the value read from the external bus, the rollback results in re-executing the memory read operation. If the error is a mismatch in the comparison, for example due to a transient fault in one of the ALUs, the two cycle rollback results in the re-execution of the ALU operation. Unfortunately, the above error detection schemes and micro rollback are not sufficient for recovering from *all* errors caused by single transient faults. One problem is caused by the potential for long latencies in detecting errors. For example, if the instruction executed is a register-register operation, a transient fault in one of the ALUs can result in an incorrect value stored in the register file. Many cycles later, a store of that register to memory will bring the value to the master/slave comparator and an error will be detected. At this point, a rollback will not restore a valid state and there is no simple way for the system to recover.

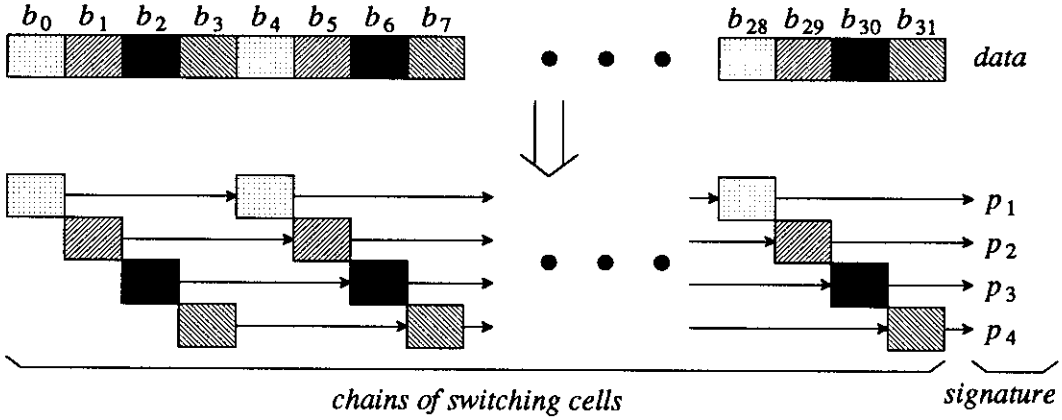

Figure 6: Generating a "signature" of a 32-bit word using four interleaved parity bits. The parity bits are computed using four interleaved chains of switching cells.

Error detection latency can be reduced by including in the master/slave comparison internal values that are normally not accessible from the pins. In particular, it can be useful to compare the ALU result as it is stored into the register file. In order to reduce the number of pins used to facilitate the comparison of internal values, a much smaller signature [8] of the internal values can be used. As shown in Figure 6, one possible way to generate a 4-bit signature is to use interleaved parity, where each signature bit is the parity of the set of bits consisting of every fourth bit of the original data [20]. A comparison of such signatures will detect all single bit errors, as well as adjacent bit errors, and numerous other multiple bit errors. Compact high-performance VLSI implementation of this signature generation circuitry is possible using chains of switching cells [13] to implement the multiple-input XOR for each signature bit [20]. In the *MP*, this basic technique is used to generate, every cycle, a 4-bit signature of the value on busD (32-bit ALU or shifter result), the 7-bit "physical" register number for the destination register of the operation (the output of RFTRAN), a bit indicating whether a value is written into the register file (the valid bit for the DWB), the eleven bits of the PSW, and all four state bits from the controller. The 4-bit signature is driven, by the master, onto output pins and is included in the master/slave comparison every cycle.

The above use of signatures does not ensure recovery from all possible transient faults in the processor. Specifically, a transient fault can invert the value of a bit in the register file. Many cycles later, a read from the register file will obtain the corrupted value. The corrupted value will, of course, cause a mismatch between the master and slave so that the error will be detected. A rollback of a few cycles will not correct this error and there is no way to determine which processor, the master or slave, has the correct value. In order to allow the two processors to determine which one has the corrected value, a single parity bit is stored with each register in the register file. Whenever the register file is read, the parity is checked (parB and parD in Figure 2). If a parity error is detected, a rollback of one cycle is initiated. In addition, the master and the slave communicate to each other the results of the respective parity checks using dedicated pins. Following rollback, if the results are clear regarding which processor has the corrupted data, two cycles are used to transfer the value from the fault-free processor to the faulty processor. If both processors detect parity errors in the same register, no *state repair* is performed following rollback (see Section V). Since a transient error in the register file decoder can cause a register to be stored in the wrong location, the parity stored with each register is calculated over the physical register number as well as the data. This error will be detected when there is an attempt to read this register. The state repair mechanism will allow recovery from this error.

It should be noted that the state repair mechanism with the dedicated error detection, as used for the register file, is not required for any other register on the *MP* chip. The reason for this is that the other registers are all modified *every cycle*. Since there is an update every cycle, all stages of the DWBs of these registers generally contain valid values. If an error occurs due to a corrupted value in one of these registers (e.g. the value of a condition code bit in the PSW is inverted, causing a conditional branch to behave incorrectly) the resulting micro rollback will invalidate the corrupted DWB stage. Thus, when the instruction is re-executed, a valid value will be obtained from the DWB.

V. The Mirror Processor Control Unit

The Berkeley RISC II processor was characterized by a simple small control unit [7, 12]. A single PLA was used to decode the opcode, producing 39 control bits, which were ANDed with different clock phases to produce approximately 100 control signals. The Mirror Processor datapath is more complex than the RISC II datapath due to the support for micro rollback, error-detection, and state repair. Hence, the number of control bits required by the *MP* is approximately double the number of control bits in RISC II. Furthermore, many of the control bits are dependent on rollback and repair signals in addition to the opcode. As a result, the *MP* control unit (Figure 7) is significantly more complex than the RISC II control.

Instructions are read from the external bus through *busIN* and are latched onto *busIR*. 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 the first or second cycle of 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. The next state, plus the opcode and the repair signals, are sent to three

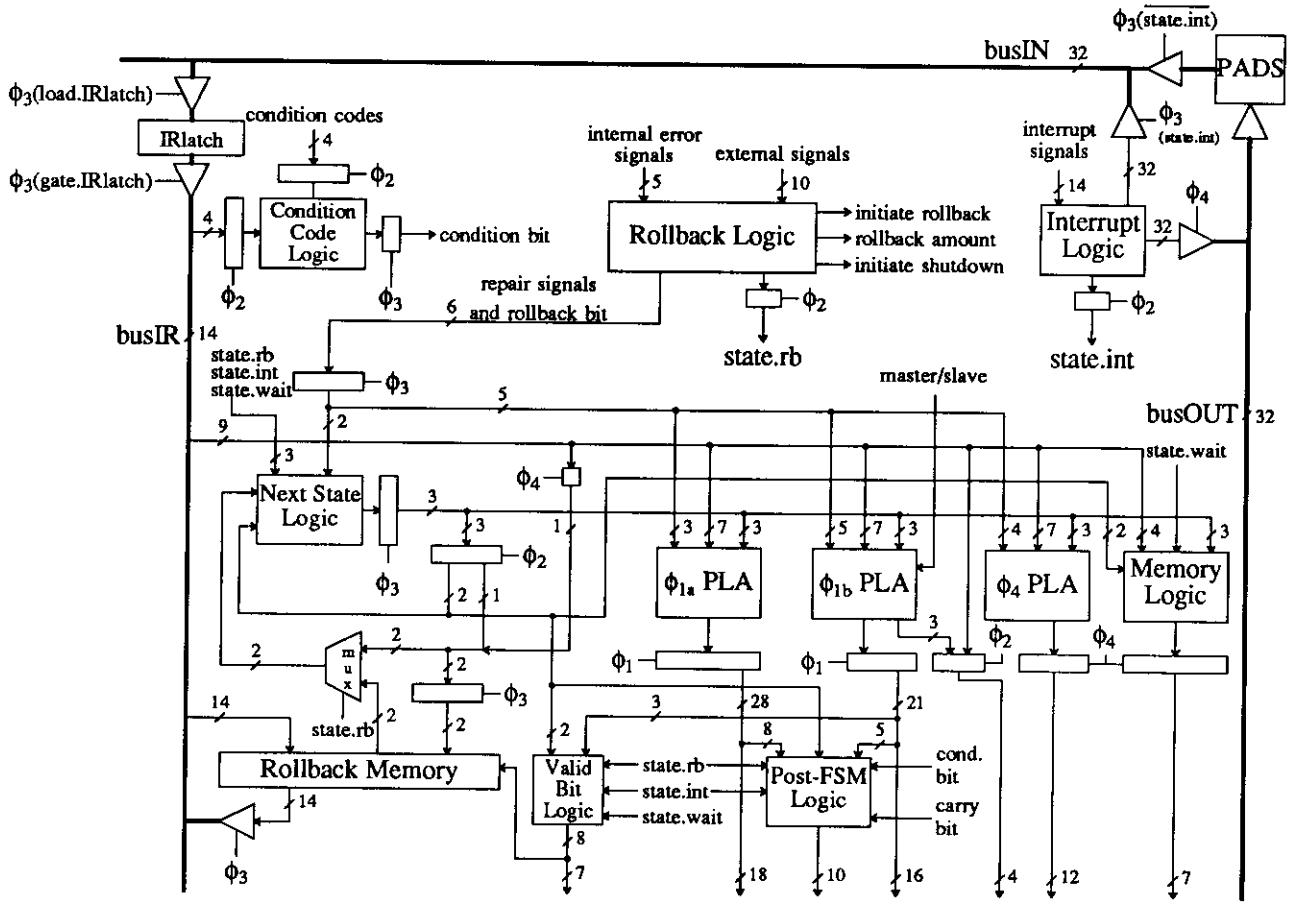

Figure 7: The Mirror Processor control unit. Most of the control signals are generated by the three PLAs. Where necessary, random logic is used for speed.

PLAs that generate the majority of the control signals used by the data path.

During ϕ_2 of every cycle, the control reads the value on the *rollback* pin and the three *rollback amount* pins. All four lines are connected to all the modules which are part of the same rollback domain. Each module connects to these lines through a bidirectional open drain pad driver. The lines are held high by external pullup resistors. At the beginning of ϕ_1 of each cycle, any module in the rollback domain can pull down the *rollback* line and pull down the *rollback amount* lines according to the number of cycles that it needs to roll back. The *MP* slave determines the results of the master/slave comparison during ϕ_3 of each cycle and the result of the parity checks in both *MP* chips are determined during ϕ_4 of the cycle. In the simplest case, if a comparison or busIN parity error is found, the chip requests a rollback of two cycles. If a register file (parB or parD) parity error is found, a rollback of one cycle is signaled.

Since the system is synchronous, when there is a rollback all the modules in the rollback domain must roll back the same distance. If several modules simultaneously request rollbacks by different amounts, the entire system must determine the *maximum* rollback distance requested and roll back by that amount. The maximum rollback distance requested is determined by a straightforward implementation of the Futurebus arbitration protocol [18]. By ϕ_2 of a rollback cycle, the rollback distance for all the

modules is available on the *rollback amount* lines.

During a normal cycle, a maximum of two values are read from the register file of each processor onto the internal buses, busA and busB (Figure 2). Four pins on the chip are dedicated to coordinating state repair between the master and slave. They signal possible parity errors in the values read from the register file. Two of the pins are driven by the master and indicate possible parity errors in the values it reads from its register file. The other two pins are driven by the slave based to the parity checks of the values read from the slave's register file. These four signals are set during ϕ_1 of a rollback cycle and are read during ϕ_2.

During ϕ_3 of the rollback cycle, the appropriate *valid bits* in the various DWBs are cleared in order to perform the rollback. At the same time, each processor's control unit independently determines whether state repair should be initiated. If none of the repair bits are set, no repair is initiated. If one of the processors detected an error in the value read on busA while the other did not, the busA repair will occur regardless of possible errors detected in the values read on busB. If both processors detect errors on busA, a repair of busA is impossible and none is attempted. If a busA repair is not needed, a busB repair may be initiated, as appropriate. Two points should be noted: (1) If repair is needed for both busA and busB, only the busA repair will be done and then normal operation will resume. When the instruction is re-executed following the repair, the busB error will be detected and, since there is no error on busA, the busB repair will be done. (2) If both the master and slave detect an error on the same internal bus, no repair will be done. However, rollback will be done so that the operation will be retried. If the error in at least one of the processors was not permanent (e.g. a transient on an internal bus), at least one of the values will be correct when the instruction is re-executed.

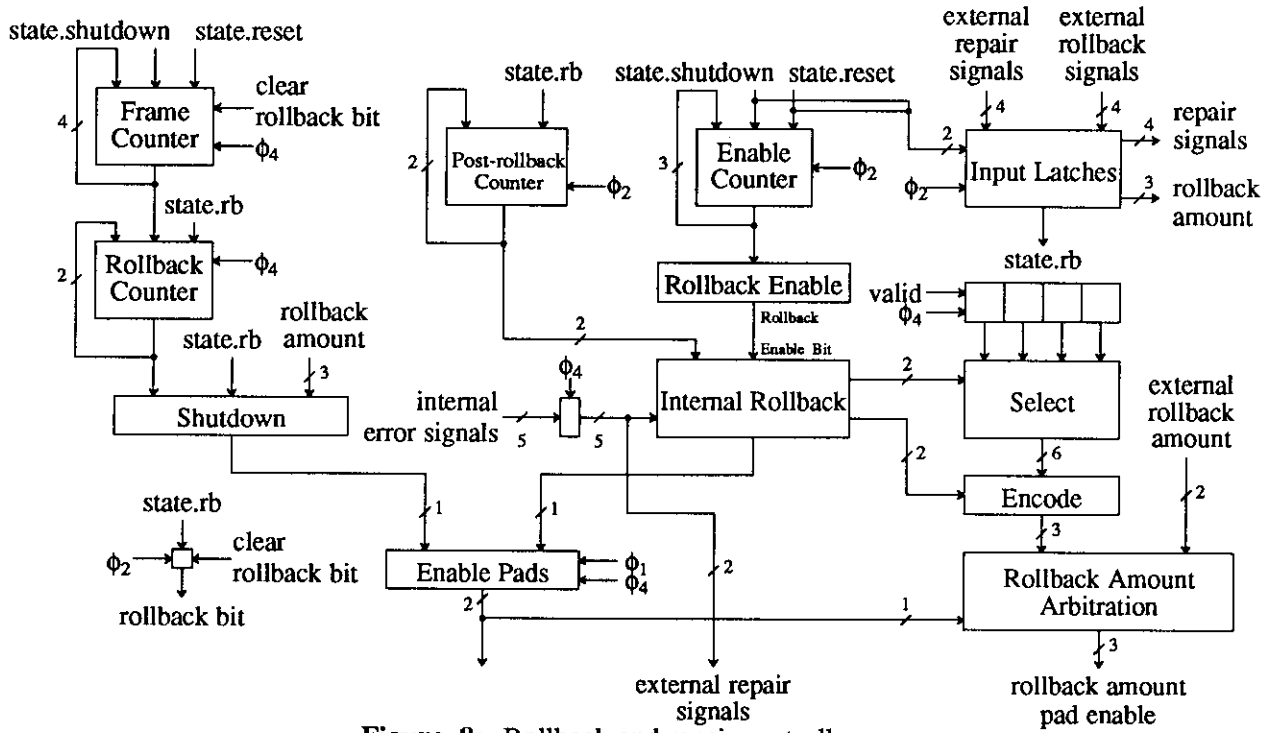

Figure 8: Rollback and repair controller.

Figure 8 shows the rollback and repair controller, which is part of the *MP* control unit. It controls the operations described above and includes many features for handling multiple faults and system-level recovery from massive errors. The rollback controller consists of two parts. The first part monitors the 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 distance to roll back and recognizes conditions where local recovery is not possible (see below). The second part of the rollback controller monitors the external *rollback*, *rollback amount*, and repair lines and sets internal rollback signals.

As discussed earlier, the *MP* was designed to recover from errors caused by single transient faults. However, several features were added to allow recovery from some multiple faults. For all the registers the DWB storage is implemented using dynamic latches. Hence this storage is susceptible to transient faults. If the value in a DWB stage is corrupted, it is possible for a rollback to restore an erroneous state in the master or slave. This will be detected one or two cycles later, triggering a rollback. Unfortunately, without a special mechanism for dealing with this case, the second rollback may restore the erroneous state. In order to prevent this situation, the rollback controller includes the *post-rollback counter* (Figure 8), which counts the number of cycles since the last rollback until it exceeds three cycles. Based on the value of this counter, the *internal rollback* block can determine whether a rollback will restore the state just restored, several cycles earlier, by a previous rollback. If this situation is detected, the *select* logic is used to determine how many DWB entries must be invalidated in order to invalidate the state restored by the previous rollback. This determination is done based on a 4-bit shift register, into which a 1 is shifted every normal execution cycle. During rollback of n cycles, the n most recent entries in this shift registers are cleared. Hence the shift register contains a record of which of the last four cycles was a normal execution cycle which has not been rolled back.

It is possible for the master/slave comparison to detect errors from which recovery is not possible using micro rollback and the state repair mechanism discussed above. For example, if two bits in one of the registers of the register file are inverted, a store of that register will bring the erroneous value to the output pins, causing a master/slave mismatch to be detected. In this situation, the parity checks on values read from the register file cannot indicate which register file is at fault. The mismatch will trigger a rollback. However, when the instruction is re-executed, the same error will be repeated. Without some additional mechanism, the processors will continuously execute the same two cycles followed by a rollback. No repair will be made and there is no way for the node to participate in system level recovery that might allow normal operation to resume.

In order to better handle the situation of useless repeated rollbacks, the node must have the capability of detecting when local recovery is impossible. In the *MP*, this is handled by detecting that the fourth rollback is being attempted within the last 16 cycles and causing the processor to trap to an error handling routine instead of performing the rollback. In the rollback controller, the *frame counter* is a simple 0 to 15 counter which is incremented every cycle. The *rollback counter* counts the number of rollbacks within the current frame. The *rollback counter* is cleared whenever the *frame counter* reaches 0. If a rollback is initiated and the value of the *rollback counter* is 3, the *shutdown* logic initiates

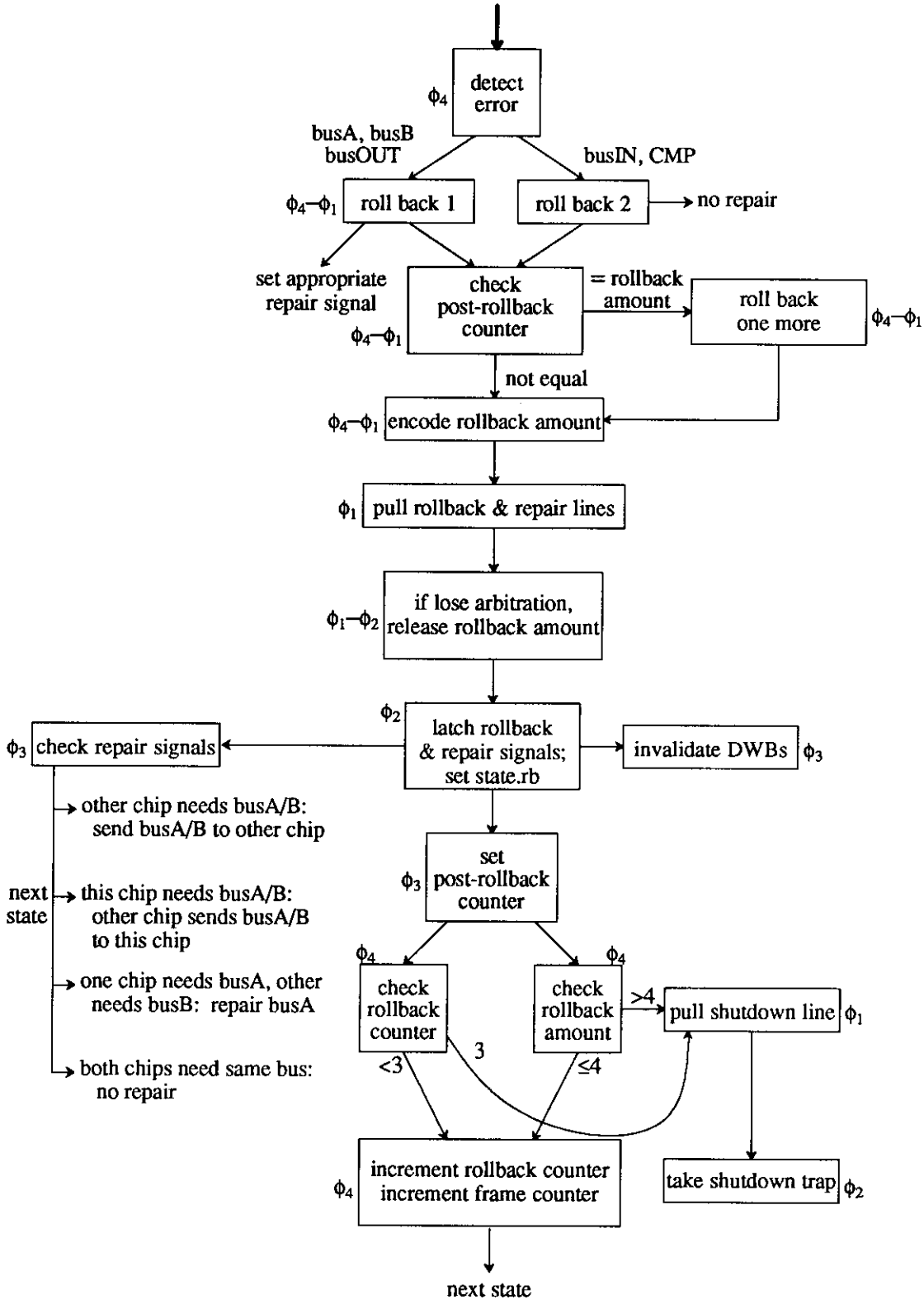

Figure 9: The sequence of operations for micro rollback.

a shutdown trap. As with any interrupt or trap, the shutdown trap causes the processor to begin executing code in a pre-determined address. The code stored in this address can, for example, perform self-diagnosis and then, if the node is operational, integrate the node back with the rest of the system. In a multiprocessor system this *self-resetting* capability [15] is essential since system-level recovery procedures require active cooperation from every operational node.

VI. Support for Periodic Self-Diagnosis

In any design of self-checking modules, there is the problem of preventing latent faults from remaining undetected for long periods. If faults remain undetected, multiple faults can eventually exist in the system simultaneously and cause the concurrent error-detection mechanism to fail, leading to an undetected error. One solution to this problem is for the system to periodically perform self-diagnosis whose purpose is to flush out latent faults. Generally, the probability of multiple faults occuring between diagnosis runs can be reduced to the required level by adjusting the frequency of the self-diagnosis runs appropriately. In a general-purpose processor, self-diagnosis can be accomplished by an operating system that periodically suspends normal execution and runs programs that were designed specifically to exercise all the features of the processor. This approach can be quite successful for simple RISC processors, where relatively short functional tests can be expected to achieve high fault coverage [19].

With the *MP*, there is a special problem of how to test the circuitry for error detection, micro rollback, and state repair. This circuitry is designed to be transparent to the application and is generally not exercised unless there is an error and rollback is initiated. Special hardware and dedicated instructions were added to the *MP* in order to permit self-diagnosis of the error-detection, repair, and rollback circuitry. The new instructions are all *priviledged* instructions which can only be executed in *kernel mode.* An unusual feature of these instructions is that they perform different actions on the master and slave processors. Furthermore, a primary consideration in the design of these instructions was to minimize the complexity of hardware modifications needed to support them. Low priority was given to the generality or "elegance" of these instructions.

The additional instructions provide two key facilities: (1) the ability of code to force apparent errors by storing incorrect parity or performing a different operation on the master and slave, and (2) the ability of code to determine whether a micro rollback has occured since a flag was last cleared. A single *rollback bit* register was added to the control unit. This register can be explicitly cleared (see below) and is set to 1 every time there is a rollback. The dedicated instrictions for self-diagnosis are as follows:

Clear Rollback Bit
clrrbm
clrrbs

Clears the rollback bit in the rollback controller. **clrrbm** clears the rollback bit on the master, while **clrrbs** clears the bit on the slave.

Add with Bad Parity
addbpm S1,S2,Rd
addbps S1,S2,Rd

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
jmprbs Rs1,Rs2,Rd

If the rollback bit is set, a PC-relative jump is performed, using the contents of register *Rs2* as the offset for the jump, while, at the same time, the contents of register *Rs1* are stored in the destination register *Rd*. 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 Rs,X
strbds Rs,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 the rollback bit is cleared, one of the processors will store the contents of the MAR instead. For **strbdm**, the master will store the bad data, while 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*.

As an example of using these special instruction, we consider testing of the handling of a parity error in the register file. The test program can use the **addbpm** or **addbps** instructions to store incorrect parity in a register of the master or slave, respectively. The rollback bits of the master and slave are cleared using **clrrbm** and **clrrbs**. Then, an instruction that reads the register stored with incorrect parity is executed. If everything is operating correctly, this instruction should cause a rollback of one cycle followed by the repair of the register. To check whether a rollback has occurred, the **jmprbm** or **jmprbs** instruction is used. In this use of the "jump if rollback" instruction, the Rs1 field is set to the same register number as the Rs2 field, so that there will not be a master/slave mismatch if the rollback bit was no set. If a rollback has occurred this is an indication that the rollback mechanism is working. The next step of this test is to clear the rollback bits again, read the register originally stored with bad parity, and check the rollback bits. In this case a rollback should not have occurred if the state repair mechanism operated correctly the first time.

It is also possible to test the master/slave comparison of the signature of internal results

(Section IV). This is done by clearing the rollback bits and then using the **jmprbm** or **jmprbs** instruction to store different values to the destination registers. Since different "ALU results" will be transmitted over busD, the signatures of the master and slave will differ, triggering a rollback of two cycles. After the rollback, the execution cycle of the **jmprbm** or **jmprbs** instruction will be repeated. However, now the rollback bits will be set, causing the branch to be taken. Thus, reaching the target of the "jump if rollback" instruction, is an indication that the signature comparison operated correctly.

VII. VLSI Implementation of the Mirror Processor

We have completed a full-custom CMOS VLSI layout of the Mirror Processor, using the MOSIS scalable CMOS design rules (SCMOS). All the features discussed in this paper are fully implemented. The chip contains 52,644 transistors, fits in an 84 pin package (76 pins are used), and, assuming 2 μm technology, the size of the chip is approximately 8.4 mm by 6.7 mm.

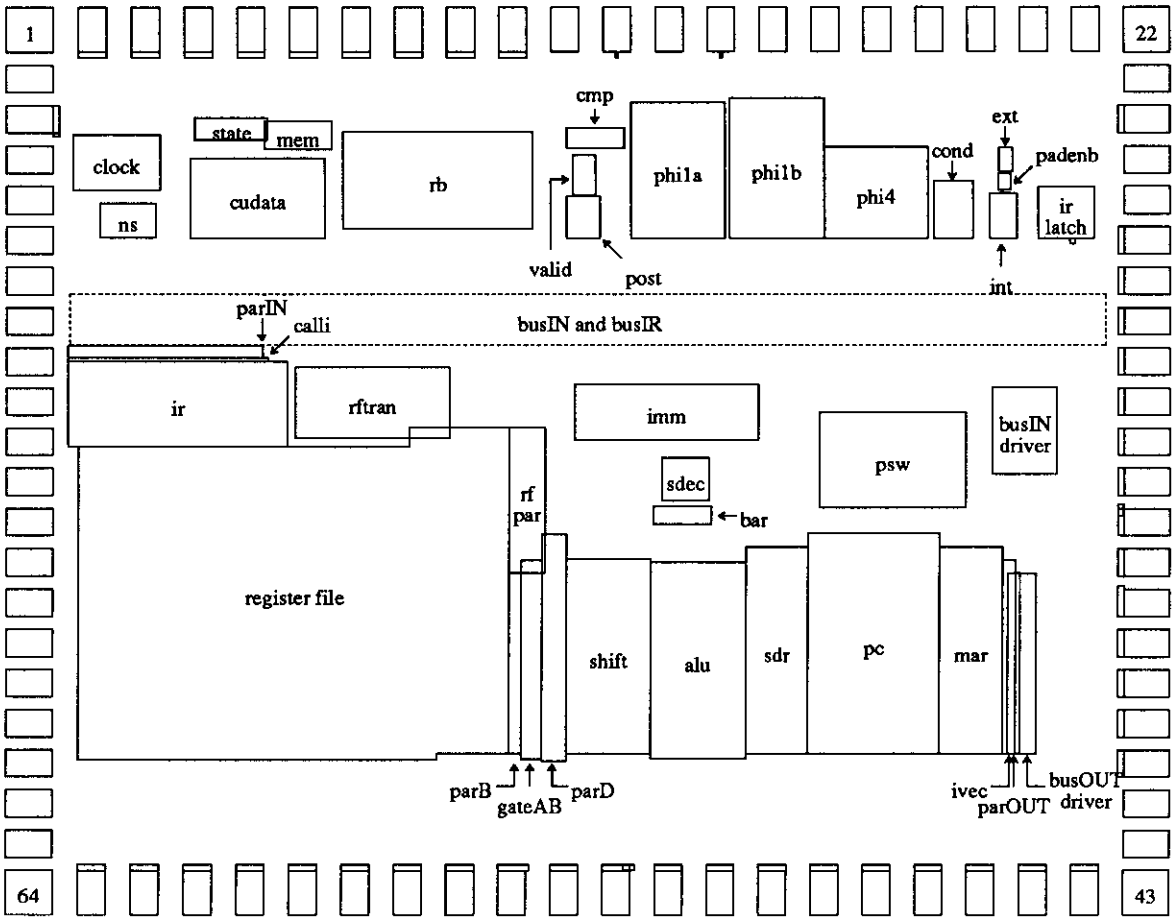

Figure 10: The floorplan of the complete Mirror Processor chip. This floorplan was extracted from the actual layout. All modules are drawn to scale. The "white space" is occupied by interconnections between modules and to the pads.

We have used the switch-level simulator *bdsim* to simulate the entire chip executing several test programs. A phase-by-phase register-transfer-level (RTL) description of the chip was written in Zycad's

ISP' hardware description language. The results of RTL simulation of the chip were compared, phase-by-phase, with the results of the switch-level simulations in order to verify the operation of the chip.

All the modules in the chip were simulated, at the circuit-level, using a version of the SPICE circuit simulator (HSPICE), assuming 2 μm technology and using pessimistic (slow) values for the device parameters. The timing analyzer *crystal* was used to determine the critical paths for each phase. Based on SPICE simulations of the critical paths, the chip will operate correctly using a 100 ns clock, with 25 ns ϕ_1 and ϕ_3, 15 ns ϕ_2 and ϕ_4, and a 5 ns non-overlap distance between phases. At this clock rate, power consumption is expected to be approximately 0.9 W per chip. The chip includes circuitry to generate the clock phases described above from an external 50% duty cycle, 40 MHz signal. This circuitry also requires an external 10 MHz signal, which is used to ensure that the master and slave execute the different phases in lock-step.

The area overhead for the error detection and recovery capabilities of the chip is significant. In particular, we estimate that by removing all the features for fault tolerance, the width of the chip could be reduced from 8.4 mm by approximately 1.8 mm and the height could be reduced from 6.7 mm by approximately 1.5 mm. On the other hand, very little performance overhead is incurred. Specifically, the only additional delay is caused by a slightly larger capacitance to be charged during register file reads due to longer buses across the register file DWB and parity circuitry. This increases the clock cycle by less than 3 ns.

VIII. Summary and Conclusions

We have presented the design and implementation of the UCLA Mirror Processor — a VLSI RISC microprocessor with extensive capabilities for fault tolerance. Our implementation of the *MP* demonstrates that micro rollback is a practical technique for minimizing the latencies normally associated with concurrent error-detection. A self-checking module, implemented with two *MP* chips operating in lock-step, can recover from most errors caused by transient faults. Micro rollback is used for re-execution of cycles (as opposed to instructions) during which errors are generated. For those cases where a transient fault may permanently modify a stored value, we introduce the use of rapid, hardware-supported state repair. Since local recovery is not sufficient for all possible errors, our design supports node *self-reset*, which guarantees that the node will re-establish a "sane" state from which it can participate in system-level recovery. The problem of latent faults in the detection and recovery mechanisms is addressed by special instructions that facilitate periodic self-diagnosis. The extensive fault tolerance features of the Mirror Processor involve significant chip area overhead but only negligible performance overhead.

References

1. X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and Calibration of a Transient Error Reliability Model," *IEEE Transactions on Computers* **C-31**(7), pp. 658-671 (July 1982).

2. M. L. Ciacelli, "Fault Handling on the IBM 4341 Processor," *11th Fault-Tolerant Computing Symposium*, Portland, Maine, pp. 9-12 (June 1981).

3. Advanced Micro Devices, *Am29000 Streamlined Instruction Processor User's Manual*, 1987.

4. R. W. Downing, J. S. Nowak, and L. S. Tuomenoksa, "No. 1 ESS Maintenance Plan," *Bell System Technical Journal* **43**(5), pp. 1961-2019 (September 1964).

5. W. W. Hwu and Y. N. Patt, "Checkpoint Repair for Out-of-order Execution Machines," *14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 18-26 (June 1987).

6. D. Johnson, "The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems," *Computer* **17**(8), pp. 40-48 (August 1984).

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

8. E. J. McCluskey, "Built-In Self-Test Techniques," *IEEE Design and Test* **2**(2), pp. 21-28 (April 1985).

9. D. A. Patterson and C. H. Séquin, "A VLSI RISC," *Computer* **15**(9), pp. 8-21 (September 1982).

10. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2), pp. 123-165 (June 1978).

11. D. A. Rennels, "Architectures for Fault-Tolerant Spacecraft Computers," *Proceedings IEEE* **66**(10), pp. 1255-1268 (October 1978).

12. R. W. Sherburne, M. G. H. Katevenis, D. A. Patterson, and C. H. Séquin, "A 32-Bit NMOS Microprocessor with a Large Register File," *IEEE Journal of Solid-State Circuits* **SC-19**(5), pp. 682-689 (October 1984).

13. M. Sievers and D. A. Rennels, "An LSI Totally Self-Checking Hamming Coded Memory Interface," *International Symposium on Circuits and Systems*, Rome, Italy, pp. 1176-1179 (May 1982).

14. J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers* **C-37**(5), pp. 562-573 (May 1988).

15. Y. Tamir and C. H. Séquin, "Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers," *International Conference on Computer Design*, Port Chester, NY, pp. 561-564 (November 1983).

16. Y. Tamir, M. Tremblay, and D. A. Rennels, "The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems," *18th Fault-Tolerant Computing Symposium*, Tokyo, Japan, pp. 234-239 (June 1988).

17. Y. Tamir and M. Tremblay, "High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback," *IEEE Transactions on Computers* **39**(4), pp. 548-554 (April 1990).

18. D. M. Taub, "Arbitration and Control Acquisition in the Proposed IEEE 896 Futurebus," *IEEE Micro* **4**, pp. 28-41 (August 1984).

19. S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors," *IEEE Transactions on Computers* **C-29**(6), pp. 429-441 (June 1980).

20. M. Tremblay and Y. Tamir, "Support for Fault Tolerance in VLSI Processors," *International Symposium on Circuits and Systems*, Portland, OR, pp. 388-393 (May 1989).