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

MICRO ROLLBACK ON A VLSI RISC: DESIGN AND IMPLEMENTATION OF THE UCLA MIRROR PROCESSOR

Mike Teh-An Liang

December 1990
CSD-900042

Micro Rollback on a VLSI RISC: Design and Implementation of the UCLA Mirror Processor

Mike Teh-An Liang

Computer Science Department
4731 Boelter Hall
University of California
Los Angeles, California 90024-1596
U.S.A.

December 1990

Table of Contents

Table of Figures

Table of Tables

Abstract

In order to achieve high reliability, computing systems that perform critical tasks must be able to continue normal operation despite component failure, *i.e.*, they must be fault-tolerant. Some methods of achieving fault-tolerance entail a long error recovery time or add considerably to the cycle time because operation cannot proceed until data has been verified. Micro rollback provides rapid restoration of previous system state based on fine-grained checkpointing done in hardware. Operation continues without delay while the data is checked, and if an error is detected a few cycles later, then the system can be rolled back to an error-free state.

This report describes the design and implementation of the UCLA Mirror Processor, a VLSI RISC processor capable of micro rollback. Its main mode of error detection is duplication and comparison. Two processors, a master and a slave, run in lockstep and perform the same operations. The slave processor compares its external signals and a signature of its internal signals with the corresponding signals from the master processor. If an error is detected, the processor state is restored to the beginning of the cycle during which the error occurred, so that correct processor state may be regenerated. Errors detected in the register file are corrected by transferring data from the fault-free processor to the one with the corrupt values. The Mirror Processor architecture, its operation, and its error detection and error recovery features are described, with an emphasis on the physical implementation of the datapath.

1. Introduction

Highly reliable computing systems are needed for critical applications such as space craft control, where human intervention may not be possible or practical after launch. Fault tolerance is often necessary in order to achieve the required reliability, since the reliability for the hardware components is insufficient.

In many fault-tolerant systems, checkers are connected between modules to verify that the data is correct before it is transferred, thus preventing error propagation through the system. These sequential checks add to the overall cycle time because data may not be transferred until the checker verifies it is correct. However, the cycle time may be reduced by adding pipeline stages. This is not a good solution because long pipelines broken by instruction branches add more cycles to the execution time, and the overall execution latency is increased.

An alternative to checking data before sending it is to perform the checking in parallel with the data transmission. Data can propagate through the system unhindered, and the results of the checking follows a few cycles later. By the time an error is detected, other modules may have already received erroneous data so that their internal state is invalid. Micro rollback[Tami88a] allows modules to roll back several cycles so that a valid system state is restored. Once the system is restored to the state it was at before the error occurred, the system can continue on with error-free execution. This report describes the design and implementation of a RISC[Patt82a] (Reduced Instruction Set Computer) processor capable of micro rollback, built to demonstrate that micro rollback is a practical, effective technique for use in high performance fault-tolerant systems.

1.1. Micro Rollback

All the registers, inputs, and outputs of a processor comprise its state, and every cycle this state changes. As a result of a fault, the state may deviate from the valid state for correct system operation. A processor can recover from such an error by resetting its state to the cycle before that in which the error occurred and then continuing on.

Micro rollback is a form of fine-grained checkpointing and error recovery at the hardware level. Every cycle, the state of a system is stored in a backup memory and the last *n* copies are kept. Should an error be detected, the state of the system is restored to that before the time the error has been determined to have occurred. Hence, error detection can be performed in parallel with instruction execution and can take as long as the number of cycles that the system saves.

In a system employing sequential data checking, data must be checked before it enters a functional unit to detect bits corrupted on the sending bus. After the operation, the results must be verified before being transferred to another unit in order to detect transient errors originating from the functional unit or the interconnecting bus. The checking step must be completed after every intermodule transaction, and the system cannot progress until after the data is verified in order to restrict the error to that module. Using micro rollback, the time the system must wait for data verification is removed. Hence the only overhead involved is during error recovery.

Micro rollback is implemented at the hardware level because of the time constraints involved with error detection and recovery. Allowing software intervention, say by a trap handler, would add more cycles to the time before the proper state can be restored. In addition, the checkpoint memory would need to be enlarged to record the increased time between the cycle of the valid state to be restored and the cycle that the system state is restored. An added benefit of a hardware implementation is that the fault-tolerance provided is transparent to both application and system software.

Micro rollback is only used to recover from errors caused by transient faults. If an error is caused by a permanent fault, this will be detected and rollback will follow. But since micro rollback only restores system state, the hard error will remain, and so the error detection and recovery sequence will be repeated. After too many rollbacks within a certain period of time, the system will assume a permanent fault has occurred and it will signal a system shutdown.

1.2. Goals

The goal of this research is to investigate the overhead associated with micro rollback. The ability to perform micro rollback was added to a processor based upon the Berkeley RISC II[Kate83a, Patt82a]. For error checking, two such processors run in lock-step and compare their states after each cycle. Reflecting upon this configuration, the chosen name for our project is the UCLA Mirror Processor (henceforth *MP*).

Specific goals for the *MP* include minimizing chip area and maintaining high processor performance. In addition, the *MP* chip-set should not involve use of glue chips in order to conserve printed circuit board real estate.

We implemented the *MP* using full custom layout to most effectively utilize chip area. Techniques such as precharged logic and dynamic latches were used to minimize module area. With the full custom layout, the modules could be tailored to fit together well, and transistors could be sized to drive their actual loads (unlike standard cell libraries where the load is unknown and so the transistors must be sized to drive a fixed pre-specified load instead of the actual load).

The time overhead for micro rollback involves two components: normal cycle operation, and error recovery time. We minimized the overhead associated with normal cycle operation because this directly translates to operating speed. In order to meet real-time constraints, error recovery time is also important, even though it may constitute a small percentage of operating time. Rapid recovery may be critical in an environment where periods of high fault rates are expected.

1.3. Report Organization

Chapter 2 gives a high level description of the *MP*'s architecture and operation and details the design decisions involved in adapting the Berkeley RISC II Processor to handle micro rollback for fault detection and recovery. Chapter 3 presents the implementation of the modules forming the datapath; details on the datapath controller implementation can be found in [Lai90a]. Strategies for ensuring the implementation is correct and testing the layout are covered in chapter 4. Results of this undertaking are shown and conclusions are drawn in chapter 5.

2. Design Decisions and Processor Description

This chapter presents the architecture and microarchitecture of the Mirror Processor. First, the *MP* architecture and instruction operation is described. After that, features pertaining to error detection, micro rollback, and state repair are covered in greater detail. Implementation details of the datapath components are in chapter 3, and details of the controller can be found in [Lai90a].

2.1. Processor Architecture

2.1.1. Processor Choice

In order to demonstrate that micro rollback is a feasible method of implementing an inexpensive fault-tolerant system, we must apply the theory to a real processor. Among the various processors available, the Berkeley RISC II stands out as the ideal choice for several reasons.

First and foremost is the documentation provided in the form of Katevenis' dissertation[Kate83a] and the various papers associated with the Berkeley RISC[Sher82a]. In addition, ease and speed of implementation is a factor; Katevenis and Sherburne designed, laid out, and had the Berkeley RISC fabricated in two years.

The Berkeley RISC has various architectural features which prove a challenge to applying micro rollback. The large register file (132 registers) cannot just be replicated n times and so a better solution must be found. In the same vein is the program counter which is set up as a three element shift register. These and other problems allow us to

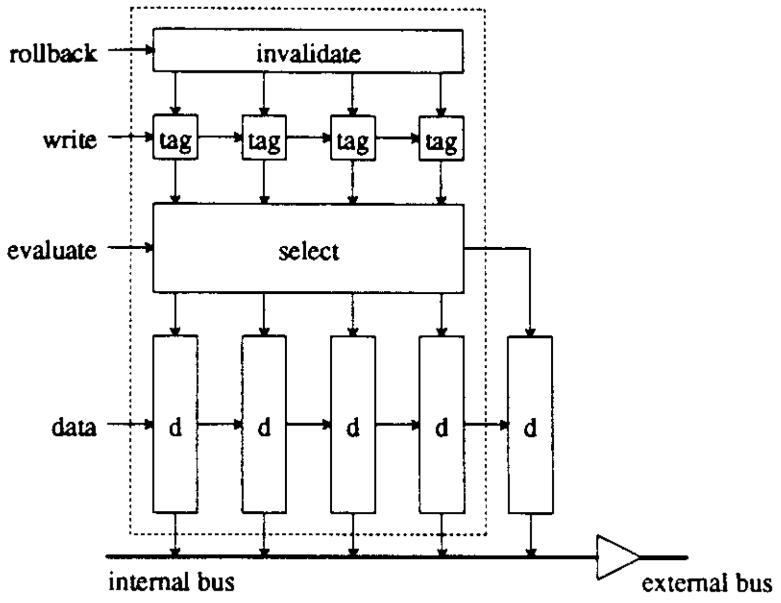

Figure 2.1: Delayed Write Buffer with Register

demonstrate the application of micro rollback on a reasonable complex processor that can be used for real world applications.

2.1.2. Micro Rollback

In order to support micro rollback on the *MP*, delayed write buffers are placed in front of every register that holds state from one cycle to the next. Figure 2.1 shows the block diagram of a delayed write buffer (DWB) placed in front of a register. The buffer is an *n*-element shift register with a priority associative tag lookup. When data is written into the buffer, it is delayed *n* cycles before it is written into the permanent register. Saved along with the data is a tag. In the case of the register file, this tag consists of a *valid bit* and the address that the data is to be store at within the register file; for single registers, only a *valid bit* is needed. For correct operation, the processor must read the most recent, valid value within the DWB/register. This requires the use of a priority selection circuit which chooses the data associated with the first valid matching tag. When the *MP* rolls back *n* cycles, it must invalidate the first *n* tags in the shift register so that on the next read these will be skipped. Consequently, only data that makes it to the end of the shift register with a valid tag is written into the permanent register.

We choose to checkpoint four cycles because this is enough time for the state checkers to detect an error and signal rollback. In addition, saving four cycles allows the *MP* to recover from two consecutive rollbacks.

It may appear to be most efficient to shift the DWBs only when writing a value into them. However, the cycle that a state belongs to is dependent upon its position within the shift register. In order to maintain the temporal information associated with the data, all DWBs must shift every cycle whether they are suppose to be written into or not. Data will always be shifted into the data portion of the DWB, but logical writing of the value is achieved by shifting a 1 into the *valid bit* of the tag. During the rollback cycle, the present instruction is cancelled, and 0's are shifted into the *valid bits* of all the registers; this effectively cancels all writes during the rollback cycle.

2.1.3. Datapath Description

The *MP* datapath, as shown in figure 2.2, operates on a four-phase clock (ϕ_1, ϕ_2, ϕ_3, ϕ_4). The implementation details can be found in chapter 3. More detailed information on the *MP* at the register-transfer level can be found in section 7.3.

The following modules do not require special changes for micro rollback:

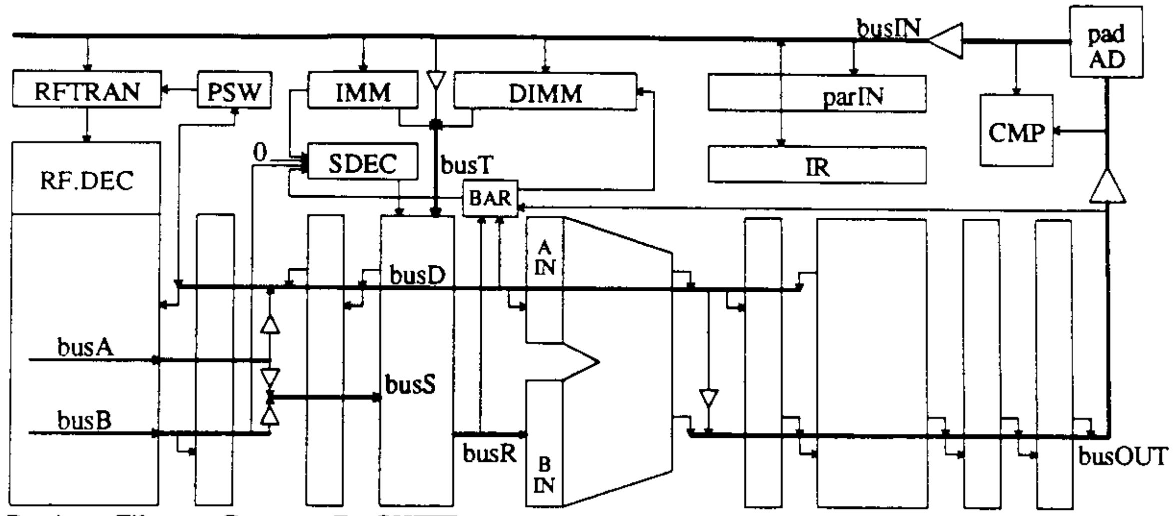

Figure 2.2: Mirror Processor Datapath

- IMM: The immediate latch, part of the distributed instruction register, is used to hold the 19-bit or 13-bit sign extended immediate constant embedded within the instruction for use the following cycle. The value is latched on ϕ_3, during the instruction fetch, and gated onto busT the following ϕ_1, to be aligned by the shifter.

- DIMM: The data immediate performs sign extension and zero extension of 8, 16, and 24 bits upon the data coming in from a load instruction. Data is read in during ϕ_3 and it is immediately deposited onto busT for data alignment during ϕ_4.

- BAR: The byte address register computes during ϕ_2 the 2 least significant bits that would have been generated by the ALU had it done an add because the ALU cannot supply this value soon enough. This is needed for load, store, and branch instructions so the *MP* can determine if it should trap on address misalignment before initiating a memory cycle. Also, the store operation performs a data alignment shift during ϕ_4 when the byte address is needed and can't be obtained from the ALU in time. The BAR holds state over a cycle for the store instruction and technically needs to be rolled back, but we can accomplish this purpose by having the BAR read from busOUT during rollback with minimal additional hardware.

- SDEC: The shift decoder takes a 5-bit shift amount and enables 1 of the 7 shift lines within the shifter. A value not corresponding to any of the shift lines generates a bad shift amount trap. Decoding two shift amounts per cycle because the shifter evaluates twice a cycle may seem like a problem, but the first shift is always by 0 or by 13. If the *shift by zero* control line is true, then the ϕ_2 shift is by 0, otherwise it is by 13. The *MP* uses *shift by zero* for the *ldhi* instruction and state repair, both of which do not use the shifter during ϕ_2.

- SHIFT: The 32-bit *MP* shifter differs from the Berkeley RISC shifter in that it only shifts by 0, 1, 2, 8, 13, 16, and 24, and the 13 bit shift is only used for immediates during ϕ_2. The dynamic shifter evaluates twice a cycle, and it is given data starting ϕ_1; this means it must precharge during ϕ_1 and ϕ_3, and evaluate during ϕ_2 and ϕ_4, the short phases. Limiting the number of positions the dynamic crossbar shifter can shift reduces internal capacitance and allows faster evaluation times. Internally, the shifter is set up as a left bus, busL, and a right bus, busR, connected together at shift points by NMOS transistors controlled by the shift decoder. A left shift can be executed by discharging busL and reading busR, and vice versa for a right shift; left shifts are zero filled and right shifts can be zero filled or sign extended. Operands read from busS can be put onto either busL or busR, and busT drives busL. The resulting shift is read off the appropriate bus and gated onto busD. During ϕ_2, data can be right shifted from busT or passed from busS onto busR and into the B input of the ALU.

- ALU: The 32-bit integer arithmetic and logic unit performs additions, subtractions, bitwise ANDs, ORs, and XORs, and calculates the conditions codes based upon its operation and the value of busD. The ALU precharges its manchester carry chain during ϕ_2, evaluates during ϕ_3, and gates the result onto the busses during ϕ_4. Condition codes for negative, overflow, and carry are easily generated, but the precharged NOR used to calculate the zero bit must evaluate on ϕ_1, when busD is stable.

In addition to the controller (described in section 2.1.5), delayed write buffers are placed in front of these registers:

- PSW: The processor status word contains the current window pointer (CWP = 2 bits), the saved window pointer (SWP = 2 bits), the interrupt enable and system status bits (Interrupt enable, System mode, Previous system mode), and the condition codes (Zero, Negative, oVerflow, Carry) for a total of 11 bits. When the PSW writes onto busD, only the lower 11 bits are driven onto busD, and so the top 21 bits contain whatever value was there from the previous cycle. When the PSW reads from busD, only the lower 11 bits are stored.

- RFTRAN: The register file works with absolute addresses, and so the register file address translator takes a register number and a register window number and determines a unique absolute address within the register file. The *MP* instruction format specifies two source register numbers and one destination register number;

these need to be translated into addresses within the register file. The two source registers are used before the next instruction is read in, and so they don't need to be saved, but the destination register number may need to be carried over a cycle for the *load* instruction, and so it must be stored. Instead of directly using a delayed write buffer, the RFTRAN reads the destination register from the controller's busIR (see figure 2.3), whose value is restored during the rollback cycle.

- Register File: The 74-word 33-bit *MP* register file is half the size of the Berkeley RISC register file which contains 132 registers. This was done to reduce the stride and the read latency. As a result of using only 4 register windows, the PSW is also smaller by two bits, one from the CWP and one from the SWP.

- PC: The program counter is logically a three element shift register containing the address of the instruction currently being fetched (NXTPC), the address of the instruction currently being executed (PC), and the address of the instruction previously executed (LSTPC).

During rollback, the *MP* may need to get data from off chip or to send data off chip. However, with the *MP*'s tight timing constraints, there isn't enough time to wait for values to come in or to regenerate these values to send out. Instead, these transactions may be simulated by using a delayed write buffer and a register to read the values going over a bus each cycle, and gating the needed signal onto the bus during the rollback cycle.

- IR: The instruction register restores the contents of the distributed instruction register (IMM, RFTRAN, Controller) by simulating an instruction fetch during ϕ_3 of rollback.

- MAR: The memory address register is used during rollback to initiate the instruction fetch of the instruction after the one being rolled back to because there isn't enough time to regenerate the address from the PC or the ALU.

- SDR: The store data register holds data for the store instruction. Normally, busD can be used as a temporary latch to hold data over for a cycle, in which case the SDR would only have to restore the value of busD, and thus, not be used in the course of normal operation. However, state repair also uses busD, and datapath timing prevents busD from being restored in time after state repair, and so the SDR is actively used as a gateway from busD onto busOUT.

State comparators and parity checkers detect errors from which the *MP* recovers by using micro rollback.

- parIN: The parity checker for busIN assures that the data sent in has not been corrupted and also checks the data from the IR during rollback.
- parB: The parity of the value read out onto busB from the register file is checked with this parity checker.
- parD: ParD performs three tasks: check the parity of the value read out onto busA from the register file, generate the parity from busD to be written into the register file, and compress the state of busD into 4 bits to be compared with the other chip.
- parOUT: To allow the receiving module (e.g. memory) to verify that the address/data sent off the chip has not been corrupted, parOUT generates a parity bit to accompany the value.
- CMP: The comparator on the slave processor compares its internal state and output signals with those of the master processor.

The Berkeley RISC uses dynamic busses to transfer data between modules in the datapath in order to save driver area, but the *MP* uses static busses to remove precharging overhead and to meet tighter timing requirements. (Internal busses within a module, on the other hand, may be precharged to reduce area (as in the register file), or to speed up function evaluation (as in the ALU).) As a consequence, the busses can be used as latches to hold values between phases rather than immediately storing them into registers.

- busIN: The input bus into the datapath is driven by tri-state buffers because during rollbacks, the IR supplies the instruction instead of memory.
- busA, busB: Values read from the register file are supplied on these busses.
- busD: Data destined for the register file and other datapath blocks is put here.

- busS, busT: Paths to the shifter.
- busR: The only way into the ALU's B input is through the shifter and onto busR.
- busOUT: This bus is for sending addresses and data to the pads.

The region between the register file and shifter in the Berkeley RISC was congested with 5 busses and interconnect; this would be further aggravated in the *MP* with its 39λ pitch. In the *MP*, the inputs to the shifter, busA and busB, are multiplexed onto busS before being sent out, and the shifter returns its results on busD rather than returning them on one of two busses like the Berkeley RISC. This reduces the maximum number of busses running through a bitslice to an acceptable number, three. In addition, all functional units place their results onto busD, and so condition code generation can be centralized in the ALU.

2.1.4. Datapath Timing

The *MP* uses a two-stage execution pipeline consisting of an instruction fetch the first cycle followed by its execution (operand read, operation, and write) the second cycle. The Berkeley RISC uses a three-state pipeline because it does not have enough time after the data operation to write the result into the register file, and so it needs to hold the value in a temporary register to be written the following cycle. This temporary register has been replaced in the *MP* by the register file's DWB, and so the operation result can be written into the register file at the end of the second cycle.

Instructions that do not load data from or store data to memory take two cycles to execute. The *load* and *store* instructions, however, need an extra cycle to perform their memory transaction. Because the *MP* can only perform one memory transaction per cycle, the instruction fetch process must be suspended for a cycle while the *load* or *store* instruction reads from or writes to memory.

Table 2.1 shows the timing of modules in the datapath. Operation is partitioned into four clock phases in order to accommodate the operation of the register file and the shifter. The register file performs a dual ported read followed by a single ported write every cycle. It's address decoders are precharged, and so it needs four phases to accomplish its task. The shifter shifts values twice a cycle and uses dynamic shift lines; it also needs four phases to operate.

The execution of an instruction begins with its instruction fetch. Because the *MP* pipeline is two stages deep, the currently executing instruction does not determine the address of instruction to be executed the following cycle, but that of the instruction two cycles down. The instruction's address is put onto busOUT by the PC or the ALU (if it was a branch) on ϕ_4 and the address goes out to the memory. On ϕ_3 of the following cycle, the instruction comes in on busIN and into the distributed IR (IMM, RFTRAN) and the controller (not shown). The controller decodes the opcode and generates control signals for the datapath to use the following cycle. During the execution cycle, operands are read out of the register file on ϕ_1, are operated upon during ϕ_3 and ϕ_4, and the results are returned to the register file on the result bus (busD) on ϕ_4. At the same time that this instruction is executing, the following instruction is being fetched from memory.

If the executing instruction is a *load* or a *store*, then the instruction fetch that would have been initiated for the next cycle during ϕ_4 is suspended, and instead, the data

Module	ϕ_1	ϕ_2	ϕ_3	ϕ_4
rftran	busIR→rftran	rftran→rf(d) psw→rftran	busIN→rftran rftran→rf(a,b)	
rf	ram read busD→rf rf→busA rf→busB	precharge decoders	ram write	precharge decoders
psw	busD→psw	psw→rftran psw→cu psw→pads		psw→busD
parB	busB→parB	parB→cu		
gate	busB→busS	busA→busD busD→busOUT	pads→busIN busIN→busT busA→busS busB→busS	busA→busD
imm	imm→busT		busIN→imm	
dimm			busIN→dimm dimm→busT	
sdec	sdec→shift	busB→sdec imm→sdec	bar→sdec sdec→shift	
parD	parD→rf parD→cu parD→state	precharge	parD→cu	precharge
shift	precharge busS→shift busT→shift	shift→busR	precharge busS→shift busT→shift	shift→busD
bar		busD→bar busR→bar	bar→sdec bar→dimm bar→cu	
alu	cc→psw	precharge busD→alu busR→alu	evaluate	alu→busD alu→busOUT
parIN	parIN→cu		busIN→parIN	
ir			ir→busIN	busIN→ir
sdr	busD→sdr	sdr→busOUT		
pc	busOUT→pc	pc→busD		pc→busD pc→busOUT
mar	busOUT→mar	mar→busOUT		mar→busOUT
parOUT	parOUT→pads parOUT→cu	busOUT→parOUT	parOUT→pads parOUT→cu	busOUT→parOUT

Table 2.1: Mirror Processor Datapath Timing

address is put out and the following cycle is used to load or store data from or to memory. After the memory transaction, the instruction fetch process is restarted again.

The *MP* runs on a four-phase clock with total cycle time of 100ns. The first and third phases, ϕ_1 and ϕ_3, take 25ns, and the second and fourth phases, ϕ_2 and ϕ_4, take 15ns. Between all four phases is a 5ns underlap. Ideally, the four clock phases are generated on-chip because the timing scales with the chip process parameters, and hence the rest of the chip. In addition, only two square-wave clock signals need to be supplied instead of four irregular phases.

The *MP* will internally generate the phase clocks or used externally supplied phase clocks based on whether the clock select pin (pad.csel) is set to 0 or 1. A phase clock (pad.phi) running four times faster than the cycle time is used to generate the four clock phases and a synchronization clock (pad.sync) is used to determine which of the phases is ϕ_1. See section 3.2 for implementation details.

Rollbacks are synchronous, and require all system modules to restore their state to that as found at the beginning of a cycle *n* cycles back from the present cycle. In the *MP*, rollback action starts during ϕ_3, when the DWBs are invalidated and the IR simulates the instruction fetch of the instruction that will be executing at the beginning of the cycle to be restored. Rollback is completed on ϕ_4, when the address of the next instruction to be fetched (or the data to read or written if the executing instruction was a load or store) is put out by the MAR. We choose the *MP* cycle boundary to be between ϕ_4 and ϕ_1 because at this point, micro rollback will be complete and the processor can continue on with normal operation. This also happens to be the start of an instruction execution cycle consisting of a register file read on ϕ_1 followed by the operation on ϕ_3 and ϕ_4 and the result returning back to the register file on ϕ_4.

2.1.5. Datapath Controller

The *MP* must be able to handle normal operations, error detection, rollback, and state repair; these four functions are partitioned into three systems:

- The error detection and recovery controller compares the two chips for mismatches and signals rollback or state repair; it will be covered in section 2.3.
- The rollback mechanism is distributed within the datapath and control; each DWB is invalidated during the rollback cycle, and the following cycle the proper value is read out of the register. This is described in section 2.4.
- The datapath controller manages the datapath during normal operation, and with the addition of two states, it handles state repair. State repair and the associated controller states are discussed in section 2.5. Section 2.2 describes normal operation of the *MP*.

When the *MP* rolls back, it restores state to that as recorded at the beginning of ϕ_1 of the destination cycle. All registers must contain the value they held at this time by the end of rollback. The controller restores the control signals by simulating an instruction fetch during the rollback cycle and regenerating the control signals from this instruction. This is simpler than rolling back the control lines because there are less inputs than outputs in the controller. Figure 2.3 shows a block diagram of the datapath and controller together.

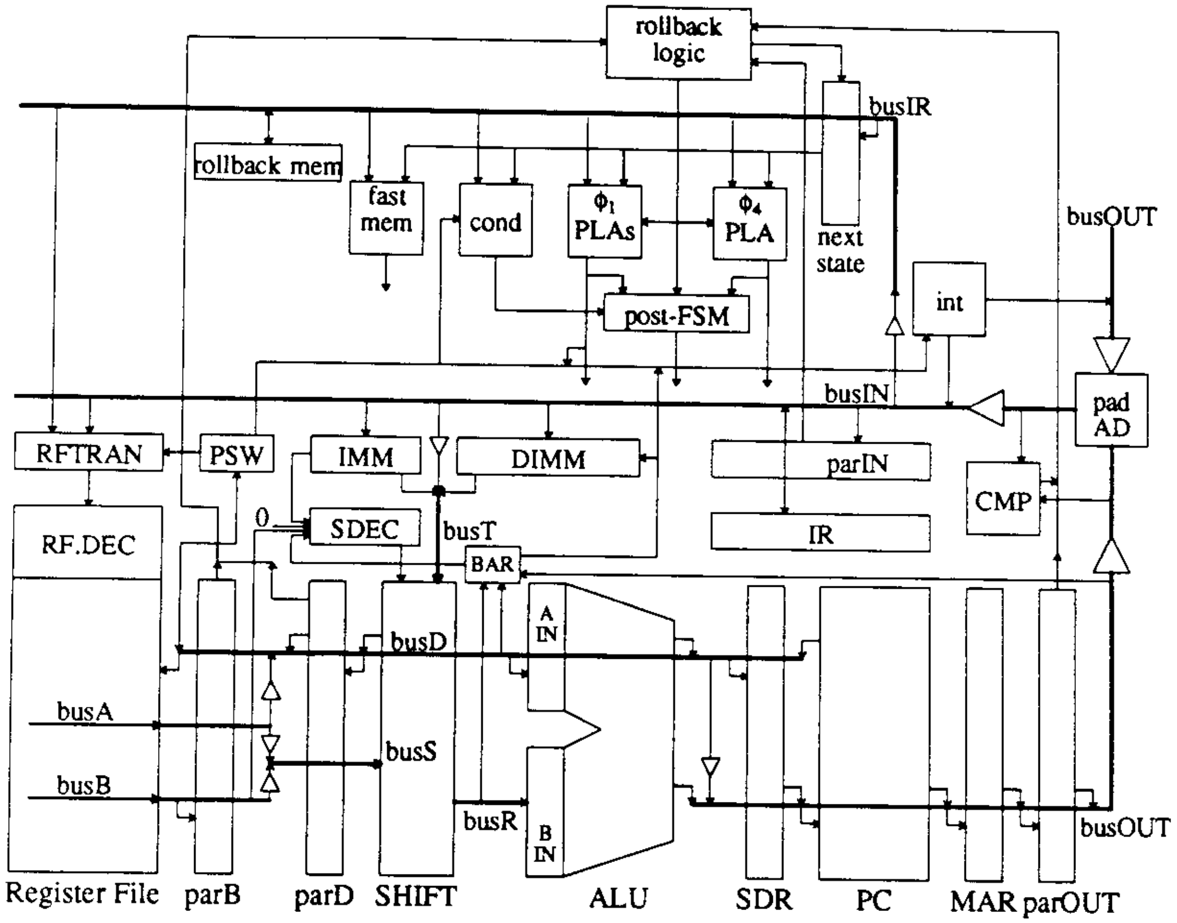

Figure 2.3: The Mirror Processor

When the instruction comes in on ϕ_3, it goes through combinational logic to determine the next state. This next state, along with the incoming opcode, is used to generate controls for the datapath. Memory control signals are needed immediately in ϕ_4, and are generated using combinational logic. The other signals aren't needed until later in the execution cycle, and so they are generated with two dynamic PLAs, one evaluating during ϕ_4, and one evaluating during ϕ_1. Some control signals are derived from inputs which aren't stable until later in the cycle or come from external sources latched in later phases; the base signals are generated by the ϕ_1 PLA and are modified by the delayed inputs using combinational logic.

The controller requires two states because the load and store instructions take two cycles to execute; during the first execution cycle the following instruction is read into the first stage of the distributed instruction register, and during the memory transaction cycle the controller must keep the opcode and control bits of the memory instruction latched in the second stage (busIR) so that it can control the memory transaction. The first stage is restored from the IR, but busIR requires an additional rollback memory because the two latches do not always contain the same data.

Two bits contain the current controller state and the previous instruction executed. These two bits need to be restored upon rollback. Two more controller states are needed for state repair; this is achieved by adding two more bits to the controller state (see section 2.5). These two bits don't need to be restored because during state repair, zeros are shifted into the *valid bits* of all the registers except for the register file, which receives the repaired data. Hence, the state from those two cycles can never be selected (except from the register file, and this is to cover erroneous data), and so the the *MP* cannot roll back to those two cycles.

The two source register fields in the *MP* instruction format indicate which registers are to be read out of the register file onto busA and busB. They are used right after the instruction is read in, and so they can be restored from the IR during the rollback cycle. However, the destination register field from the instruction may need to be held over a cycle for the load instruction. BusIR is used by the controller to latch information from the *load* or *store* instruction (including the destination register number) to be used during the memory data transaction. The RFTRAN reads the destination register from busIR, and as a result, is restored from the datapath controller during rollback.

More operation and implementation details can be found in [Lai90a].

2.1.6. External Interface

Table 2.2 shows the signals interfacing the *MP* with the outside world. These are categorized as power, clocks, I/O, interrupts, and error detection and recovery. Clocks, interrupts, and error detection are covered in sections 2.1.4, 2.2.8, and 2.3. I/O deals with memory transactions and the bus protocol.

The Berkeley RISC performs a memory access every cycle, and if there was a cache miss or memory delay, then the clocks are stopped until the problem is resolved. Stopping the clocks for an extended memory fetch is not an acceptable solution, especially with micro rollback, where every cycle, the system state must be checkpointed. In addition, the *MP* doesn't perform memory accesses during state repair.

We expect memory to be asynchronous with regard to the *MP*'s clocks, and so the *MP* supplies two control signals to manage the time-multiplexed address/data bus, and it expects a wait signal to indicate if the transaction must be delayed for any number of cycles. The address is put out during ϕ_4-ϕ_1, and the data is read in or written out during ϕ_2-ϕ_3.

Controls to the memory are the read/write pin (0/1), the data size pin (byte=00, halfword=01, word=10), the instruction/data pin (0/1), and the system mode pin (kernel=0, user=1). In order to simplify byte transactions, the *MP* performs its own data

Pin			Size	Type
Power	Vdd	Vdd	5	input
	GND	GND	6	input
Clocks	pad.csel	clock selection	1	input
	pad.phi	internal clock phase generation	1	input
	pad.sync	internal clock synchronization	1	input
	pad.phi1	external ϕ_1	1	input
	pad.phi2	external ϕ_2	1	input
	pad.phi3	external ϕ_3	1	input
	pad.phi4	external ϕ_4	1	input
I/O	pad.AD	address/data bus	33	i/o
	pad.rw	read/write	1	output
	pad.size	data size	2	output
	pad.enb.addr	memory address enable	1	output
	pad.enb.data	memory data enable	1	output
	pad.wait	wait	1	input
	pad.id	instruction/data	1	output
	pad.sysmode	system mode	1	output
Interrupts	pad.reset	reset	1	input
	pad.irr	interrupt request	1	input
	pad.ira	interrupt acknowledge	1	output
Error Detection and Recovery	pad.ms	master/slave	1	input
	pad.state	compressed internal state	4	output
	pad.rb	rollback	1	i/o
	pad.RB	rollback amount	3	i/o
	pad.repairAm	repair busA on master	1	master
	pad.repairAs	repair busA on slave	1	slave
	pad.repairBm	repair busB on master	1	master
	pad.repairBs	repair busB on slave	1	slave
	pad.shutdown	shutdown request	1	i/o
TOTAL			76	

Table 2.2: Mirror Pin Count

alignment from memory reads and writes. Table 2.3 shows the positions that words, halfwords, and bytes occupy in memory. When reading bytes and halfwords from memory, the *MP* only requests the word which the data is embedded in and internally aligns the data; memory is expected to ignore the two least significant bits of address (byte address) and the data size for this transaction. As for writing bytes and halfwords

data size	pad.size	padAD<1:0>	<31:24>	<23:16>	<15:8>	<7:0>
word	10	00	d	d	d	d
halfword	01	00			d	d
		10	d	d		
byte	00	00				d
		01			d	
		10		d		
		11	d			

Table 2.3: Data Size and Alignment

to memory, the *MP* aligns the data onto byte or halfword boundaries before sending it out so memory only has to determine which bytes to write based upon the byte address and data size. The instruction/data pin is used to determine whether to look in an instruction cache or not, and the system mode pin is used for memory protection.

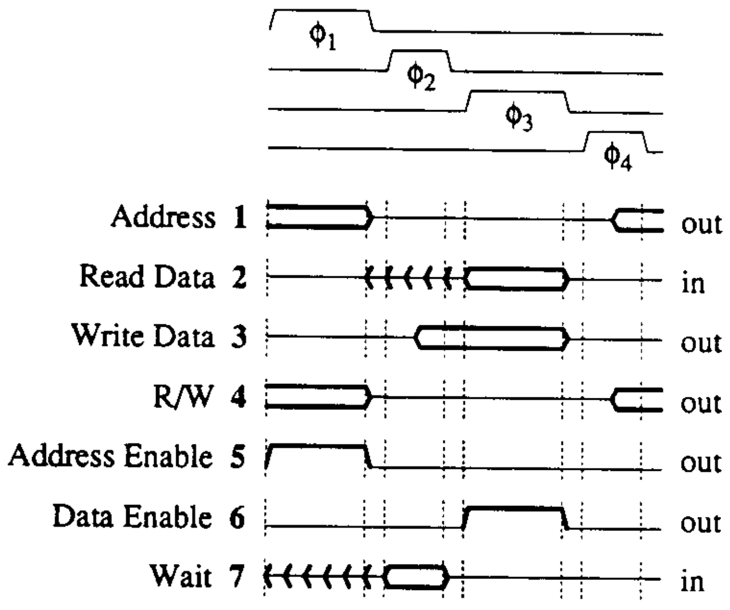

Figure 2.4: Memory Interface Timing

Figure 2.4 shows the memory timing. For a write cycle, the destination address (**1**) is put out during ϕ_4-ϕ_1 along with memory control signals (*e.g.* R/W (4), data size). The address enable signal (**5**) is asserted on the leading edge of ϕ_1 after all controls and data are stable and held until the falling edge of ϕ_1. Memory latches the address on the rising edge of the address enable signal, and in the event that it cannot respond in that cycle it must assert the wait line (7) within 25ns of the leading edge. The wait signal is latched on ϕ_2 and it determines whether the *MP* should go into a wait state; 0's are written into the *valid bits* of the the registers during all cycles that the wait line is asserted so that no new state is logically stored. (The data itself is stored into the register, but it will never be selected.) Once the wait line is released, the data enable signal (6) is asserted on the rising edge of ϕ_3 and held until the falling edge of ϕ_3. On ϕ_2-ϕ_3 the data (3) is put out to memory and is latched using the data enable signal.

For a read cycle, the address and memory enable pins are asserted as in the write cycle and the same protocol for wait states is observed. However, the trailing edge of the address enable signal (5) tells memory to start putting the data onto the bus (2) and to keep it there until the falling edge of the data enable signal. During wait states, the data won't be valid, but as long as it is correct by the setup time before the end of ϕ_3 there should be no problems.

2.1.7. Testing

Once the *MP* is fabricated, it must be tested to make sure that it is free of fabrication defects. To verify that the datapath works after fabrication, we considered putting in scan-in scan-out (SISO) latches[McCl86a]. However, this would entail either making just the register itself a SISO, or the register and its delayed write buffer into SISOs. The first option only adds overhead to the permanent register to make it controllable, but does not help with the DWBs. The second option allows full control, but at the expense of adding even more overhead to the space taken up by the DWBs. In any case, the internals of the *MP* are easily visible because of the changes made for error detection.

Instead, our strategy is to employ techniques found in [Brah84a] to test the processor under normal operation. The test procedure involves partitioning the processor into various logical units and using the instruction set to exercise these components to make sure everything is correct. Before we send the *MP* out for fabrication, we must thoroughly verify through simulation that it will work (see chapter 4 for details). Once the *MP* comes back, the only problems that should crop up will stem from manufacturing defects. Using the instruction set to test the chips will cull out the bad ones.

If there is a defect in the pads or the datapath controller, then we should be able to know right away that the chip is bad because it won't be able to execute any instructions at all. If there is are problems with the controller, but the processor still correctly executes most instructions, then we can proceed to test the chip on the assumption that the controller is functional. Later on in the testing, if the chip should fail any of the tests, it will not matter what the source of the problem is because we just want to know if the chip is defective.

In order to test the error detection and rollback circuitry, we added test instructions to the *MP* instruction set (see section 2.2.6). These instructions are used to verify that the chip works after fabrication. Since they are just instructions and require no special hardware setup, they may also be used for run-time testing of the *MP*.

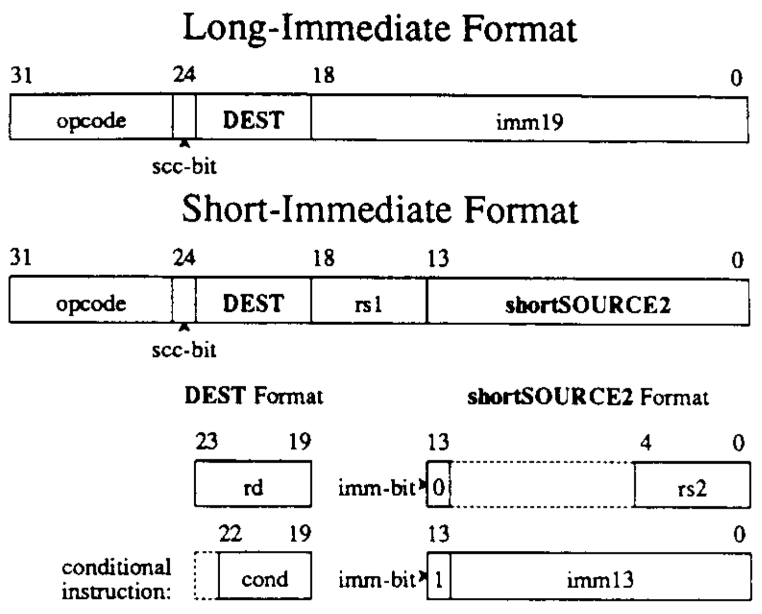

Figure 2.5: Mirror Instruction Format

2.2. Normal Operation

This section describes the operation of the *MP* without use of error detection, micro rollback, nor state repair. We describe the *MP* instruction formats and show how each instruction exercises the datapath.

2.2.1. Instruction Format

Figure 2.5 shows the *MP* instruction formats and table 2.4 shows the various instructions. The **l** in the opcode table indicates whether the instruction is to be decoded as a long-immediate format instruction or a short-immediate format instruction. The **p** in the opcode table indicates a privileged instruction; executing this instruction with the processor in user mode (S=1) causes a trap. The **t** in the opcode table indicates a test instruction used for run-time testing of micro rollback and error detection hardware. The **c** indicates that instruction may cause a branch to the address calculated from the source

	000xxxx	001xxxx	010xxxx	011xxxx
xxx0000				
xxx0001	calli(**p**)	sll	ldrbpm(**lpt**)	strbdm(**lpt**)
xxx0010	getpsw	sra		
xxx0011	getlpc(**p**)	srl	ldrbps(**lpt**)	strbds(**lpt**)
xxx0100	putpsw(**p**)	ldhi(**l**)		
xxx0101		and		
xxx0110	clrrbm(**pt**)	or	ldxw	stxw
xxx0111	clrrbs(**pt**)	xor	ldrw(**l**)	strw(**l**)
xxx1000	callx	add	ldxhu	
xxx1001	callr(**l**)	addc	ldrhu(**l**)	
xxx1010	jmprbm(**lpt**)	addbpm(**pt**)	ldxhs	stxh
xxx1011	jmprbs(**lpt**)	addbps(**pt**)	ldrhs(**l**)	strh(**l**)
xxx1100	jmpx(**c**)	sub	ldxbu	
xxx1101	jmpr(**cl**)	subc	ldrbu(**l**)	
xxx1110	ret		ldxbs	stxb
xxx1111	reti(**p**)		ldrbs(**l**)	strb(**l**)

Empty boxes are illegal opcodes.
Opcodes with high bit set (1xxxxxx) are illegal opcodes.

c	Conditional instruction; DEST-field is cond.
l	Long-immediate format instruction.
p	Privileged instruction.
t	Test instruction.

Table 2.4: Mirror Opcodes

registers if the condition codes in the PSW meet certain conditions; the **DEST** field is interpreted as a condition code for conditional instructions and as the destination register otherwise.

Within the short-immediate format, the **shortSOURCE2** field is interpreted as source register rs2 if the immediate bit is 0, and as a 13-bit sign extended to 32-bits immediate value if it is set to 1. If the set condition code bit (scc) is 1, then the condition codes generated by the ALU during the execution cycle will be saved into the PSW.

For the instruction descriptions, s1 refers to the value of rs1 in the register file, s2 refers to the value of **shortSOURCE2**, and d refers to the destination in the register file indexed by rd unless it is used as a source, in which case it is the value in that location.

2.2.2. Register to Register Instructions

The arithmetic, logical, and shift instructions fetch their operands from the register file or use imm13, perform the appropriate operation, and store the result back into the register file. The instructions are shown in table 2.5. (The Berkeley RISC also has inverted versions of subtract, subi and subci, but removing those simplified ALU implementation.)

	opcode	operation	description
arithmetic:	add	$d \leftarrow s1 + s2$	add
	addc	$d \leftarrow s1 + s2 + c$	add with carry
	sub	$d \leftarrow s1 - s2$	subtract
	subc	$d \leftarrow s1 - s2 - \bar{c}$	subtract with carry
logical:	and	$d \leftarrow s1 \wedge s2$	bitwise and
	or	$d \leftarrow s1 \vee s2$	bitwise or
	xor	$d \leftarrow s1 \oplus s2$	bitwise xor
shift:	sll	$d \leftarrow s1 << s2$	logical shift left by s2 mod 32
	sra	$d \leftarrow s1 >> s2$	arithmetic shift right by s2 mod 32
	srl	$d \leftarrow s1 >> s2$	logical shift right by s2 mod 32

Table 2.5: Register to Register Instructions

Internally, subtraction is performed as addition using the complemented value of the subtractor. The ALU's manchester carry chain produces carries from each bit pair for the next pair ($c_1 \cdots c_{32}$). If the scc bit is 1, then the condition codes in the psw will be generated for arithmetic operations as $z \leftarrow d = 0$, $n \leftarrow d<31>$, $v \leftarrow c_{31} \oplus c_{32}$, and $c \leftarrow c_{32}$. For logical and shift operations the condition codes will be set to $z \leftarrow d = 0$,

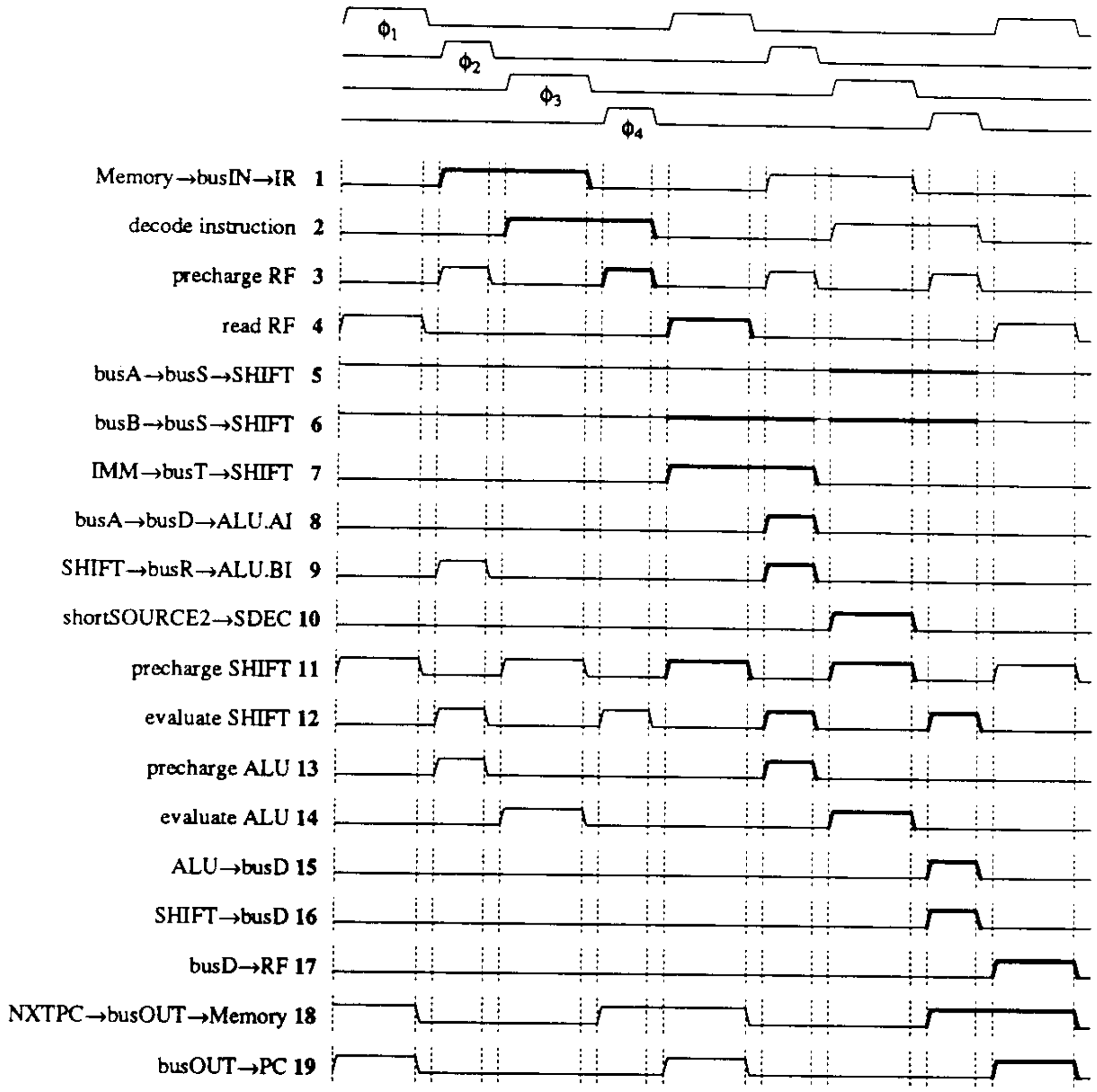

Figure 2.6: Register to Register Instruction Execution

n ← d<31>, v ← 0, and c ← 0.

Figure 2.6 shows the timing diagram of ALU and shift instruction execution. The instruction comes into the chip from the address/data bus (1) and is decoded for the execution cycle (2). If the instruction is an ALU operation, the immediate bit determines whether busB (6) or the immediate (7) goes to the shifter (12) after the register file operand read (**4**). A shift instruction puts busA onto busS instead (**5**), and shortSOURCE determines the shift amount (**10**). The shifter evaluates ϕ_2 to fill the ALU's B latch (**9**), while busA is gated onto busD and into the ALU's A latch (**8**). Both the ALU and the shifter evaluate every cycle (**14 12**), but only one gates out onto busD for the appropriate instruction (**15 16**). The result is written into the register file's DWB (**17**). Meanwhile, the address of the instruction to be executed two cycles down is put onto the address/data bus for the instruction fetch next cycle (**18**) and the program counter is updated (**19**).

2.2.3. Load Instructions

The *MP* uses a time-multiplexed address/data bus to perform one memory transaction per cycle. This means that during the read cycle of a load instruction, the *MP* cannot be fetching another instruction, and so it must suspend the execution pipeline until the memory cycle has completed. The ten load instructions are shown in table 2.6. If the scc flag is 1, then the condition codes saved will be $z \leftarrow d = 0$, $n \leftarrow d<31>$, $v \leftarrow 0$, and $c \leftarrow 0$.

opcode	operation	description
ldxw	d ← M[s1 + s2]	load absolute word
ldrw	d ← M[PC + sxt(imm19)]	load pc relative word
ldxhu	d ← M[s1 + s2]<15:0>	load absolute halfword unsigned
ldrhu	d ← M[PC + sxt(imm19)]<15:0>	load pc relative halfword unsigned
ldxhs	d ← sxt(M[s1 + s2]<15:0>)	load absolute halfword signed
ldrhs	d ← sxt(M[PC + sxt(imm19)]<15:0>)	load pc relative halfword signed
ldxbu	d ← M[s1 + s2]<7:0>	load absolute byte unsigned
ldrbu	d ← M[PC + sxt(imm19)]<7:0>	load pc relative byte unsigned
ldxbs	d ← sxt(M[s1 + s2]<7:0>)	load absolute byte signed
ldrbs	d ← sxt(M[PC + sxt(imm19)]<7:0>)	load pc relative byte signed

Table 2.6: Load Instructions

Figure 2.7 shows the timing diagram of load instruction execution. The load instruction comes into the chip from the address/data bus (**1**) and is decoded for the execution cycle (**2**). If the load is from an absolute address, then the short-immediate format is used and the immediate bit determines whether busB (**5**) or imm13 (**6**) goes to the shifter (**11**) after the register file read (**4**); a PC relative load puts imm19 into the shifter (**6**). The shifter evaluates during ϕ_2 to fill the ALU's B latch (**9**) while busA is gated onto busD and into the ALU's A latch (**7**) for an absolute load; a PC relative load gates the PC onto busD (**8**). The ALU calculates the address (**13**) and passes it out on the address/data bus (**14**) while the controller initiates a read cycle. The byte address calculated in the BAR is sent to the shift decoder (**16**) and the DIMM along with the size and sign in preparation for the incoming data so it can be properly aligned and sign extended. The data comes in and is sign extended in the DIMM (**15**), right aligned through the shifter (**11 17**), and stored into the register file (**18**) during the read cycle. Note that during the read cycle, instruction fetches are suspended (**19 20**).

2.2.4. Store Instructions

Like load instructions, stores also suspend the execution pipeline during the write cycle because no instruction can be fetched. The six store instructions are shown in table 2.7. The scc bit should not be set, but if it is, then the condition codes saved will be $z \leftarrow ?$, $n \leftarrow ?$, $v \leftarrow 0$, and $c \leftarrow 0$.

Figure 2.8 shows the timing diagram of store instruction execution. The store instruction comes into the chip from the address/data bus (**1**) and is decoded for the

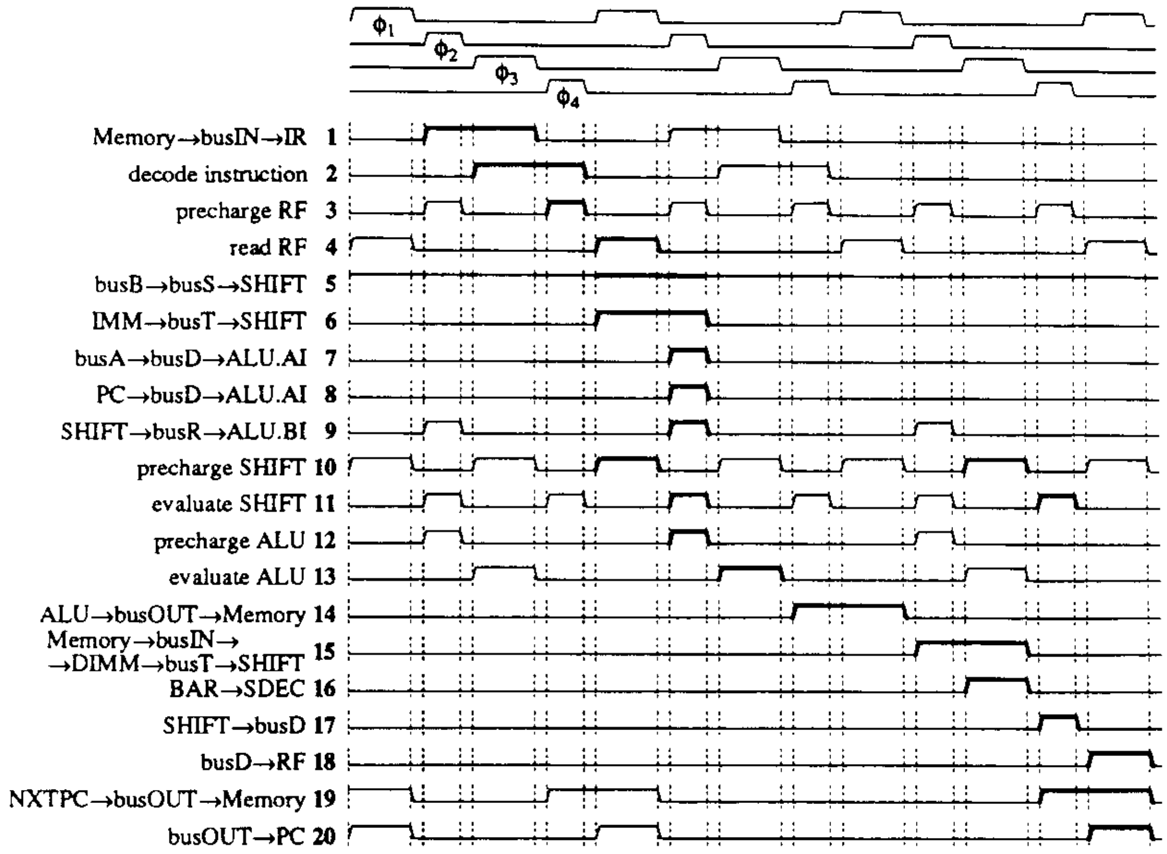

Figure 2.7: Load Instruction Execution

opcode	operation	description
stxw	d → M[s1 + s2]	store absolute word
strw	d → M[PC + sxt(imm19)]	store pc relative word
stxh	d → M[s1 + s2]<15:0>	store absolute halfword
strh	d → M[PC + sxt(imm19)]<15:0>	store pc relative halfword
stxb	d → M[s1 + s2]<7:0>	store absolute byte
strb	d → M[PC + sxt(imm19)]<7:0>	store pc relative byte

Table 2.7: Store Instructions

execution cycle (2). Stores are an exception to the instruction format in the sense that rd is used to read onto busB rather than rs2 (4) because only two values can be read out of the register file at a time. Although it is possible to use rs2 for an absolute store, this will give a bad address (because rd will actually be used instead of rs2), and so imm13 should always be used (**5**). A PC relative store presents no problems and puts imm19 into the shifter (**5**). The shifter evaluates ϕ_2 to fill the ALU's B latch (**8**) while busA is gated onto busD and into the ALU's A latch (**6**) for an absolute store; a PC relative store gates the PC onto busD (**7**). The ALU calculates the address (**14**) and passes it out on the address/data bus (**15**) while the controller initiates a write cycle. In the meantime, the byte address calculated in the BAR is sent to the shift decoder (**9**) so that the data to be stored (**10**) can be aligned with word boundaries (**12**), put in the SDR (**16 17**), and eventually memory (**18**). Note that during the write cycle, instruction fetches are suspended (**19 20**).

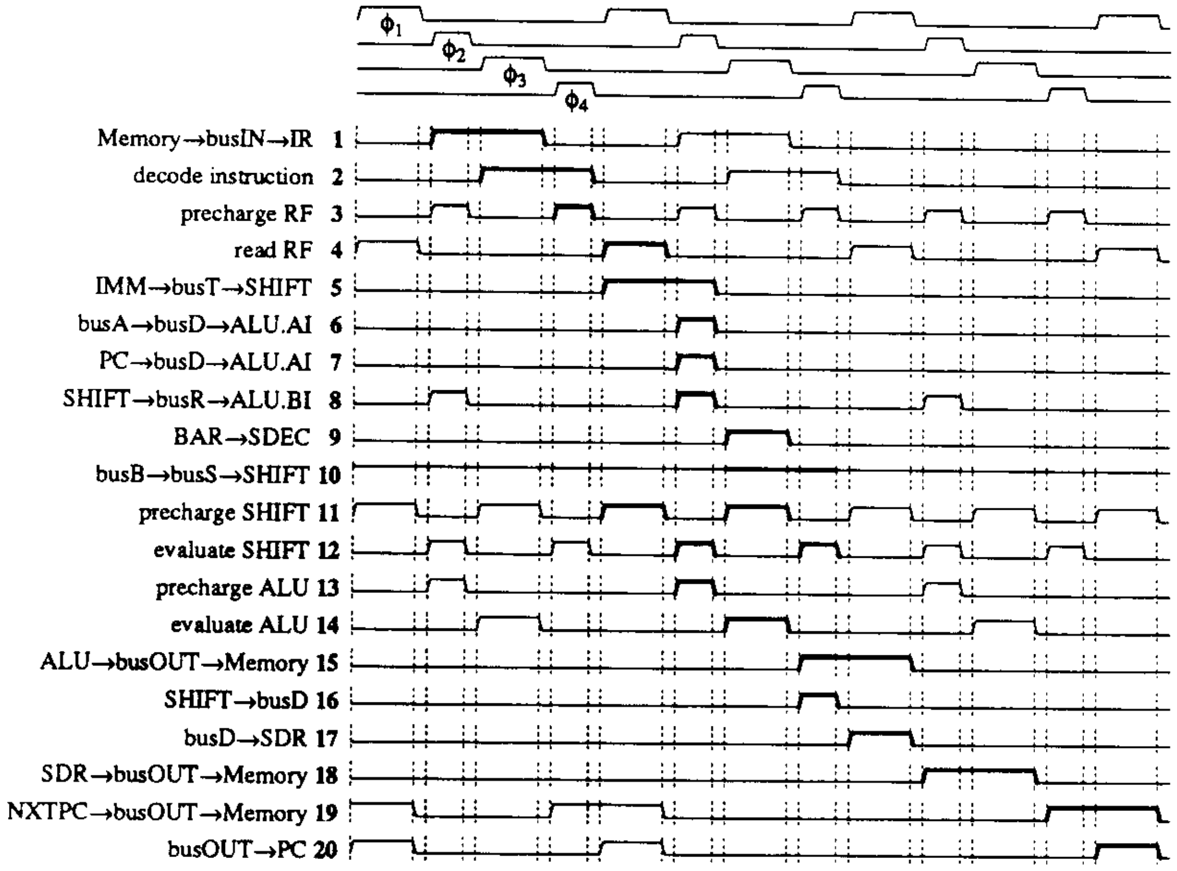

Figure 2.8: Store Instruction Execution

2.2.5. Branch Instructions

Instruction addresses put out the current execution cycle fetch the instruction the next cycle, which is then executed the following cycle; any changes to instruction sequencing do not take effect for two cycles. The *MP* uses this delayed control transfer due to its two-cycle execution pipeline. Table 2.8 shows the branch instructions. Jumps are conditional and use the conditions in table 2.9. Call and return instructions decremented and incremented the current window pointer in addition to changing the flow of execution as a hardware assist for procedure calls. The Berkeley RISC allowed conditional returns, but ran into timing difficulties determining if it should return and if this would cause a trap; we encountered the same problems, and as a result implemented unconditional returns. The return from interrupt, in addition to incrementing the current window pointer, also enables interrupts (I=1) and restores the system mode bit (S=P) in the PSW. The scc bit should not be set for the branch instructions, but if it is, then the condition codes saved will be z $\leftarrow$ X, n $\leftarrow$ X, v $\leftarrow$ X, and c $\leftarrow$ X (where X may be 0 or 1 because two different logic levels (0V and 5V) may be gated onto the bus used for generating the next cycle's condition codes, resulting in an indeterminate value (2.5V)). In addition, the effective address for conditional instructions (jump) will be unknown.

opcode	operation	description
jmpx	PC $\leftarrow$ s1 + s2	jump absolute
jmpr	PC $\leftarrow$ PC + sxt(imm19)	jump pc relative
callx	CWP–; d $\leftarrow$ PC; PC $\leftarrow$ s1 + s2	call absolute
callr	CWP–; d $\leftarrow$ PC; PC $\leftarrow$ PC + sxt(imm19)	call pc relative
ret	PC $\leftarrow$ s1 + s2; CWP+	return from call
reti	PC $\leftarrow$ s1 + s2; CWP+; PSW(I,S)	return from interrupt

Table 2.8: Branch Instructions

code	symbol	name	value
0000	nev	never	0
0001	gt	greater than (signed)	$\overline{(n \oplus v) \vee z}$
0010	le	less or equal (signed)	$(n \oplus v) \vee z$
0011	ge	greater or equal (signed)	$\overline{n \oplus v}$
0100	lt	less than (signed)	$n \oplus v$
0101	hi	higher than (unsigned)	$\overline{\bar{c} \vee z}$
0110	los	lower or same (unsigned)	$\bar{c} \vee z$
0111	lo nc	lower than (unsigned) no carry	$\bar{c}$
1000	his c	higher or same (unsigned) carry	c
1001	pl	plus (signed)	$\bar{n}$
1010	mi	minus (signed)	n
1011	ne	not equal	$\bar{z}$
1100	eq	equal	z
1101	nv	no overflow (signed)	$\bar{v}$
1110	v	overflow (signed)	v
1111	alw	always	1

Table 2.9: Mirror Condition Codes

Figure 2.9 shows the timing diagram of branch instruction execution. The branch instruction comes into the chip from the address/data bus (1) and is decoded for the execution cycle (2). If the branch is to an absolute address, then the short-immediate format is used and the immediate bit determines whether busB (5) or imm13 (6) goes to the shifter (11) after the register file read (4); A PC relative branch puts imm19 into the shifter (6). The shifter evaluates ϕ_2 to fill the ALU's B latch (9) while busA is gated onto busD and into the ALU's A latch (7) for an absolute branch, and the PC is sourced for a PC relative branch (8). The ALU calculates the instruction address two cycles away (13) and passes it out on the address/data bus (17) and updates the PC (19) for unconditional branches and branches fulfilling conditions. Conditional branches failing conditions do not branch and default to the PC for the instruction fetch (**18 19**). The current window pointer is effectively changed on ϕ_3 for calls and returns (**14**) so that for call instructions, the PC can be saved in a new register window (**15 16**).

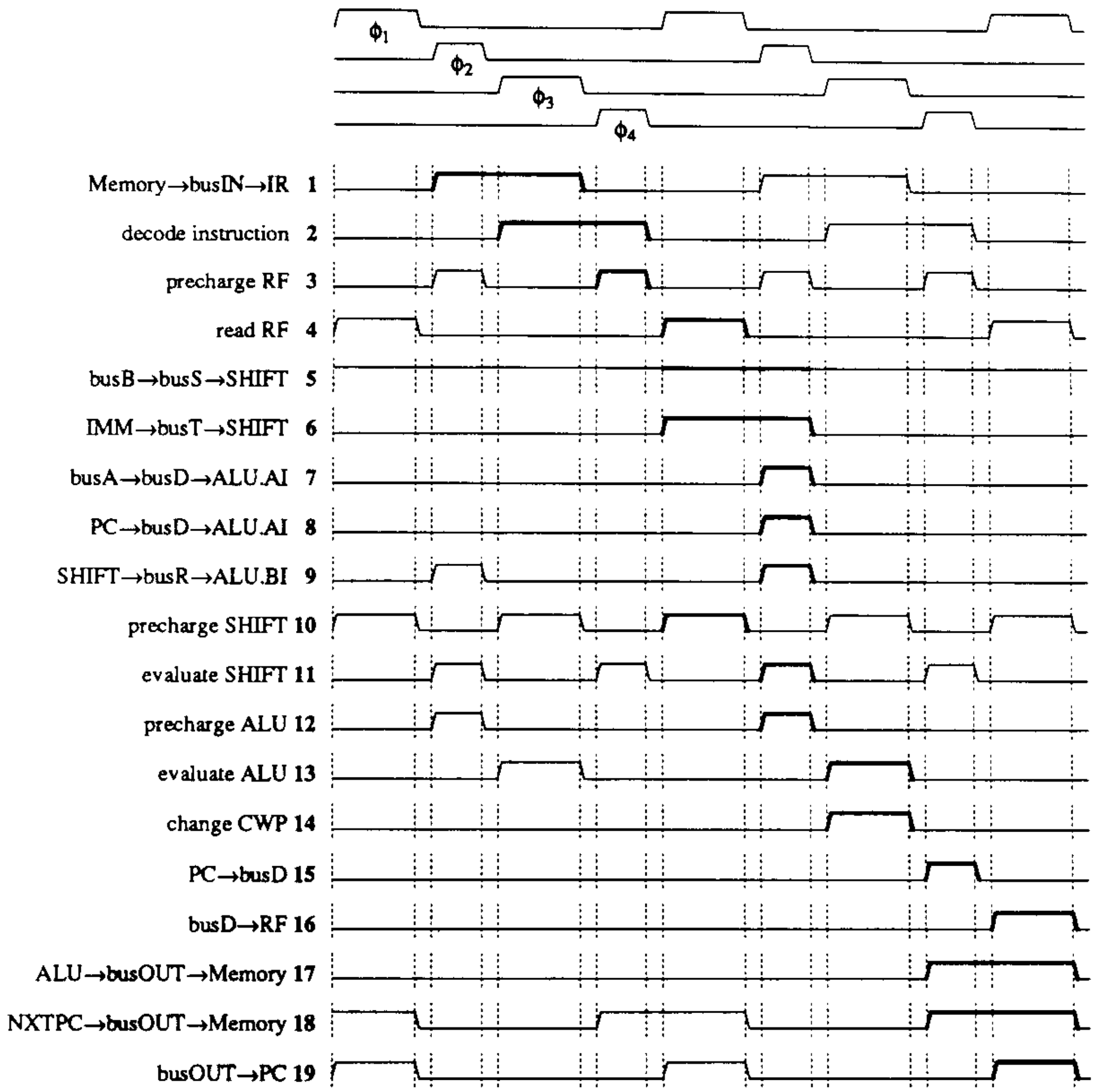

Figure 2.9: Branch Instruction Execution

2.2.6. Test Instructions

The following instructions were added for run-time testing of the error detection and micro rollback circuitry. They operate as variations of the standard instructions, only they intentionally inject errors into the system. Table 2.10 summarizes the instructions and is followed by a more detailed description.

opcode	operation	description
clrrbm	rb $\leftarrow$ 0	clear rollback bit on master
clrrbs	rb $\leftarrow$ 0	clear rollback bit on slave
jmprbm	d $\leftarrow$ s1; PC $\leftarrow$ PC + s2	jump if rollback bit is set, master
jmprbs	d $\leftarrow$ s1; PC $\leftarrow$ PC + s2	jump if rollback bit is set, slave
addbpm	d $\leftarrow$ s1 + s2; $\bar{p}$	add with bad parity on master
addbps	d $\leftarrow$ s1 + s2; $\bar{p}$	add with bad parity on slave
ldrbpm	d $\leftarrow$ M[PC + sxt(imm19)]; $\bar{p}$	load with bad parity on master
ldrbps	d $\leftarrow$ M[PC + sxt(imm19)]; $\bar{p}$	load with bad parity on slave
strbdm	d $\rightarrow$ M[PC + sxt(imm19)]	store bad data, master
strbds	d $\rightarrow$ M[PC + sxt(imm19)]	store bad data, slave

Table 2.10: Test Instructions

- *clrrbm, clrrbs*: After every rollback, the rollback bit in the controller is set as an indication that a rollback has occurred. This bit can only be reset by executing the *Clear Rollback Bit* instruction. *Clrrbm* resets the rollback bit on the master processor and does nothing on the slave processor, while *clrrbs* acts on the slave processor and ignores the master processor.

- *jmprbm, jmprbs*: *Jump if Rollback Bit is Set* is used to force a state compression error, and in addition can be used to verify that a rollback has occurred. If the rollback bit is set, then a PC relative jump is taken using the value of rs2 as the offset, and s1 is stored in the register file (PC $\leftarrow$ PC + rs2; d $\leftarrow$ s1). If the rollback bit is not set, then rs1 is placed onto busD of one processor and rs2 is placed onto busD of the other. If we choose the values in rs1 and rs2 so that the four-bit signature does not match, this will cause a comparison error to be detected the following cycle, and rollback will ensue followed by re-execution of the branch instruction. The second time around, the rollback bit will be set and so the branch will be taken. It is only practical for rs2, the branch offset, to assume several small values, and so there are two variations of the branch instruction to allow either processor to gate any value from rs1 onto busD to fully test the state compression logic. *Jmprbm* causes the master processor to gate rs1 onto busD and the slave processor to gate rs2 onto busD, and vice versa for *jmprbs*. If we choose values for rs1 and rs2 such that the four-bit signatures match, then we can use this instruction solely for testing if a rollback has occurred. This application is used in conjunction with the other test instructions to verify that the subsystem tested actually did cause a rollback.

- *addbpm, addbps*: *Add with Bad Parity* is used to inject a parity error into the register file in order to test state repair. The instruction acts like the normal *add* instruction, only the inverted parity of the sum is stored into the register file. *Addbpm* causes only the master processor to invert the parity, and *addbps* causes the slave to store bad parity.

- *ldrbpm, ldrbps*: *Load with Bad Parity* functions like *ldrw*, except that one of the processors will invert the parity of the data read in as it is gated onto busIN if the rollback bit is not set. *Ldrbpm* flips the parity on the master processor and *ldrpbs* flips the parity on the slave processor. After the parity error on the incoming data is detected, a rollback will set the rollback bit, and the parity bit will not be changed the second time the instruction is executed.

- *strbdm, strbds*: *Store Bad Data* acts like *strw*, only one of the processors will attempt to store the contents of the MAR instead of busD if the rollback bit is clear. This will cause a comparison error on the pad checkers, and rollback will follow. On the second execution of the store instruction, the rollback bit will be set, and so both processors will store the contents of busD and there will be no error. For *strbdm*, the master processor stores the MAR, while *strbds* causes the slave processor to store the MAR.

2.2.7. Miscellaneous Instructions

Miscellaneous instructions not fitting any of the above categories are shown in table 2.11. They are all similar in timing to the register to register instructions.

opcode	operation	description
calli	CWP–; d ← LSTPC	call interrupt
getpsw	d ← PSW	get psw
getlpc	d ← LSTPC	get last pc
putpsw	PSW ← s1 + s2	put psw
ldhi	d ← imm19 << 13	load immediate into bits 31:13

Table 2.11: Miscellaneous Instructions

Referring to the branch timing diagram (figure 2.9), the *calli* instruction decrements the register window pointer (**14**) and stores the LSTPC into the destination register (**15 16**). This instruction is used by the hardware for handling interrupts and traps and should not be used by software. If the scc bit is set, then the condition codes stored will be $z \leftarrow ?$, $n \leftarrow ?$, $v \leftarrow 0$, and $c \leftarrow 0$.

Looking at the register to register timing diagram (figure 2.6), *getpsw* gates the PSW onto busD during ϕ_4 (**15 16**) to be written into the register file (**17**). Condition codes are $z \leftarrow ?$, $n \leftarrow ?$, $v \leftarrow 0$, and $c \leftarrow 0$.

Because the PSW is only drives its 11 bits onto busD, the top 21 bits are whatever was placed upon busD from the previous cycle. In normal operation, this would be no problem because the same data would be present on both the master and the slave processors. However, the *jump if rollback bit is set* test instruction forces an internal state comparison error by placing different values onto busD of the master and the slave processors if the rollback bit is not set. If the intent of the test instruction is to force a rollback, then after the rollback, the values placed upon busD of both processors will be the same and so there will be no problem. The only case where this will be a problem is if the test instruction is used to detect if the rollback bit has been set by placing two different values onto busD whose compressed state matches. If a *getpsw* instruction follows, then the two resulting values on busD will not match. The problem can be remedied by either not following any *jump if rollback bit is set* instructions with a *getpsw* instruction, or if the test instruction is used only to detect if a previous rollback has occurred, to gate the same value onto busD of both processors.

Getlpc operates the same way as *getpsw*, only it gates out LSTPC from the PC onto busD on ϕ_4 (**15 16**). Condition codes are $z \leftarrow d = 0$, $n \leftarrow d<31>$, $v \leftarrow 0$, and $c \leftarrow 0$. The *getlpc* instruction is used as the first instruction of every interrupt handler in order to save the LSTPC; when the *getlpc* instruction is executed, the LSTPC is the address of the instruction following the interrupted instruction. The *calli* instruction saves the address of the interrupted instruction, and so the way to return from the interrupt handler is to perform a *jmpx* to the address of the interrupted instruction followed by a *reti* to the address of the instruction following the interrupted instruction. The *jmpx* refetches the interrupted instruction, the *reti* in the *jmpx*'s delayed slot fetches the following instruction, and the interrupted instruction restarts execution in the *reti*'s delayed slot.

Putpsw is like an add instruction, only instead of the destination being the register file, busD is written into the PSW during ϕ_1 (**17**). The scc bit should not be set because that would put the PSW in a write-write conflict and the results are undefined.

Ldhi takes imm19 as data and shifts it on ϕ_4 (**12**) onto busD (**16**) and into the register file (**17**) with imm19 occupying bit positions <31:13> and bit positions <12:0> being zero filled. This is accomplished by having the IMM gate onto busT left shifted by 13. The right shift onto busD is by zero. (Immediates used during ϕ_2 are right shifted by 13 so that the resulting value is properly aligned.) Condition codes are $z \leftarrow d = 0$, $n \leftarrow d<31>$, $v \leftarrow 0$, and $c \leftarrow 0$.

Type	Cause	Vector
interrupt	reset	00000000_{16}
trap	illegal instruction	00000000_{16}
trap	privileged instruction	00000000_{16}
trap	address misalignment	00000000_{16}
trap	bad shift amount	00000000_{16}
interrupt	interrupt request	00000010_{16}
trap	register window overflow	00000020_{16}
trap	register window underflow	00000030_{16}
interrupt	shutdown	00000040_{16}

Table 2.12: Interrupt Vectors

2.2.8. Interrupts

In the event of a trap or an interrupt with the interrupt enable bit set, the currently executing instruction is cancelled (by not writing the results into the registers) and the fetched instruction is discarded. Instead, a hard-wired *calli* instruction with rd=25 and scc=0 is read into the instruction register and controller; its execution is similar to a *call* instruction except that the ALU does not gate out the address, but instead, the PC is used to fetch the next consecutive instruction.

The interrupt signal also causes the interrupt enable bit in the PSW to be disabled, the *MP* to assume kernel mode, and the previous system mode bit to be saved (PSW(I=0, S=0, P=S)). During ϕ_4, the interrupt vector address is put out so the *MP* can start executing the interrupt handler after the *calli*. If the cause was an external interrupt, then the interrupt request acknowledge line (pad.ira) is asserted to indicate receipt of the signal.

The interrupt causes and corresponding vectors are shown in table 2.12. Address alignment is caused by a word operation not being on a word boundary or a halfword operation not being on a halfword boundary. The shifter only allows shifts by six amounts, and so anything other than those causes a bad shift amount trap. A register window overflow trap can be caused by a call instruction if the new current window pointer equals the saved window pointer; this means all available register windows are taken and it's time to spill a window to memory to make sure there's always a free window around. Likewise, a register window underflow trap can be caused by a return instruction if the new current window pointer equals the saved window pointer; the new window is in memory and needs to be restored. Reset and shutdown are treated like interrupts, only reset has highest precedence and shutdown has next highest.

The implementation of the datapath controller requires that interrupts and traps be generated by the end of ϕ_2 so that the internal interrupt signal can be ready by the beginning of ϕ_3 to initiate the interrupt process. Unfortunately, the bad shift amount and address misalignment signals cannot be resolved until the middle of ϕ_3. However, the write signal to all the registers can be delayed until ϕ_4 and the memory address enable signal is not asserted until the falling edge of ϕ_4; the solution is to cancel all register writes and memory transactions which may occur and to invoke the trap the following cycle for these two events.

2.3. Error Detection Scheme

The *MP* performs error detection by running two processors, a master and a slave, in lockstep. A difference between the observable state of the two chips indicates that an error has occurred and the checker signals rollback to the cycle before the cycle the error occurred. The master processor performs all transactions with the memory and the rest of the system while the slave processor reads in the master's visible states and compares them with its own independently generated values.

The slave processor is silent except in the event of a discrepancy, and so it is conceivable to configure a system with one master and multiple slaves. This would work fine except for the present scheme of state repair which requires exactly two processors.

In order to minimalize the number of chips in our system, the checkers must be on-chip one of the processors. The master and slave processor datapaths are identical; the only difference is that the slave has comparators and doesn't drive values on its pins. We can take advantage of this common denominator by implementing one chip that can be configured as a master or a slave by setting a pin (pad.ms) to 0 or 1. In fact, the area taken by this two chip system can be further reduced by taking two chips and bonding all pads identically except for the master/slave pin, which is bonded to different pins for the master and slave processor packages. The master and the slave can now be mounted piggy-back, and thus, present the PCB footprint of a single processor.

Portions of a processor's state are not externally visible, and so in order to detect internal errors within a few cycles instead of many cycles later, the *MP* must export this internal state to the checker. A total of fifty-five bits of internal state from all the modules within the *MP* must be compared between the master and the slave processors. Dedicating that many pins for error detection is not a reasonable solution; we reduce the number exported to four by using data compression techniques. The internal state is reduced to four bits of interleaved parity by taking the state from each module and calculating the parity of every fourth bit to obtain four bits of compressed state. These four-bit bit vectors from all the modules are then all XORed together to obtain one four-bit bit vector for internal state exportation. Four bits are enough to show some differences should the *MP* be hit by multiple bit errors, yet not so many as to consume a sizable proportion of the pin allocation.

During rollback, a value read out of a rollback memory may have been corrupted. A solution would be to protect it with a parity bit, but this turns out to be unnecessary as all state will be checked within a few cycles, and so the problem will be caught by the error checkers and another rollback will be signaled to a cycle before the bad data within the rollback memory.

State comparison of some modules must be temporarily suspended during rollback because the state between the two processors are not expected to be identical. During state repair, the bad data from the processor needing to be repaired will be read out again from the register file, and if this error is not masked, the *MP* will reinitiate rollback and state repair instead of proceeding with the current state repair.

The additions to the *MP* datapath for error detection are shown in figure 2.10. The external state comparators (CMP), the internal state compressors (parD, PSW, RFTRAN), and the parity generators (parIN, parB, parD, parOUT) will be explained in the following three sections.

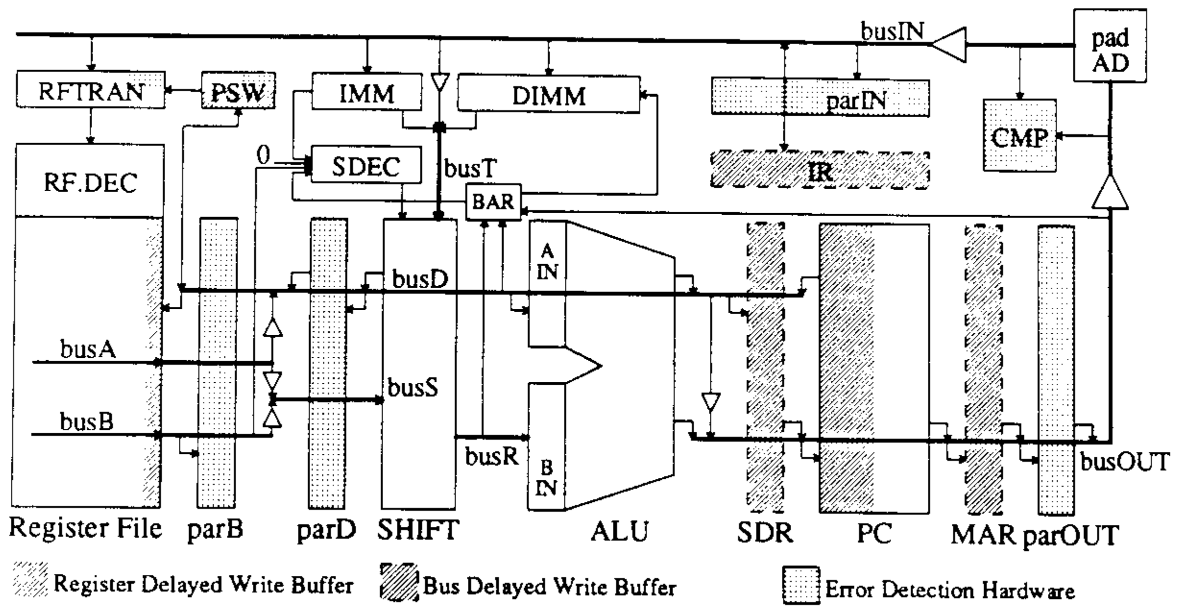

Figure 2.10: Mirror Datapath Additions for Micro Rollback

2.3.1. External State Comparison

The slave compares the output pins shown in the I/O section of table 2.2 and the interrupt acknowledge pin. The error detection and recovery pins are not checked because an error may be detected on one processor and not the other and then they would differ, whereas pins associated with normal Operation should always match.

A pad comparator is merely an XOR sourcing from a pad's input and outputs (see figure 2.11). On the master processor, the output pads will be enabled, and so the input should be identical to the output and no errors will be signaled by the master. The slave's pads are disabled, and so input from the pad comes from the master; this configuration forms an active comparator on the slave processor.

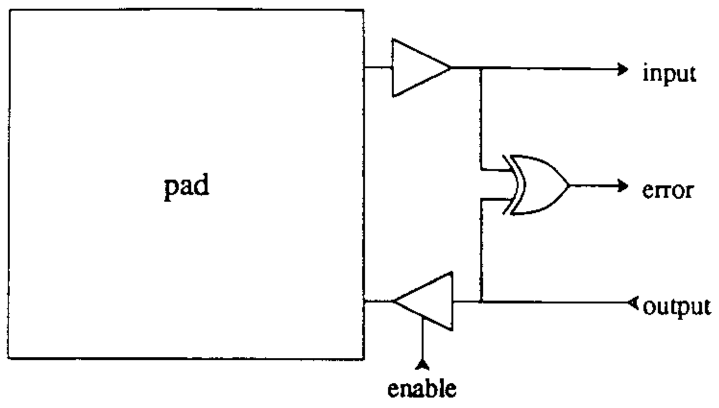

Figure 2.11: Output Comparator

2.3.2. Internal State Comparison

Comparing only externally visible state may not catch all errors that develop on-chip because the fault may not propagate off-chip for many cycles. To ensure early detection of errors, the *MP* exports internal state for comparison. We only have to look at registers which contain the results of an executed instruction and whose values cannot be verified directly by external state comparison.

Possible register candidates are the RF, SDR, PSW, PC, MAR, the datapath controller state, and the distributed IR. Data written into the SDR, PC and MAR is also put out to the pads, where any errors can be caught by external comparison. Any changes in the distributed IR will cause the controller to behave differently if the error is in the opcode; this can be caught by external state comparison. If it is the data portion that is changed, then the result will propagate to the register file or the PSW where it will be exported. This leaves the register file, the PSW, and the controller for internal state exportation.

Input ports to the register file are busD (32 bits) for the data, the register file translator (7 bits) for the address, and the register file write signal (1 bit). All these values must be monitored because once data is written into the DWB, it possibly may not be accessed and compared again until it is committed into the permanent register file. The value of the PSW for the next cycle is calculated every cycle and written into the PSW's DWB; we must add the 11 bits from the PSW to the checking list. Normal controller operation uses two bits of state; adding the two bits used for state repair gives a total of four bits to be compared from the controller.

Four-way interleaved parity generators are installed in the PSW and RFTRAN as indicated by the stipples in figure 2.10. In addition to generating the parity of busD, parD also generates the four-way interleaved parity of busD. The four bits of compressed state from these three modules, along with the four bits of controller state, are merged together to form the four bits of exported internal state by XORing together the four four-bit bit vectors.

2.3.3. Parity Checking

The *MP* bus protocol includes supplying parity to protect the address/data bus during memory transactions; this entails checking the incoming parity on busIN with parIN and generating parity to accompany the data on busOUT using parOUT. ParIN serves a dual purpose during rollback by checking that the data read out of the IR during rollback has not been corrupted.

ParD and parB are used to generate the parity of the data read out of the register file onto busA and busB. This parity, along with the parity read out of the register file, is sent to controller for error detection and state repair determination. ParD also generates the parity written into the register file to be used for error detection and state repair.

During state repair, the *MP* transfers data from the processor with good data to the processor with bad data. Data comparison of the data busses is not in effect at this point because the two processors do not have the same state. The transferred data is verified the same way as with a memory transaction, by using parOUT and parIN and sending one bit of parity with the data.

2.3.4. Rollback Arbitration

Once an error is detected, the *MP* initiates rollback to the cycle before the cycle that the error occurred. Any chip in the fault-tolerant system may signal a rollback, and we simplify interconnection of the rollback signals (rollback, rollback amount, and shutdown) by using wired-ORs.

However, two chips may detect errors with different detection latencies and will signal rollback to two different cycles at the same time. Simply ORing the two amounts together will not work because the result may be much larger than either one (*e.g.* 011 OR 100 = 111). In this case, we want the system to roll back the largest amount requested so that all errors may be undone. We can use the same arbitration scheme used by the Futurebus protocol[Taub84a] for determining the next bus master. The chips that want control of the system bus put their priority number on the arbitration bus and asynchronously determine the highest priority. At the end of arbitration, the arbitration bus contains the winning processor's priority number. Likewise, at the end of rollback

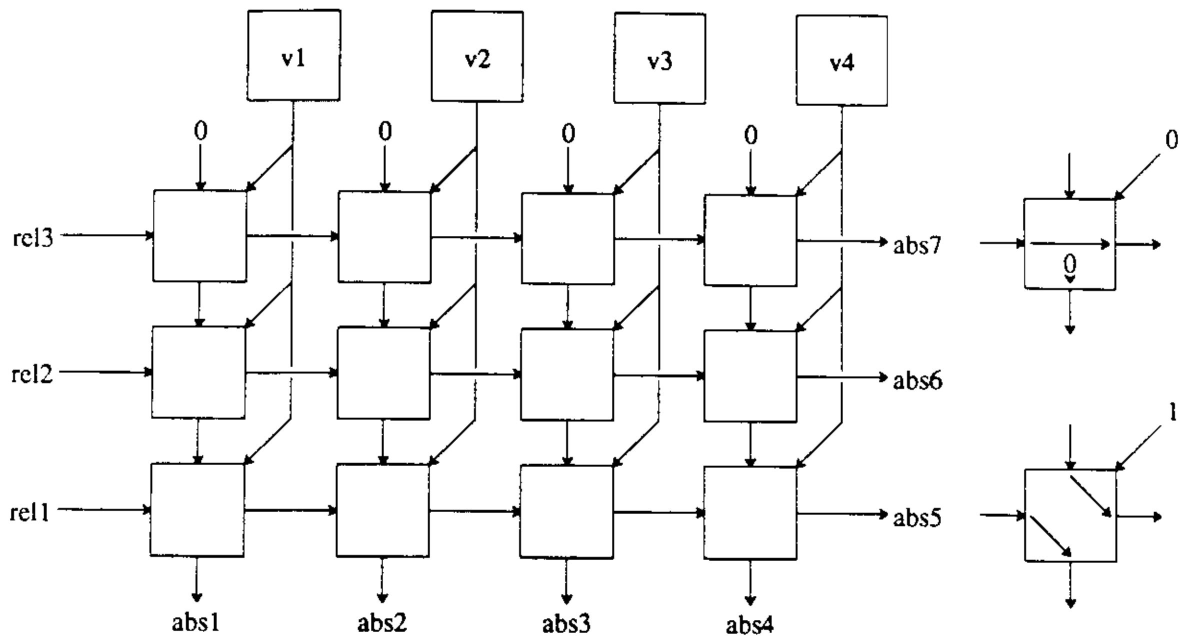

Figure 2.12: Relative to Absolute Rollback Amount Translator

amount arbitration, the largest rollback amount will remain on the rollback amount bus.

The rollback controller uses a relative to absolute rollback translator, as shown in figure 2.12, to handle interactions with the outside world. The *valid bits* indicate which cycles contain good data. One of the three inputs is set to 1 and the other two are set to 0. The switching network passes the 1 to the output representing the absolute rollback amount that corresponds to the relative rollback amount at the input. The first *n valid bits* are invalidated for a rollback of *n* by resetting the bits to 0. The array is shifted once per cycle, and a 1 is shifted in if the system state is to be recorded and a 0 if it is not.

Externally, rollback amounts are given as a number of cycles from the present cycle. The rollback controller, however, counts cycles in terms of a relative rollback amount. After a rollback, the *MP* considers the last *n* cycles that it rolled back over to be invalid because they hold corrupt data stemming from the bad state. If the restored state also happens to be bad, then the *MP* should roll back one more cycle from the last rollback, or a relative amount of one cycle. If all the states are valid, then the translator will return the same number put in. The only difference is if some states are not valid, in which case the number of invalid states is added to the relative rollback amount in order to obtain the absolute rollback amount.

The rollback controller determines how many cycles to roll back. If the error was detected from on-chip, then the rollback amount is set to one relative cycle because the processor (either the master or the slave) was able to detect the error right away. If the error was detected from off-chip, then the rollback amount is set to two relative cycles because the bad state took one cycle to get off the master processor and one cycle to get onto the slave processor to be compared. Rolling back within two cycles of a previous rollback causes an additional cycle to be added to the relative rollback amount because the error may have been in a DWB, and so rolling back an extra cycle will erase the bad data. In any case, the maximum number of absolute rollback cycles the *MP* requests is clamped to 4. If too many rollbacks are encountered within a 16-cycle frame, then the controller signals a shutdown trap because the fault is probably caused by a bad value in a permanent memory or stems from a hardware fault rather than a transient error. More details on the rollback controller can be found in [Lai90a].

2.4. Rollback Scheme

The rollback control mechanism, as described in section 2.1.2, is distributed throughout the datapath in order to allow for one cycle recovery. When a system unit signals rollback, this is latched in the *MP* on ϕ_2, and rollback proceeds on ϕ_3. Registers holding values between cycles are restored and outstanding memory transactions are restarted. In the case that the rollback amount is more than the *MP* can handle, a shutdown trap is signaled. The rest of this section deals with the changes made to the datapath, how the changes affect instruction operation, and what occurs during a rollback cycle.

2.4.1. Datapath Changes

When a processor rolls back to a cycle, it restores all state to that which was recorded at the beginning of ϕ_1 of the destination cycle. For the *MP*, this includes the IMM, BAR, PSW, RFTRAN, RF, SDR, PC, and the datapath controller. In addition, any outstanding memory transactions must be restarted. Not all the units mentioned need DWBs; some registers can be restored from values put out onto busses during rollback by other registers. The following modules shown in figure 2.10 are augmented for micro rollback:

- RF, PC: To cancel the last *n* cycles of writes into these registers we just need to place a delayed write buffer before the input of each register. Invalidating the values means they won't be written into the permanent register nor read out of the DWB.

- PSW: The processor status word register evaluates twice in a rollback cycle. It must evaluate on ϕ_1 for normal operation because it is written into on ϕ_1 and needs to be read out immediately, and it must evaluate on ϕ_3 because the bits are needed to initiate the memory cycle and register file read.

- CU: (control unit not shown) Load and store instructions take two execution cycles, and so the controller must restore its state and second stage instruction register. (See [Lai90a] for more details.)

- IR: The RFTRAN, IMM, and the first stage of the controller instruction register store relevant pieces of the incoming instruction and form a distributed instruction register. We can take advantage of this and reduce the amount of rollback control logic by creating a central instruction register (IR) to put out the proper instruction during rollback and simulate an instruction fetch.

- SDR: Without micro rollback or state repair, the *MP* would temporarily store data upon busD before a write to memory the following cycle for a *store* instruction. The SDR was created to restore the value of busD. State repair, however, also uses busD to transfer data, and because there isn't enough time after state repair to restore busD to its original value, the SDR must instead be used as an active gateway from busD onto busOUT for the *store* instruction (*i.e.* the data for the *store* is saved in the SDR instead of on busD).

- MAR: In order to restart the last memory access within one cycle, the *MP* cannot evaluate the ALU or PC but must add another rollback memory, the memory address register. If the BAR should be used, it will contain the same value as the two least significant bits of the MAR. We take advantage of this fact by restoring the BAR from busOUT during rollback when the MAR gates out onto busOUT.

2.4.2. Datapath Execution

The delayed write buffers are two-stage shift registers. Figure 2.13 shows the two types of cells used for both the data and tag portions. The left cell is a one-stage shift register cell (basically a latch), and the right cell is a two-stage shift register cell. Table 2.13 shows at what times the delayed write buffer stages shift and when the registers evaluate and gate onto the external busses. *Load* and *update* mean loading the first stage and updating the second stage of the two-stage shift register cell. For some registers, the value to be written into the first element is not ready until after the first stage of the shift register cell is loaded; a different *load1* phase indicates that a one-stage cell is to be used as the first element so that it can be loaded later and the output will be immediately updated. The evaluation phase is when the first valid value is found and put on the DWB's internal bus (figure 2.1). *Gate* indicates the phase that the register writes onto the external busses. Implementation details can be found in section 3.4.1.

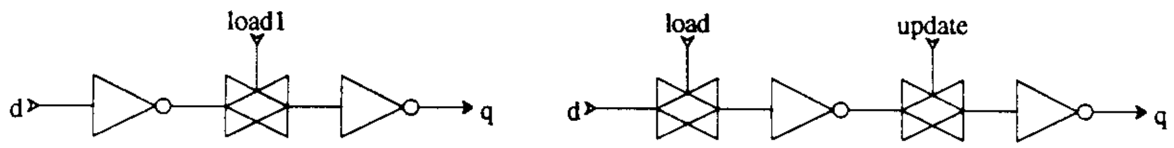

Figure 2.13: Shift Register Cell

register		RF	SDR	PC	MAR	IR	PSW	CU
valid	load1	ϕ_4	ϕ_4	ϕ_4	ϕ_1	ϕ_4	ϕ_4	ϕ_4
	load	ϕ_3	ϕ_3	ϕ_3	ϕ_1	ϕ_4	ϕ_4	ϕ_4
	update	ϕ_4	ϕ_4	ϕ_4	ϕ_2	ϕ_2	ϕ_1	ϕ_2
data	load1	ϕ_1	ϕ_1	ϕ_1	ϕ_1	ϕ_4	ϕ_1	ϕ_4
	load	ϕ_3	ϕ_3	ϕ_3	ϕ_1	ϕ_4	ϕ_4	ϕ_4
	update	ϕ_1	ϕ_1	ϕ_1	ϕ_3	ϕ_2	ϕ_1	ϕ_2
evaluate		ϕ_1	ϕ_1	ϕ_1	ϕ_3	ϕ_3	ϕ_1/ϕ_3	ϕ_3
gate		ϕ_1	ϕ_2	ϕ_2/ϕ_4	ϕ_2/ϕ_4	ϕ_3	ϕ_4	ϕ_3

Table 2.13: Delayed Write Buffer Timing

Figures 2.14, 2.15, 2.16, and 2.17 show register to register, load, store, and branch instruction execution along with DWB timing. The DWBs shift every cycle and load data into the stages every cycle, but logical writing into the register is controlled by shifting a 1 into the *valid bit* of the tag. Register evaluations take place every cycle, but the results are only seen if the values are gated out onto the external busses (*e.g.* busD, busOUT).

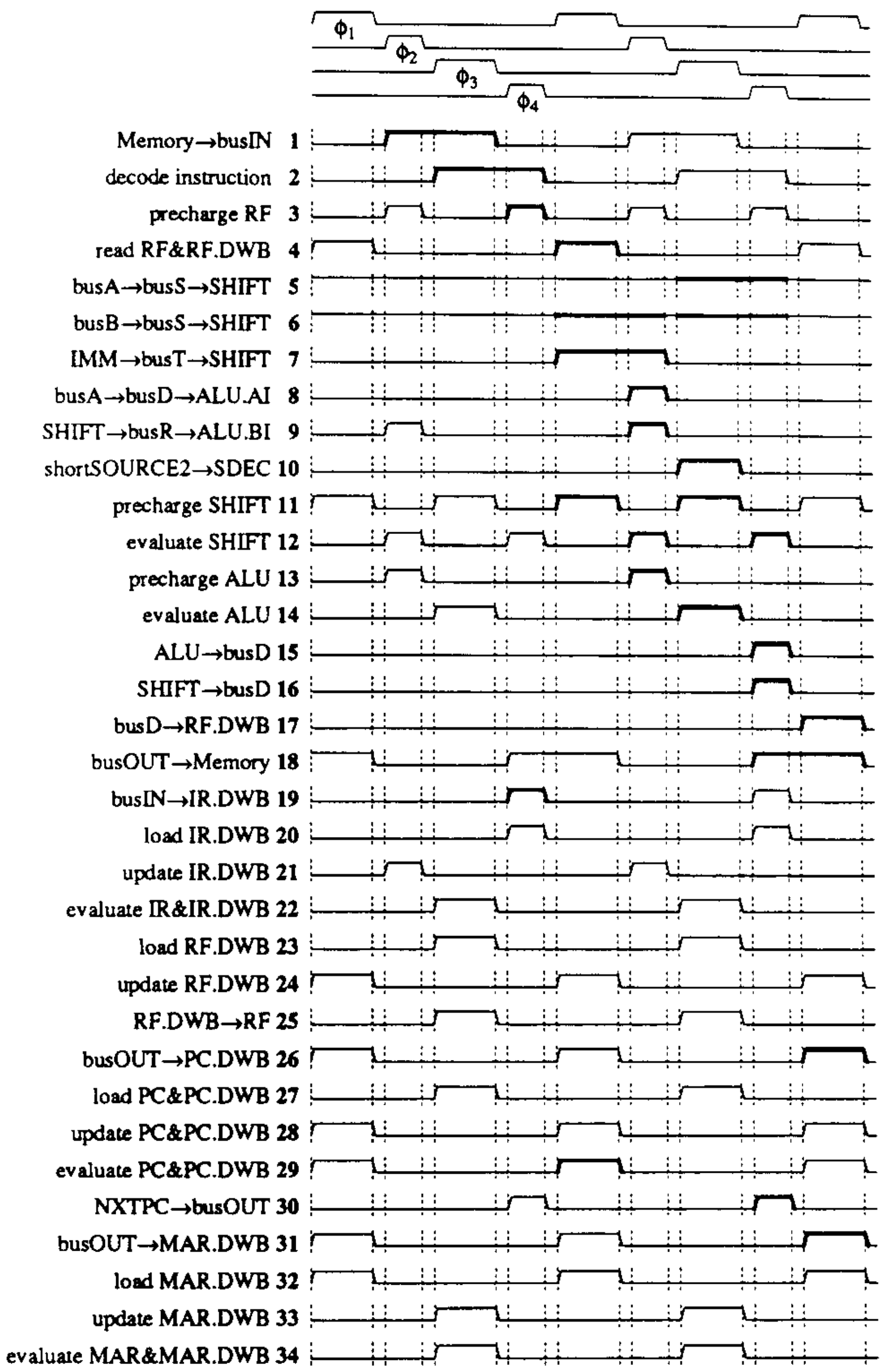

Figure 2.14: Register to Register Instruction Execution with Micro Rollback

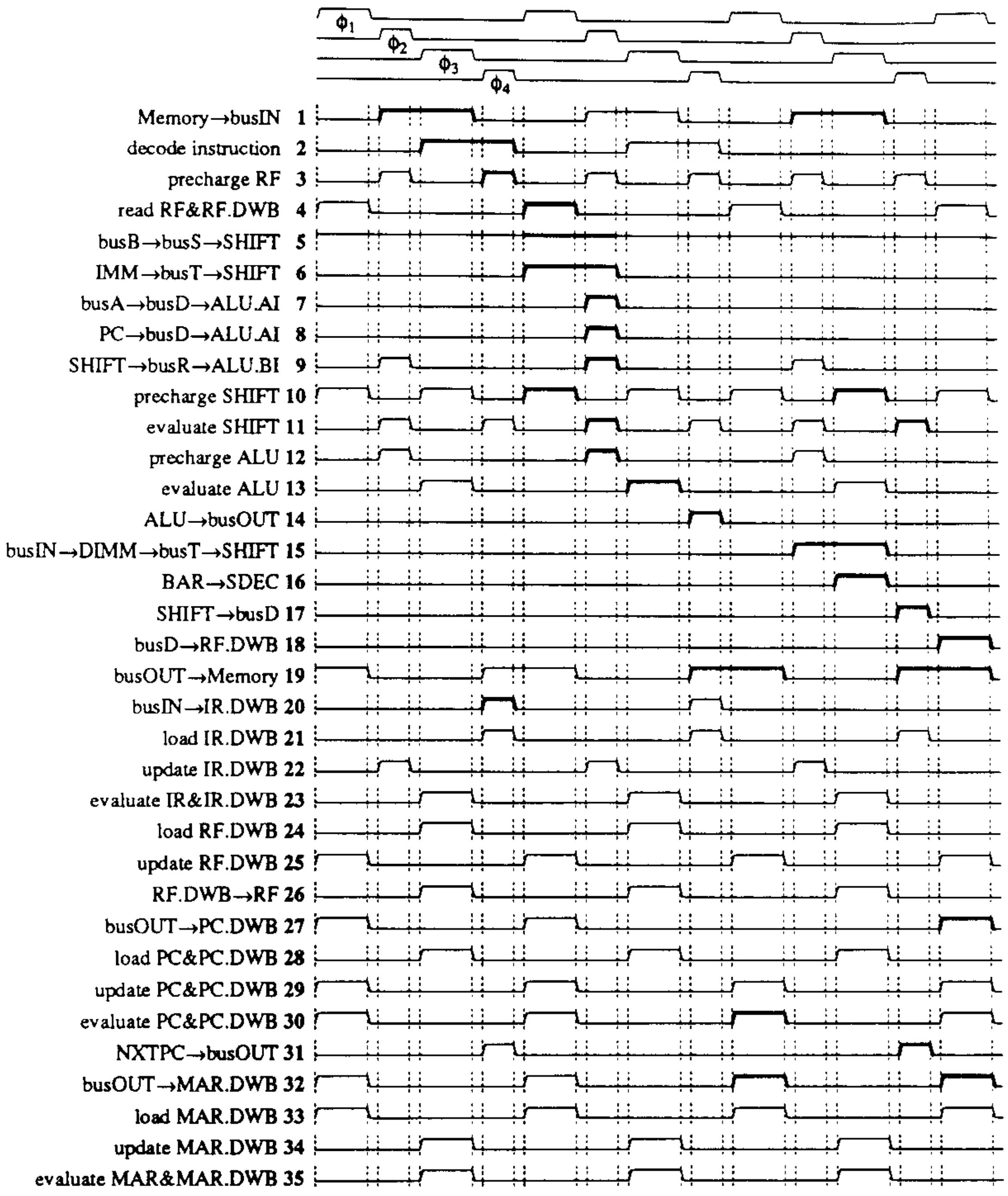

Figure 2.15: Load Instruction Execution with Micro Rollback

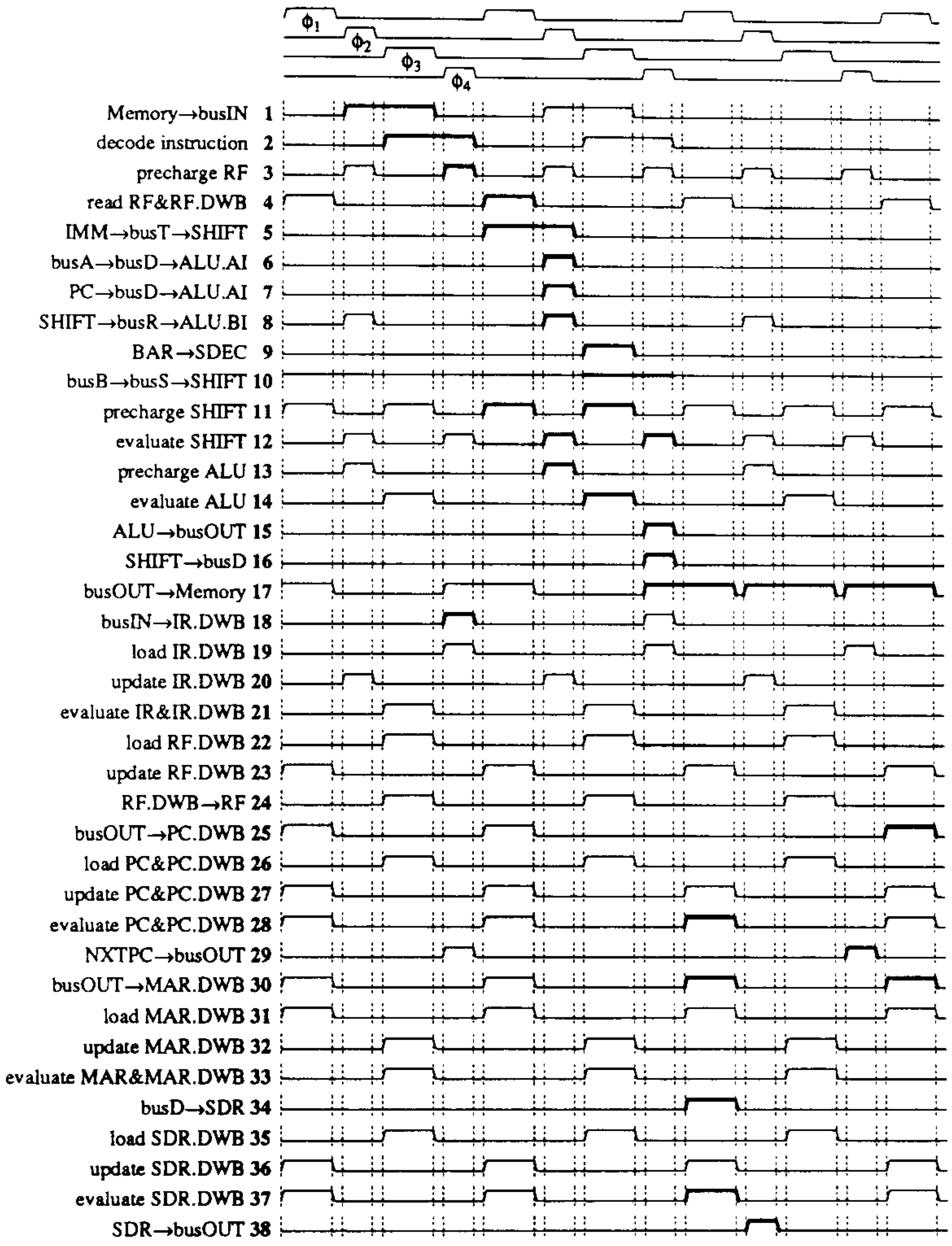

Figure 2.16: Store Instruction Execution with Micro Rollback

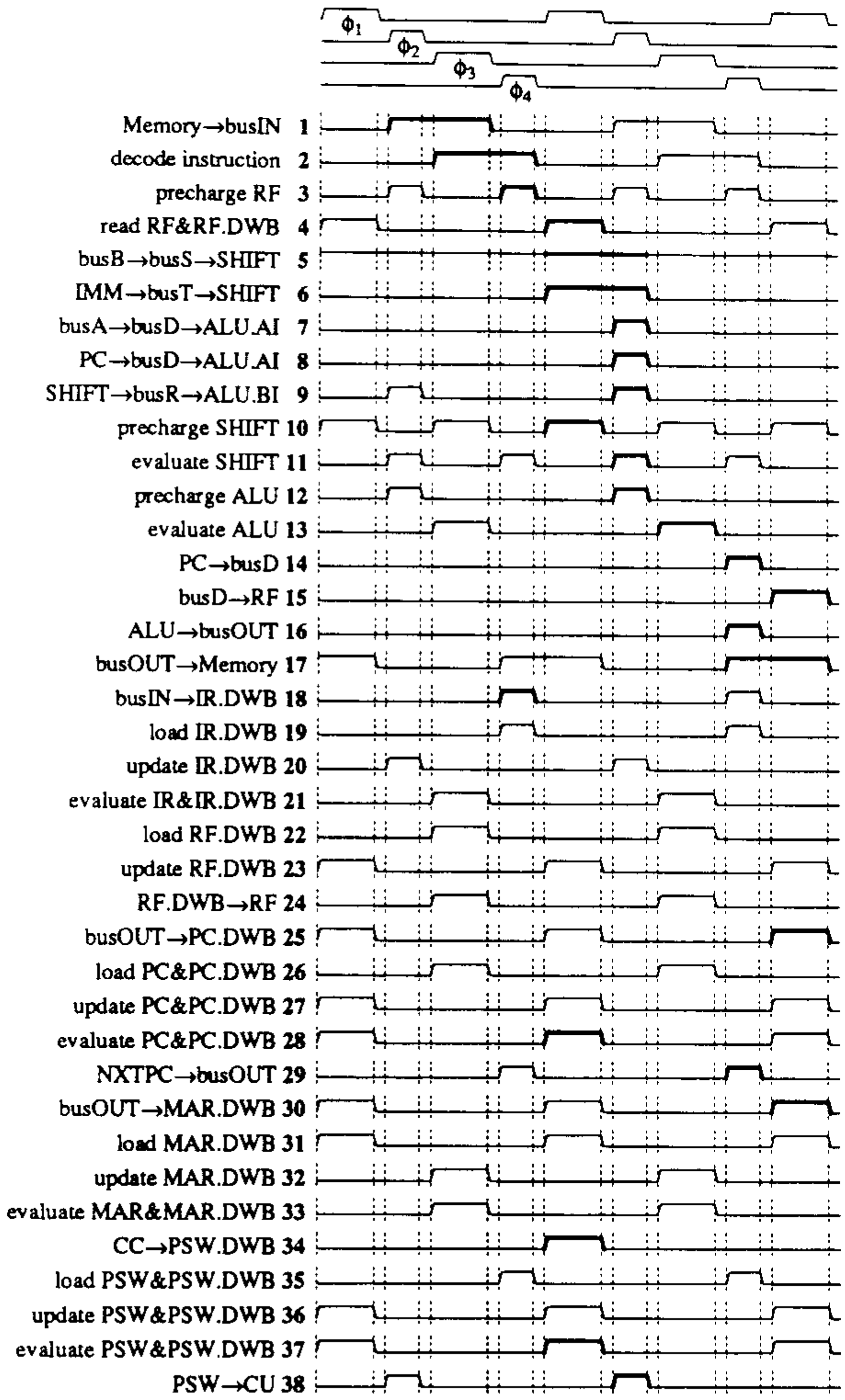

Figure 2.17: Branch Instruction Execution with Micro Rollback

2.4.3. Rollback Cycle

Figure 2.18 shows the rollback execution timing diagram. The first cycle shown is a normal operation cycle, and the second cycle is the rollback cycle. The rollback signal is latched ϕ_2 and determines whether a rollback should occur that cycle or not. On ϕ_3, the valid bits corresponding to the number of cycles to roll back are invalidated, and any the transfer of data from the pads to busIN is cancelled (**1**). Instead, the IR supplies the instruction that will be executing at the beginning of the cycle rolled back to so that the distributed instruction register (RFTRAN, IMM) and the controller can be restored (**6 7 2**). On ϕ_4, the instruction fetch or data transfer is reinitiated from the MAR (**22 23 24**). At the same time, the lower two bits of the address are used to restore the BAR. Note that on a rollback cycle, invalid bits are shifted into all the DWBs so that the cycle will not be recorded (**3 8 14 19 25**). Though the PSW normally evaluates on ϕ_1, it must also evaluate ϕ_3 during rollback because status bits are needed for the memory transaction and the current window pointer is needed for the next register file read (**28 29**).

2.5. State Repair Scheme

Micro rollback will handle transient errors that are detected a few cycles after they happen, and even those that develop in the delayed write buffers, but only if rollback is signaled before the bad data is committed to permanent memory. This means that should a bit flip in a permanent register, the *MP* cannot recover. In this case, error detection will catch the error and signal rollback; the error will be detected again after rollback, and the

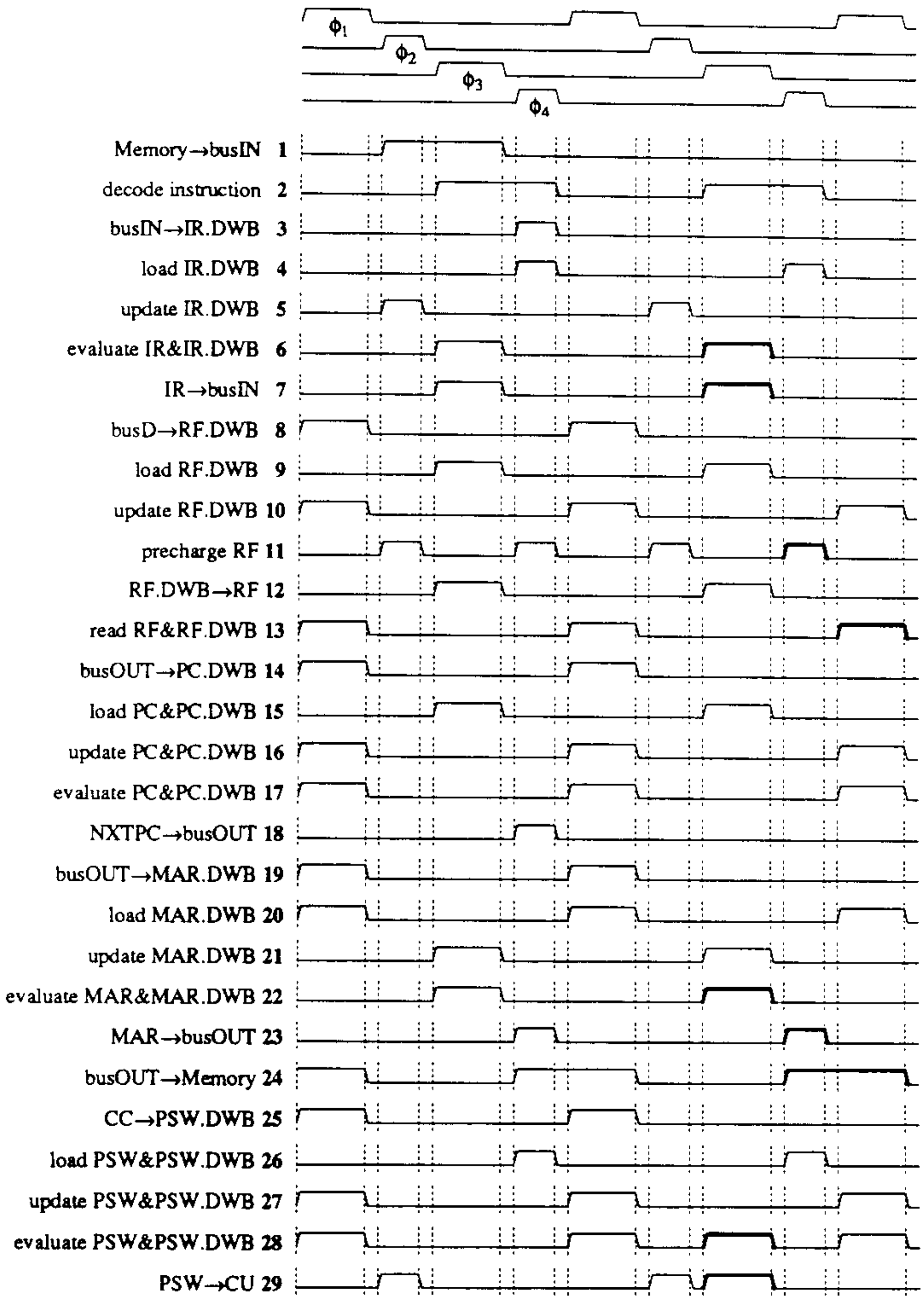

Figure 2.18: Rollback Execution

error detection circuitry will again signal rollback. After the fourth rollback, as described in section 2.3.4, shutdown will be signaled.

The register file occupies a large percentage of chip real estate, and hence, the probability of the register file being corrupted is greater than that of any single register. In order to effect better coverage, we invest extra resources into protecting this module by performing state repair upon the register file should it be the source of the error. Rather than continuing on from a rollback after an error is detected in the register file, the chip with the correct data transfers the value to the chip with the bad data in order to repair the corrupt register file.

We want to effect state repair with a minimum of additional hardware and cycle overhead. We do not want to use external glue chips to control state repair because they will comprise another system which is subject to failure, and also they take up PCB real estate and routing.

One solution is to add a new instruction which will be executed in the event of state repair. The *MP* will trap to a software state repair handler employing a conditional store instruction after rollback. The state repair handler will determine which register caused the problem by examining the instruction that caused the trap, and then execute a conditional store instruction which will allow only the processor with a good value to write into memory. Next, both processors will load the value back into their register files and return from the trap.

For the lower bound, this sequence will take 2 cycles for the *calli* instruction, 2 cycles for the store instruction, 2 cycles for a load instruction, and 2 cycles for the *reti* instruction for total overhead of 8 cycles. More cycles are needed to extract the proper register and manipulate the register window in order to access the register. Furthermore, state repair may be needed during the register window overflow handler routine which should never be interrupted under any circumstances.

A better solution is to add states to the controller finite state machine which will perform state repair without changing any other registers on the chip. Only two cycles are needed (read the data and transfer between chips) and they leave no recorded trace except for the corrected data. When we detect an error on the register file, we roll back to the execution cycle of the instruction that detected this error. This will also read out from the instruction register the two possible registers at fault. During the rollback, the two processors communicate to each other with four lines which one had the error and needs to be repaired and which source register it was read out of. In case of multiple errors, priority goes toward first repairing the register read onto busA and then the register read onto busB. In the event that the same register needs to be repair on both chips, repair is impossible; instead, the instruction is retried in hope that the error is transient. If re-execution of the instruction does not clear the error, state repair will be repeatedly signaled. This policy leads to a shutdown after four rollbacks within a 16 cycle time frame are performed (enforced by the rollback controller).

The rest of this section describes how errors is detected within the register file, what changes are needed to allow state repair, and what goes on during state repair.

2.5.1. Error Detection

For our error model we assume one bit will be flipped in the register file. Using a single bit of parity will detect this and also all odd numbers of flipped bits.

In the event that an even number of bits are flipped on one processor, the error will not be caught, but it will be detected by state comparison and a rollback will ensue. Since this error will not be fixed, the cycle will repeat until the *MP* signals shutdown. Even if the *MP* can't recover from this error, it will be able to detect the problem and signal shut down.

There are three cases to examine in the rare event that the same register on both processors is corrupt. If an even number of bits flip on both, then the case defaults to that of only one processor with an even number of bits flipped. If both have an odd number of bits flipped, then the *MP* can't recover because both processors are known to be corrupt. Instead, the *MP* re-executes the instruction in hope that one of the errors was caused by a transient error on a bus or a parity generator and is not an error with the stored data. After enough repeats the *MP* will initiate shutdown. The one fatal case is if an odd number of bits is flipped on one processor and an even number on the other. The even number flipped will not flag an error, and so the *MP* will assume this is the good value and repair both processors with the bad data.

2.5.2. Datapath Changes

The *MP* derives the parity to be written into the register file from both the data and the register address. This is because while the address is within the delayed write buffer it may be changed and this can cause the two processors to store the data into different addresses. The data that is stored into the wrong address will trigger a parity error when it is read out because its parity will not match its corresponding address.

Additions to do this (figure 2.10) are parD and parB. ParD serves double duty by both generating the parity to save in the register file and checking the parity of busA read out of the register file (by gating busA onto busD). ParB checks the parity of busB read out of the register file.

During state repair, the parity of busOUT is compared with the parity read out of the register file to make sure that the correct data has not been corrupted on the way out, and parIN on the destination processor verifies the data safely made it between chips; any errors start another rollback.

In order to transfer data with a minimal number of extra controls, a gateway is added from busIN to busT to avoid use of the DIMM, which is controlled by a combination of the BAR and data size. The SDR was originally used during micro rollback to restore the value of busD, but since state repair uses busD to transfer data between the two processors, the SDR was changed to directly gate the data for a store instruction onto busOUT rather than temporarily storing it on busD so that busD wouldn't have to be restored again after state repair. The good data to be sent to the faulty processor still needs to be sent to busOUT from busD, and so the gateway from busD to busOUT was added for state repair. In this case, busD is used as a dynamic latch to hold the repair data over a cycle. The shifter needs to shift by zero in order to put the value read from the register file onto busD for sending to the other processor, and it also needs to shift by zero when it brings the data into the main datapath from busT after the processors receive the correct value; fortunately, the *shift by constant zero* signal is already provided for use by the *ldhi* instruction.

2.5.3. Controller Changes

Once the on-chip parity error is detected, the rollback controller signals a rollback of one cycle, back to the instruction that read from the bad register, and signals state repair by asserting the state repair line.

There are four signals associated with state repair (table 2.2). Pad.repairAm and pad.repairBm are asserted by the master processor if it needs the registers read out of the register file onto busA or busB repaired. Pad.repairAs and pad.repairBs are asserted by the slave processor if it needs the registers read out of the register file onto busA or busB repaired. If any of these lines are asserted during rollback, then state repair is initiated after the rollback (except when both are asserted for the same bus as described above for errors in both processors). If both registers read out of the register file need to be repaired, then priority goes toward first repairing the register read out onto busA and then the register read out onto busB.

The datapath controller reads the repair signals and controls state repair. Two more states are needed and the logical step would be to add another bit to the one bit of controller state to get four states. However, the *MP* needs to be in the correct state after state repair to resume execution and there isn't enough time to restore the state. Instead, three bits are used for four states with redundant encodings. 000 is the normal state, 001 is the suspend state, 10x is the first state repair state (repair1), and 01x is the second state repair state (repair2). Because the last bit can be either 0 or 1, the *MP* saves the normal controller state within the redundant state during state repair. Only the last bit needs to be restored on rollback because the *MP* doesn't roll back into a state repair cycle; zeros are shifted into the *valid bits* of all registers (except for the register file) during state repair so that the two cycles are not recorded.

During rollback amount arbitration, the system unit that requests the largest rollback amount wins. It is possible that the *MP* will request rollback to a certain cycle in order to perform state repair and end up rolling back to the wrong cycle. The instruction will be different, and so the registers repaired will be the wrong ones, but since there's no errors in those registers no harm is done. Once the flow of execution comes back to the instruction that caused the state repair, the bad data will be read out again and the *MP* will again initiate state repair.

2.5.4. State Repair Cycle

Figure 2.19 shows the timing diagram for state repair. An error is detected by the rollback controller and the *MP* initiates rollback and state repair. On ϕ_3, all registers are invalidated (**1**) and the IR simulates an instruction fetch (**2**). As part of rollback, the MAR gates its value onto busOUT (**14**), but the controller knows it's a state repair and

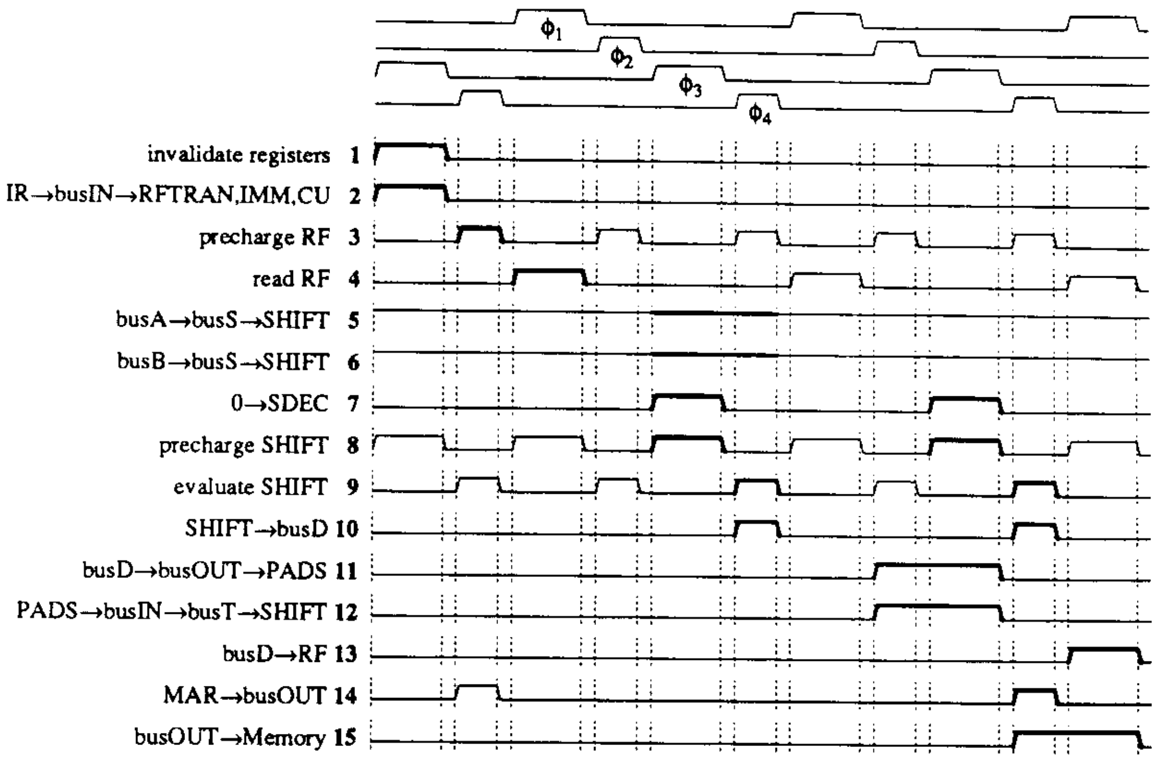

Figure 2.19: State Repair Execution

doesn't initiate a memory transaction. The register file is read (4) and the proper bus is gated to the shifter (5 6).

A path through the shifter must be taken because that is the only way to get a value from busB onto busD. All shifts are by zero so that the data won't be changed (7). Reading from the register file on the corrupt chip will result in another parity error which is masked by the controller.

The shifter evaluates ϕ_4 (9) and puts the value to be repaired onto busD (10). The following cycle, the data is gated onto busOUT and to the pads (11); only the processor with the good data will write onto the bus. Both processors read the data into the shifter (12) and shift it into the datapath ϕ_4 (9 10), where it is written into the register file (**13**). Now the memory transaction occurring during ϕ_1 must be restarted from the MAR (**14 15**), and normal operation can resume. When the instruction reads out the data it will be correct.

If both registers had needed to be repaired, then at this point the register read out onto busA from the register file will be correct, but the one read out onto busB will still be wrong. The instruction will re-execute, and the *MP* will initiate state repair for the other register. However, the rollback will be for one cycle, and thus it will not erase the data from the first repaired register.

3. Processor Implementation and Timing

This chapter describes the implementation of the various modules in the datapath. The blocks are broken down into six categories: pads, the clock, busses, registers, error detection, and combinational logic. Following those descriptions is a summary of the critical paths, the geometry and overhead of the various blocks, and the power consumption estimates for the *MP*.

3.1. Pads

We used the 2µm scalable CMOS N-well TinyChip pad set from MOSIS. These pads are designed for use with a 44 pin package and so pad power and chip power are supplied over the same 33µm Vdd/GND rails. However, for our implementation, we used separate power lines for the pads and the processor core in order to isolate the Mirror from pad voltage transients. We needed to widen the pad power rails to 100µm (the size of a pad) in order to handle the current draw of up to 80 pads.

Our pad set consists of six cells: the tri-state I/O pad from the TinyChip pad set with power rails redrawn to a width of 100µm, a corner pad supplying Vdd to the pad power rails, a corner pad supplying GND to the pad power rails, a middle pad supplying GND to the pad power rails, a pad to supply power to the chip core, and a blank pad with only rails running through and a disconnected pad. All power pads are drawn with connections 100µm wide.

The tri-state I/O pad consists of a tri-state driver and a buffered input as shown in figure 3.1 (taken from the TinyChip documentation). We use the I/O pad to fulfill all

```
                                                        +------+
                                                   +----| PAD  |
                                                   |    +------+
                                                 +-+        |
                     Vdd                          |         \
                 +---------+                     +-+        / 150 ohm
                 |         |                       |GND     \
             |+--+         +--+|  ENABLE                    |
         +--0||               ||0---------          Vdd     |
         |   |+--+         +--+|                     |      |
         |       |         |                     |+--+      |
         |       +---------+--------------------0|| P       |
         |       |         |                     |+--+      |
 ENABLE  |   |+--+         +--+| ENABLEbar           |      |
---------|---||               ||0--+------           +------+
         |   |+--+         +--+|   |                 |      | IN_unbuffered
         |       |         |       |             |+--+      |
         |       +---------+-------|-------------|| N     +---+
         |       |         |       |             |+--+    |   |
   OUT   |   |+--+         +--+|   |                 |     \ /
   ------+---||               ||---+                GND     0
             |+--+         +--+|                            | INbar
                 |         |                              +---+
                 +---------+                              |   |
                    GND                                    \ /
                                                            0
                                                            |
                                                           IN
```

Figure 3.1: TinyChip Tri-State I/O Pad Schematic

four functions we require from an active pad. For an input pad, the enable line is hardwired to GND. For an output pad, the enable line is controlled by pad.ms; a 0 means the chip is the master and so the pad should be enabled and a 1 means the chip is the slave and so the pad should be disabled. Pads tied to pullup resistors (wired-NOR) have the OUT line connected to GND and the enable line is asserted to activate the pad. For the bidirectional address/data bus, the datapath controller controls the enable line and both components of the pad are used.

Timing through the pads can be found in section 3.3.

3.2. Clock

Our problem is to take a 25ns square wave phase clock (pad.phi) and generate our four phase clocks (ϕ_1, ϕ_2, ϕ_3, and ϕ_4). In order to differentiate which of the phases is ϕ_1, we also require a synchronization clock (pad.sync) whose leading edge must precede the leading edge of the phase designated ϕ_1 and must be after the falling edge of the previous phase.

Figure 3.2 shows the clock circuit used to generate the internal clock phases from external clocks. In.phi and in.sync are the phase and synchronization clocks and are buffered inputs from the pads. We use a 4 element shift register to generate the 4 non-overlapping phase clocks; after powerup, all 4 phases may be active but after 4 cycles of the phase clock the internal clocks will be in order. An $\overline{S}$-$\overline{R}$ flip-flop is used to generate a 1 every four cycles. The synchronization clock sets the flip-flop once every four phases using a leading edge pulse generator, and after the signal is latched into the shift register it is cleared by a falling edge pulse generator based upon the phase clock.

We must somehow load and update a master-slave shift register using a single clock; this is done by using the phase clock to load the master stage and generating a pulse from the falling edge of the phase clock to update the slave stage. Throughout the circuit we used delay inverters to generate a suitable propagation delay. These consist of one NMOS transistor switched by the input and one PMOS transistor switch by the input in series with several PMOS transistors switched on to add to the rise time delay.

From the shift register we can pick off the raw clock phases. Since ϕ_1 and ϕ_3 are 25ns long and the shift register shifts every 25ns, we can use the values directly from the

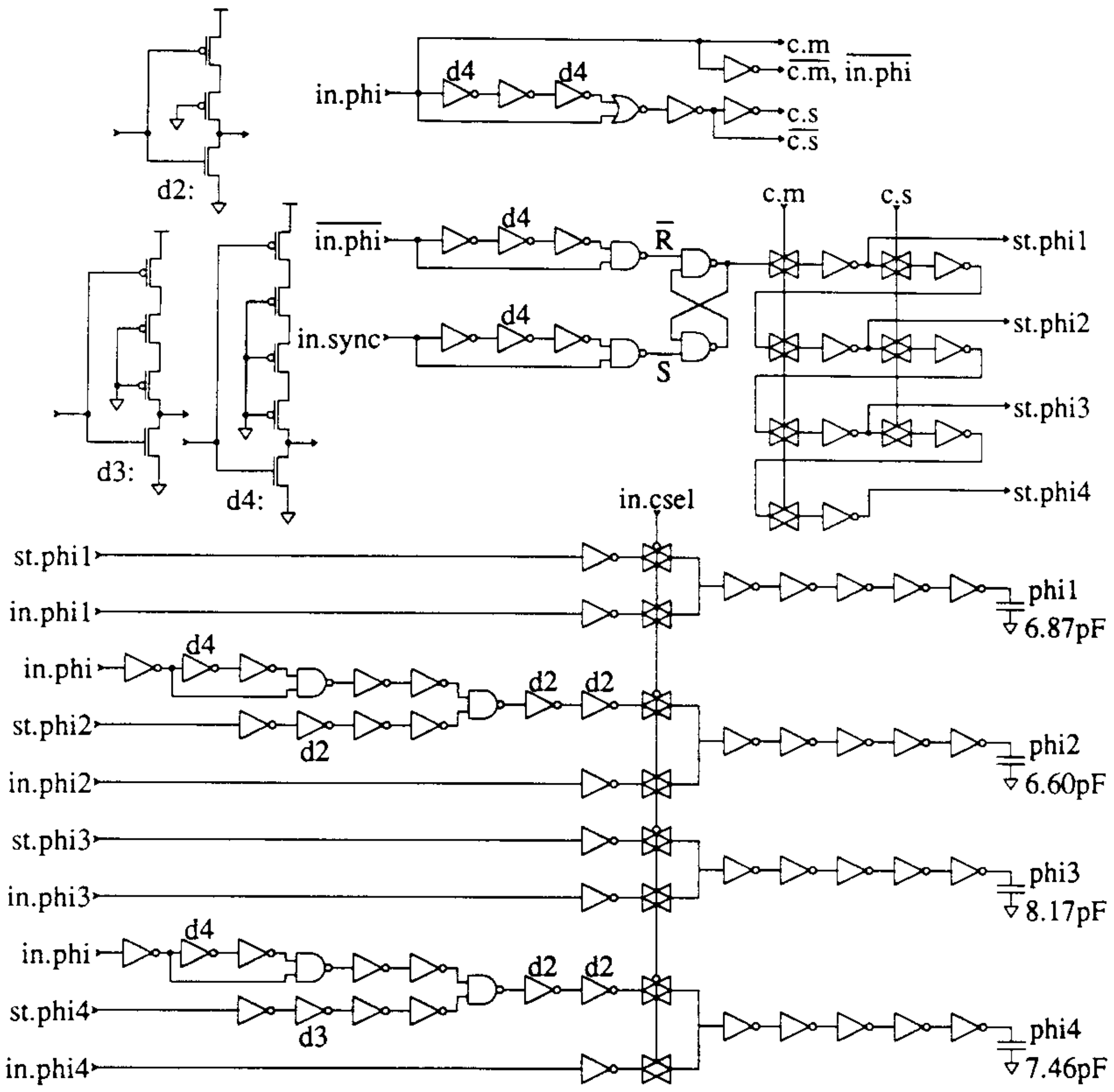

Figure 3.2: Clock Generation Circuit

shift register to generate those clocks. The short clocks, ϕ_2 and ϕ_4, are 15ns and a bit tricker to generate. The phase clock with its falling edge delayed is used as a base and it is ANDed with a delayed st.phi2 or st.phi4 signal to generate the ϕ_2 or ϕ_4 phase clock. This signal is actually a bit ahead in time and so it is put through two delay inverters before being fed to the clock driver.

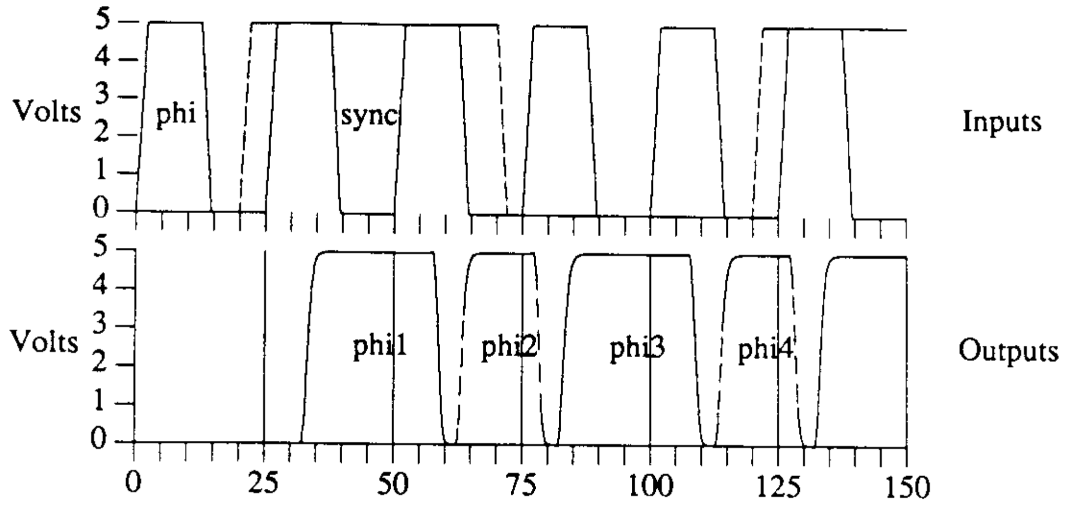

Figure 3.3: Clock Generation using Slow-Slow Spice Parameters

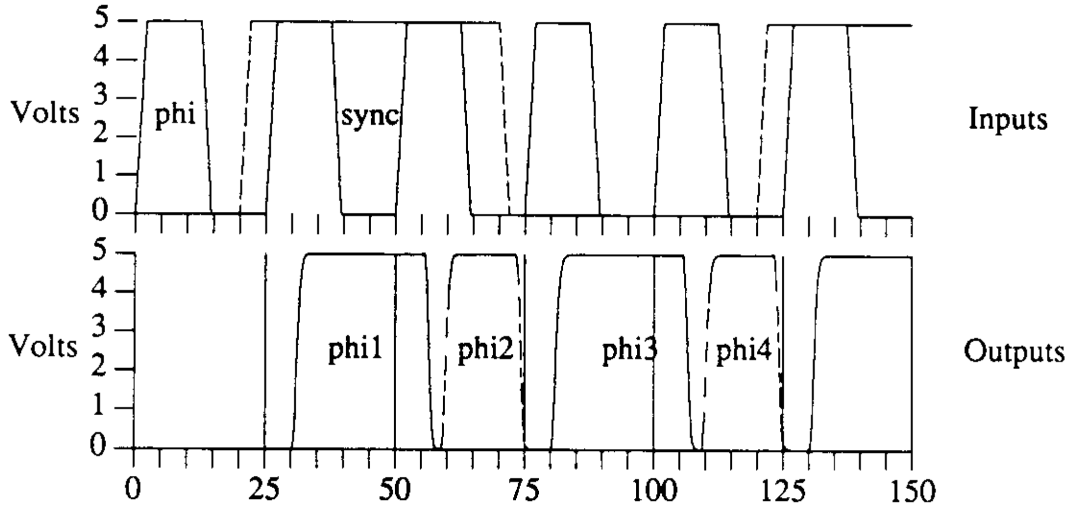

Figure 3.4: Clock Generation using Fast-Fast Spice Parameters

In the event that our clock generation scheme doesn't work, we have the ability to override internal clock generation by setting pad.csel (clock selection) to 5V and directly supplying the phase clocks through 4 pads (pad.phi1, pad.phi2, pad.phi3, pad.phi4). The clock driver is a 5-stage logarithmically ramped inverter chain: 8:4→24:12→64:32→176:88→472:236 (P:N, w=*n*um).

Figures 3.3 and 3.4 are plots of spice runs using the slow-slow and fast-fast corner parameters. The capacitive loads seen by the clock drivers are calculated by summing up the total capacitance of all routing and gates.

3.3. Bus

Three main data busses run through the *MP* datapath: busIN to carry incoming data from the pads, busD to transfer data between modules, and busOUT to carry outgoing data to the pads. Figure 3.5 shows the controls and drivers used to put data onto these busses. The top two circuits control gating during one or two clock phases. The metal enable lines (enb and $\overline{\text{enb}}$) are driven by double-sized inverters (PMOS w=16um, NMOS w=8um) and control one of the two bus drivers shown in the bottom half of figure 3.5. These circuits are used in all registers and functional blocks.

Figure 3.6 shows the circuitry involved in transferring data to or from the datapath. Blocks that gate onto busOUT use the tri-state driver configuration shown in the lower right corner of figure 3.5. A double-sized tri-state inverter is used to put data onto busOUT within the datapath. Routing to the pads (out.AD) is driven by a buffer consisting of a 24:12 inverter (PMOS w=24um, NMOS w=12um) feeding a 64:32 inverter. For our simulations, we use a 20pF capacitive load on the pads.

Once data comes onto the *MP*, it is driven through the routing (in.AD) by the pad input buffers to the busIN driver. We need to bring data in quickly and we do this using the driver shown in the lower left corner of figure 3.5. The driving transistors are sized 64:32 in order to handle busIN, which runs across the width of the datapath. The IR simulates an instruction fetch during rollback and so it too uses the same bus drivers as

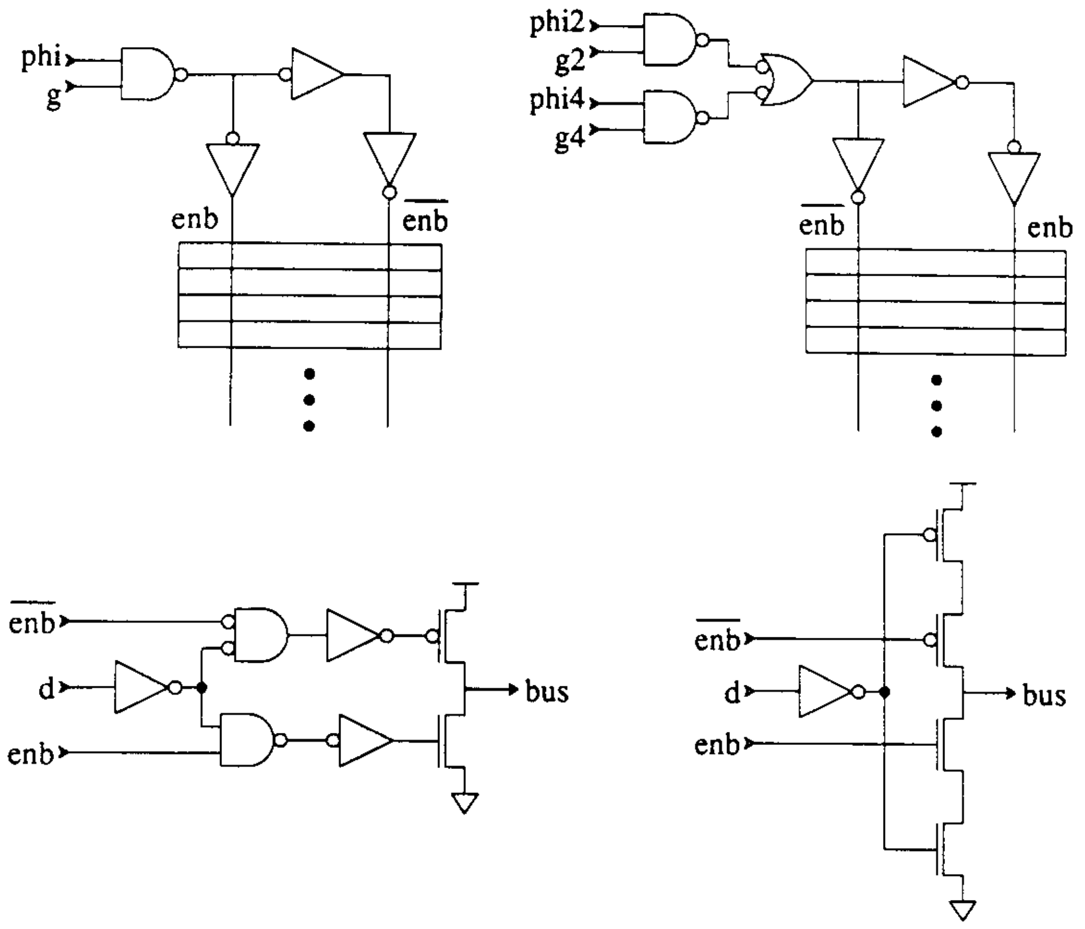

Figure 3.5: Bus Gating Circuitry

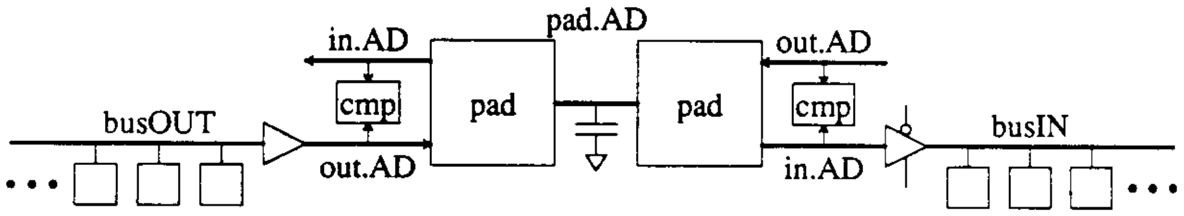

Figure 3.6: BusOUT to Pads to BusIN Path

the busIN driver. The *MP* knows by the beginning of ϕ_3 if there's going to be an interrupt and so the *calli* instruction to be gated onto busIN can be driven by a 32:16 driver.

All modules gating onto busD use double-sized tri-state inverters except for the

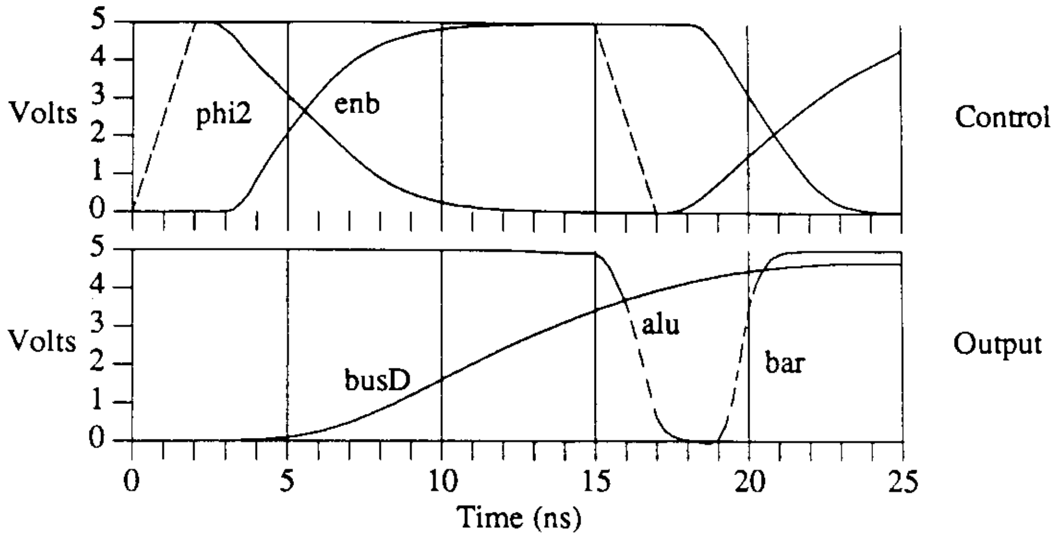

Figure 3.7: BusA Gating onto BusD

shifter; it uses a triple-sized driver because it evaluates in the same phase it is gating onto the bus (unlike the other modules which already have their data ready at the beginning of the phase).

The following timing diagrams are representative of most module behavior upon busIN, busD, and busOUT. The exceptions are the shifter and the IR which will be covered in their own sections.

Figure 3.7 shows busA gating onto busD during ϕ_2. Data put on busD during ϕ_2 is destined for the ALU, whose data must be valid by the beginning of ϕ_3 (20ns mark), and for the BAR, which must also be ready ϕ_3. If a module puts data onto busD during ϕ_4, it just needs to be stable before the beginning of ϕ_1 (20ns mark). The inputs to the bus drivers are assumed to be stable at the beginning of the driving phase.

Figure 3.8 shows a data transfer between two Mirror Processors as would occur during state repair. The data panel shows a 1 from busD gating onto busOUT, being

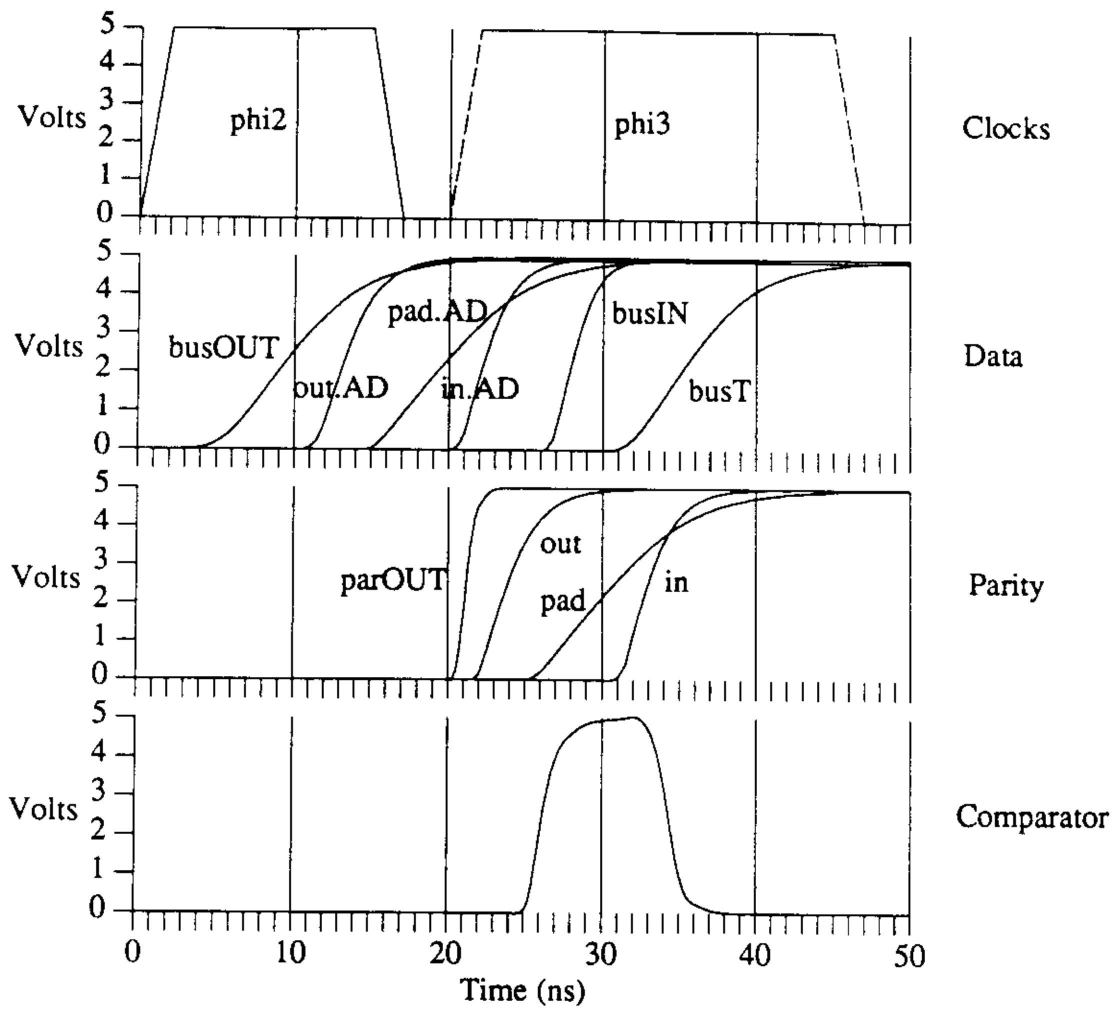

Figure 3.8: Data Transfer Between Two Mirror Processors

driven to the pads, off the source chip and onto the destination chip, and onto busIN and finally busT. The shifter needs the data by ϕ_4 which gives this transaction plenty of time to complete.

Reading at the 2.5V mark, it takes 20ns from the beginning of ϕ_2 or ϕ_4 to write data or an address to the pads for memory to use. Values put out onto pad.AD are checked by the comparator on the slave; the difference between out.AD on the master and in.AD on the slave as shown on the data panel is the time it takes before the comparator can be sure

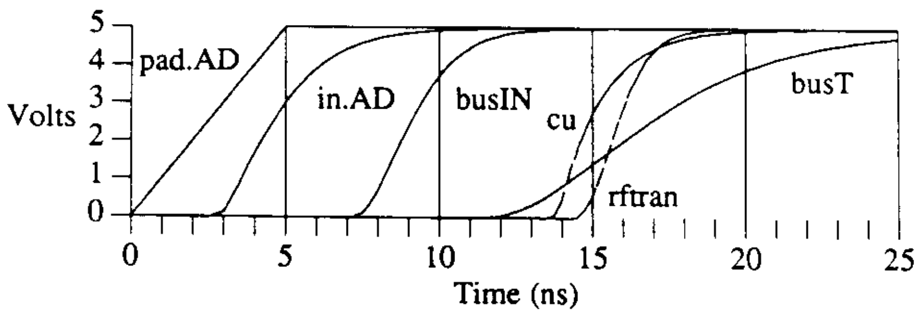

Figure 3.9: Pad.AD Gating onto BusIN

there is an error on one of the chips. Parity takes about 12ns to generate, but, as shown on the parity panel, it makes it to the other chip in time for the comparator (latched ϕ_1 and ϕ_3) to catch any errors.

In order to determine to setup time required of incoming data, pad.AD was driven with a 5ns slope and we measured how much time it took for the signal to reach its three destinations: the datapath controller (cu), the register file translator, and busT. Figure 3.9 shows the results.

For an instruction fetch, the instruction must go through the controller to the ϕ_4 PLA, it must go through the rftran and to the address decoder of the register file, and it must be stored into a 19-bit immediate latch reading off of busIN (not shown). Measuring from 2.5V to 2.5V, worst case is 13.25ns for the rftran.

During a load operation, data must come onto busIN and go through the data immediate sign extender onto busT (with worst case being 8 bits sign extended to 32 bits). This transaction takes 14.25ns to complete. Since we need these values by the beginning of ϕ_4, this gives memory [ϕ_1 + underlap + ϕ_2 + underlap + ϕ_3 + underlap - 14.25ns =] 65.75ns after the beginning of the address enable signal to put the requested data on the pads.

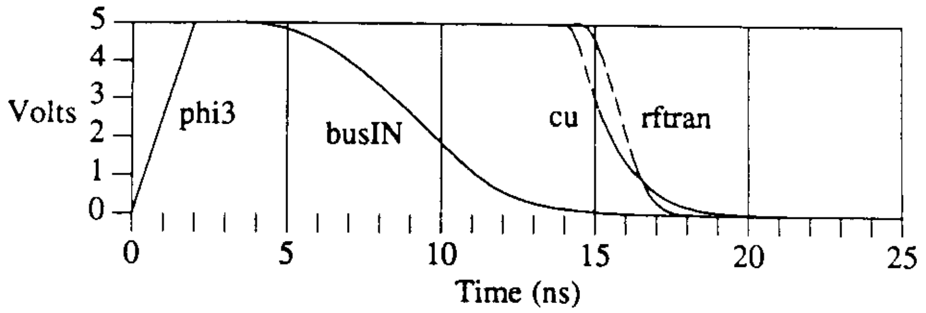

Figure 3.10: Calli Gating onto BusIN

The last case we have to look at (because it uses different sized bus drivers) is the *calli* instruction override. Figure 3.10 shows the *calli* instruction being put on busIN during an interrupt cycle with plenty of time to spare.

3.4. Registers

This section covers the registers in the *MP* datapath that need extra hardware added on for micro-rollback. These registers are: the register file (RF), the store data register (SDR), the program counter (PC), the memory address register (MAR), the instruction register (IR), and the processor status word (PSW).

Section 3.4.1 covers the implementation and operation of all registers and delayed write buffers except for the RF, which is described in section 3.4.2. The PC and PSW contain extra combinational logic which are detailed in sections 3.4.3 and 3.4.4.

3.4.1. Single Registers

We implement micro rollback upon all the registers in the Mirror by placing a delayed write buffer (DWB) before the register into which we write instead of the register itself. The DWB is set up as a shift register consisting of a data portion and a tag portion. Selection circuitry chooses the most recent, valid entry which is gated onto the register's internal bus, and from there onto an external data bus upon the register read. We roll back by invalidating the first *n* tags during ϕ_3 of the rollback cycle. Figure 3.11 shows a prototypical register with a DWB.

All delayed write buffers and permanent registers except for the register file are implemented with dynamic storage. This is possible because all the single registers are usually written into at least once every other cycle in normal operation, and so the permanent registers will be constantly refreshed. The only time that a stream of 0's will be written in the the *valid bits* of the single registers is during *wait states* when the *MP* is waiting for memory to finish a transaction. No valid data is written into the DWBs, and so the permanent registers will not be refreshed *n* cycles later when the data reaches the end of the DWB. Dynamic memory, however, can last for at least a millisecond, and no memory access should last 1000 cycles.

Operation of a register and DWB is divided into four parts: DWB shift, DWB and register evaluation (finding the current value), DWB invalidation, and register value gating onto the external bus (register read).

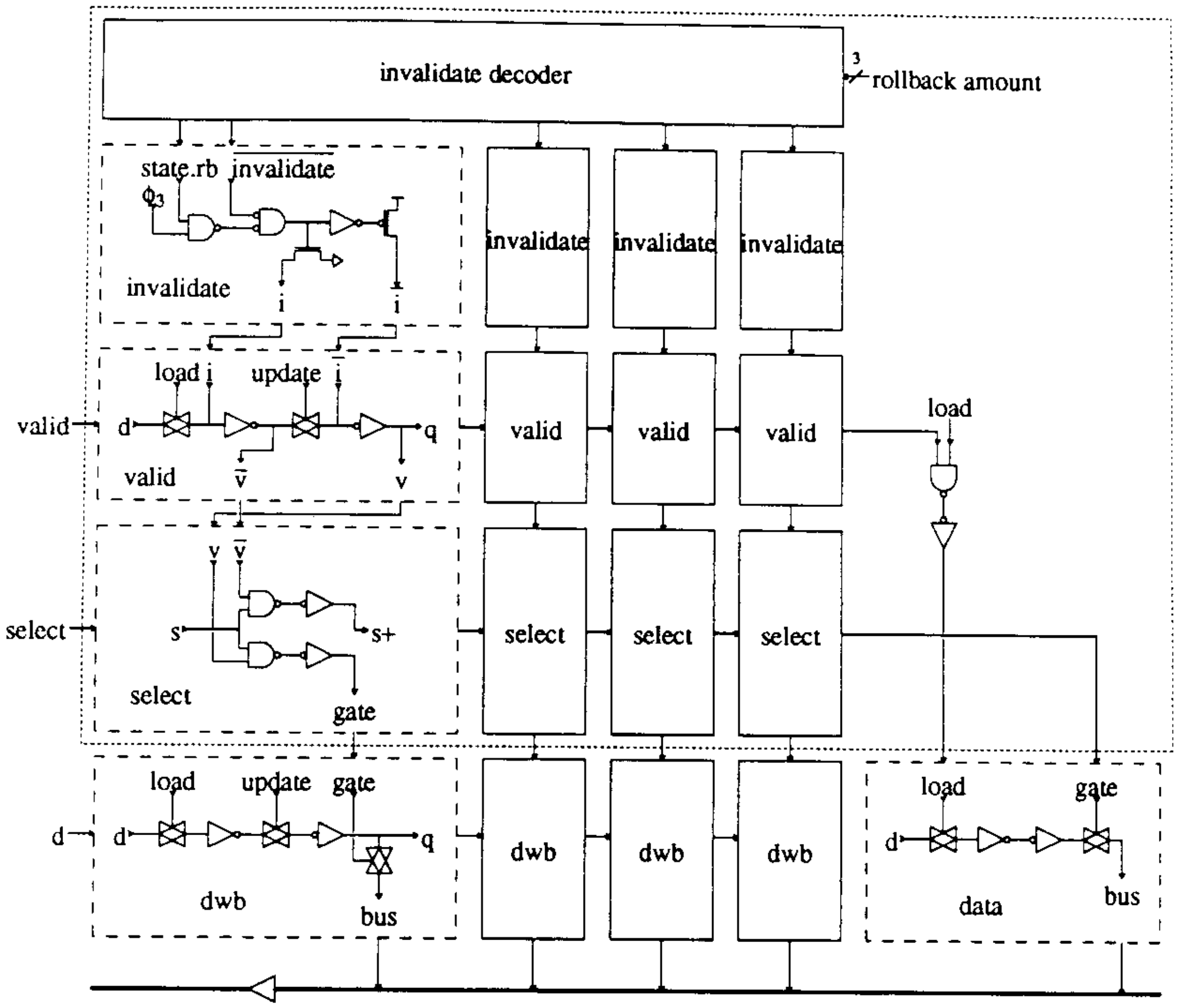

Figure 3.11: Prototypical Register with Delayed Write Buffer

3.4.1.1. DWB Shift

In order to minimize the stride, the DWB cell was laid out with poly select lines; I was able to design a cell with a pitch of 39λ and a stride of 72λ. The load and update control lines are driven by buffers connected to the phase clocks (refer to figure 3.11). Poly has a high RC constant and so we will see large propagation delays at the end of the lines; for this reason we use non-adjacent clock phases to load and update the data cells to prevent data shoot-through. The exception to this is the PSW; it is only 11 bits so the

register		RF	SDR	PC	MAR	IR	PSW
valid	load1	ϕ_4	ϕ_4	ϕ_4	ϕ_1	ϕ_4	ϕ_4
	load	ϕ_3	ϕ_3	ϕ_3	ϕ_1	ϕ_4	ϕ_4
	update	ϕ_4	ϕ_4	ϕ_4	ϕ_2	ϕ_2	ϕ_1
data	load1	ϕ_1	ϕ_1	ϕ_1	ϕ_1	ϕ_4	ϕ_1
	load	ϕ_3	ϕ_3	ϕ_3	ϕ_1	ϕ_4	ϕ_4
	update	ϕ_1	ϕ_1	ϕ_1	ϕ_3	ϕ_2	ϕ_1
evaluate		ϕ_1	ϕ_1	ϕ_1	ϕ_3	ϕ_3	ϕ_1/ϕ_3
gate		ϕ_1	ϕ_2	ϕ_2/ϕ_4	ϕ_2/ϕ_4	ϕ_3	ϕ_4

Table 3.1: Datapath Delayed Write Buffer Timing

propagation delay is small enough to fit within the phase underlap. Section 2.4.2 describes the dwb operation the high design level. Table 3.1 shows the timing for each of the registers in the datapath.

In order to read from the DWB, the second half of the shift register must contain the proper value, and so the update phase cannot be after the evaluate phase. Several registers (RF, SDR, PC, PSW) must read available input from the bus while the DWB update and evaluate occurs; this is handled by removing the update transmission gate from the first cell so that basically becomes a latch. This cell is loaded during the update phase because its outputs will change immediately; the phase the first cell loads is indicated by *load1*.

BusD and busOUT are used twice a cycle with values being gated out ϕ_2 and ϕ_4 and written into the register ϕ_1 and ϕ_3. Registers that read from these busses must use metal control lines for the load transmission gate of the first stage so that the gate won't still be enabled the next phase. This doesn't increase the stride for the registers which already have one transmission gate removed, but for the MAR, the overall stride is increased

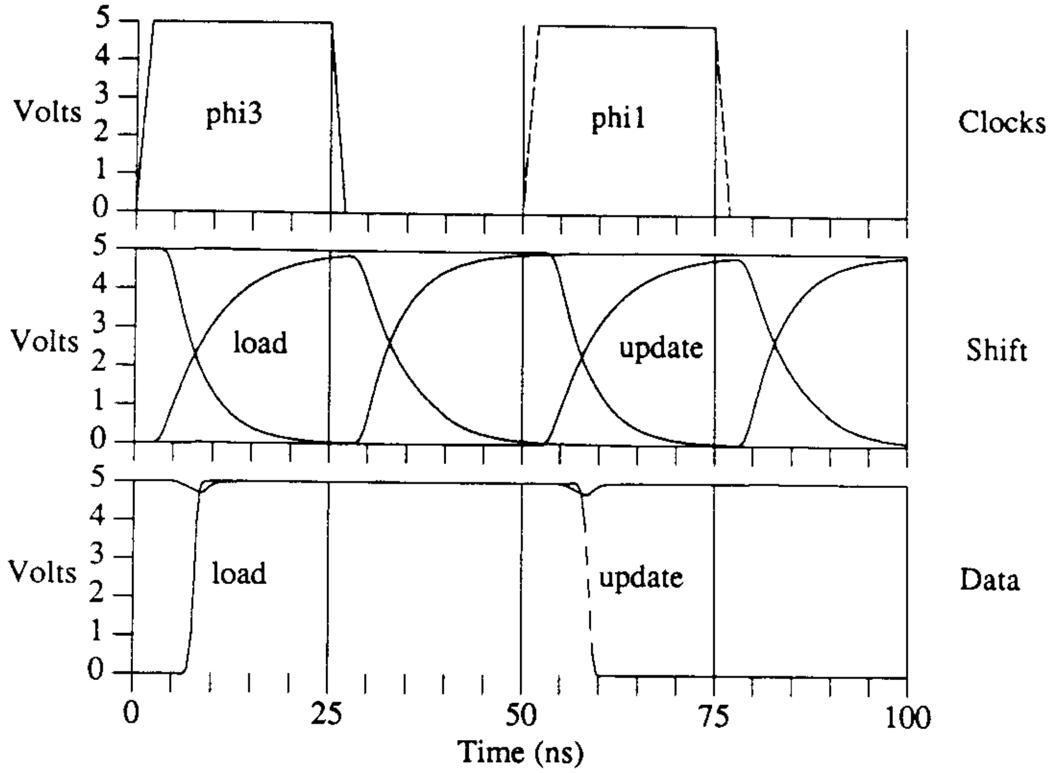

Figure 3.12: Delayed Write Buffer Shift

by 10λ.

Once data has made it to the end of the shift register, it is written into the permanent register only if the valid bit is 1. For the single registers (SDR, MAR, IR, PSW) the load signal is simply generated by ANDing the valid bit with the clock. For the PC, we must generate load and update signals because the permanent PC is a three element shift register itself.

The time it takes to shift is the same for all the DWBs because the shift control lines are buffered clock signals. Figure 3.12 shows the PC's DWB shift operation for a 100ns cycle. The clocks are asserted ϕ_3 and ϕ_1, and the shift lines which are driven by the buffered clocks change value accordingly. The second panel shows the shift lines at the end of the poly line (32nd bit) where the delay is the longest. During ϕ_3, the first stage of the shift register loads, and during ϕ_1, the second stage updates. The data loads and updates within 15ns after the clock edges, which covers the case of the IR which shifts during ϕ_4 and ϕ_2, the 15ns phases.

3.4.1.2. DWB Evaluation

In order to choose the proper value from a single register and its DWB, we apply a 1 to the select line and the selection circuit chooses the correct register and gates it onto the internal bus where it is held until the next evaluation. Evaluation is performed during ϕ_1 and ϕ_3, the 25ns phases, to give it enough time to find the correct value; the select line is tied to one of those two clocks. The 1 ripples through selection circuit until it reaches the first valid bit holding a 1, then the corresponding register is enabled and a 0 is passed on down the remainder of the selection chain. The control lines enabling each register to gate onto the internal bus are drawn in poly to reduce the pitch; as a consequence, the propagation delay causes the last bit to be the worst case.

The PC is the general case to be considered for register evaluation because in addition to selecting the first valid register, it also selects the second or third valid register to be gated onto another bus. Figure 3.13 shows a block diagram of the PC. Three permanent registers, npc, pc, and lpc, are needed because three different values may be chosen after all the DWBs are invalidated. The value of NXTPC is gated onto busNPC, where it is passed through an an incrementer (inc) before being sent to busOUT. There are two selection circuits to detect both the first valid bit for the NXTPC and the

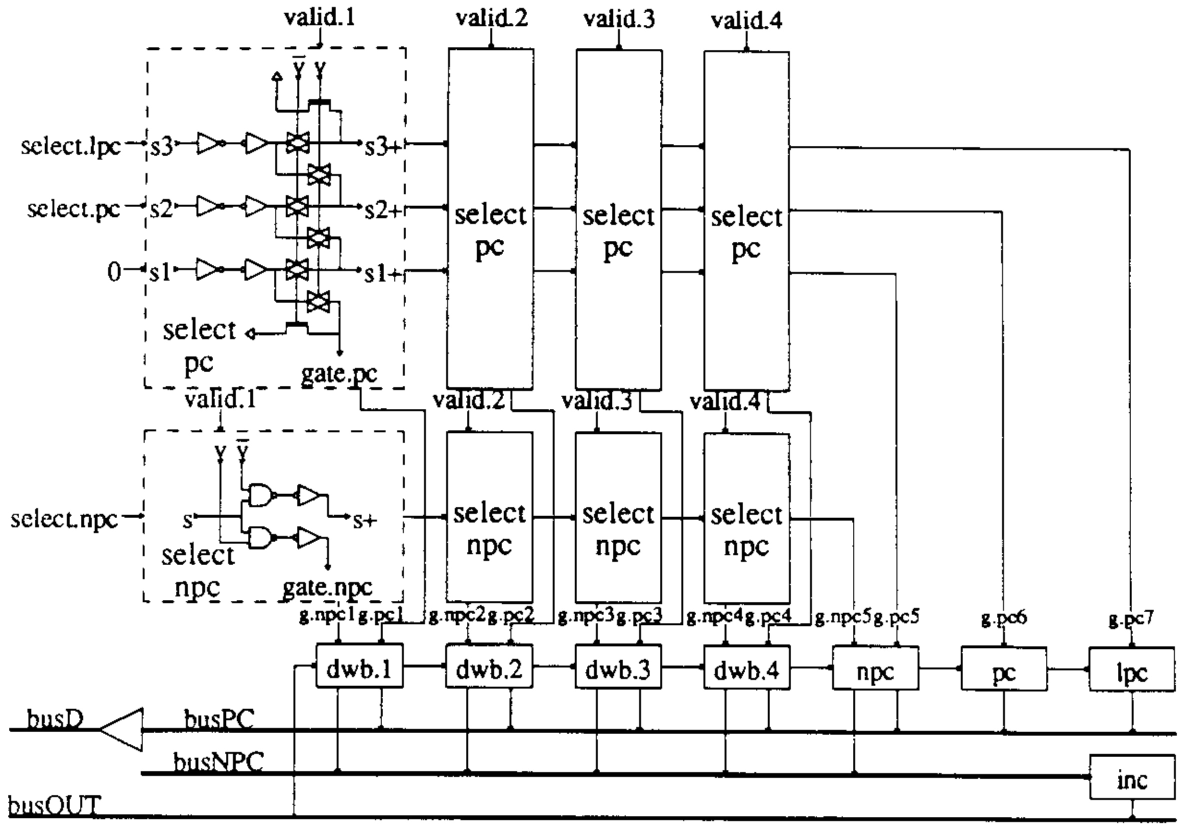

Figure 3.13: Program Counter with Delayed Write Buffer

second or third valid bit for the PC or LSTPC. In order to accomplish the latter operation, we use a three level selection circuit. We set exactly one of the three inputs to 1 and the rest to 0. At each stage, if the valid bit is 1, then the three inputs are shifted down one step with the bottom input going to gate the register. If the valid bit is 0, then we pass the inputs through and the associated register is not enabled. Two values may be read from the PC, and for this reason the first five DWB cells must be dual ported. Adding another transmission gate to the data cell increases the stride from 72λ to 92λ.

Timing of the PC evaluation also covers that of the SDR and PSW because 1) they all evaluate during ϕ_1, 2) the PC performs a first valid register selection, and 3) the PC's internal bus is longer than that of the SDR and PSW. Figure 3.14 shows the timing for

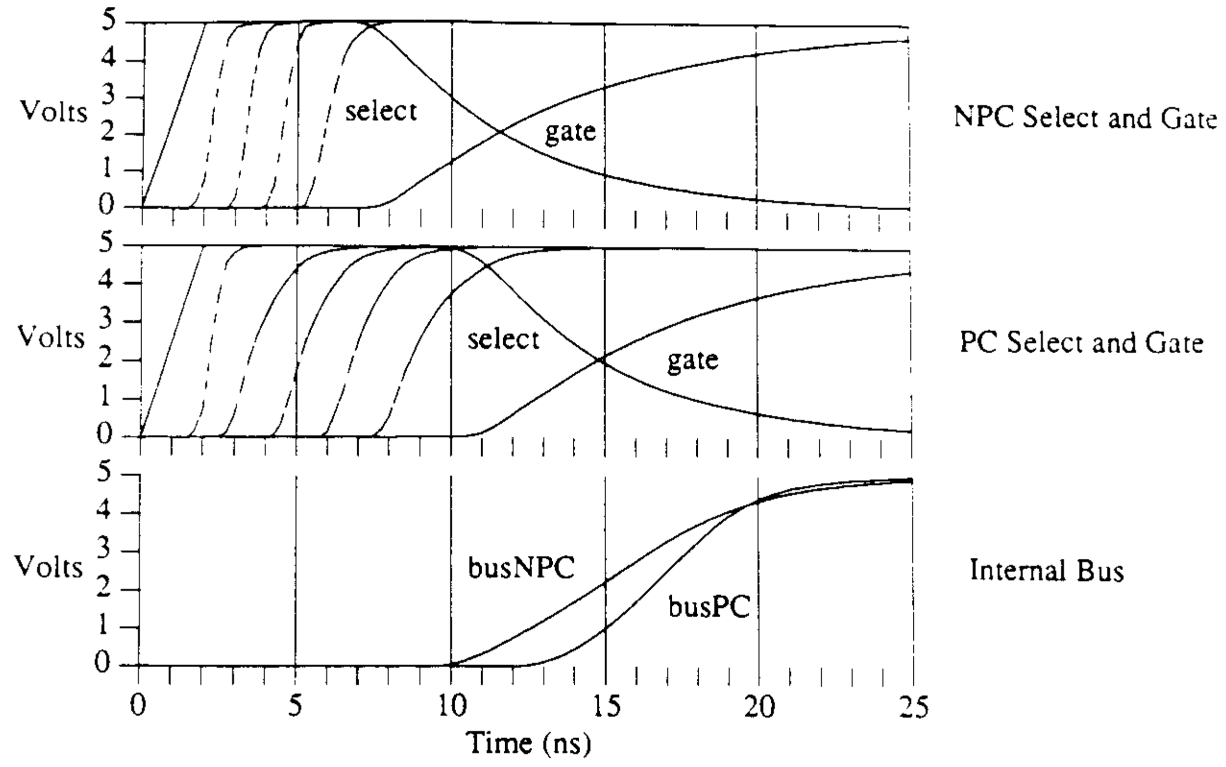

Figure 3.14: Program Counter Evaluation

the PC evaluation of the NXTPC and PC with all the valid bits cleared so that the selection circuit must ripple all the way through to the permanent registers. The first panel shows the selection circuitry rippling to the end where the permanent register gate line is enabled at the 32nd bit. The second panel shows similar happenings for the three level selection circuit to select the PC or NXTPC. The last panel show the internal busses set to the proper values within 25ns.

Both the IR and MAR operate under more tight timing constraints because they are used during the rollback cycle and must evaluate during ϕ_3, while the valid bits are being cleared. The MAR can meet its timing criteria using poly gate lines because it has 30ns (ϕ_3 + underlap) before busMAR's value is gated onto busOUT. Figure 3.15 shows

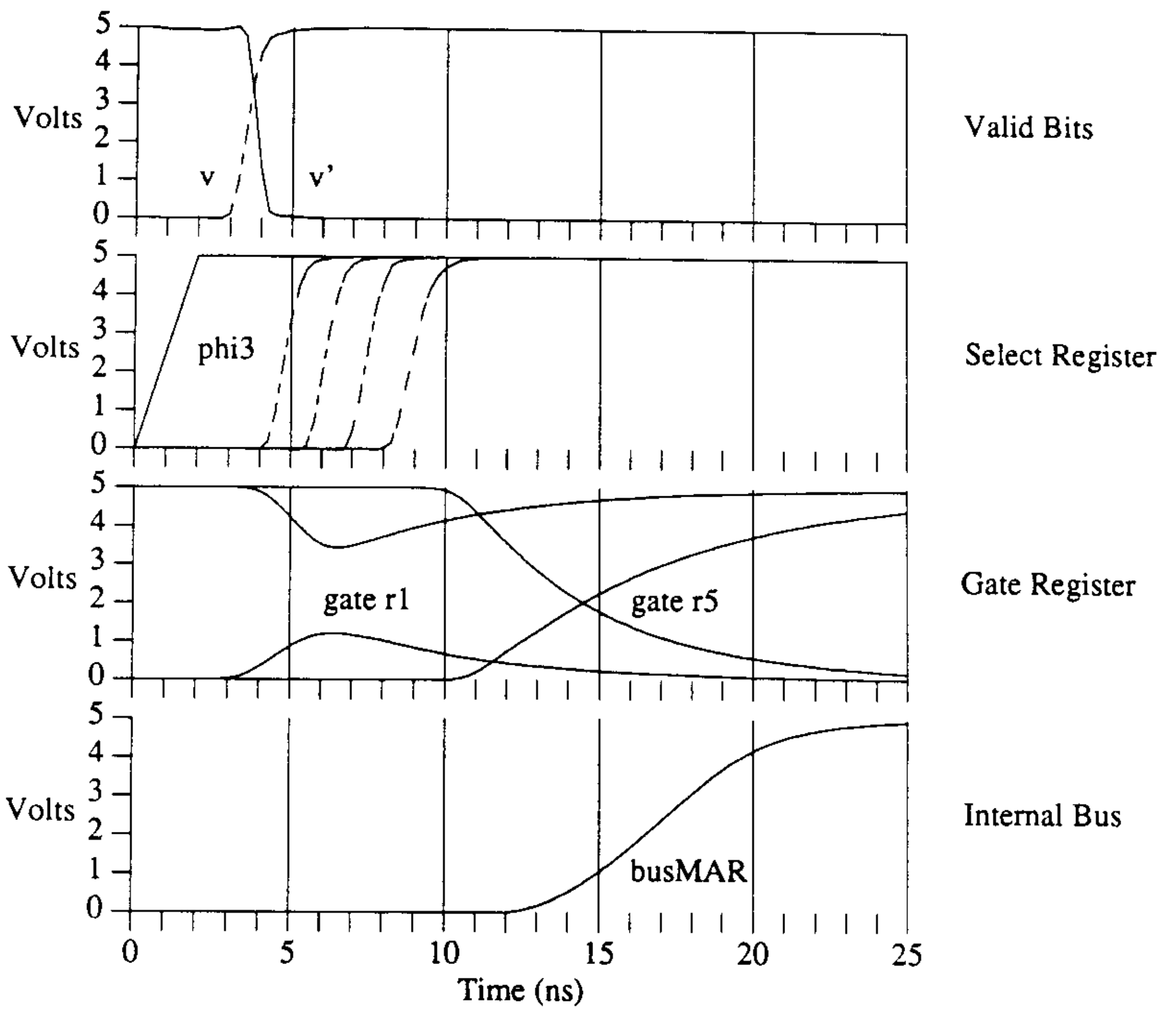

Figure 3.15: Memory Address Register Evaluation

evaluate of the MAR during the rollback cycle. Initially, all the valid bits are 1 and so the first register (R1) is starting to be gated onto the internal bus. All four valid bits are then invalidated so the selection circuit now must propagation to the permanent register (R5) which is then enabled while the first register is disabled.

The IR has even tighter timing because it must invalidate, evaluate, and gate onto busIN to simulate an instruction fetch all during ϕ_3 of the rollback cycle. The inputs to the controller's PLAs must be set up and the absolute address for the register file read

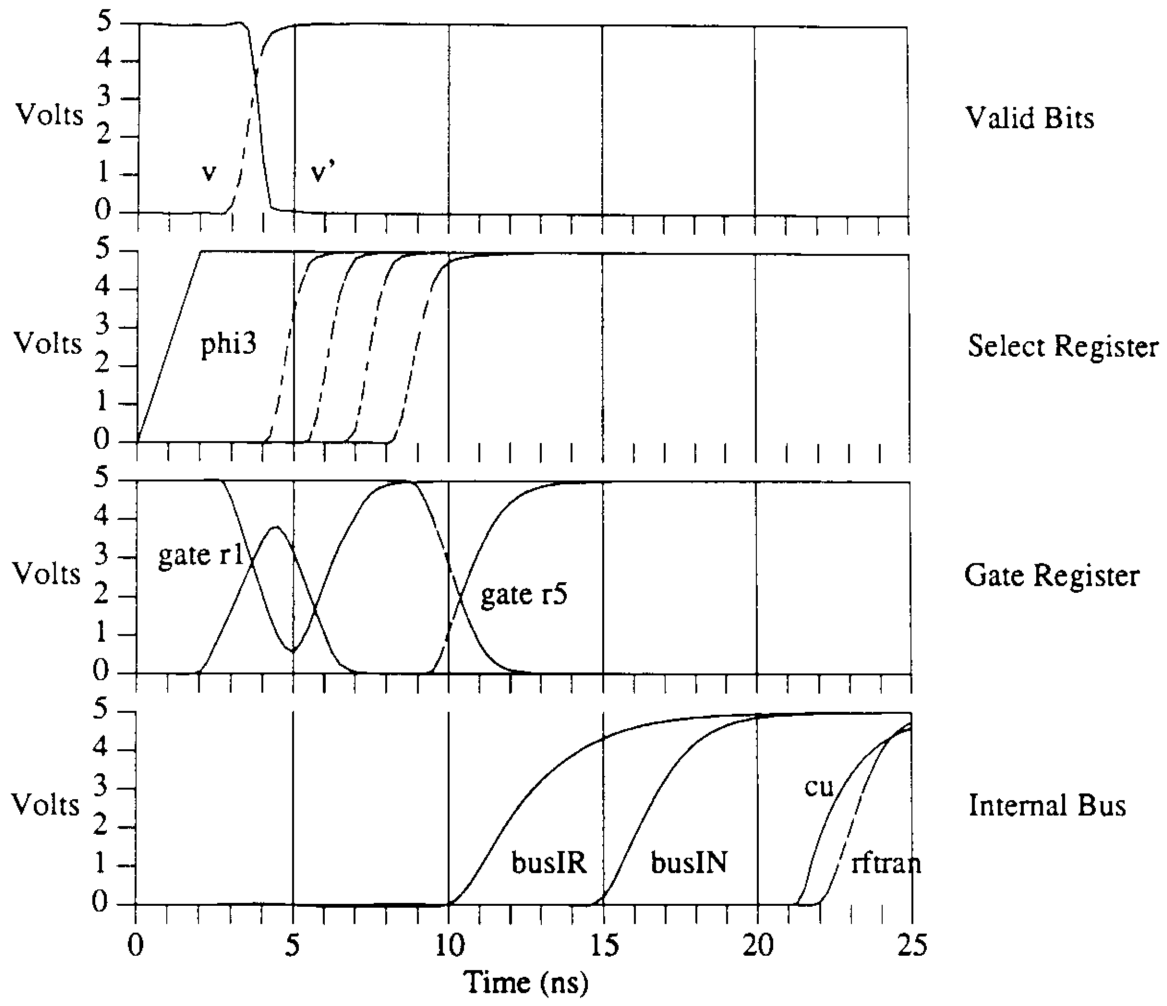

Figure 3.16: Instruction Register Evaluation

must be calculated by the register file translator during this "instruction fetch." Metal gate lines are used instead of poly to speed up gating the proper register onto the internal bus and busIN. Figure 3.16 shows the timing for this process. Note that the gate lines change value faster than those for the MAR because we replaced the poly lines with metal. The cost to stride is an increase from 72λ to 82λ for every DWB cell.

3.4.1.3. DWB Invalidation

During ϕ_3 of the rollback cycle the first n tags in every DWB are invalidated according to the rollback amount. The invalidate decoder is simply a piece of combinational logic that clears no valid bits for a rollback of 0, the first valid bit for a rollback of 1, the first two valid bits for a rollback of 2, and so forth.

The first tag of every register is written in on ϕ_4 or ϕ_1 instead of during ϕ_3 due to the delayed interrupt scheme which needs until ϕ_4 to cancel a write. In any case, all writes into the registers are cancelled by the controller during rollback, and so there's no Vdd/GND short circuit caused by the controller trying to write a 1 into the first register's valid bit while the register's invalidation circuitry is trying to set it to 0.

As shown before in the case of the IR and MAR, invalidation of the tags takes about 4ns. This is no problem for the registers that evaluate ϕ_1, but the invalidation causes reselection of a new value as shown in the timing of the IR and MAR. The PSW evaluates during ϕ_3 of the rollback cycle in addition to ϕ_1 because the proper register window is needed by the register file translator to calculate the register to read out of the register file the following cycle. We can use poly gate enable lines for the PSW because it is only 11 bits wide and the delay is minimal.

3.4.1.4. Register Gating

Register gating onto busD (PC, PSW) and busOUT (SDR, PC, MAR) is covered in section 3.3. Timing of the IR gating onto busIN is shown in section 3.4.1.2 because all three events (invalidation, evaluation, and gating) happen in one phase. The description of IR bus driving circuitry is shown in section 3.3.

3.4.2. Register File

The register file consists of the permanent register file and its delayed write buffer. Figure 3.17 shows a block diagram of the register file. The permanent register file is a dual-port read, single-port write, 33 bit by 74 address random access memory (ram). A ram column is selected with the decoder (dec), and the read and write operations are performed with the read/write circuitry (rw). Because there is an address associated with every piece of data written into the DWB, the tag is comprised of seven address bits and one valid bit.

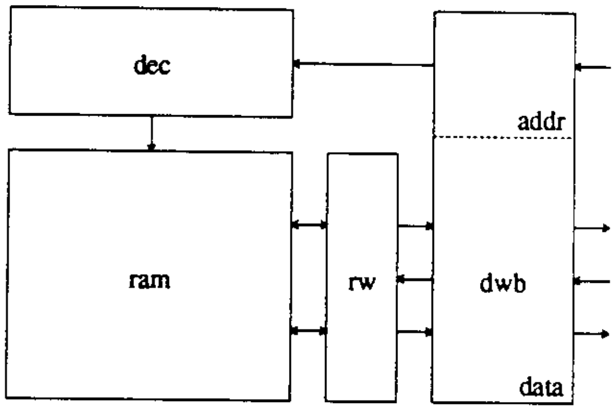

Figure 3.17: Register File Block Diagram

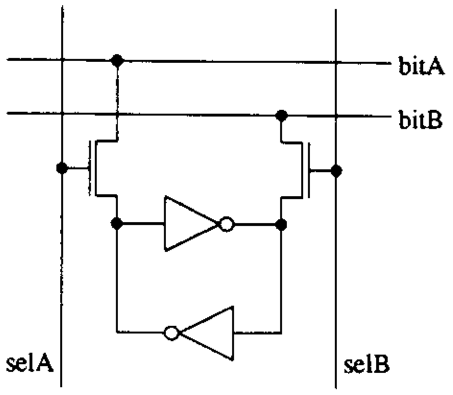

Figure 3.18: Register File RAM

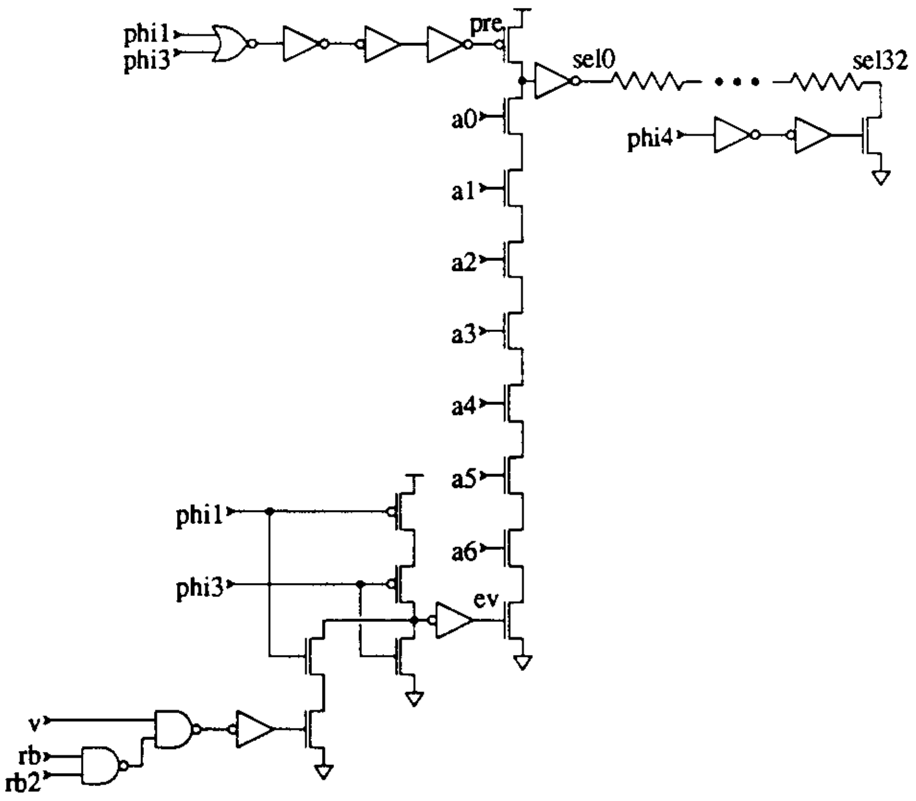

Figure 3.19: Register File Decoder

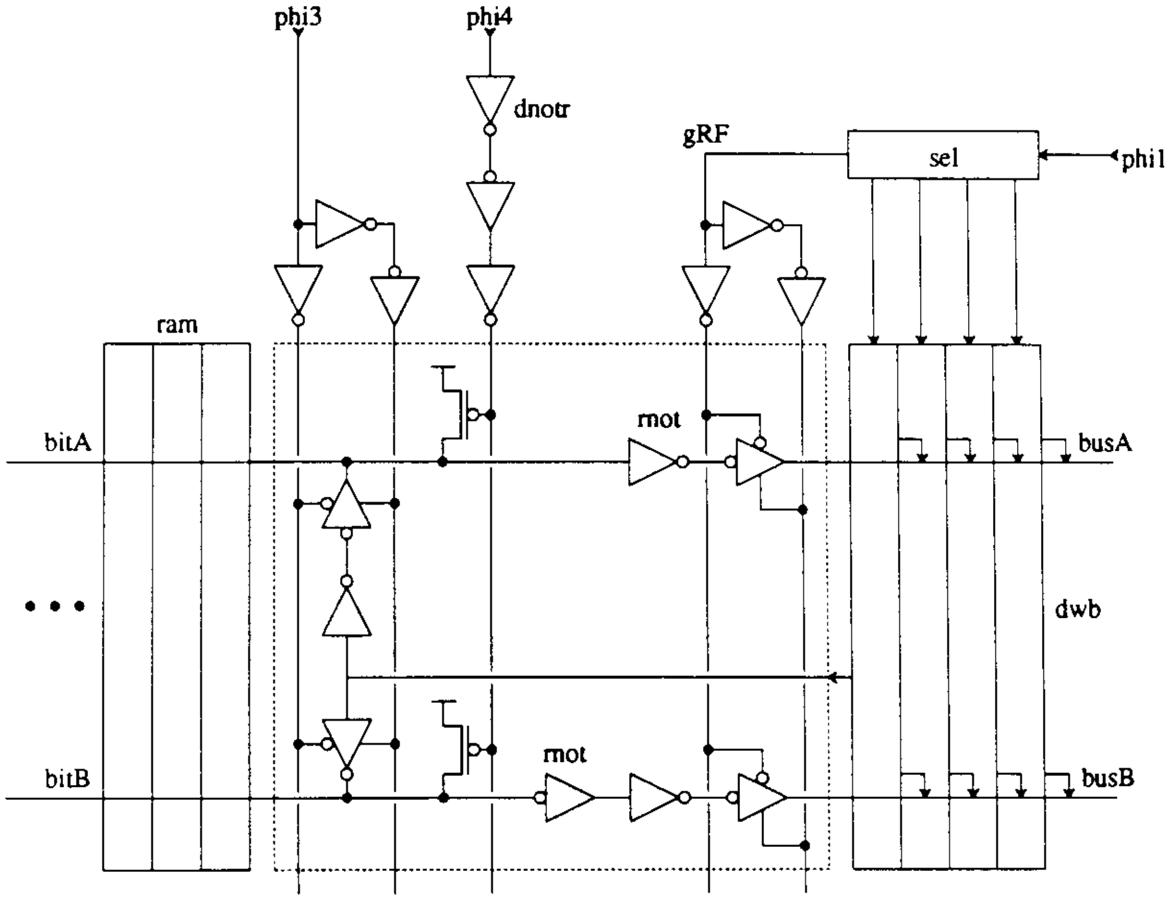

Figure 3.20: Register File Read/Write

3.4.2.1. Register File Circuitry

Figure 3.18 shows a register file ram cell, a six transistor static ram. Its dimensions are 39λ×32λ. The bit lines (bitA, bitB) run across in metal2, and the select lines (selA, selB) run up and down in poly. Because the select lines are in poly, we wanted to minimize the pitch; the register file's 39λ pitch determined the datapath's pitch.

Figures 3.19 and 3.20 show the decoder to drive the ram's select lines and the read/write circuitry to drive the ram's bit lines. The address space consists of locations 0-9 and 64-127. The register file translator (section 3.6.3) takes a register number and the current window and derives a unique seven bit address falling within this range. This

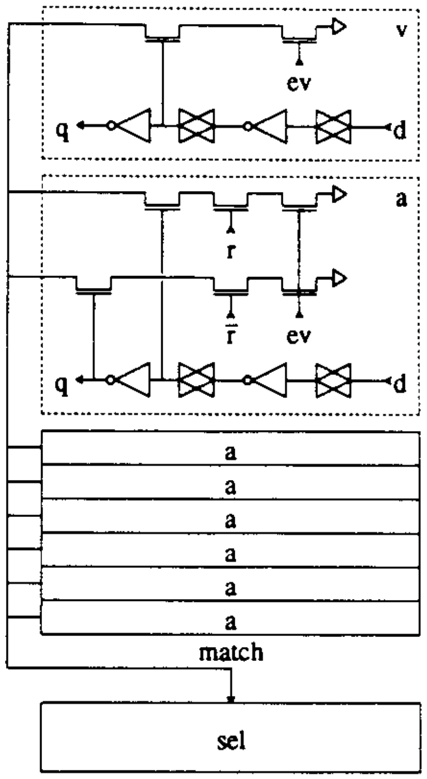

Figure 3.21: Register File Delayed Write Buffer Tag

is then used by the precharged decoder to select the proper register. Every register has two decoders, one to connect to bitA, and one to connect to bitB. BitB is an inverted bus because it is connected to the inverted value in the ram cell. The selection circuit (sel) shown in figure 3.20 is the same static selection circuit used by the prototypical DWB.

The RF's DWB is dual ported, just like the PC's, only the tag is more complex. Each data entry has a corresponding address, and so the tag consists of a seven-bit address and a valid bit. Figure 3.21 shows the RF DWB tag circuitry. Because the RF is dual ported, there is actually two sets of comparison logic, but only one set is shown for clarity. We use a precharged match line to determine if a DWB entry is valid. If any of the address bits in the address registers does not match with the requested address (r, $\bar{r}$), or if the entry is not valid, then the match line will be discharged and the selection circuitry (like that used in figure 3.11) will skip that register. If none of the DWB's entries match the address, then the permanent register is selected (figure 3.20).

Figure 3.22 shows the circuitry to manage the parity on the data and also shows how the register file actually interfaces with the rest of the datapath. The bottom slice represents the 32 bits of data and how they are gated onto the busses. BusA and busB are the internal busses used by the DWB and permanent register file to store the value read out. The inverted value is read out and used by tri-state inverters to drive the busses (see section 3.3 for gating onto busses). Not shown are the lower five bits read off of $\overline{\text{busB}}$ for the shift decoder.

The middle slice is the parity bit. Before the parity is written in from parD, it is XORed with the parity of the destination address (rd) generated by parity generator p.d. This same parity generator also generates the 4-bit state of the address, state.rd, for state exportation. At this point, the parity can also be inverted (invert) for the *add with bad parity* instruction. So that the controller only has to deal with parity of the data, the address parity is removed from the parity read out of the register file by XORing it again with the parity of the address (ra, rb) it was read out of. The source address parity generation circuits, p.a and p.b, actually reside in the register file address translator. The parity read out of the register file (rfA.par, rfB.par), along with the parity generated from the data read out of the register file by parB (busB.par) and parD (busD.par) is sent to the controller for error checking.

Data is logically written into the register file by writing a 1 into the valid bit.

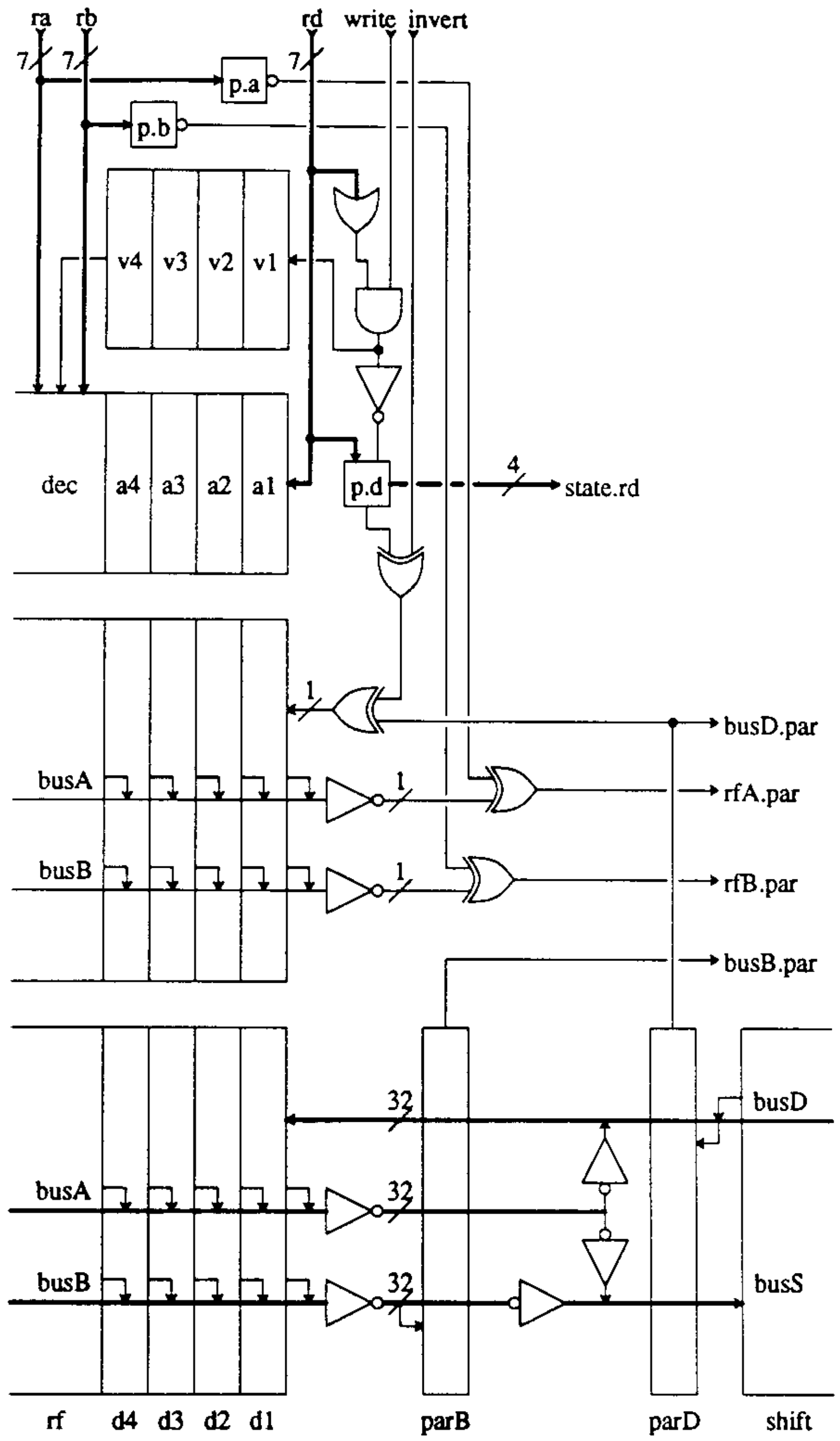

Figure 3.22: Register File Parity

Because r0 is permanently hardwired as zero, if an instruction should try to write any value into that register, then the write will be cancelled so that any reads from r0 will always result in zero. Upon powerup, the register file's state will be unknown, so at least four cycles must pass before r0 may be reliably read.

The parity of the address to be written to is generated from the four bits of compressed address; this state also includes the valid bit. Only data with a valid bit of 1 will be read out of the DWB, and so the bit is inverted before being used to generate the state so that the parity of the address will not be changed. (Only v=1 will be read out, $\bar{v}$=0, and par(addr) XOR $\bar{v}$ = par(addr) XOR 0 = par(addr).)

3.4.2.2. Register File Timing

Figure 3.23 shows the timing diagram of a read from the register file's last entry in the DWB. The first panel shows the match lines for the first three register initially charged to 5V, and then being discharged. Initially, the first register is valid because the match line is valid; after evaluation, the match lines become discharged and so the selection propagates to the last register as shown in the second panel. The third panel shows the poly gate lines; note the same behavior as that seen for the MAR (figure 3.15); the first register is initially chosen and then it becomes invalid with time.

BusA needs to be stable by the beginning of ϕ_2 so that it can be gated onto busD. BusB needs to be gated onto busS as it is read out so that the input to the precharged shifter is ready by the beginning of ϕ_2. In addition, the lower five bits are loaded into the shift decoder and the parity of busB is generated and driven to the controller. These

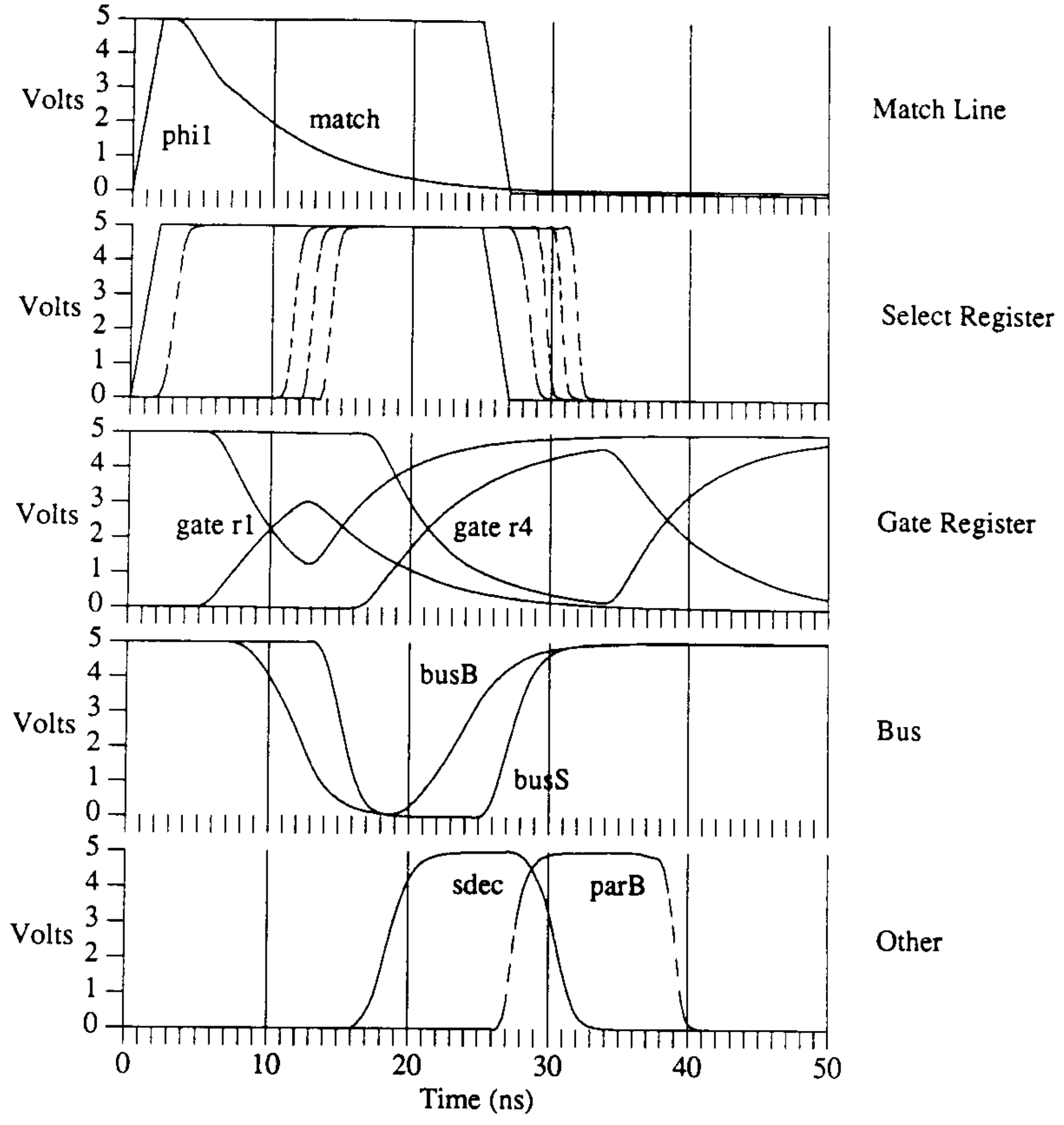

Figure 3.23: Register File Delayed Write Buffer Read

latter two actions have all of ϕ_2 to finish and are no problem.

Every cycle, two values are read from the ram during ϕ_1. Both bit lines (bitA, bitB) are precharged on ϕ_4, and then the decoder is evaluated on ϕ_1. The proper registers will be connected to the bit lines, and if the value is 0 then the bit line will be discharged.

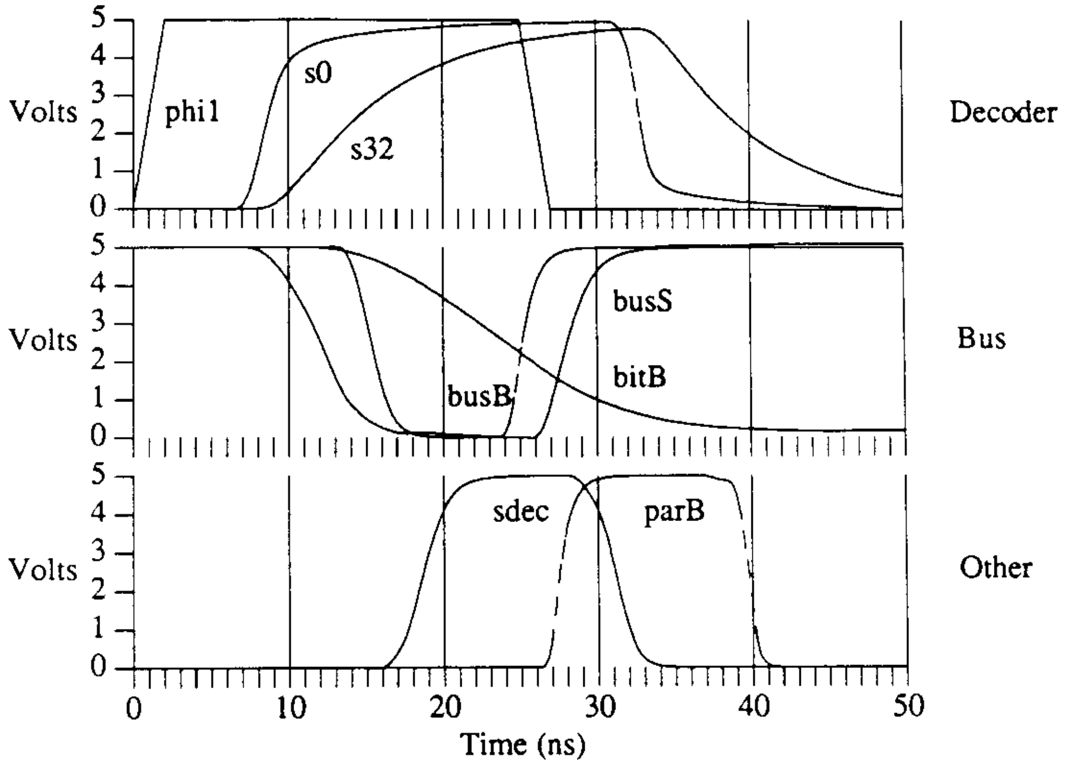

Figure 3.24: Register File Ram Read

Figure 3.24 show the worst case read onto busB. The first panel shows the select lines from the decoder going into the ram. S0 is at the first cell, and s32 is the signal at the end of the poly line. BitA and bitB are read by ratioed inverters (rnot) with a P:N::4:1 ratio in order to trigger quickly on the falling edge of the bit lines being discharged. Even so, the value of busS is not quite at 5V by the beginning of ϕ_2 (30ns). However, this is acceptable because the precharged shifter will only discharge a line if the corresponding input is a 1, and so the shifter can tolerate a late 0 to 1 transition.

During ϕ_1, the value on busD (which was placed there during ϕ_4) is written into the first register of the register file's DWB. This action, consisting of charging an inverter gate to a new value, will take the same amount of time as the update in the other registers. The parity, however, is calculated by parD during ϕ_1; its writing into the register file and reading out again is covered in section 3.5.2. If the data to be read out of the register file resides in the DWB, then its match line will not be pulled low during the evaluation phase, and so the selection circuitry will select the register containing the data to be gated onto busA or busB. If there are no matches in the DWB, then the data read from the permanent register file will be gated onto busA or busB instead.

Once a cycle, a value from the last register in the DWB is written into permanent storage if it is valid. The data and its complement is driven onto bitA and bitB by the write circuitry on ϕ_3 regardless if it is valid or not. Choosing whether to write it into the ram or not is done in the decoder (figure 3.19). The destination address is driven onto the address lines for both address selectors, and the decoder evaluates if the valid bit (v) is 1. If a rollback of four cycles happens, then the data to be written into the permanent register file will become invalid, but there is not enough time to cancel the write. Instead, the rollback signal (rb) and rollback amount (rb2) are also used to determine if there should be a write or not. Once the target register is connected to the bit lines, the inverters in a feedback loop will be quickly overwritten by the capacitance of the bit lines and the large drivers. Figure 3.25 shows this action. The third panel shows the select lines being enabled as the bit lines are driven to their proper value in panel 4 and the ram quickly changes value.

After a read on ϕ_1, the select lines have ϕ_2 to become disabled. However, following a write, the bit lines must be precharged for the following read, and so the select lines must be discharged faster. We add pulldown transistors to the ends of the poly select

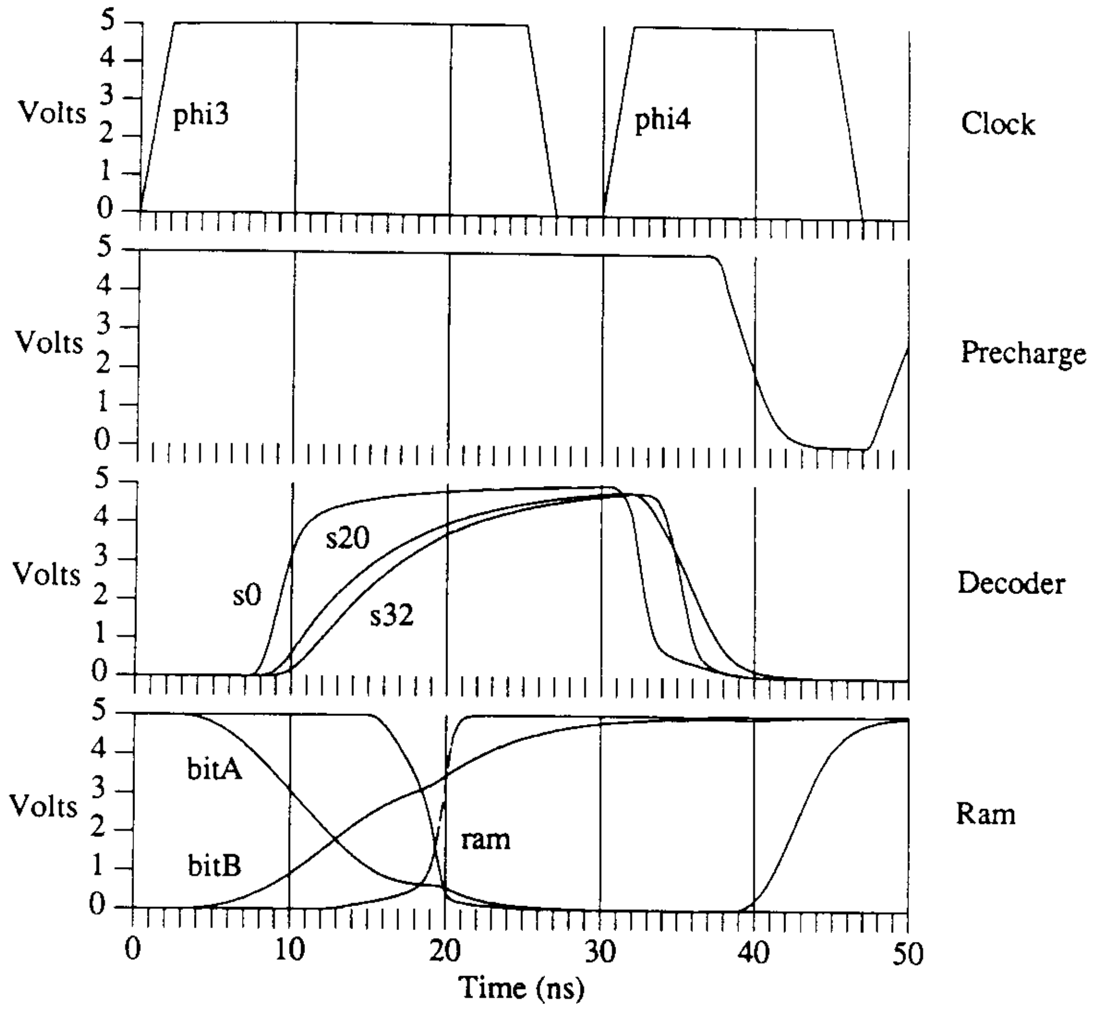

Figure 3.25: Register File Ram Write

lines to discharge on ϕ_4; this moves the slowest point to bit 20. In order to insure that the ram won't be affected by the precharging bit line, we delay the precharge using a delay circuit (dnotr) similar to the delay inverters used in the clock generator (figure 3.2).

3.4.3. Program Counter

In order to prepare for the next instruction fetch, the PC reads the NXTPC during ϕ_1, and increments the value by four using a modified manchester carry chain. The NXTPC contains the value of the currently fetched instruction, and so four bytes down is the next instruction to be fetched if there is no branch taken.

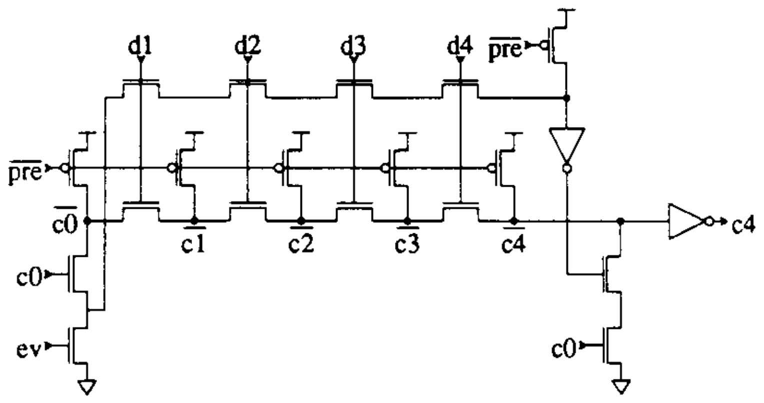

Figure 3.26: Program Counter Manchester Carry Chain

Figure 3.26 shows the manchester carry chain used in the incrementer. Because only a constant 4 is being added every time, only the third bit is 1 and the rest are 0. This means there is no need for the generate signal, and the data (d#) can be directly used for the propagate signal (see figure 3.40 for a full manchester carry chain).

Figure 3.27 shows the timing involved in the NXTPC increment. The value to be incremented is read out of the PC's NXTPC during ϕ_1 and will be stable by the end of ϕ_1. During ϕ_2, the chain is precharged, and it is evaluated on ϕ_3. The second panel shows the worst case of the carry propagating from the third bit down the chain using the carry bypass. At the end, the carry is XORed with the operand to obtain the sum. This is needed by the beginning of ϕ_4 to be gated onto busOUT, and so there are no problems

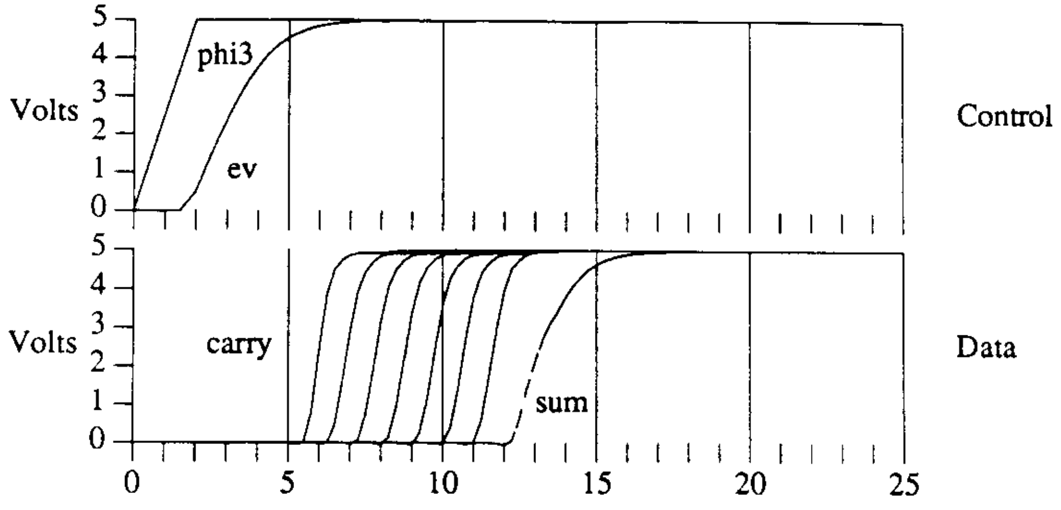

Figure 3.27: Program Counter Incrementer Evaluation

with timing.

3.4.4. Processor Status Word

Figure 3.28 shows a block diagram of the PSW. At the left is the PSW register and its DWB, as described in section 3.4.1. In the middle is the latch used to hold the value read from the PSW register stable while the next PSW value is generated in the NXTPSW block, shown on the right.

In a normal cycle, the value from the previous cycle is read out onto PSW.data on ϕ_1. On ϕ_2, the latch updates busPSW with this value, and it is used by the NXTPSW block to generate the processor status for the current cycle. This new value is returned to the PSW's DWB on busNXTPSW, ready to be written in on the following ϕ_1. The latch is needed because the first element in the DWB may be simultaneously written into and read out of on ϕ_1, and since the NXTPSW block consists of pure combinational logic, a potential feedback loop exists. The latch loads on ϕ_2 to break this loop. In addition, a

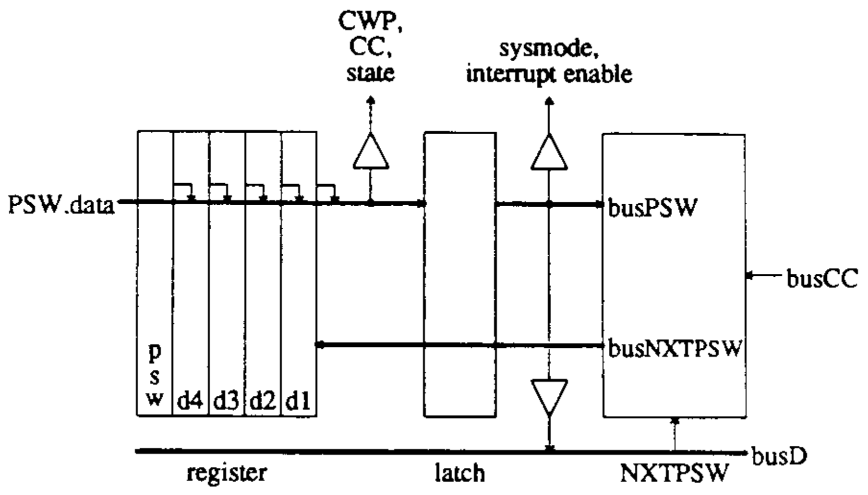

Figure 3.28: Processor Status Word

reset causes the output of the latch to be set to all zeros so that busPSW will be in a known state for the next PSW generation. During ϕ_3 of the rollback cycle, the PSW rolls back, and so the latch also loads at this time.

The PSW must export its internal state to the other chip for comparison. The four-way interleaved parity is generated from the value read out onto PSW.data on ϕ_1, and it is sent to the controller where it is merged with the other compressed state from the rest of the datapath.

The next PSW value is generated by selectively gating values from different sources onto busNXTPSW. Figure 3.29 is an ISP description of this process. Note that though each field in the ISP description is generated by a mutually exclusive if-then-else clause, it is possible in the actual circuit to gate several of these values onto busNXTPSW to get an indeterminate value.

The NXTPSW block is divided into four sections corresponding to the four fields within the PSW (condition codes, system modes, saved window pointer, and current

```
PSW_CWP          :=      PSW<10:9>,
PSW_SWP          :=      PSW<8:7>,
PSW_ISP          :=      PSW<6:4>,
PSW_I            :=      PSW<6>,
PSW_S            :=      PSW<5>,
PSW_P            :=      PSW<4>,
PSW_CC           :=      PSW<3:0>,

busD_CWP         :=      busD<10:9>,
busD_SWP         :=      busD<8:7>,
busD_ISP         :=      busD<6:4>,
busD_CC          :=      busD<3:0>,

NXTPSW_CWP       :=      NXTPSW<10:9>,
NXTPSW_SWP       :=      NXTPSW<8:7>,
NXTPSW_ISP       :=      NXTPSW<6:4>,
NXTPSW_CC        :=      NXTPSW<3:0>;

/*
 *       Condition Codes
 */
if (load.PSW_busCC and (not state.int))
        NXTPSW_CC=busCC
else if (load.PSW_busD and (not state.int))
        NXTPSW_CC=busD_CC
else
        NXTPSW_CC=PSW_CC;

/*
 *       Interrupt, System Mode, Previous System Mode
 */
if (state.int)
        NXTPSW_ISP=0 concat 0 concat PSW_S
else if (load.PSW_busD and (not state.int))
        NXTPSW_ISP=busD_ISP
else if (ctrl.PSW_reti and (not state.int))
        NXTPSW_ISP=1 concat PSW_P concat PSW_P
else
        NXTPSW_ISP=PSW_ISP;

/*
 *       Saved Window Pointer
 */
if (load.PSW_busD and (not state.int))
        NXTPSW_SWP=busD_SWP
else
        NXTPSW_SWP=PSW_SWP;

/*
 *       Current Window Pointer
 */
if (load.PSW_busD and (not state.int))
        NXTPSW_CWP=busD_CWP
else if (state.int)
        NXTPSW_CWP=PSW_CWP
else
        NXTPSW_CWP=PSW_CWP+ctrl.CWP_inc;
```

Figure 3.29: Next Processor Status Word Generation ISP Description

window pointer). The generation of the values for each field is independent of the others. The condition codes are updated with those from the ALU sent on busCC if the *set condition code* bit is set in the executing instruction. If the instruction is a *putpsw*, then the condition code field is taken from busD (which is why the *scc* bit should never be set on a *putpsw* instruction). If neither of these conditions are fulfilled, or if an interrupt occurs, then the old value of the condition code field is used instead.

The system mode field (interrupt enable, system mode, and previous system mode) is changed in the case of an interrupt, a *reti* instruction, or a *putpsw* instruction. An interrupt causes the interrupt enable bit to be cleared, the system mode bit to be set to kernel mode(0), and the old value of the system mode bit to be saved in the previous system mode bit. Returning from an interrupt procedure with *reti* will cause the interrupt enable bit to be set and the system mode bit to be restored from the previous system mode bit. The *putpsw* instruction causes this field to be read from busD, and if none of these conditions apply, then no changed is made by using the old value.

The only way the saved window pointer may be changed is through the *putpsw* instruction, and this is cancelled upon an interrupt. Ctrl.CWP_inc is a two bit bus containing 11 (−1), 00, or 01 and is used to decrement, leave alone, or increment the current window pointer. If the executing instruction is a *putpsw*, then the current window pointer will be set from busD. An interrupt will override this change, as well as the default action of adding ctrl.CWP_inc to the current window pointer, and instead, cause the old current window pointer value to be used.

3.5. Error Detection

This section describes the modules used for error detection in the *MP*. They can be divided into three categories: those that perform parity generation using static circuits, those that perform parity generation using dynamic circuits, and those that perform state comparison.

3.5.1. Static Parity Generation

ParIN, parOUT, and parB (see figure 2.2) are used to generate the parity of the data placed upon busIN, busOUT, and busB. The parity can be quickly and efficiently generated using a two level compressed XOR tree[Trem89a] as shown in figure 3.30. The layout only requires two levels of XOR cells, and no crossover of signals means only one layer of routing is needed. The ten transistor static XOR circuit used through the *MP* design is shown in figure 3.31.

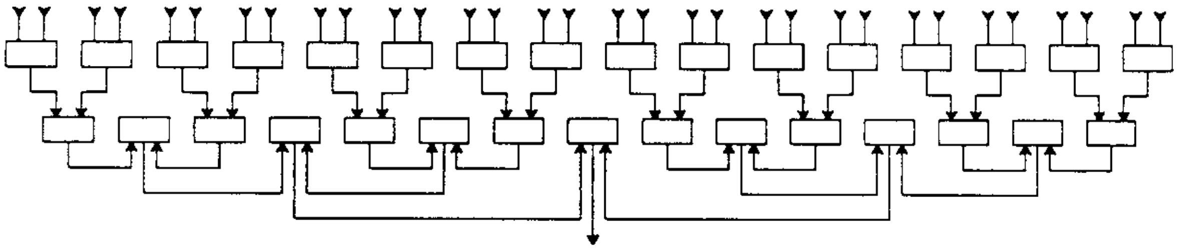

Figure 3.30: Compressed XOR Tree

Each XOR takes about 2.5ns to evaluate, and a path through the 32 leaf XOR tree is five nodes for a 12.5ns propagation time. The stride of each XOR is 28λ, and the stride of the parity generator (2 XORs + routing) is 87λ.

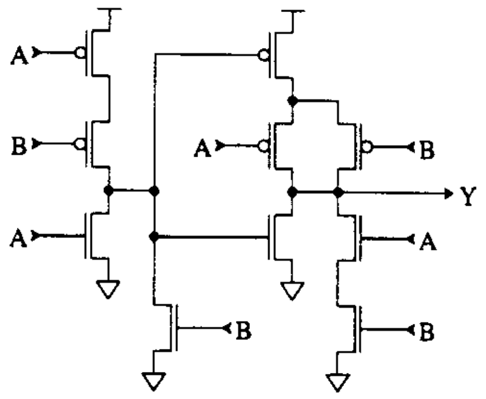

Figure 3.31: Static XOR

3.5.2. Dynamic Parity Generation

Generating the parity of busD is a special case, because not only is one bit of parity generated to protect the register file, but the four-way interleaved parity is also needed for internal state compression. One way to do this is use a modified XOR tree with at least six more lines of routing between the nodes to accommodate the four-way interleaving. However, this adds too much to the stride of parD, which we want to minimize in order to minimize the overall stride of the datapath.

For the parD implementation, we use a dual-rail precharged parity chain[Trem89a]. Figure 3.32 shows one cell in this chain. Both p and $\overline{p}$ are precharged, and upon evaluation, the $\overline{p}$ at the end of the chain is discharged. If d is zero, then the values on p and $\overline{p}$ will be passed on to p+ and $\overline{p+}$. If d is one, then the parity passing through the cell will be inverted by passing p and $\overline{p}$ on to $\overline{p+}$ and p+. We implement four-way interleaving by connecting every fourth cell together into a chain of eight cells. Figure 3.33 shows how this can be done with a minimum of routing crossover. Once the four bits of parity are generated, they can be merged together with an XOR tree to get the one

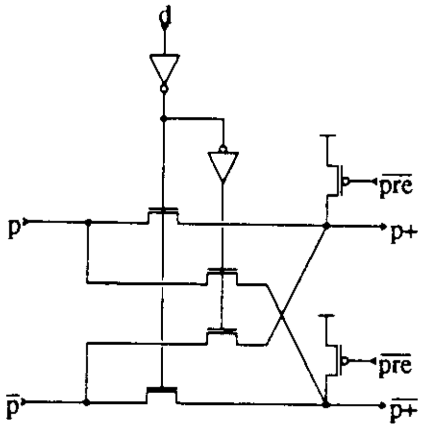

Figure 3.32: Dynamic Parity Cell

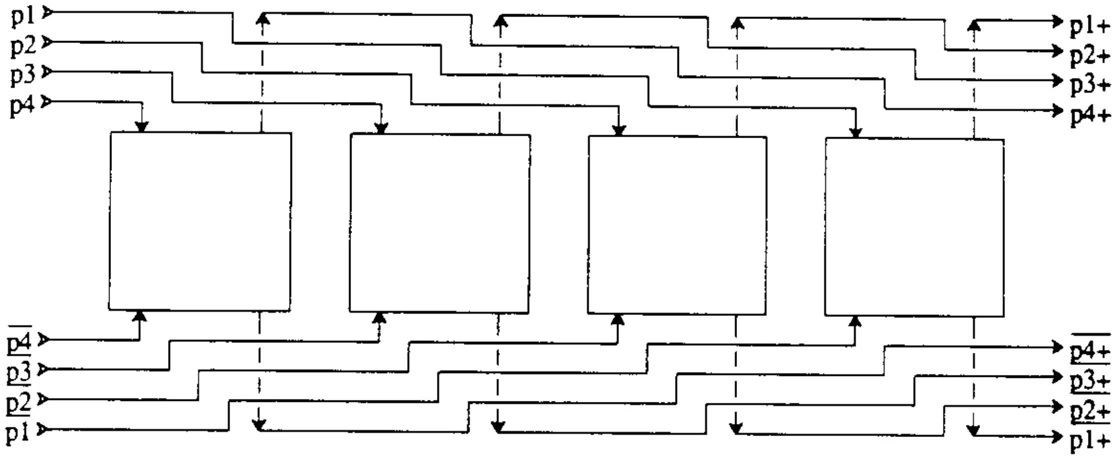

Figure 3.33: Parity Chain Using Dynamic Parity Cells

bit needed by the register file and controller. The resulting stride (including 24λ for vertical power lines) is 176λ.

ParD evaluates on ϕ_1 to calculate the compressed state of busD (state.busD). These four bits are then reduced to one bit for writing into the register file. The critical path exists when the first DWB register is selected so that the parity written into the register file on ϕ_1 is immediately read out and sent to the controller. On ϕ_3, parD evaluates to generate the parity of busD, which is usually the value of busA read out of the register file.

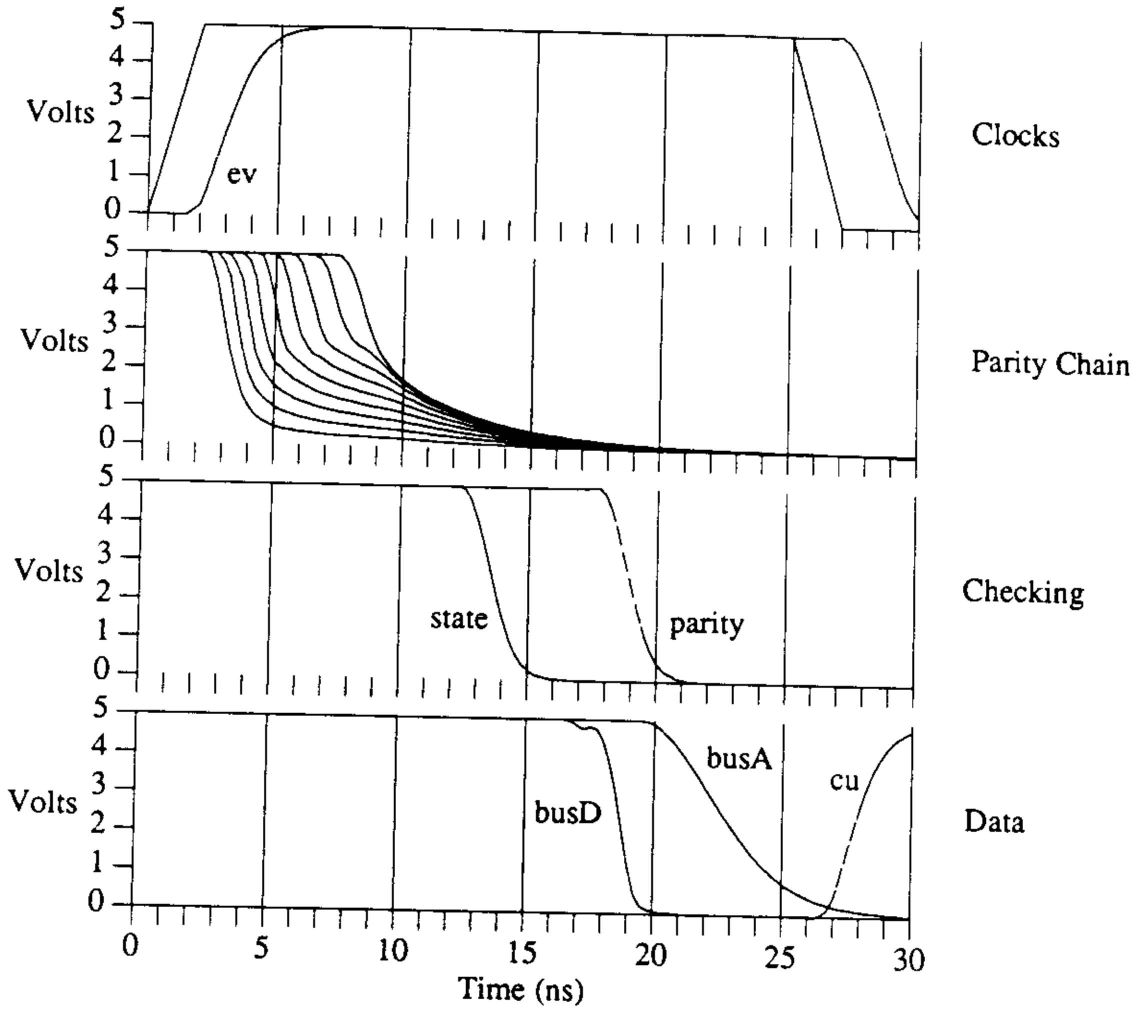

Figure 3.34: BusD State Compression and Parity Generation

Figure 3.34 shows the timing of parD. The second panel shows the parity chain being discharged upon evaluation. In the third panel, the line labeled state shows the amount of time it takes to send the four-bit compressed state to the controller on ϕ_1, where it will be merged with the compressed state from the rest of the datapath. The line marked parity shows the time it takes to generate the parity of busA on ϕ_3 and to send it to the controller. The critical path, shown in the fourth panel, is a write of the parity bit into the register file (busD), the read out (busA), and the drive to the controller (cu). Since the controller does not need the parity read out of the register file until the middle of ϕ_2, there are no timing problems.

3.5.3. State Comparison

The master processor exports all its states to the slave processor where it is compared. The slave processor, because it is an identical chip except for the setting of pad.ms, also sends all its states to the pads. Thus, the logical place to compare the processor states is at the slave processor's pads. Placing an XOR between the input from the pads and the output to the pads as shown in figure 2.11 and described in section 2.3 is enough to catch any differences in state.

Errors arising from differences in control outputs are sent directly to the controller. Any errors from the 33 bit address/data bus, however, needs to be merged before being sent to the controller. We accomplish this using two two-level OR trees (figure 3.35) to reduce the 33 bits (16+16+1) down to 3 bits for the controller. Each OR node is positioned next to the comparators by the pads, and so the only extra area taken up is by the routing between the nodes.

3.6. Combinational Logic

This section describes the modules in the *MP* that are mainly combinational in nature and either don't need to be rolled back, or restore their values from other rollback modules. The modules described are: the shifter, the arithmetic and logic unit (ALU), the register file address translator (RFTRAN), the immediate register (IMM) and the data immediate sign extender (DIMM), the byte address register (BAR), and the shift amount

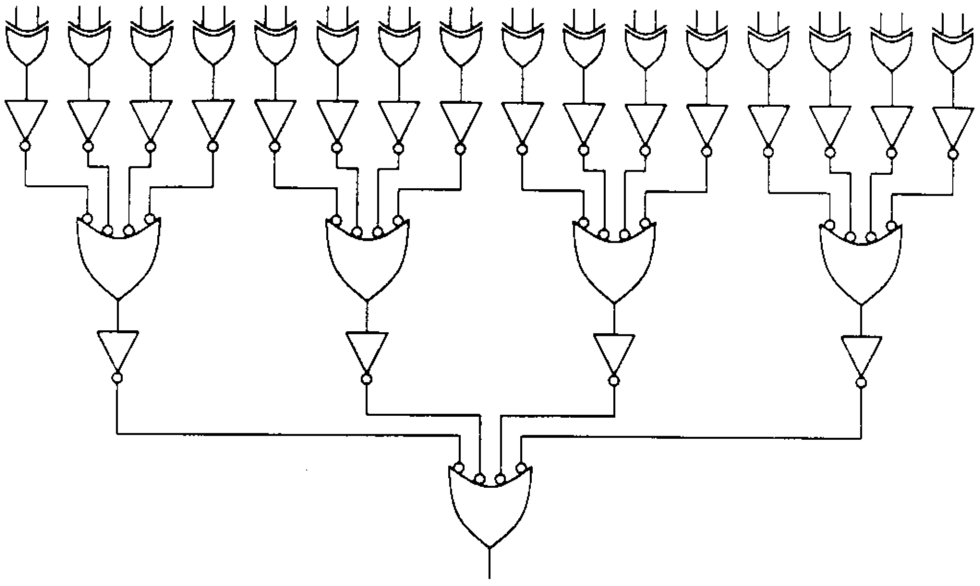

Figure 3.35: OR Error Merging Tree

decoder (SDEC).

3.6.1. Shifter

The *MP*'s 32-bit dynamic shifter performs left shifts, logical right shifts, and arithmetic right shifts by 0, 1, 2, 8, 13, 16, and 24 bits. Figure 3.36 shows a four bit arbitrary amount shifter representative of the *MP* shifter.

Two internal busses, $\overline{\text{busR}}$ and $\overline{\text{busL}}$, are precharged during every ϕ_1 and ϕ_3. For right shifts, a fill bus ($\overline{\text{busF}}$) used to extend the sign bit is precharged at the same time. The shift amount decoder (section 3.6.6) drives the poly shift amount lines controlling the transistors that connect the two internal busses together at various shift points. The shift amount lines are set during ϕ_1 and ϕ_3, and the shift evaluation happens during ϕ_2 and ϕ_4.

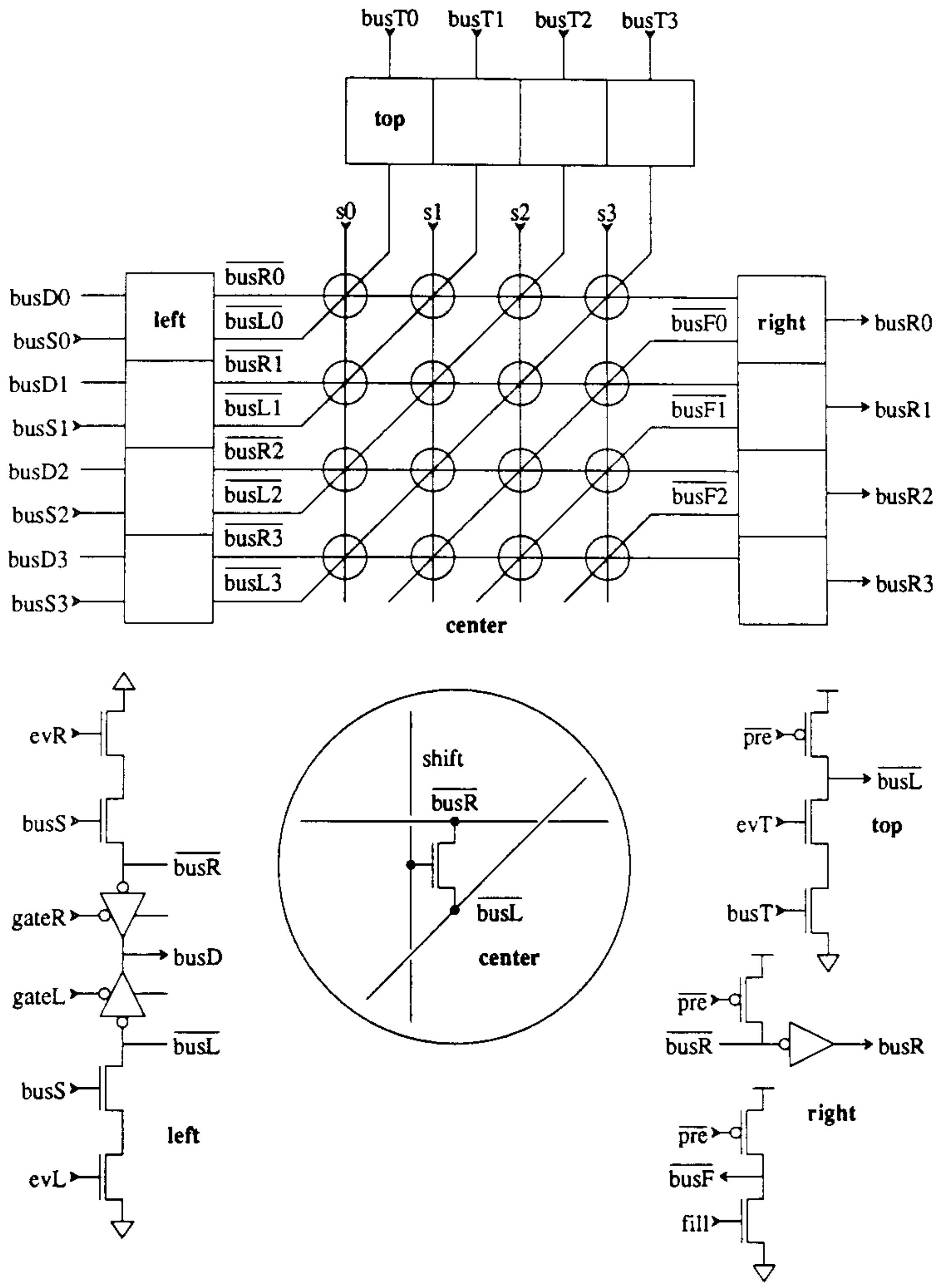

Figure 3.36: Shifter

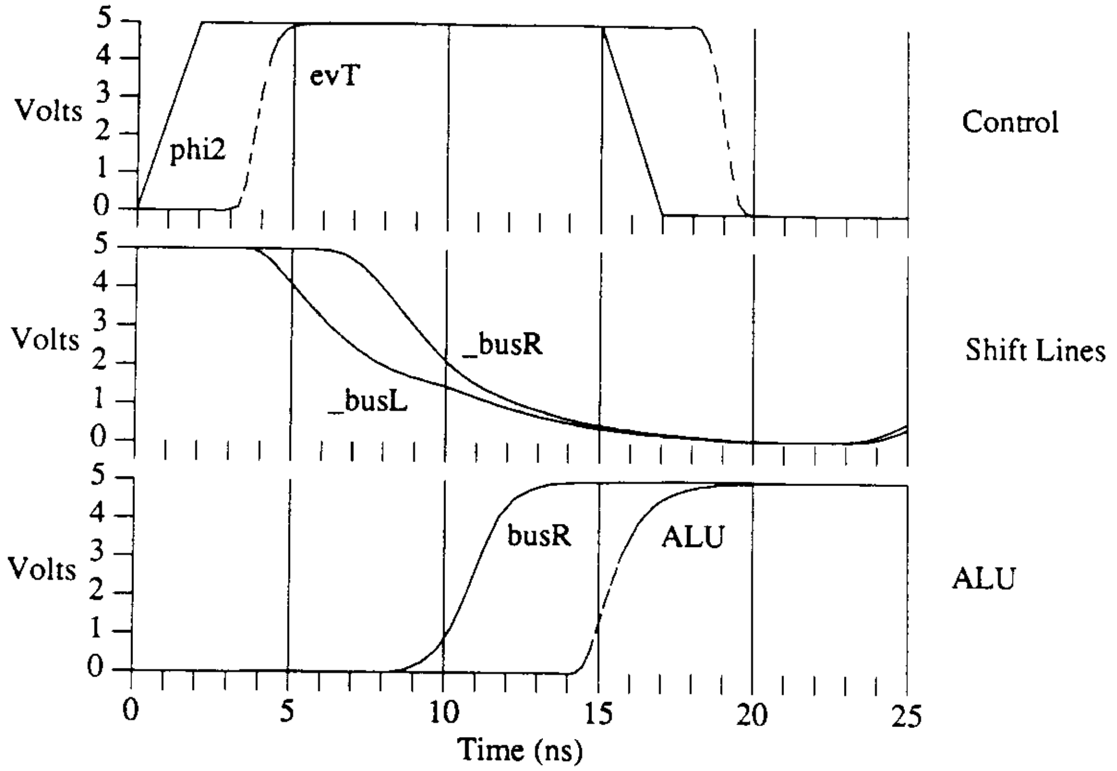

Figure 3.37: Shifter Shifting from BusT to BusR to ALU

Five kinds of shifts are used in the shifter operation. During ϕ_2, two of the shifts are used to transfer data from busT or busS onto busR, the B input of the ALU. If the source is an immediate from busT (evT), then each bit in $\overline{\text{busL}}$ is discharged if the corresponding bit in busT is a 1. The shift by 13 line will be set by the shift decoder while all the other shift amount lines will be cleared, and so the bit pattern discharged from $\overline{\text{busR}}$ through the shift transistor will be shifted right by 13. If the shift is a signed operation and the sign bit is 1, then $\overline{\text{busF}}$ will be discharged so that 1's will be shifted in from the left. If the operand is read from busS, then $\overline{\text{busR}}$ is evaluated from busS (evR) so that no shifting occurs. $\overline{\text{BusR}}$ is then read by an inverter to drive busR which feeds the B input of the ALU.

During ϕ_2, the two ALU operands must pass through the shifter and busD to set up

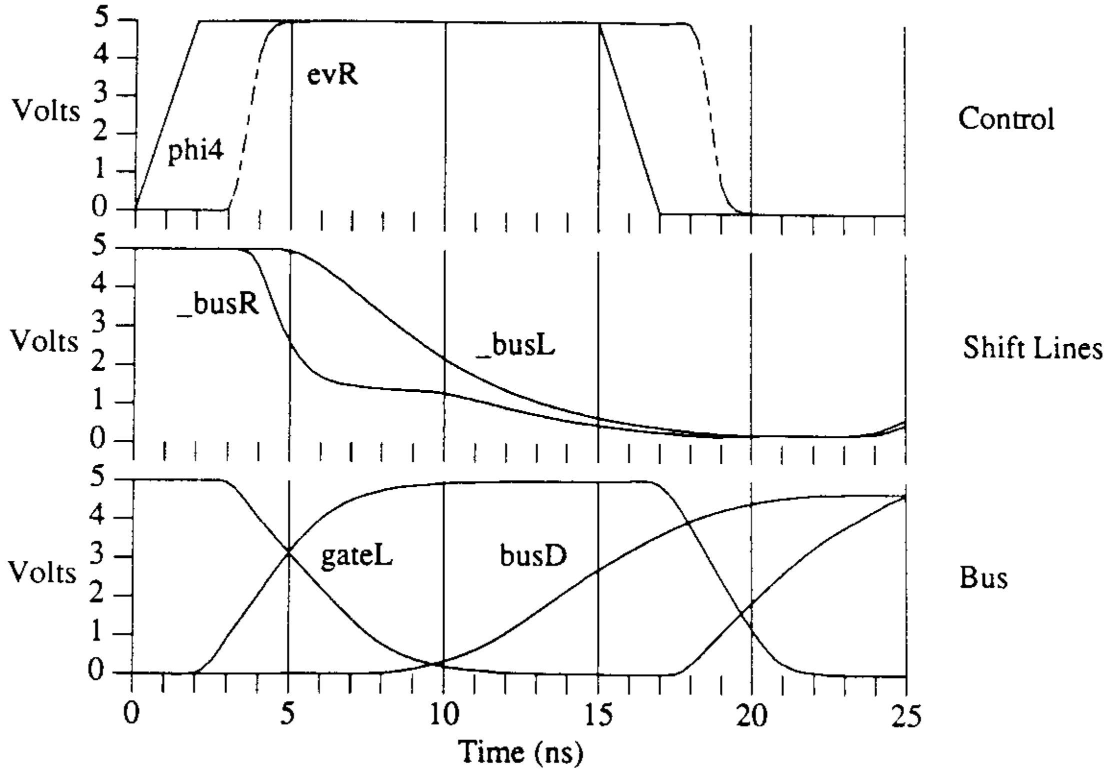

Figure 3.38: Shifter Shifting Right to Left onto BusD

the generate and propagate controls of the ALU's manchester carry chain (figure 3.39). Figure 3.37 shows the timing path through the shifter. The shift from busT to busR takes longer than the shift from busS to busR because $\overline{\text{busR}}$ must be discharged through a shift transistor in the former case. The first panel shows the control lines to discharge $\overline{\text{busL}}$, the second panel shows the busses discharging, and the third panel shows the input to the ALU and the ALU's manchester carry chain control set up by the beginning of ϕ_3 (20ns).

The other three shifts happen during ϕ_4. The left to right (evL, gateR) and right to left (evR, gateL) shifts from busS onto busD are used for the three shift instructions, and the left to right shift from busT to busD (evT, gateR) is used to bring data into the main datapath from the the pads for the load instructions and state repair. Figure 3.38 shows the worst case shift during ϕ_4 of a right to left shift from busS onto busD. $\overline{\text{BusL}}$ runs stepwise diagonally across the shifter and therefore is more capacitive than $\overline{\text{busR}}$. Evaluating from $\overline{\text{busR}}$ forces more charge to flow through the shifting transistor, and hence, cause a longer shifting delay. The first panel shows the control lines to discharge $\overline{\text{busR}}$, the second panel shows the busses discharging, and the third panel shows the shifted value being driven onto busD and ready by the beginning of ϕ_4 (20ns).

3.6.2. Arithmetic and Logic Unit

The *MP*'s 32-bit integer arithmetic and logic unit (ALU) performs additions, subtractions, bitwise ANDs, ORs, and XORs, and calculates the conditions codes based upon its operation and the value of busD. Figure 3.39 shows a bitslice of the ALU.

At the top are the latches, loaded during ϕ_2, that hold the inputs stable while the ALU evaluates on ϕ_3. The MUX at the B input selects between the true and complement value for an add or subtract operation. Next are the three logic gates that perform the ALU's logic operations; these results are fed into a precharged MUX which evaluates on ϕ_3. At this same level is the precharged manchester carry chain with a carry bypass every four bits (figure 3.40). The AND and XOR gates are also used as the generate (g#) and propagate (p#) inputs to the manchester carry chain, which evaluates on ϕ_3. Upon evaluation during ϕ_3, the final MUX selects between an arithmetic operation or a logic operation. The result is gated out to the busses on ϕ_4.

In order to perform a logic operation, the B input MUX selects the complemented value of B.in, the second MUX selects the chosen operation, and the last MUX selects

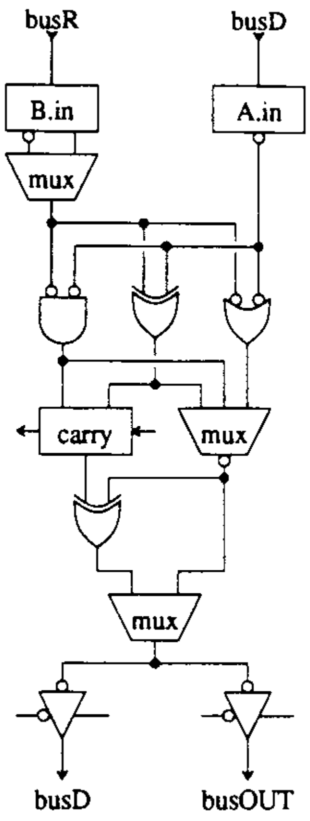

Figure 3.39: Arithmetic and Logic Unit

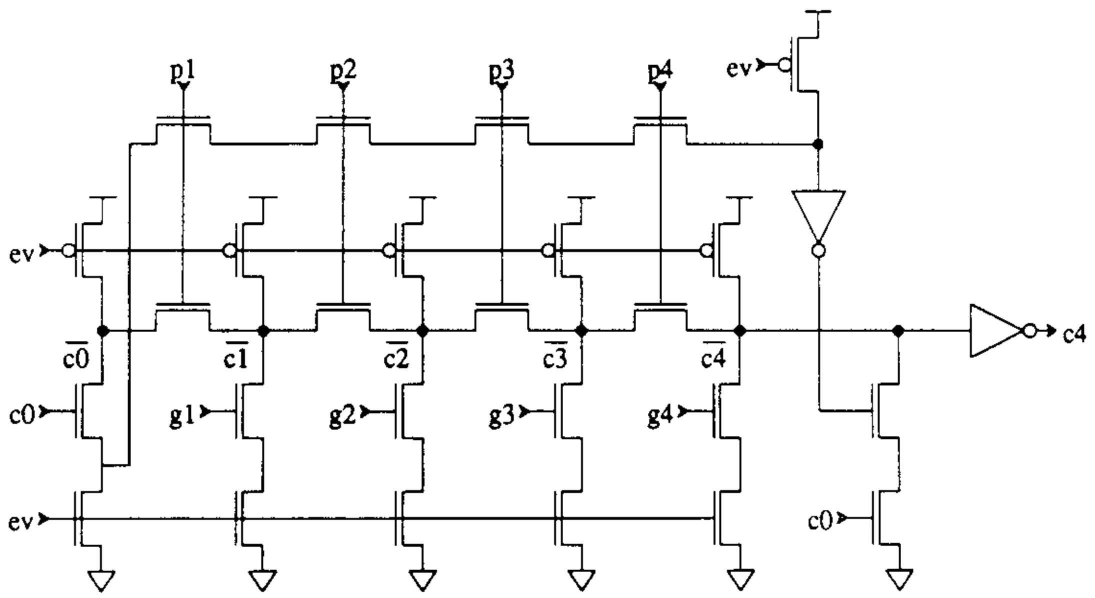

Figure 3.40: Manchester Carry Chain with Four Bit Carry Bypass

the logic operation MUX. For an arithmetic operation, the B input MUX selects the complemented value for an add and the true value for a subtract. The second MUX selects the XOR gate to do the one-bit adding of the two operands, and after the manchester carry chain evaluates, the carry from the previous bit is added with another XOR. The final MUX then selects this sum for gating onto the busses.

The negative and zero condition code bits are generated from busD during ϕ_1. The negative bit is simply the most significant bit of busD, but the zero bit must be generated from busD using a precharged 32 input NOR gate; this takes 8.5ns to evaluate and drive to the PSW.

For an arithmetic operation, the overflow bit is determined by XORing the last two carry bits (c31 and c32) from the manchester carry chain together, and the carry bit is the carry out (c32) from the manchester carry chain. For a logic operation, these bits are defined to be zero. During ϕ_4, these two condition code bits calculated from the manchester carry chain are ANDed together with the control line that selects between logic (0) and arithmetic (1) operation to obtain the correct condition codes.

Setup of the manchester carry chain inputs during ϕ_2 is detailed in sections 3.3 and 3.6.1 for busD and shifter operation. Figure 3.41 shows the critical path during ϕ_3, when the manchester carry chain evaluates and the sum is generated. The first panel shows the evaluate line charging up. The second panel shows the worst case manchester carry chain configuration of the first bit in generate mode and the next 31 bits in propagate mode. A carry is generated at bit 0 and propagates to bit 3, where it skips by the next six groups of four bits to bit 29, and then it propagates up to bit 31. The carry is then

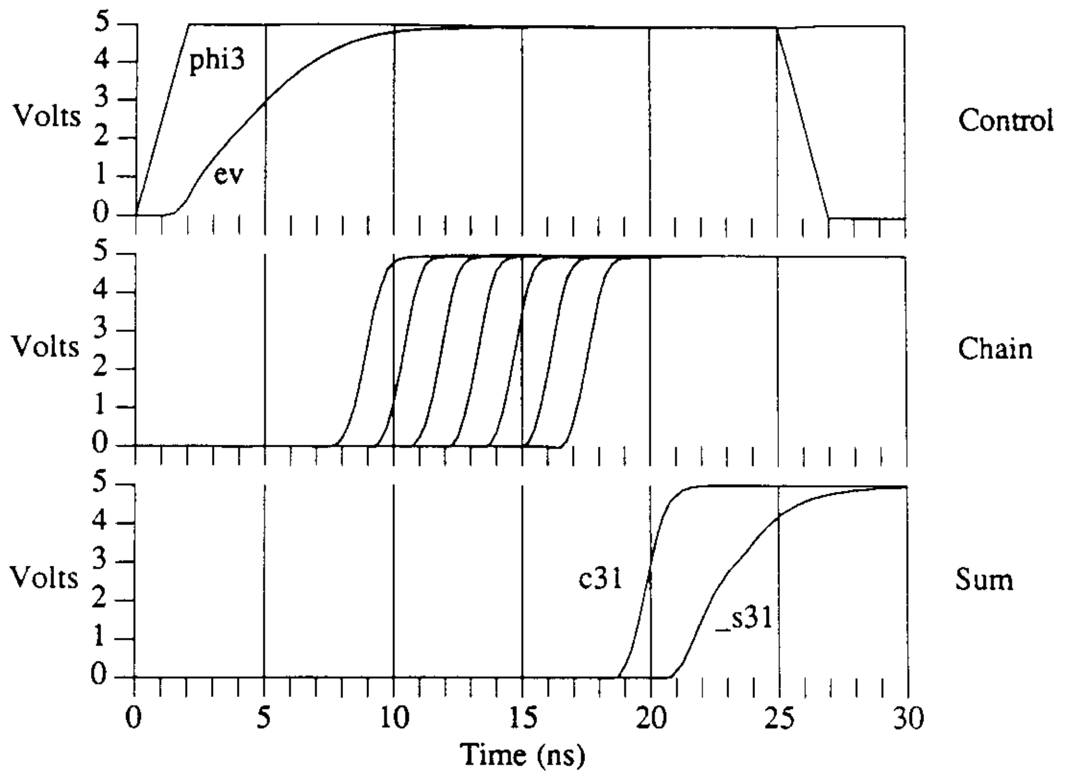

Figure 3.41: Arithmetic and Logic Unit Evaluation

XORed with the partial sum at bit 31 as shown in the third panel, and the result is ready to be gated onto the busses by the tri-state inverters during ϕ_4.

3.6.3. Register File Address Translator

The Register File Address Translator (RFTRAN) takes a register number and the current window pointer, and calculates a unique address within the register file. We use the same register file windowing scheme as the Berkeley RISC[Kate83a], only instead of using 8 windows, we implement 4 windows on the *MP* in order to reduce the size of the register file from 138 registers to 74 registers.

Figure 3.42 shows the RFTRAN. Rs1 and rs2, from the instruction source register

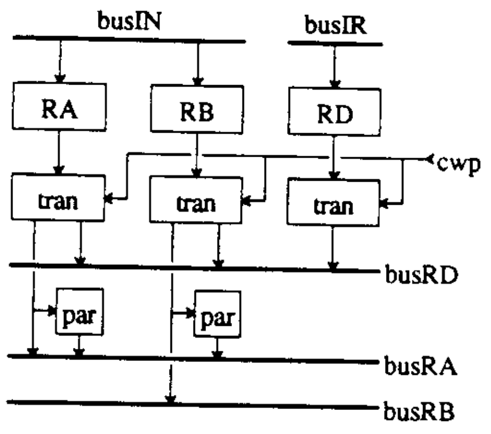

Figure 3.42: Register File Address Translator

fields, are latched from busIN into registers RA and RB during the instruction read on ϕ_3. The register numbers are then translated into register file addresses (tran) and sent to the register file on busRA and busRB along with the parity (par) of the addresses. The register file needs these values by the beginning of ϕ_4, so that it can start to precharge its address decoder in preparation for the register file read on ϕ_1. BusRA selects the register read onto busA and busRB selects the register read onto busB. Figure 3.9 shows the amount of time it takes to read in a value from busIN, translate the address, and send it to the register file.

Rd, the destination register field, is used for the register file write. However, the next instruction is read in before the result from the current operation is written into the register file, and so the destination address must be latched for an extra cycle and restored on rollback. Instead, as described in 2.1.5, the destination register is read from busIR and latched into RD on ϕ_2. The translated address is written into the register file's DWB on ϕ_4, so there's no timing problem.

In order to support state repair, the *MP* must be able to write data into one of the

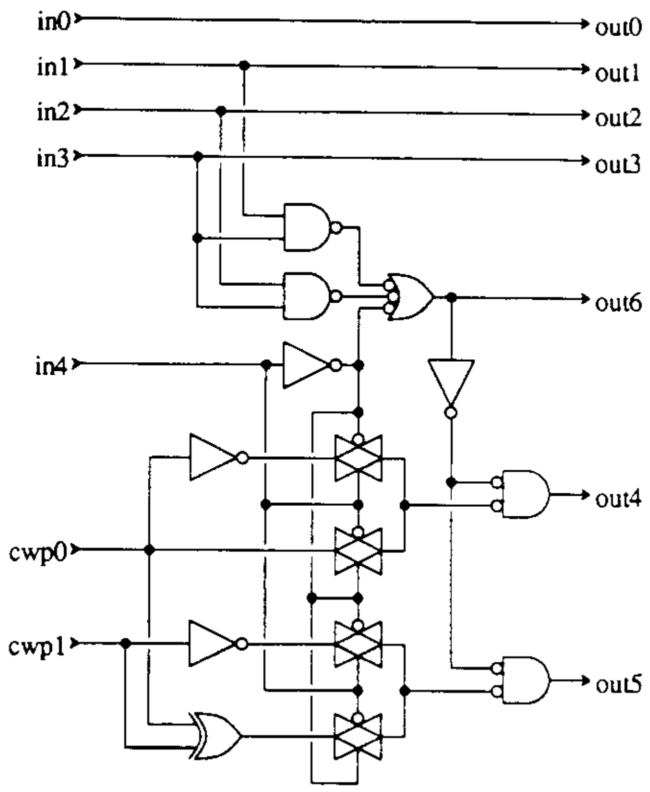

Figure 3.43: Relative to Absolute Address Translator

two source registers indicated in the fetched instruction in addition to the destination register. The controller chooses whether the translated address from RA, RB, or RD will be gated onto the destination address bus, busRD.

Figure 3.43 shows the circuitry used to translate the register address, and figure 3.44 shows the ISP description of the translator circuit. If the requested register is from 0 to 9, then one of the ten global registers is selected. Otherwise, bit 6 is set to indicate a read into the local registers and the correct window and register in the window is calculated.

```
/* pass through bottom four bits */
out<3:0>=in<3:0>;

/* determine if register is global or local */
out<6>=not ((in leq 9) and (in geq 0));
next;

/* if global, then zero extend address */
if (out<6> eql 0)
       out<5:4>=0

/* else find the proper window */
else
       if (in<4> eql 0)
              out<5:4>=cwp<1:0>+1
       else
              out<5:4>=cwp<1:0>;
       next
```

Figure 3.44: Relative to Absolute Address ISP Description

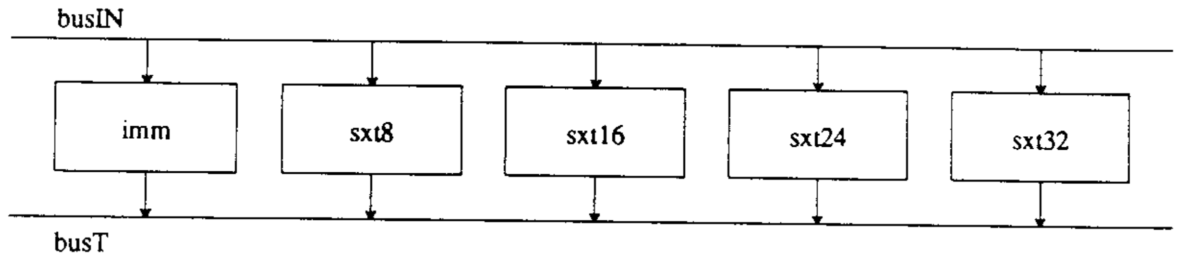

Figure 3.45: Immediate Register and Data Immediate Sign Extender

3.6.4. Immediate Register and Data Immediate Sign Extender

The immediate register (IMM) and data immediate sign extender (DIMM) are used to bring data from busIN down into the main datapath via the shifter. Figure 3.45 shows a block diagram of the IMM and DIMM.

The IMM loads the lower 19 bits of the instruction coming into the *MP* during ϕ_3 of the instruction fetch. During ϕ_1 of the following cycle, the data is gated onto busT left shifted by 13 bits. If the immediate is to be used as an operand for the ALU, then only the lower 13 bits are put onto busT and the upper 6 bits are filled by the sign of the 13 bit immediate (busT<31:26> = busIN<12> (sign), busT<25:13> = busIN<12:0>, busT<12:0> = 0). The shifter will perform an arithmetic right shift by 13 bits on the data so that when it reaches the datapath it will be properly aligned. If the immediate is to be used for the *ldhi* instruction, then all 19 bits are gated onto busT (busT<31:13> = busIN<18:0>, busT<12:0> = 0). The shifter will then perform a right shift of zero bits so that the immediate will appear on busD left shifted by 13 bits as needed by the *ldhi* instruction.

The DIMM consists of four buffers from busIN to busT that drive the lower 8, 16, 24, or 32 bits (sxt8, sxt16, sxt24, sxt32) and either zero extend or sign extend the remaining top 24, 16, 8, or 0 bits. Extending the sign is accomplished by driving the sign bit to the upper bits and selecting between the sign or GND to gate out. Table 3.2 shows which module will be selected based upon the data word size and its byte address for the load operations. During state repair, the data transferred from the other chip is put onto busT via sxt32. Combinational logic selects the proper driver for these two operations. Figure 3.9 shows the amount of time it takes to load data into the *MP* and pass it through the DIMM (sxt8) and onto busT.

data size	out.size	busBAR	31:24	23:16	15:8	7:0	module
word	10	00	d	d	d	d	sxt32
halfword	01	00			d	d	sxt16
		10	d	d			sxt32
byte	00	00				d	sxt8
		01			d		sxt16
		10		d			sxt24
		11	d				sxt32

Table 3.2: Data Immediate Sign Extension Module Selection

3.6.5. Byte Address Register

The byte address register (BAR) is used to calculate the byte address of the next memory operation because the ALU cannot finish before ϕ_3, when the address is needed by the shift decoder. Its functionality is that of a two-bit adder, though because it needs to be rolled back, the implementation is a bit more complicated. Instead of dedicating a rollback memory for two bits, we can restore the proper value from the MAR as it gates onto busOUT during a rollback because it is the same value.

The BAR holds it value on its two-bit bus, busBAR. During ϕ_2 and the underlap between ϕ_2 and ϕ_3, the BAR gates the calculated value of busD<1:0>+busR<1:0> onto busBAR, and during ϕ_4 of the rollback cycle the BAR gates busOUT<1:0> onto busBAR. Figure 3.46 shows a description of the BAR. See figure 3.7 for timing from busD onto busBAR, which takes longer to evaluate than from busR onto busBAR.

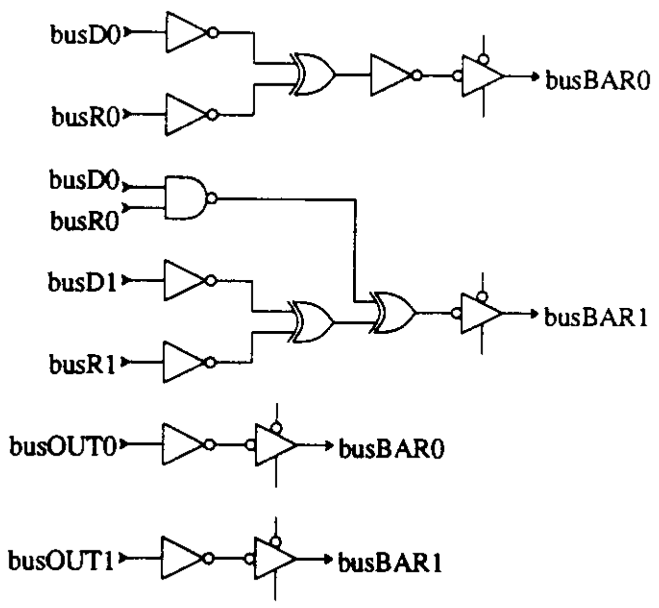

Figure 3.46: Byte Address Register

3.6.6. Shift Amount Decoder

The shift amount decoder (SDEC) takes a shift amount and enables the proper shift line in the shifter. If the shift amount doesn't match any of the amounts that the shifter can shift by (0, 1, 2, 8, 16, 24), then the badshift signal is asserted and the *MP* traps the following cycle on a bad shift amount trap.

Figure 3.47 shows the shift amount decoder and shift line drivers. During ϕ_2, the shifter only shifts by 0 or 13 bits. The SDEC either enables the *shift by 0 bits* shift line, busSDEC0, or the *shift by 13 bits* shift line, busSDEC13. The *load shift amount with zero* line serves double duty by selecting whether to shift by 0 or 13 bits.

For the ϕ_4 shift, a shift amount is loaded into the shift amount register (SHam) during ϕ_2 and ϕ_3 and is processed by the SDEC. The controller determines which of four sources is loaded into the SHam. During ϕ_2, the SHam can be loaded with values from busB (busB<4:0>), the IMM (IMM<4:0>), or with zero. The BAR evaluates during ϕ_2, and so it is loaded into the SHam on ϕ_3 (left shifted by 3 bits so that $0 \rightarrow 0$, $1 \rightarrow 8$, $2 \rightarrow 16$, and $3 \rightarrow 24$). The shift amount decoder circuitry then evaluates and the proper shift line is enabled while all the rest are disabled. If the shift amount is not one that the shifter can handle, then the badshift error is signaled to the controller.

Figure 3.48 shows the SDEC timing. On ϕ_3, the SHam is loaded with the value of the BAR*8. This value propagates through the shift amount decoder and the shift line drivers onto busSDEC. The shift control lines are drawn in poly, and so there is a propagation delay from the first bit to the last bit within the shifter. Shift0 and shift31 in figure 3.48 are the shift control line at the 1st and 32nd bits. The last bit is ready by the

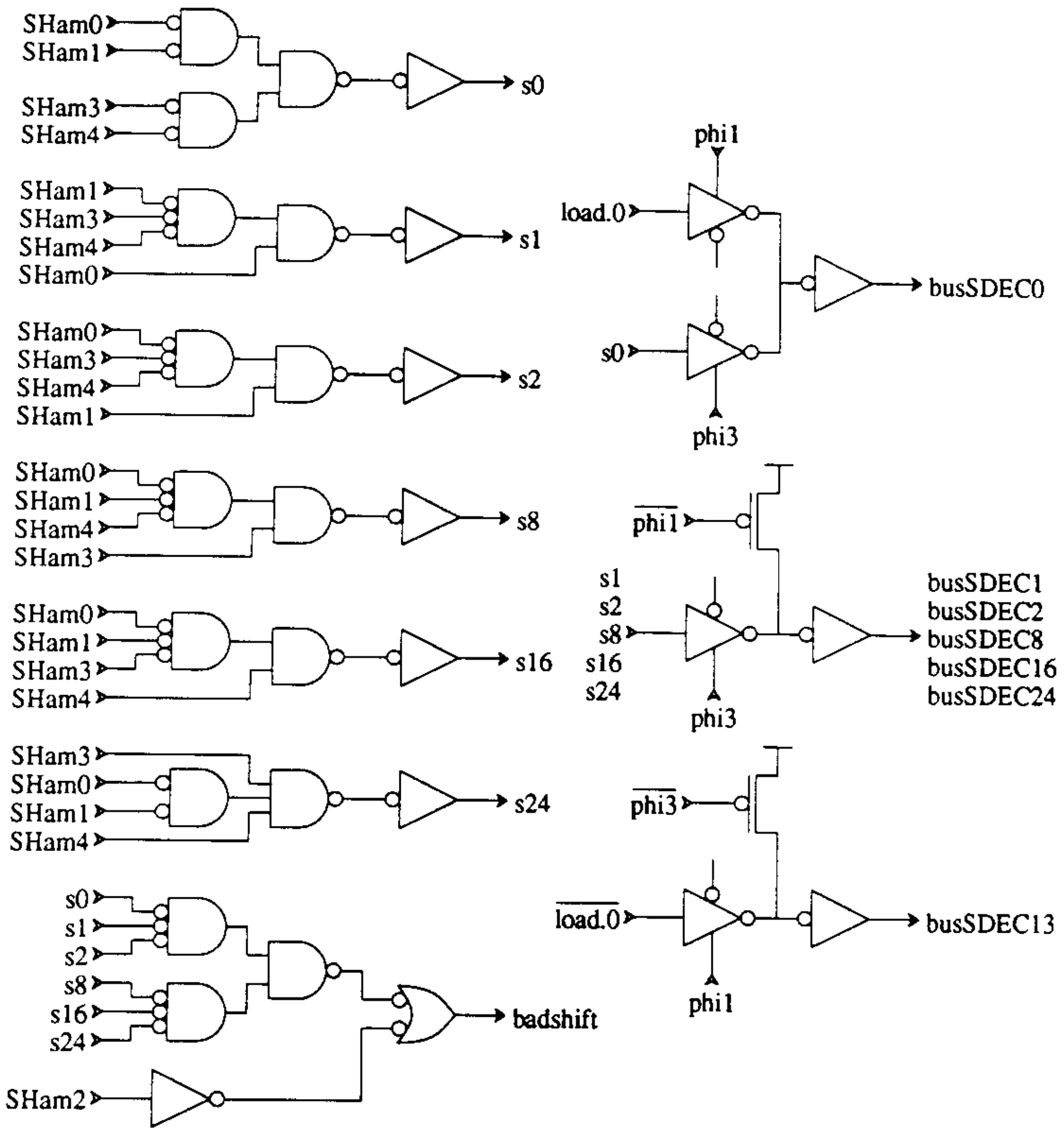

Figure 3.47: Shift Amount Decoder

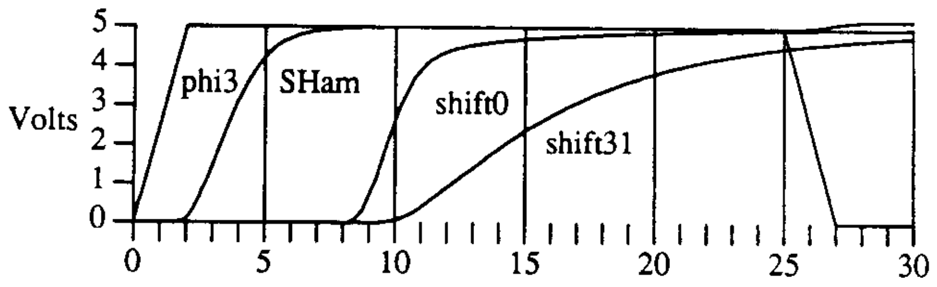

Figure 3.48: Shift Amount Decoder Evaluation

beginning of ϕ_4 (30ns), when the shift begins.

3.7. Critical Paths

We aimed at an operating cycle time of 100ns for the *MP*, and have achieved this level of performance in our circuit simulations. In addition, the timing of the delayed write buffers and error detection circuitry does not figure into the critical path of each clock phase. It takes 10ns to load and update the DWBs, which is well within the duration of each of the four clock phases, and a single register DWB evaluation takes 20ns, which is less than the length of ϕ_1 or ϕ_3.

The critical paths for each clock phase within the datapath are:

ϕ_1 The read from the permanent register file, followed by a read from the register file's DWB (section 3.4.2.2). The value is read onto busB, which then must make its way onto busS and to the shifter.

ϕ_2 A tie between the shift from busT onto busR and to the ALU (section 3.6.1), and any register or bus gating onto busD and to the ALU (section 3.3).

ϕ_3 The ALU manchester carry chain evaluation and operand summation (section 3.6.2). The next closest contender is the write into the permanent register file from its DWB (section 3.4.2.2).

ϕ_4 A shift operation gating onto busD (section 3.6.1), followed by any register or bus gating onto busD (section 3.3). The shifter was the original critical path, and so its drivers were increased to reduce the gating time. The drivers for the other registers and busses only have to place the value onto busD no slower than the shift operation, and so we minimized the size of those drivers in order to reduce the stride of the datapath.

The critical paths for the controller[Lai90a] are:

ϕ_1-ϕ_2 The rollback controller determines the absolute number of cycles to roll back on ϕ_1 and sends to rollback amount to the rollback amount bus for arbitration. The resulting rollback amount is read in, and sent to all the modules before the beginning of ϕ_3.

ϕ_3 On ϕ_3, the next instruction to be executed is read in and sent to the ϕ_4 PLA to be decoded. It takes about 12.5ns for the instruction to propagate to the control lines of the PLA from the pads (section 3.3), and the controller needs the input lines stable by 23ns after the beginning of ϕ_3. The memory must place the instruction on the pads by 10.5ns after the rising edge of ϕ_3 in order to maintain this critical path. During rollback, the IR simulates an instruction fetch; this process brings the instruction to the controller 23ns after the beginning of ϕ_3.

ϕ_4 The signals need by the datapath during ϕ_1 are generated by the ϕ_4 PLA. The PLA evaluates on ϕ_4, and drives the control signals to the datapath in 13ns.

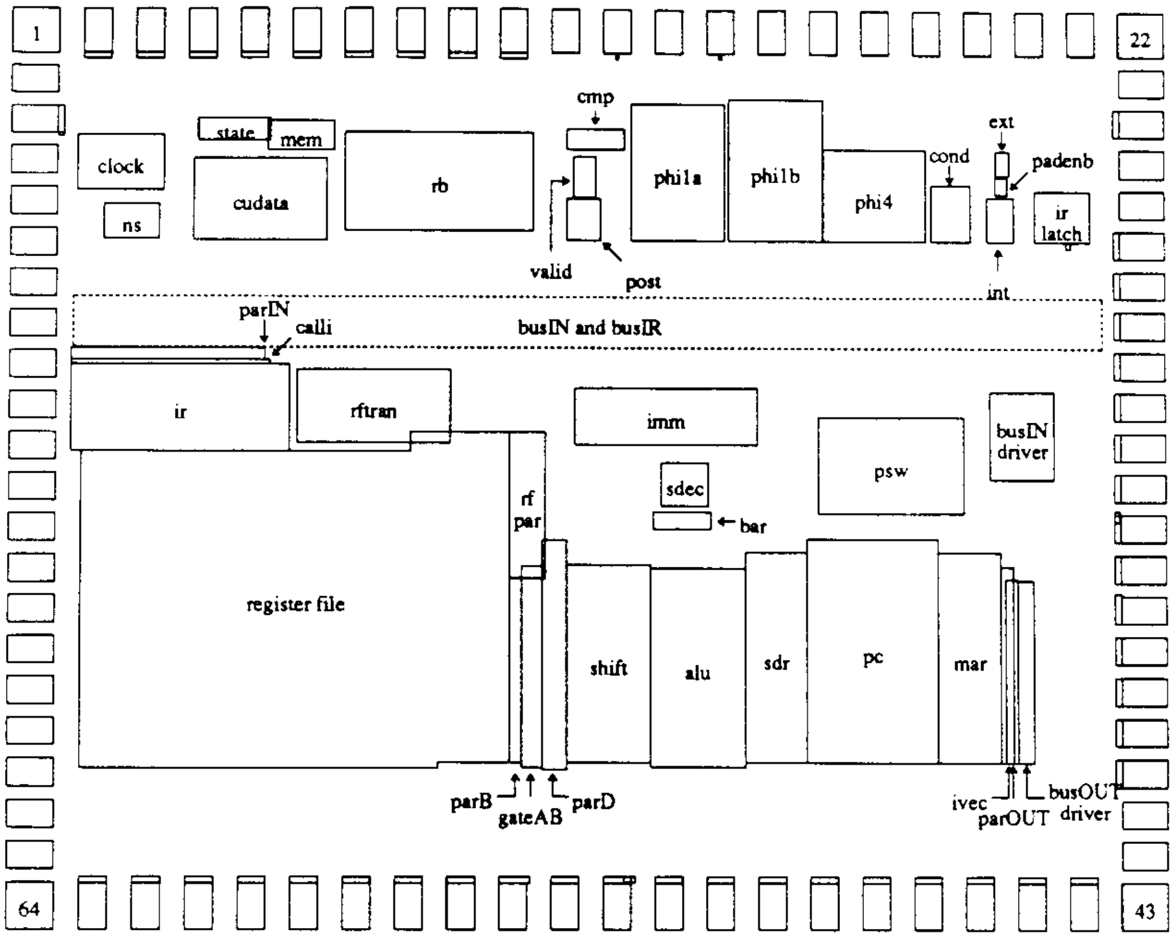

Figure 3.49: Mirror Processor Layout

3.8. Geometry

Figure 3.49 shows the relative sizes and positions of the modules in the mirror processor as derived from the Magic layout editor. The rectangles are the bounding boxes of each module, including some routing for power lines and intermodule connection. The routing to the pads and the control lines are not shown. The upper cluster of blocks form the controller, and the lower half is the datapath. The small blocks adjacent to the pads are the pad comparators (padcmp). Calli and ivec are used to gate the *calli* instruction onto busIN and the interrupt vector onto busOUT to be sent to the pads. GateAD are drivers to gate busA onto busD and to gate busA and busB onto busS. The busIN driver block drives busIN from in.AD, and the busOUT driver block drives out.AD to the pads from busOUT.

The following list is a brief description of the controller blocks:

- padenb: generate the *memory address enable* and *memory data enable signals.*
- ext: synchronize external signals to the internal clock.
- valid: determine what values to write into the *valid bits* of the DWBs.
- cmp: process comparisons in the datapath and from the pads.
- int: handle traps and interrupts.
- state: perform state compression bit vector merging from the datapath.
- ns: determine the next processor state while the instruction is read in on ϕ_3.
- post: adjust the datapath control signals after they are generated by the PLAs for signals that depend upon input that isn't stable until after the PLAs evaluate.
- mem: generate memory control signals needed during ϕ_4 by using combination logic instead of PLAs.
- cond: determine if the condition code bits from the PSW meet the conditions specified by the conditional instruction.
- irlatch: drive busIR and also latch the instruction following a *load* or *store* instruction for temporary storage during the memory transaction cycle.
- phi4: PLA evaluating ϕ_4 for control signals needed by the beginning of ϕ_1.
- cudata: rollback memory to restore busIR and the controller state.
- phi1a: PLA to generate half of the control signals needed after ϕ_1. There were so many signals that the original phi1 PLA was split into two PLAs in order to reduce the overall height of the controller.
- phi1b: The PLA to generate other half the control signals needed after ϕ_1.
- rb: determine if a rollback should occur and arbitrate the rollback amount.

Table 3.3 shows the *MP* pad assignments. The corner pads in figure 3.49 are numbered for indexing into the pad assignment table. The Vdd/GND pins are clustered together to minimize PC board routing. There are two Vdd clusters in the upper left corner (84, 1) and the lower right corner (41, 42, 43), and two GND clusters in the upper right corner (16, 22, 23) and the lower left corner (64, 65, 66). Eight pads remain unused

1	Vdd (pad)	24	pad.ira
2	pad.state1	25	pad.irr
3	pad.state2	26	pad.wait
4	pad.state3	27	pad.sysmode
5	pad.id	28	pad.AD
6	pad.rw	29-40	pad.AD
7	pad.sz1	41-42	Vdd (dp)
8	pad.sz0	43	Vdd (pad)
9	pad.enb.addr	44-63	pad.AD
10	pad.enb.data	64	GND (pad)
11	pad.repairAm	65-66	GND (dp)
12	pad.repairAs	67-74	blank
13	pad.repairBm	75	pad.ms
14	pad.repairBs	76	pad.csel
15	pad.reset	77	pad.phi4
16	GND (pad)	78	pad.phi3
17	pad.rb0	79	pad.phi2
18	pad.rb1	80	pad.phi1
19	pad.rb2	81	pad.sync
20	pad.rb	82	pad.phi
21	pad.shutdown	83	pad.state0
22	GND (pad)	84	Vdd (cu)
23	GND (cu)		

Table 3.3: Mirror Processor Pad Assignment

(67-74), and they are positioned to the left of the register file where no control signal routing needs to be done.

Figure 3.50 and table 3.4 show the position and dimensions of the datapath and controller halves within the chip. The sizes given are of the entire chip, the entire chip without the pads but with routing to the pads and the pad comparator, the *MP* core without routing to the pads, the datapath without routing to the pads but with busIN, and the controller without routing to the pads but with control routing between its blocks.

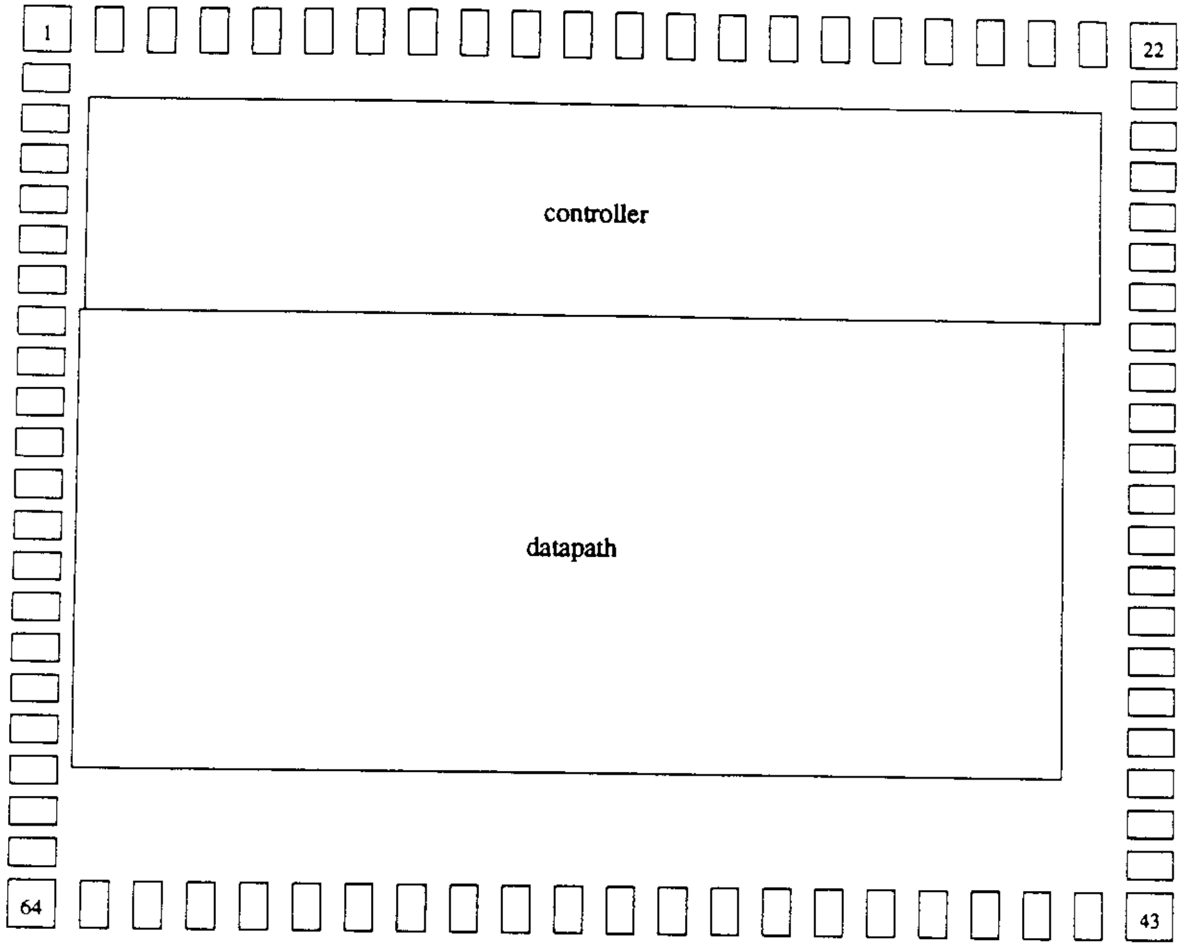

Figure 3.50: Mirror Processor Area Breakdown

Area	Width (λ)	Height (λ)
chip	8395	6715
chip – pads	7725	6045
datapath + controller	7400	4890
datapath	7138	3337
controller	7363	1553

Table 3.4: Mirror Processor Area Breakdown

The transistors are distributed within the chip as shown in table 3.5. Numbers are given for the chip with and without the pads. The core is broken down into the datapath, the controller, the clock generator, and the comparators at the pads.

Area	Number
chip	52644
chip – pads	49609
datapath	41063
controller	7247
clock	389
padcmp	910

Table 3.5: Mirror Processor Transistor Breakdown

Module	Width (λ)	Height (λ)	Area (λ^2)	Transistors
rf	3006	2409	7241454	20441
rfpar	188	772	145136	
parB	87	1340	116580	322
gateAB	122	1467	178974	430
parD	179	1671	299109	532
shift	581	1436	834316	980
alu	612	1447	885564	2671
sdr	412	1527	629124	1922
pc	854	1629	1391166	3689
mar	423	1527	645921	2000
ivec	39	1429	55731	51
parOUT	87	1337	116319	316
busOUT	100	1330	133000	136
ir	1566	561	878526	2434
calli	1428	32	45696	76
parIN	1389	87	120843	340
rftran	1112	342	380304	822
imm	1330	381	506730	1213
busIN	464	612	283968	677
bar	421	130	54730	144
sdec	342	318	108756	254
psw	1007	706	710942	1613
datapath			16135489	41063

Table 3.6: Mirror Processor Datapath Module Geometry and Transistor Count

Table 3.6 gives the bounding box of the active area and the transistor count for each module in the datapath. The active area is the smallest box that fits around a module but excludes routing for power and busses that run vertically to the upper portion of the datapath. Rfpar corresponds to the circuitry shown in figure 3.22. This region extends to the right of the register file and occupies space above the main datapath, so it was counted separately instead of including it in the active area of the register file along with extra empty space.

Table 3.7 shows the active area bounding boxes and the transistor counts of each controller module. The datapath and controller active areas, along with miscellaneous modules, are summarized in table 3.8. Dimensions are given for one pad comparator and one pad, but the area and transistors of all the pad comparators and pads are accounted for.

In order to add the ability to perform micro rollback to a RISC processor, delayed write buffers are added to the registers in the datapath. Figures 3.51, 3.52, 3.53, and 3.54 show the register breakdown and the actual ratio of control area to data area. The register file layout is partitioned into four sections, analogous to the block diagram of the register file shown in figure 3.17.

Each subcell is labeled with a letter legend. The *d*'s indicate data cells, the *g*'s are the drivers onto the external busses, and the *b*'s are drivers to buffer the control lines. The *v*'s are valid bits which are used by the selection circuitry (*s*) to select the proper register. Invalidation circuitry (*i*) resides at the top of the cell along with the clock and control inputs. In the PC layout, *mc* and *xor* denote the manchester carry chain and adder

Module	Width (λ)	Height (λ)	Area (λ^2)	Transistors
padenb	124	74	9176	44
ext	176	86	15136	58
valid	293	151	44243	166
cmp	116	417	48372	180
int	167	316	52772	196
state	132	500	66000	228
ns	189	371	70119	210
post	308	230	70840	218
mem	189	471	89019	248
cond	420	287	120540	248
irlatch	365	362	132130	378
phi4	593	674	399682	610
cudata	560	905	506800	1328
phi1a	994	679	674926	948
phi1b	1028	682	701096	1044
rb	690	1371	945990	1143
controller			3946841	7247

Table 3.7: Mirror Processor Controller Module Geometry and Transistor Count

Module	Width (λ)	Height (λ)	Area (λ^2)	Transistors
datapath			16135489	41063
controller			3946841	7247
clock	620	404	250480	389
padcmp $\times$ 45	200	46	41400	910
pads $\times$ 65	200	335	4355000	3055
chip			24729210	52664
chip – pads			20374210	49609

Table 3.8: Mirror Processor Module Geometry and Transistor Count

used by the NXTPC to increment the address for the next instruction. In the PSW layout, *l* is the latch used to break the feedback loop. *Cwp*, *swp*, *isp*, and *cc* are combinational logic used to calculate the next state of the PSW, and *st* is the state compression logic used for error checking.

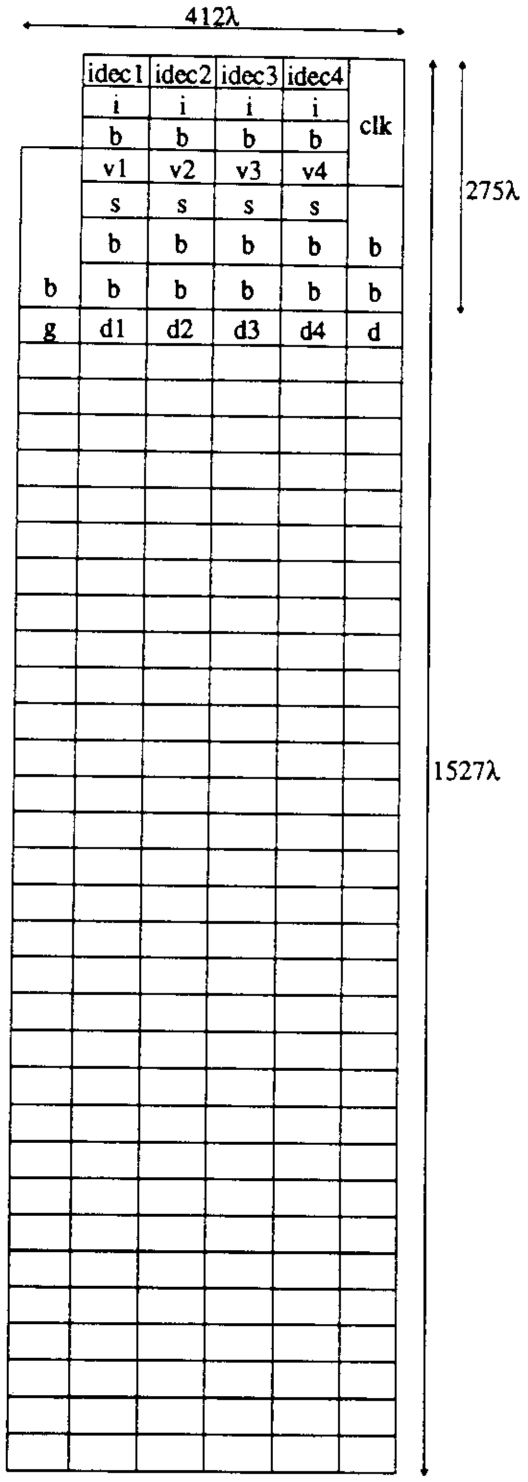

Figure 3.51: Single Register Layout

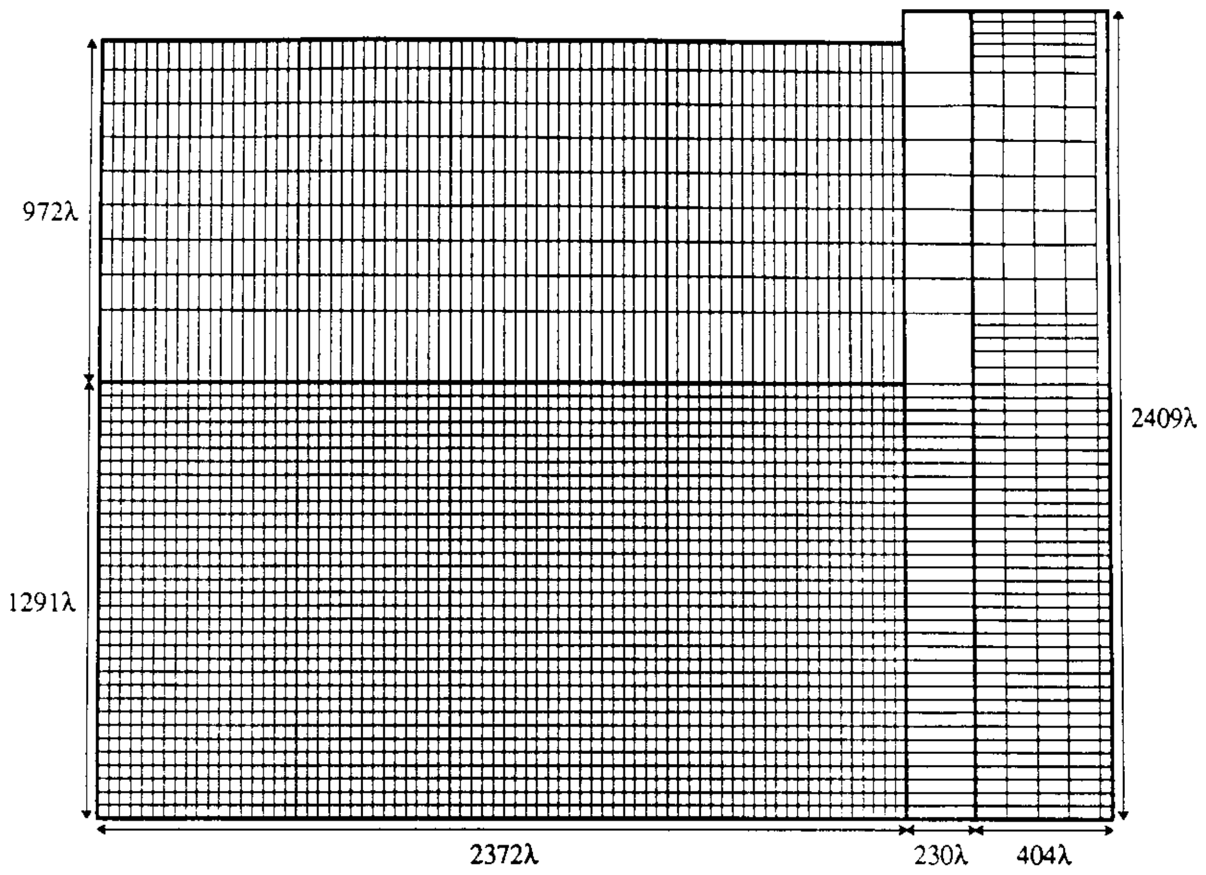

Figure 3.52: Register File Layout

Table 3.9 shows the active area overhead that error detection, micro rollback, and state repair take up within the datapath. The overhead of the various modules was found by calculating the area that would have been saved had the width and height of the module been reduced. If W and H are the width and height, and dx and dy are the reduction in width and height due to a feature being being removed, then the overhead of that feature is $OV=1-\frac{(W-dx)\times(H-dy)}{W\times H}$. Some modules are used for more than one purpose; only after all functions are removed is the total overhead calculated in order to avoid redundant area. The register file contains one bit of parity used for error detection and state repair, and the PSW contains the state compression circuitry used for error

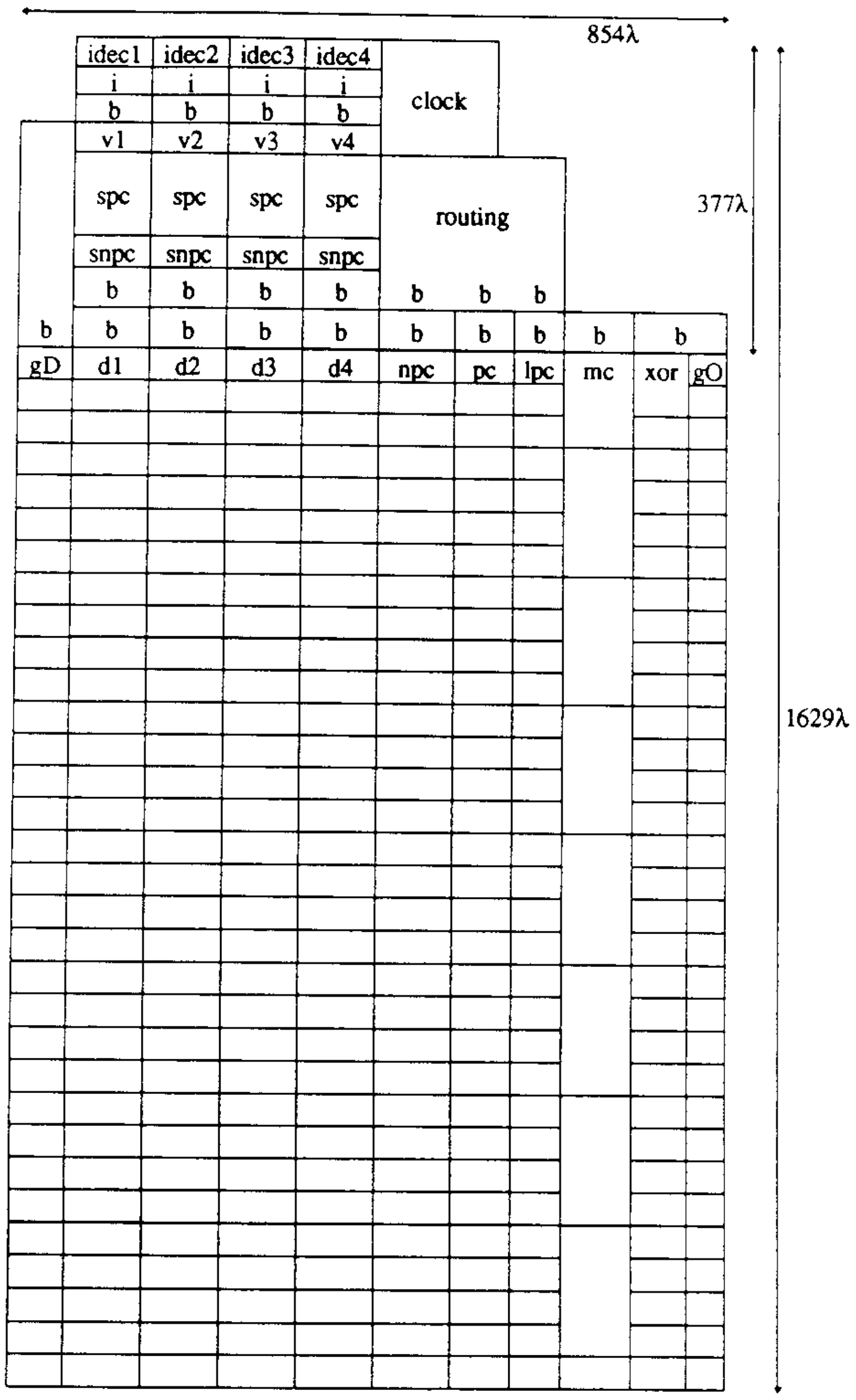

Figure 3.53: Program Counter Layout

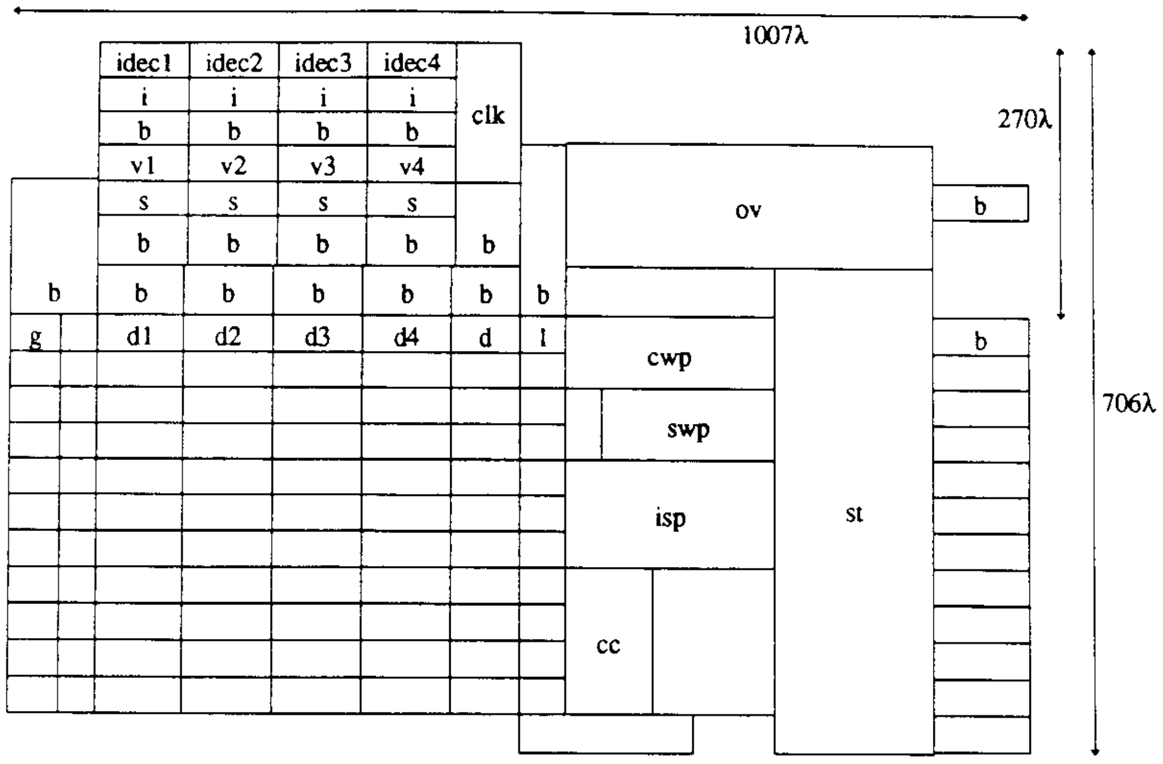

Figure 3.54: Processor Status Word Layout

detection in addition to the DWBs. As shown, the total overhead of the datapath portion of the *MP* is 41.3%.

The active area overhead in the controller is shown in table 3.10. Though it has a higher overhead of 67.1%, the total active area is less than the datapath. As shown in table 3.11, the total overhead of error detection, micro rollback, and state repair in the Mirror Processor is 41.6% counting the pads, and 45.9% not counting the pads. Though state repair accounts for 8.4% of the active chip area, almost all of this area is shared with error detection, and so the cost considerably less. Most of the active area overhead is taken up by the DWBs used for micro rollback; in some cases, an entire register was added in order to implement micro rollback.

Module	Error Detection	Micro Rollback	State Repair	Total
parD	100%		100%	100%
parOUT	100%		100%	100%
parIN	100%		100%	100%
rftran	21.9% (83400)			21.9% (83400)
psw	15.6% (110842)	44.2% (314012)		57.6% (409154)
rf	1.6% (117234)	32.3% (2335644)	1.6% (117234)	33.7% (2442114)
rfpar	100%		100%	100%
sdr		100%		100%
pc		54.5% (758880)		54.5% (758880)
mar		100%		100%
ir		100%		100%
bar		32.1% (17550)		32.1% (17550)
parB			100%	100%
datapath	6.2% (992883)	34.5% (5579657)	5.7% (915221)	41.3% (6662656)

Table 3.9: Mirror Processor Datapath Module Overhead

Table 3.12 shows what the reduction in chip width and height would have been had error detection, micro rollback, state repair, or all three been removed from the chip. The reduction in width came entirely from the datapath and the pad comparators. The reduction in height came from the datapath, the controller, and the pad comparators, though the only reduction from the controller came from the PLAs.

Module	Error Detection	Micro Rollback	State Repair	Total
padenb			13.7% (1258)	13.7% (1258)
valid		100.0%	5.5% (2416)	100.0%
cmp	100.0%	4.8% (2320)		100.0%
int		5.4% (2839)	5.4% (2839)	5.4% (2839)
state	100.0%	6.6% (4356)	6.6% (4356)	100.0%
ns		31.0% (21735)	74.2% (52059)	83.1% (58239)
post	39.0% (27600)	20.8% (14720)	13.0% (9200)	41.6% (29440)
mem			20.0% (17766)	20.0% (17766)
irlatch		100.0%		100.0%
phi4			17.4% (69538)	17.4% (69538)
cudata		100.0%		100.0%
phi1a		17.9% (121142)	27.9% (188116)	41.0% (276626)
phi1b	48.4% (339040)	25.8% (181014)	56.0% (392464)	64.1% (449696)
rb	76.9% (727486)	100.0%	3.8% (35880)	100.0%
controller	30.6% (1208498)	50.1% (1977289)	19.7% (775892)	67.1% (2648937)

Table 3.10: Mirror Processor Controller Module Overhead

Module	Error Detection	Micro Rollback	State Repair	Total
datapath	6.2% (992883)	34.5% (5579657)	5.7% (915221)	41.3% (6662656)
controller	30.6% (1208498)	50.1% (1977289)	19.7% (775892)	67.1% (2648937)
clock				
padcmp	100%			100%
pads	7.7% (335000)	7.7% (335000)	7.7% (335000)	21.5% (938000)
chip	9.1% (2577781)	32.1% (7933346)	8.4% (2067513)	41.6% (10290993)
chip – pads	11.0% (2242781)	37.3% (7598346)	8.5% (1732513)	45.9% (9352993)

Table 3.11: Mirror Processor Module Overhead

	Width (λ)	Height (λ)	Area Overhead
chip	8395	6715	
– error detection	–312	–358	8.8%
– micro rollback	–1488	–1217	32.6%
– state repair	–353	–207	7.2%
– all three	–1887	–1504	39.8%

Table 3.12: Mirror Processor Chip Overhead

3.9. Power Consumption

We need to know the current draw of a module so that we can size its power lines in order to prevent electromigration. According to [West85a], the average current density should be limited to a range from 0.5mA/μm to 1.0mA/μm. We generate a worst case estimate of the total current a module draws by calculating the amount of capacitance that must be charged and discharge each clock cycle.

Table 3.13 shows the average current draw of each of the modules in the datapath.

Module	Current (mA)			
	Transistors	Bus	Extra	Total
rf	32.75*			32.75
parB	0.74			0.74
gateAB	1.20	1.97		3.17
parD	1.18			1.18
shift	3.17	1.97	1.40	6.54
alu	6.52	1.97		8.49
sdr	4.58	1.50	0.32	6.40
pc	8.83	3.47	0.46	12.76
mar	4.75	1.50	0.32	6.57
ivec	0.16	1.50		1.66
parOUT	0.71			0.71
busOUT	1.37	1.45		2.82
ir	6.31	2.63	0.33	9.27
calli	0.24	2.63		2.87
parIN	0.78			0.78
rftran	1.90			1.90
imm	2.75			2.75
busIN	2.15	2.63		4.78
bar	0.39			0.39
sdec	0.66			0.66
psw	3.81	1.97	0.33	6.11
datapath	84.95	12.47	3.16	100.58

Table 3.13: Mirror Processor Datapath Average Current Consumption

Module	Current (mA)			
	Transistors	Bus	Extra	Total
dwb	6.47		0.29	6.76
rw	4.02		0.44	4.46
dec	5.03		5.03	10.06
ram		10.74	0.47	11.21
sel			0.26	0.26
rf	15.52	10.74	6.49	32.75

Table 3.14: Register File Average Current Consumption

For most of the modules, finding the average current can be easily done by counting the number of transistors and basing the total charge being moved upon that transistor capacitance. For each transistor, its source, gate, and drain capacitance is calculated to find the total capacitance of that transistor. The total capacitance of the module then can be found by summing together the capacitances of all the transistors. Vdd is 5V and the cycle time is 100ns, and so we can find the average current draw of each module using $I=\frac{CV}{T}$.

Some modules drive busses (busIN, busD, busOUT); the charge to fill these capacitances needs to be added to the total current draw of those modules. The total capacitance of the bus is due to modules reading from the bus, modules driving the bus, and the metal lines of the bus itself. This extra current is added to the average current draw of applicable modules.

Some modules contain capacitances that change value more than once per cycle. The shift lines in the DWBs are charged and then discharged once per cycle, and the shifter shifts twice per cycle. This extra current draw is taken into account by calculating the capacitance of the area that changes value more than once per cycle.

The register file is a special case to consider because though it contains almost half the transistors in the datapath, not all of them are used each cycle. Table 3.14 shows how the register file average current calculation breakdown. The register file's DWB's current is calculated like the other DWBs. The read/write circuitry (RW) operates once per cycle and so a transistor count suffices (with adjustments for control lines charging and discharging). The decoder (DEC) operates twice a cycle, once for the read, and once

Module	Current (mA)
padenb	0.13
ext	0.18
valid	0.46
cmp	0.42
int	0.49
state	0.64
ns	0.57
post	0.61
mem	0.70
cond	0.60
irlatch	1.12
phi4	1.62
cudata	3.59
phi1a	2.34
phi1b	2.52
rb	2.83
controller	18.82

Table 3.15: Mirror Processor Controller Average Current Consumption

for the write, and so it is accounted for the second time in the extra column. The ram is static, and so the main current draw from this region comes from the read/write circuitry driving the bit lines. For the read cycle, both bit lines are precharged to 5V and then either one or both may be discharged. For the write cycle, the two are driven to complementary levels. Each bit line will experience a maximum of two voltage transitions per cycle. Also included in the ram current is the extra current for charging and discharging three sets of select lines and the current from 33 bits flipping on a write. The ends of the poly select lines are clamped to 0V at the end of ϕ_3; the SEL entry accounts for this current usage.

The current used by the controller modules was calculated by just taking the transistor count of each module. Table 3.15 shows these values.

The datapath and the controller use separate rails to supply Vdd and GND to the modules. The datapath draws 100.58mA; this is handled by two 200μm wide power rails, each supplied by two pads (pins 41, 42, 65, and 66). The upper chip power supply is handled by two 100μm wide power rails sourcing from pins 23 and 84.

Two aspects of clock power consumption must be considered: the minimum width of the clock lines, and the average current the clock generator uses. In order to avoid electromigration, not only must the width be wide enough to handle the average current, but any current peaks should be no larger than ten times the average current. We apply this rule of thumb in reverse to find a minimum wire width. Spice simulations show that the maximum current peak is 33mA on the ϕ_3 clock line. The two peaks occur on the rising and falling clock edges. So, the minimum line width needed is 3.3μm; the lines leave the clock generator as 8μm wide lines and split up into two sets of 4μm lines going into the datapath and the controller.

Current derived by a transistor count of the clock generator gives 1.80mA. The circuitry operates four times a cycle and drives each of the four clock lines (6.87pF, 6.60pF, 8.17pF, 7.46pF) twice, once for the rising clock edge and once for the falling clock edge. The total current consumed by the clock generator is 10.11mA.

The pad power rails are sized to handle two pad voltage transitions per cycle with two Vdd and two GND pads at the four chip corners. Spice simulation of an I/O pad driving a 20pF load shows a total draw of 1.26mA for a 0V to 5V change and a 5V to 0V change. If the pad frame were fully filled, then each Vdd/GND pair would need to

Module	Current (mA)
datapath	100.58
controller	18.82
clock	10.11
padcmp	3.77
pads	49.77
chip	183.05
chip – pads	133.28

Table 3.16: Mirror Processor Average Current Consumption

handle 40 I/O pads for a total average current of 50.40mA. The pad power rails are drawn 100μm wide in order to handle this worse case. The I/O pad is also used as a wired-OR pulldown; spice simulations show this draws 6.81mA through the GND rail. The *MP* uses five pads in such a manner, and we handle this 34.05mA current draw by placing the five pads between one of the corner GND pads (pin 22) and an extra GND pad rail pad (pin 16). During normal operation without any interrupts, nine pads will make one voltage transition and 35 pads will make two voltage transitions. This gives a worst case average current draw from the pads of 49.77mA.

The pad comparators are split into two groups: one set draws off the datapath power lines, and one set draws off the controller power lines. The datapath portion consists of the 33 bit pad.AD comparator with 33 XORs and two sets of 16-1 OR trees. Current estimated by transistor count gives 1.65mA, and double this is used because the pads may change value twice a cycle. The controller portion consists of XORs on the control output pads; this gives a transistor count current of 0.47mA. The total pad comparator current draw is 3.77mA.

Table 3.16 shows the average current used by the *MP* with and without pads. With

Vdd=5V, the Mirror Processor average power consumption is 915.25mW.

4. Verification

This chapter deals with the methodologies used to ensure that the Mirror Processor's design and implementation are correct. The goal was to determine that the architecture was functionally correct before layout began and to verify that the final layout corresponds to the architectural description. Before layout, we used Endot[Zyca88a], a hardware register-transfer level (RTL) simulation system, to model a functional description of the system. A commercial version of Spice, Hspice[Meta87a], was used for preliminary timing estimates. Layout was performed with Magic[Scot85a], a physical layout editor, and afterwards Bdsim[Segaa], a switch level simulator, was used to verify functionality. Actual circuit timing was determined with Hspice, and Crystal[Scot85a], a timing analyzer, was used to verify the critical paths in the resulting layout.

4.1. Register-Transfer Level Simulations Using Endot

Before we started any layout of the *MP*, we wanted to make sure the architecture was correct and that the modules (*e.g.* register file, ALU) would properly interact together. We used Endot to simulate the processor at the register-transfer level. All blocks within the processor were written up in ISP'[Zyca88b], a hardware description language based upon ISP[Bell70a]. We determined if various circuit configurations were feasible for our timing goal of 100ns by running preliminary Spice circuit simulations of key building blocks.

The system is modeled as a console, a processor pair, and a memory. With this setup, we can assemble small programs, load those into the memory, and actually run them. The modules are listed in section 7.3.

After each module was written, it was tested by feeding it various test vectors to make sure it conformed to the architectural specifications. The entire system as a whole was tested by running a series of small test programs and observing the interactions of the modules on a phase by phase basis. Details of the test procedures are covered in [Lai90a].

4.2. Circuit Simulations Using Spice

It's not practical to run Spice on an entire chip, so we only simulate modules and paths between modules. A clock phase is long enough for most control and data signals to stablize, so that at the beginning of a phase, the only signal that should change is the clock. However, should a transaction should take longer than one phase (*e.g.* PC writing onto busOUT and to the pads), then we simulate the whole operation and include as many clock phases as needed. We simulate the circuits using a 2ns rising and falling clock edge, and we define the beginning and end of a phase to be the point in time where the clock starts to rise or fall.

Should a circuit's input values change, we supply the actual driving circuit rather than using a spice voltage source in order to obtain a more accurate time. An example of this is the ALU's manchester carry chain. The chain consists of eight identical subcircuits, each of which produce 4 carry bits and a carry bypass. The first subcircuit in the chain is set up to generate a carry, and the other seven are set up to bypass the carry

```
* NMOS Theoretical Slow Model
.MODEL CMOSN NMOS
+LEVEL=2 VTO=0.9 TOX=430E-10 NSUB=1.0E+16 XJ=.15U
+LD=.20U UO=620 UCRIT=.62E5 UEXP=.125 VMAX=5.1E4
+NEFF=4.0 DELTA=1.4 RSH=38 CGSO=2.1E-10 CGDO=2.1E-10
+CJ=215U CJSW=540P MJ=.76 MJSW=.30 PB=.8

* NMOS Theoretical Fast Model
.MODEL CMOSN NMOS
+LEVEL=2 VTO=0.65 TOX=370E-10 NSUB=6.0E15 XJ=.15U
+LD=.20U UO=680 UCRIT=.62E5 UEXP=.125 VMAX=5.1E4
+NEFF=4.0 DELTA=1.4 RSH=34 CGSO=1.8E-10 CGDO=1.8E-10
+CJ=175U CJSW=460P MJ=.76 MJSW=.30 PB=.8

* PMOS Theoretical Slow Model
.MODEL CMOSP PMOS
+LEVEL=2 VTO=-0.9 TOX=430E-10 NSUB=6.6E+15 XJ=.05U
+LD=.20U UO=240 UCRIT=.86E5 UEXP=.29 VMAX=3.0E4
+NEFF=2.65 DELTA=1.0 RSH=110 CGSO=2.05E-10 CGDO=2.05E-10
+CJ=275U CJSW=400P MJ=.535 MJSW=.34 PB=.8

* PMOS Theoretical Fast Model
.MODEL CMOSP PMOS
+LEVEL=2 VTO=-0.6 TOX=370E-10 NSUB=5.4E+15 XJ=.05U
+LD=.20U UO=270 UCRIT=.86E5 UEXP=.29 VMAX=3.0E4
+NEFF=2.65 DELTA=1.0 RSH=92 CGSO=1.75E-10 CGDO=1.75E-10
+CJ=225U CJSW=300P MJ=.535 MJSW=.34 PB=.8
```

Table 4.1: Spice Transistor Models

directly from the carry-in to the carry-out. We could just simulate one of these seven subcircuits and multiply the resulting timing by seven, but then we would introduce errors by choosing a start and end time. Instead, we simulate the whole chain to find the critical path.

We hand extract the critical paths from our layout rather than let the CAD tools do it for us because they do not handle the extraction properly. In particular, Magic will extract a poly wire controlling many gates as a point node when in fact it should be done

Area Capacitance	ndiff	pdiff	poly	metal1	metal2	units
subs	0.250	0.300	0.057	0.026	0.016	fF/μm^2
diff	-	-	0.930	0.047	0.022	fF/μm^2
poly	-	-	-	0.047	0.022	fF/μm^2
metal1	-	-	-	-	0.044	fF/μm^2

Edge Capacitance	ndiff	pdiff	poly	metal1	metal2	units
subs	0.600	0.400	0.080	0.075	0.068	fF/μm
diff	-	-	0.085	0.086	0.073	fF/μm
poly	-	-	-	0.086	0.073	fF/μm
metal1	-	-	-	-	0.085	fF/μm

Resistance	ndiff	pdiff	poly	metal1	metal2	units
sheet	40	150	30	0.050	0.040	Ω/□
2×2 via	50	100	7	-	0.1	Ω

Table 4.2: Spice Parasitics

as a chain of resistors, capacitors, and gates. In addition, the tools extract the whole circuit when we are only interested in simulating a particular path.

Capacitances of local interconnect are small enough to be ignored; we only take into account long signal lines and busses. For metal lines, the propagation delay is negligible, and so only the capacitance is included. However, poly lines have a high RC constant. In the layout, we use long poly control lines that run across the bit slices in order to reduce the stride. This is modeled with a chain of transistor gates and resistors using Hspice's wire resistor model, which handles capacitance and RC delays.

We used the SPICE level 2 corner parameters provided by VTI for their 2μm CMOS N-well process (table 4.1). These transistor models represent the slowest and fastest transistor switching speeds for the NMOS and PMOS transistors, and hence allow for the four possible ranges of operation: slow NMOS and slow PMOS, fast NMOS and fast PMOS, slow NMOS and fast PMOS, and fast NMOS and slow PMOS. The parasitic parameters we use are derived from VTI's 2μm N-well CMOS/BULK wafer acceptance specifications and are summarized in table 4.2.

Transistor parasitics are taken care of by using transistor subcircuits from a library; each subcircuit consists of a transistor with full parasitics on the source and drain. Hspice allows four parasitic parameters to be attached to a transistor instance. These are: source and drain diffusion perimeter capacitance (ps, pd), source and drain area capacitance (as, ad), equivalent squares from source or drain via contact to transistor channel (nrs, nrd), and source and drain via contact resistance (rsc, rdc). Figure 4.1 illustrates the first three regions for three transistors with channel widths of 4μm, 6μm, and 8μm. The fourth parasitic is obtained by dividing a 2×2 via resistance by the number of vias in the source or drain. Figure 4.2 shows four sample transistors defined as subcircuits and an inverter defined in terms of these transistors. The perimeter is given in microns, the area in square microns, and the resistance in ohms. Even if a resulting circuit does not exactly reflect the layout (*e.g.* shared drain rather than two drains), the actual worst case capacitance is still be better than the modeled case.

4.3. Switch Level Simulations Using Bdsim

To verify that our layout is logically correct, we used the extractor from Magic and an accompanying filter (Ext2sim) to obtain a netlist suitable for simulation. We generated Bdsim test scripts that separately exercised all the modules in the datapath and controller and then the whole chip together. Our strategy is to test the chip hierarchically

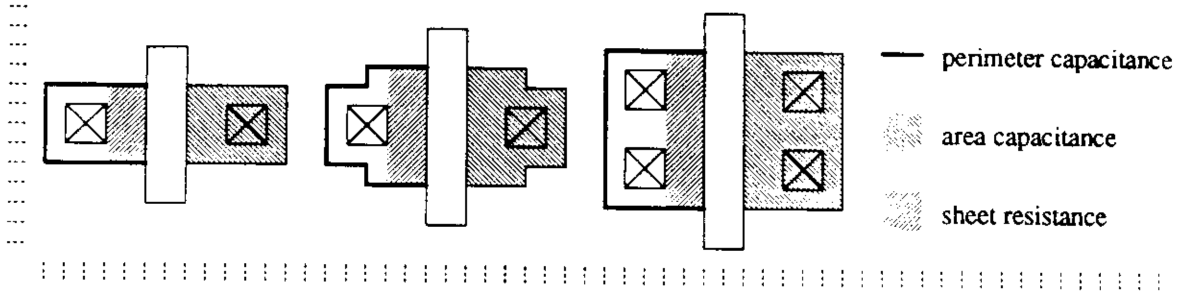

Figure 4.1: Transistor Parasitics from Layout Geometry

```
vpbulk vpbulk 0 dc 5v
vnbulk vnbulk 0 dc 0v

* n4: drain gate source
m1 2 3 4 vnbulk cmosn l=2u w=4u
+ad=20p as=20p pd=14u ps=14u
+nrd=.5 nrs=.5 rdc=50 rsc=50

* p6: drain gate source
m1 2 3 4 vpbulk cmosp l=2u w=6u
+ad=26p as=26p pd=16u ps=16u
+nrd=.33 nrs=.33 rdc=100 rsc=100

* n6: drain gate source
m1 2 3 4 vnbulk cmosn l=2u w=6u
+ad=26p as=26p pd=16u ps=16u
+nrd=.33 nrs=.33 rdc=50 rsc=50

* p8: drain gate source
m1 2 3 4 vpbulk cmosp l=2u w=8u
+ad=40p as=40p pd=18u ps=18u
+nrd=.25 nrs=.25 rdc=50 rsc=50

***

vdd 1 0 dc 5v

*
* inverter example
*
x1 out in 1 p8
x2 out in 0 n4
```

Figure 4.2: Transistor Subcircuits with Parasitics

from the bottom up. At the bottom level, we exercise a module through its external interface and verify that its internal values and external outputs are correct. At the top level, we know the modules work correctly internally, and so we only have to monitor the interfacing control and data signals and supply the proper values on the pads.

Each module consists of latches and combinational circuits. For registers such as the register file and the PC, which consist mainly of latches, we break functionality down into four parts: writting out onto the external bus, writting into the register, reading out the correct register from the DWB, and invaliding valid bits for rollback. The circuitry consist of multiple instances of a few subcircuits, and so the modules are simple to test. These portions were separately tested with arbitrary bit vectors (0xAB5C155A, 0xCC00FFEE, 0xDEADC0DE, 0xGE0DE51C, 0x6005EE66), and then the whole module was exercised for several cycles to verify that it operated correctly.

Large combinational circuits like the ALU and the shifter are a bit harder to verify because of the potentially large test space. We can fully test small combinational circuits like the shift decoder by using all possible input vectors, but for larger ones like the 32-bit ALU, we must choose some critical test vectors and cover the rest of the input space with randomly generated vectors.

Once we have verified the internal workings of each module, we must verify that they will properly interact together. In this case, only the signals going in to and out of the module need to be checked. To manually create test vectors for the whole processor would be a nontrivial task. Instead, we take advantage of the fact that we already have a working RTL simulation of the chip. In order to generate test vectors for use with

Bdsim, we run the Endot code with some test programs, set the pads of the Bdsim simulation to the same values found on the pads in the Endot model, and verify that the signals transmitted between the modules at the switch-level match with the corresponding signals at the register-transfer level.

We wrote four test programs (section 7.2) to exercise the modules and interconnections between the modules in the processor. Linear.ras is a sequence of all the instructions in the *MP* instruction set except for the jump and test instructions. Jmp.ras is a small application to extract the condition codes from the PSW using the conditional jump instructions. Int.ras is used to test all possible interrupts and traps. Test.ras exercises the test instructions, and in the process, tests the error detection, micro rollback, and state repair circuitry. By executing every instruction and forcing every exception that the *MP* would encounter in actual operation, we can verify that all the modules will work together properly in all situations.

Simulation time was broken up into eight phases: the four clock phases and the four underlap times. These are named phi11, phi12, phi22, phi23, phi33, phi34, phi44, and phi41. Phi(i)(i) corresponds to clock phase ϕ_i, and phi(i)((i mod 4)+1) corresponds to the underlap time following ϕ_i. In order to generate the Bdsim test scripts, we first compiled a list of all the control and data signals within the datapath and controller and determined at what times they are relevant. For instance, the PC determines if it should gate onto busOUT during ϕ_4 by taking the control signal gate.INC_busOUT and ANDing it with the ϕ_4 phase clock. If the PC is supposed to gate onto busOUT, then gate.INC_busOUT will be 1, otherwise it will be 0. Since the PC has already been tested in isolation, we can be sure that no erroneous gating will result if gate.INC_busOUT is a 1 on any clock phase other than ϕ_4. This signal will be used during ϕ_4, and so the times that the signal needs to be verified are ϕ_4 and the underlap times before and after (phi34, phi44, phi41). Data signals are transferred between modules, and so the busses need be verified for the times that they are written into each module connected to the databus.

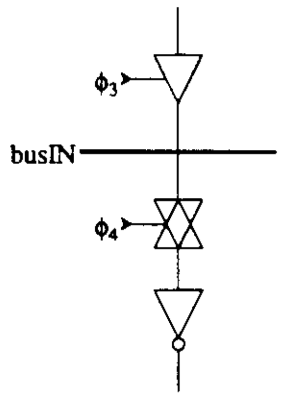

Figure 4.3: Data Transfer Through Pass Transistor Logic (Charge Sharing)

The signals are grouped into three categories: signals coming into the chip that need to be set, internal signals that need to be verified, and certain internal signals propagated through pass transistor logic. This last category is needed because Bdsim does not handle charge sharing, so that if a transistor connects two capacitors with different values together, charge from the larger capacitance will not flow to the smaller capacitance. Only a driving transistor or a Bdsim *set* command will cause a node to change value. Charge sharing is used to transfer the values on busIN, busOUT, and busIR into the IR, the MAR, and the controller rollback memory. Figure 4.3 shows the case of busIN and the IR. Data is placed onto the bus during ϕ_3 by a tri-state driver, and when the IR's transmission gates conduct in order to store the data into the register, no transistors are driving the bus. This is not a problem with the other modules because they read off the bus directly with an inverter and this inverter will drive a value to the transmission gate in that module. Our fix is to first verify that the value on the bus is correct, then explicitly set the value so that it will be transferred into the register, and then release the bus.

A translation file with the names of the signals in the Endot and Bdsim models and the times they are active (section 7.2) is used to create Bdsim scripts from the Endot simulations of the four test programs. This signal list was manually generated by scanning the ISP code for the times when the control or data signal is used. It does not matter when a signal is generated, as long as the signal line contains the proper value at the time it is needed. The translation file is passed through a filter to create Endot commands, which, when executed during the simulation of the test programs, will write Bdsim commands to a file using values from the Endot simulation. In order to simplify the RTL model, the ISP specification does not exactly match the physical layout (*e.g.* events may happen on the trailing edge of the clock). Assuming that the phases when a signal is used are correctly identified, a signal will only be set or verified the simulation phases before, during, and after it is actually needed in order to avoid verification problems between the Endot simulation and the Bdsim simulation. The layout is then simulated with Bdsim using the controlling test scripts. When the layout passes these tests, then we know that the layout at the switch-level conforms to the register-transfer level model, which has already been verified in greater detail.

In order to create the Bdsim scripts, a test program is first assembled and the execution order of the test program is determined. The Endot control script consists one Endot macro call (*cycle*) to generate Bdsim commands for every cycle that the program executes. As a debugging aid, each cycle is annotated with the disassembled listing of the executing instruction. The *cycle* macro itself consists of eight Endot macro calls to deal with each of the eight clock phases and underlap times. For each phase, a command is sent to Endot to set its clock and evaluate until all signals are stable. Then, the corresponding Bdsim command is written to the Bdsim script file. Using the results from the translation file, Bdsim *set* commands are written to the script file using values from the Endot simulation for signals coming into the chip and *verify* and *set* commands are written for the signals propagated by charge sharing. The Bdsim *evaluate* command is then written into the Bdsim script. If the Bdsim simulation were to execute its script up to this point, then the clock and inputs will have been set and the system will have evaluated and become stable, thus matching the Endot simulation state. Now, Bdsim *verify* commands are written to the script using the values from the Endot simulation. This process is repeated for the other seven macro calls, and the *cycle* macro call is repeated for as many cycles as many cycles as the program executes.

When the Bdsim simulator is started, all the registers are in an unknown state. Single registers are easy to initialize by using short Bdsim initialization scripts, but the register file is harder because all the memory can only be accessed through one port. We initialize the register file at the beginning of each test program by writing 0's into all registers within the register windows that will be used, so that the values put on busA and busB, even though they may not be needed, will match that from the Endot simulation.

During this initialization sequence, we used the *scycle* Endot macro instead of *cycle*. The only difference is that *scycle* does not write *verify* commands to the Bdsim script so that there can not be any errors during initialization.

4.4. Critical Path Analysis Using Crystal

Though we may have determined the critical paths of each clock phase by analyzing what modules operate that phase, a longer critical path may be missed due to human error. Crystal is a timing analyzer that will determine the critical path in a circuit without considering any specific input values. We use Crystal to verify that there are no hidden critical paths in the layout that are longer than the ones that we have identified earlier.

Crystal is guaranteed to find the worst-case timing behavior of the circuit because it doesn't depend on user input to find the critical paths. Instead, Crystal tries all possibilities at each point unless told otherwise, and picks the worst branch. The problem with this approach is that Crystal may examine paths that could not possibly occur in actual operation. We prune away these false critical paths by selectively setting node values to constant values so that impossible branches won't be taken.

The following are the paths found by Crystal for each of the four clock phases. The clock phase name indicates which clock phase the path belongs to, and the path description is in the form of $\cdots \rightarrow$element$\rightarrow \cdots$, where element is a functional block or a bus. Entries of the form element(subelement) indicate actions within the element. The paths are listed in the order they were found, and the last one in each sequence is the critical path for that clock phase.

ϕ_1 rf→busB→parB→cmp
The comparator doesn't need the parity of the value read out of the register file onto busB until ϕ_3. Section 3.4.2.2 shows that this value becomes stable 40ns after ϕ_1. Since this isn't the critical path within ϕ_1, we prune away this path by clamping parB after the first level of XOR gates so that Crystal will not follow this path.

ϕ_1 rf(rfB.par)→cu
The controller doesn't need the parity read out of the register file until ϕ_2. This parity is generated by reading the parity from the register file and passing it through and XOR gate to remove the parity due to the address. It takes 25ns to read a value onto busB from the register file (section 3.4.2.2) and to pass it through an XOR takes an additional 2.5ns. This path is removed by clamping the parity bits read out of the register file.

ϕ_1 rf→busB→busS31s→shift(sign)
This is a false path because the sign bit is not used for a shift from busS to busR during ϕ_2. It is removed by clamping the busS sign bit.

ϕ_1 rf→busB→busS→shift
The critical path during ϕ_1 consists of reading a value onto busB and sending it on busS to the shifter. Section 3.4.2.2 shows that this process takes slightly more than 30ns, which is acceptable.

ϕ_2 pc→busD→alu(ai)→alu(mcc)
A false path because once data is loaded into the ALU, it starts evaluation when in fact the manchester carry chain is actually precharging during ϕ_2. This is remedied by clamping the manchester carry chain.

ϕ_2 pc→busD→busOUT→parOUT→out.AD32→pads→cmp→cu
No operations cause a module to write onto busD and then gate onto busOUT. Clamping the busD to busOUT gateway fixes this problem.

ϕ_2 state.reset→state.rb→out.state→cmp(error)→cu
The comparison isn't needed until the following ϕ_2. Clamp state.reset and state.rb.

ϕ_2 rftran→busRD→rf(par)
Not needed until ϕ_4. Clamp busRD.

ϕ_2 sdr→busOUT→parOUT→out.AD32→pads→cmp→cu
This critical path to send data off the chip with the parity takes all of ϕ_2 and half of ϕ_3 and is shown in section 3.3. This path is removed by clamping parOUT.

ϕ_2 psw(out.sysmode)→cu→state.int→valid→ctrl.MAR_write
Cancelling a write due to an interrupt is done during ϕ_4 when the *valid bits* are shifted. This path is removed by clamping the privileged opcode error signal.

ϕ_2 pc(mcc) precharge
False path. Precharge does not take this long. Clamp precharge control.

ϕ_2 mar→busOUT→pads→cmp→cu
Not needed until ϕ_3. Clamp busOUT.

ϕ_2 ns→state→pad.state→cmp
Next state is generated during ϕ_3, and its comparison isn't done until the next ϕ_4. Clamp state.repair1 and state.repair2 signals.

ϕ_2 padenb→pad→cmp
The pad comparison isn't needed until ϕ_4. Clamp pad enable signal.

ϕ_2 psw→int→state.int→valid→ctrl.MAR_write
Not needed until ϕ_4. Clamp psw.

ϕ_2 pc→busD→alu(ai)→alu(mcc)→carry
False path. Once again, the manchester carry chain tries to evaluate while it is suppose to be precharging. Clamp carry out.

ϕ_2 ns→state→pad.state→cmp→cu
False path. Not done until ϕ_3 and not needed until the following ϕ_2. Clamp state.suspend.

ϕ_2 pc→busD→bar→busBAR→imm
Not needed until ϕ_3. Clamp BAR after busD inputs.

ϕ_2 rftran
Not needed until ϕ_4. Clamp cwp.

ϕ_2 ext→state.shutdown→int→state.int→valid→ctrl.MAR_write
Not needed until ϕ_4. Clamp state.shutdown.

ϕ_2 ext→state.wait→mem→out.enb.data→cmp→cu
The comparison isn't needed until the following ϕ_2. Clamp state.wait.

ϕ_2 shift→busR→bar→busBAR→imm
Not needed until ϕ_3 next cycle. Clamp BAR after busR inputs.

ϕ_2 busB→sdec→ctrl.badshift→cu
Not needed until ϕ_3. Clamp shift lines.

ϕ_2 int→state.int→valid→ctrl.MAR_write
Not needed until ϕ_4. Clamp state.int.

ϕ_2 pc→busD→alu(ai)→alu(mcc(propagate))
The critical path during ϕ_2 consists of the PC gating onto busD, latching the value into the A input of the ALU, and generating the propagate inputs to the manchester carry chain. Section 3.3 shows that this takes 17ns, which is before the ALU evaluates 20ns after ϕ_2.

ϕ_3 ir→busIN→rftran→busRB→rf(dec→write)
False path. The destination register is selected by the address from the register file's DWB. Clamp busRA and busRB.

ϕ_3 sdec→shift→busR→bar→busBAR→sdec→shift→busR→bar→
→sdec→ctrl.badshift→cu→state.int3→int→pad.ira→cmp→cu
False path. The BAR is generated during ϕ_2 and the underlap following ϕ_2. Clamp busR.

ϕ_3 rf(dec→ram→bitA→rw) (read)
False path. Reads are not done during ϕ_3. Set bitA and bitB to 1 and 0 in case of a write.

ϕ_3 psw→state.PSW→state→pad.state→cmp→rb
Not needed until ϕ_4. Clamp psw.

ϕ_3 pc(mcc→xor) not using bypass
False path. Clamp PC's manchester carry chain.

ϕ_3 ir→busIN→rftran(busRB→busRD)→busRD→rf
False path. If busRD was used (for state repair), then the RD register would not be loaded during the data transfer. Also, not used during normal operation. Select busRD.

ϕ_3 ir→busIN→rftran(busRB→par)
Not needed until ϕ_4. Clamp busRA.par and busRB.par signals.

ϕ_3 alu(mcc→xor) using bypass
The ALU's manchester carry chain takes 27ns to evaluate on ϕ_3 as shown in section 3.6.2.

ϕ_4 pc→busD→bar→busBAR→imm
False path. The BAR does not change on ϕ_4. Clamp BAR.

ϕ_4 rb→pad.RB2→rb→pad.RB1→rb→pad.RB0
Arbitration isn't complete until ϕ_2. Clamp rollback amount enable signals.

ϕ_4 alu→busOUT→parOUT→out.AD32→pads→cmp→cu
This critical path to send to address off the chip with parity takes all of ϕ_4 and half of ϕ_1. The timing is covered in section 3.3. This path is removed by clamping parOUT.

ϕ_4 mem→gate.busOUT_padAD4→pad.AD→cmp→cu
This is part of the previous path. Clamp pad enable lines.

ϕ_4 pc→busD→psw
The PC gates onto busD and the input to the PSW is set. Section 3.3 shows this is ready by the beginning of ϕ_1.

5. Conclusion

As shown in chapter 3, it is possible to apply micro rollback, error detection, and state repair to a RISC style processor without any significant degradation in performance. We were able to achieve our goal of a 100ns cycle time without significantly adding to the critical paths of a processor based upon the Berkeley RISC II, and we also minimized the number of extra cycles needed for micro rollback and state repair. In fact, the only incurred cycle overhead is during ϕ_1, when the read from the register file is delayed by less than 3ns due to the increased capacitance from the register file's DWB.

The area overhead, however, is a large percentage of the active area on the chip. The active area overhead of error detection, micro rollback, and state repair is 41.6%. If these features were removed from the *MP* chip, the the chip dimensions of $8395\lambda \times 6716\lambda$ would be reduced in width by 1887λ and in height by 1504λ for a overhead of 39.8%. One must keep in mind though, that the initial area of the Mirror Processor is less than that of the Berkeley RISC II because the *MP*'s register file contains half the number of registers that the Berkeley RISC's register file contains. If the *MP* were implemented with 132 registers, then the overhead would be less because more area would be dedicated toward normal functionality.

The *MP* chip is designed so that the master and slave processors use identical pins except for the master/slave pin; this allows a slave chip package to be mounted on top of a master chip package so as to present the footprint of a single chip on a printed circuit board. Though the overhead of micro rollback in terms of silicon real estate is high, at the higher level of the printed circuit board, the Mirror Processor presents a profile comparable to a RISC type processor without micro rollback.

If we were to design a second generation of the Mirror Processor, the first problem we would work on would be the critical paths during ϕ_1, when the register file is read and during ϕ_3, when the ALU evaluates its manchester carry chain. The poly select lines within the register file ram constitute a considerable delay which could be alleviated by redrawing them in metal. In order to keep the stride of the ram cell minimal (because there are 74 addresses), we would have to increase the pitch of the datapath. An implication of changing the register file to metal control lines is that the evaluation control lines in the delayed write buffers may also have to be redone in metal to keep up with the increase in speed.

The ALU uses a manchester carry chain, which is an optimized precharged ripple carry adder. Faster results can be gained by using a carry look-ahead adder, but at the expense of an increase in pitch and stride.

The shifter performs the dual role of data alignment for immediate operands and load and store instructions, and data shifting for the shift instructions. Adding a data aligner to the datapath would allow the shifter to move its operation out of the critical path in ϕ_4 to ϕ_3, where it can operate in parallel with the ALU.

Any processor design is a series of tradeoffs, and implementing micro rollback on a RISC style processor is no exception. Micro rollback allows error detection to be performed in parallel with normal processor operation so that error detection time will not be added to the overall cycle time. Instead of waiting for data verification to complete before proceeding with the next cycle, the Mirror Processor proceeds with execution on the assumption that it can recover from an error detected several cycles later. In order to do this, area must be dedicated toward error recovery. With the Mirror Processor, we have traded off area in order to maintain high performance.

6. References

Tami88a. Yuval Tamir, Marc Tremblay, and David A. Rennels, "The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems," *18th Fault-Tolerant Computing Symposium*, pp. 234-239 (June 1988).

Patt82a. D. A. Patterson and C. H. Séquin, "A VLSI RISC," *Computer* **15**(9), pp. 8-21 (September 1982).

Kate83a. 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).

Lai90a. Titus Lai, "The UCLA Mirror Processor: Control Implementation, Functional Simulation, and Testing Considerations for a RISC with Micro Rollback," UCLA/CSD Master's Report, University of California, Los Angeles, CA (December 1990).

Sher82a. R. W. Sherburne, M. G. H. Katevenis, D. A. Patterson, and C. H. Séquin, "Datapath Design for RISC," *M.I.T. Conference on Advanced Research in VLSI*, pp. 53-62 (January 1982).

McCl86a. Edward J. McCluskey, *Logic Design Principles*, Prentice-Hall (1986).

Brah84a. D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors," *IEEE Transactions on Computers* **C-33**(6), pp. 475-485 (June 1984).

Taub84a. D. M. Taub, "Arbitration and Control Acquisition in the Proposed IEEE 896 Futurebus," *IEEE Micro* **4**(4), pp. 28-41 (August 1984).

Trem89a. Marc Tremblay and Yuval Tamir, "Support for Fault Tolerance in VLSI Processors," *International Symposium on Circuits and Systems*, Portland, OR, pp. 388-393 (May 1989).

West85a. Neil H. E. Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley (1985).

Zyca88a. Zycad Corporation, "ENDOT_ISP' User's Documentation," Part #ND1115 REV 1.0 (1988).

Meta87a. Meta-Software, Inc., *HSPICE User's Manual*, June 1987.

Scot85a. Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout, editors, "1986 VLSI Tools: Still More Works by the Original Artists," Report No. UCB/CSD 86/272, Computer Science Division (EECS), University of California, Berkeley (December 1985).

Segaa. Russell Segal, *Bdsim: a multi-level simulator*, University of California, Berkeley.

Zyca88b. Zycad Corporation, "N.2 ISP' User's Manual," Document #101, Version 1.14 (1988).

Bell70a. C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill (1970).

7. Appendix

7.1. Instruction Reference

Table 7.2 summarizes the Mirror Processor instruction set. Table 7.1 lists the condition codes for the conditional branch instructions. See section 2.2 for more details.

code	symbol	name	value
0000	nev	never	0
0001	gt	greater than (signed)	$\overline{(n \oplus v) \lor z}$
0010	le	less or equal (signed)	$(n \oplus v) \lor z$
0011	ge	greater or equal (signed)	$\overline{n \oplus v}$
0100	lt	less than (signed)	$n \oplus v$
0101	hi	higher than (unsigned)	$\overline{\bar{c} \lor z}$
0110	los	lower or same (unsigned)	$\bar{c} \lor z$
0111	lo nc	lower than (unsigned) no carry	$\bar{c}$
1000	his c	higher or same (unsigned) carry	c
1001	pl	plus (signed)	$\bar{n}$
1010	mi	minus (signed)	n
1011	ne	not equal	$\bar{z}$
1100	eq	equal	z
1101	nv	no overflow (signed)	$\bar{v}$
1110	v	overflow (signed)	v
1111	alw	always	1

Table 7.1: Mirror Condition Codes For Conditional Branch Instructions

opcode	operation	description
add	$d \leftarrow s1 + s2$	add
addbpm	$d \leftarrow s1 + s2; \bar{p}$	add with bad parity on master
addbps	$d \leftarrow s1 + s2; \bar{p}$	add with bad parity on slave
addc	$d \leftarrow s1 + s2 + c$	add with carry
and	$d \leftarrow s1 \wedge s2$	bitwise and
calli	$CWP-; d \leftarrow LSTPC$	call interrupt
callr	$CWP-; d \leftarrow PC; PC \leftarrow PC + sxt(imm19)$	call pc relative
callx	$CWP-; d <- PC; PC \leftarrow s1 + s2$	call absolute
clrrbm	$rb \leftarrow 0$	clear rollback bit on master
clrrbs	$rb \leftarrow 0$	clear rollback bit on slave
getlpc	$d \leftarrow LSTPC$	get last pc
getpsw	$d \leftarrow PSW$	get psw
jmpr	$PC \leftarrow PC + sxt(imm19)$	jump pc relative
jmprbm	$d \leftarrow s1; PC \leftarrow PC + s2$	jump if rollback bit is set, master
jmprbs	$d \leftarrow s1; PC \leftarrow PC + s2$	jump if rollback bit is set, slave
jmpx	$PC \leftarrow s1 + s2$	jump absolute
ldhi	$d \leftarrow imm19 << 13$	load immediate into bits 31:13
ldrbpm	$d \leftarrow M[PC + sxt(imm19)]; \bar{p}$	load with bad parity on master
ldrbps	$d \leftarrow M[PC + sxt(imm19)]; \bar{p}$	load with bad parity on slave
ldrbs	$d \leftarrow sxt(M[PC + sxt(imm19)]<7:0>)$	load pc relative byte signed
ldrbu	$d \leftarrow M[PC + sxt(imm19)]<7:0>$	load pc relative byte unsigned
ldrhs	$d \leftarrow sxt(M[PC + sxt(imm19)]<15:0>)$	load pc relative halfword signed
ldrhu	$d \leftarrow M[PC + sxt(imm19)]<15:0>$	load pc relative halfword unsigned
ldrw	$d \leftarrow M[PC + sxt(imm19)]$	load pc relative word
ldxbs	$d \leftarrow sxt(M[s1 + s2]<7:0>)$	load absolute byte signed
ldxbu	$d \leftarrow M[s1 + s2]<7:0>$	load absolute byte unsigned
ldxhs	$d \leftarrow sxt(M[s1 + s2]<15:0>)$	load absolute halfword signed
ldxhu	$d \leftarrow M[s1 + s2]<15:0>$	load absolute halfword unsigned
ldxw	$d \leftarrow M[s1 + s2]$	load absolute word
or	$d \leftarrow s1 \vee s2$	bitwise or
putpsw	$PSW \leftarrow s1 + s2$	put psw
ret	$PC \leftarrow s1 + s2; CWP+$	return from call
reti	$PC \leftarrow s1 + s2; CWP+; PSW(I,S)$	return from interrupt
sll	$d \leftarrow s1 << s2$	logical shift left by s2 mod 32
sra	$d \leftarrow s1 >> s2$	arithmetic shift right by s2 mod 32
srl	$d \leftarrow s1 >> s2$	logical shift right by s2 mod 32
strb	$d \rightarrow M[PC + sxt(imm19)]<7:0>$	store pc relative byte
strbdm	$d \rightarrow M[PC + sxt(imm19)]$	store bad data, master
strbds	$d \rightarrow M[PC + sxt(imm19)]$	store bad data, slave
strh	$d \rightarrow M[PC + sxt(imm19)]<15:0>$	store pc relative halfword
strw	$d \rightarrow M[PC + sxt(imm19)]$	store pc relative word
stxb	$d \rightarrow M[s1 + s2]<7:0>$	store absolute byte
stxh	$d \rightarrow M[s1 + s2]<15:0>$	store absolute halfword
stxw	$d \rightarrow M[s1 + s2]$	store absolute word
sub	$d \leftarrow s1 - s2$	subtract
subc	$d \leftarrow s1 - s2 - \bar{c}$	subtract with carry
xor	$d \leftarrow s1 \oplus s2$	bitwise xor

Table 7.2: Mirror Processor Instruction Set

7.2. Bdsim Test Programs

The following four Mirror Processor assembly language programs and the following signal list are used to test the signals between all the modules in the datapath and the controller.

Linear.ras tests all the normal instructions except for the jump instructions, which are covered by jmp.ras. Interrupts and traps resulting from normal operation are tested by int.ras. Error detection, micro rollback, and state repair are tested by running the test instructions in test.ras. The first section of code in each program is used to initialize the register file to a known state so that the endot simulation state and the bdsim simulation state will match before the blocks are tested. Verification is not performed during this initialization.

Mirror.signals is an equivalency list of bdsim signals and endot signals and the times that the signals need to be valid.

linear.ras

```
#
#       test linear instructions
#
#       getpsw, getlpc, putpsw, callx, callr, ret
#       sll, sra, srl, ldhi, and, or, xor, add, addc, sub, subc
#       ldxw, ldrw, ldxhu, ldrhu, ldxhs, ldrhs, ldxbu, ldrbu, ldxbs, ldrbs
#       stxw, strw, stxh, strh, stxb, strb
#

init:
        add     r0,r0,r1
        add     r0,r0,r2
        add     r0,r0,r3
        add     r0,r0,r4
        add     r0,r0,r5
        add     r0,r0,r6
        add     r0,r0,r7
        add     r0,r0,r8
        add     r0,r0,r9
        add     r0,r0,r10
        add     r0,r0,r11
        add     r0,r0,r12
        add     r0,r0,r13
        add     r0,r0,r14
        add     r0,r0,r15
        add     r0,r0,r16
        add     r0,r0,r17
        add     r0,r0,r18
        add     r0,r0,r19
        add     r0,r0,r20
        add     r0,r0,r21
        add     r0,r0,r22
        add     r0,r0,r23
        add     r0,r0,r24
        add     r0,r0,r25
        add     r0,r0,r26
        add     r0,r0,r27
        add     r0,r0,r28
        add     r0,r0,r29
        add     r0,r0,r30
        add     r0,r0,r31
        putpsw  0x68a(r0)           # cwp=11, swp=01, isp=000, znvc=1010
        add     r0,r0,r10
        add     r0,r0,r11
        add     r0,r0,r12
        add     r0,r0,r13
        add     r0,r0,r14
        add     r0,r0,r15
        add     r0,r0,r16
        add     r0,r0,r17
        add     r0,r0,r18
        add     r0,r0,r19
        add     r0,r0,r20
        add     r0,r0,r21
        add     r0,r0,r22
        add     r0,r0,r23
        add     r0,r0,r24
        add     r0,r0,r25

endinit:
        add     r0,r0,r0

                                    # getpsw, putpsw, getlpc
misc:
        getpsw  r1
        putpsw  0x085(r0)           # cwp=00, swp=01, isp=000, znvc=0101
        getlpc  r1

                                    # ldxw, ldrw, ldxhu, ldrhu, ldxhs, ldrhs
                                    # ldxbu, ldrbu, ldxbs, ldrbs
                                    # add, sll, or, and
load:
        add     r0,$abscissa,r2
        ld      0(r2),r5            # r5     AB5C155A
        ldr     coffee,r6           # r6     CC00FFEE
        add     r0,$deadcode,r2
        ldhu    0(r2),r7
        sll     r7,$16,r7
        ldrhu   deadcode+2,r1
        or      r7,r1,r7            # r7     DEADC0DE
        add     r0,$geodesic,r2
        ldh     0(r2),r8
        sll     r8,$16,r8
        ldrh    geodesic+2,r1
        and     r1,$0xffff,r1
        add     r8,r1,r8            # r8     6E0DE51C
        add     r0,$gooseegg,r2
        ldbu    0(r2),r9
        sll     r9,$8,r9
        ldrbu   gooseegg+1,r1
        or      r9,r1,r9
        sll     r9,$8,r9
        ldb     2(r2),r1
        and     r1,$0xff,r1
        add     r9,r1,r9
        sll     r9,$8,r9
        ldrb    gooseegg+3,r1
        and     r1,$0xff,r1
        or      r9,r1,r9            # r9     6005EE66

                                    # xor, ldhi, sra, srl, and
parity:                             # parity(DEADCODE)
        ldhi    $0x8000,r2
        sra     r2,$24,r2           # 16
        srl     r7,r2,r1
        xor     r7,r1,r1
        sra     r1,$8,r2
        xor     r1,r2,r1
        srl     r1,$2,r2
        srl     r1,$2,r2
        xor     r1,r2,r1
        sra     r1,$2,r2
        xor     r1,r2,r1
        srl     r1,$1,r2
        xor     r1,r2,r1
        and     r1,$0x1,r1

                                # add, addc
dadd:                           # AB5C155A.CC00FFEE+DEADC0DE.6E0DE51C
        add     r6,r8,r1,(c)
        addc    r5,r7,r2,(c)

                                # sub, subc
dsub:                           # CC00FFEE.DEADC0DE-6E0DE51C.6005EE66
        sub     r7,r9,r1,(c)
        subc    r6,r8,r2,(c)

                                # stxw, strw, stxh, strh, stxb, strb, add
store:
        add     r0,$mem,r1
        st      r5,0(r1)
        str     r6,mem
        sth     r7,0(r1)
        strh    r8,mem+2
        stb     r9,0(r1)
        strb    r5,mem+3

                                # call, callr, add
subr:
        add     r7,r0,r15
        add     r0,$inc,r1
        call    0(r1),r26
        add     r0,r0,r0
        add     r14,r0,r15
        callr   inc,r26
        add     r0,r0,r0
        add     r14,r0,r1

end:
        add     r0,r0,r0
        add     r0,r0,r0
        add     r0,r0,r0

                                # ret, add
inc:
        add     r31,$1,r30
        ret     alw,8(r26)      # unconditional
        add     r0,r0,r0

abscissa:
        .word   0x5A155CAB      # 0xAB5C155A
coffee:
        .word   0xEEFF00CC      # 0xCC00FFEE
deadcode:
        .word   0xDEC0ADDE      # 0xDEADC0DE
geodesic:
        .word   0x1CE50D6E      # 0x6E0DE51C
gooseegg:
        .word   0x66EE0560      # 0x6005EE66
mem:
        .word   0x00000000
```

jmp.ras

```
#
#       test conditional branch instructions
#
#       jne, jpl, jnv, jult
#


init:
        add     r0,r0,r1
        add     r0,r0,r2
        add     r0,r0,r3
        add     r0,r0,r4
        add     r0,r0,r5
        add     r0,r0,r6
        add     r0,r0,r7
        add     r0,r0,r8
        add     r0,r0,r9
        add     r0,r0,r10
        add     r0,r0,r11
        add     r0,r0,r12
```

```
                add     r0,r0,r13
                add     r0,r0,r14
                add     r0,r0,r15
                add     r0,r0,r16
                add     r0,r0,r17
                add     r0,r0,r18
                add     r0,r0,r19
                add     r0,r0,r20
                add     r0,r0,r21
                add     r0,r0,r22
                add     r0,r0,r23
                add     r0,r0,r24
                add     r0,r0,r25
                add     r0,r0,r26
                add     r0,r0,r27
                add     r0,r0,r28
                add     r0,r0,r29
                add     r0,r0,r30
                add     r0,r0,r31
                putpsw  0x68a(r0)       # cwp=11, swp=01, isp=000, znvc=1010
                add     r0,r0,r10
                add     r0,r0,r11
                add     r0,r0,r12
                add     r0,r0,r13
                add     r0,r0,r14
                add     r0,r0,r15
                add     r0,r0,r16
                add     r0,r0,r17
                add     r0,r0,r18
                add     r0,r0,r19
                add     r0,r0,r20
                add     r0,r0,r21
                add     r0,r0,r22
                add     r0,r0,r23
                add     r0,r0,r24
                add     r0,r0,r25

endinit:
                add     r0,r0,r0

scc:
                putpsw  0x083(r0)       # cwp=00, swp=01, isp=000, znvc=0011
                call    cc,r26
                add     r0,r0,r0
                add     r15,r0,r1
                putpsw  0x087(r0)       # cwp=00, swp=01, isp=000, znvc=0111
                call    cc,r26
                add     r0,r0,r0
                add     r15,r0,r1
                putpsw  0x08D(r0)       # cwp=00, swp=01, isp=000, znvc=1101
                call    cc,r26
                add     r0,r0,r0
                add     r15,r0,r1

end:
                add     r0,r0,r0
                add     r0,r0,r0
                add     r0,r0,r0

cc:
                add     r0,r0,r31
z:
                jmpr    ne,n
                add     r0,$v,r16
                or      r31,$0x8,r31
n:
                jmp     pl,0(r16)
                add     r0,r0,r0
                or      r31,$0x4,r31
v:
                jmpr    nv,c
                add     r0,$ccend,r16
                or      r31,$0x2,r31
c:
                jmp     ult,0(r16)
                add     r0,r0,r0
                or      r31,$0x1,r31
ccend:
                ret     alw,8(r26)      # unconditional
                add     r0,r0,r0
```

int.ras

```
#
#       test interrupts, traps, and waits
#
#       reset, badop, privop, badaddr, badshift, int, over, under, wait
#

trapv:
        getlpc  r24
        jmpr    alw,trap
        getpsw  r23
ntrap:  .word   0x4C000000      # init:

intv:
        getlpc  r24
        jmpr    alw,int
        getpsw  r23
        .space  4

overv:
        getlpc  r24
        jmpr    alw,over
        getpsw  r23
        .space  4

underv:
        getlpc  r24
        jmpr    alw,under
        getpsw  r23
        .space  4

trap:
        ldr     ntrap,r22
        jmp     alw,0(r22)
        add     r0,r0,r0

init:
        add     r0,r0,r1
        add     r0,r0,r2
        add     r0,r0,r3
        add     r0,r0,r4
        add     r0,r0,r5
        add     r0,r0,r6
        add     r0,r0,r7
        add     r0,r0,r8
        add     r0,r0,r9
        add     r0,r0,r10
        add     r0,r0,r11
        add     r0,r0,r12
        add     r0,r0,r13
        add     r0,r0,r14
        add     r0,r0,r15
        add     r0,r0,r16
        add     r0,r0,r17
        add     r0,r0,r18
        add     r0,r0,r19
        add     r0,r0,r20
        add     r0,r0,r21
        add     r0,r0,r22
        add     r0,r0,r23
        add     r0,r0,r24
        add     r0,r0,r25
        add     r0,r0,r26
        add     r0,r0,r27
        add     r0,r0,r28
        add     r0,r0,r29
        add     r0,r0,r30
        add     r0,r0,r31
        putpsw  0x680(r0)       # cwp=11, swp=01, isp=000, znvc=0000
        add     r0,r0,r10
        add     r0,r0,r11
        add     r0,r0,r12
        add     r0,r0,r13
        add     r0,r0,r14
        add     r0,r0,r15
        add     r0,r0,r16
        add     r0,r0,r17
        add     r0,r0,r18
        add     r0,r0,r19
        add     r0,r0,r20
        add     r0,r0,r21
        add     r0,r0,r22
        add     r0,r0,r23
        add     r0,r0,r24
        add     r0,r0,r25
        putpsw  0x280(r0)       # cwp=01, swp=01, isp=000, znvc=0000
        add     r0,r0,r16
        add     r0,r0,r17
        add     r0,r0,r18
        add     r0,r0,r19
        add     r0,r0,r20
        add     r0,r0,r21
        add     r0,r0,r22
        add     r0,r0,r23
        add     r0,r0,r24
        add     r0,r0,r25
        add     r0,r0,r26
        add     r0,r0,r27
        add     r0,r0,r28
        add     r0,r0,r29
        add     r0,r0,r30
        add     r0,r0,r31
        add     r0,r0,r0

reset:
        add     r0,$badop,r22
        st      r22,ntrap
        putpsw  0x0E0(r0)       # cwp=00, swp=01, isp=110, znvc=0000
        add     r0,r0,r0        # assert reset line
        add     r0,r0,r0

badop:
        add     r0,$privop,r22
        st      r22,ntrap
        putpsw  0x0E0(r0)       # cwp=00, swp=01, isp=110, znvc=0000
        .word   0x00000000      # badop
        add     r0,r0,r0

privop:
        add     r0,$badaddr,r22
        st      r22,ntrap
        putpsw  0x0E0(r0)       # cwp=00, swp=01, isp=110, znvc=0000
        putpsw  0x0E0(r0)       # privop
        add     r0,r0,r0

badaddr:
        add     r0,$badshift,r22
        st      r22,ntrap
        putpsw  0x0E0(r0)       # cwp=00, swp=01, isp=110, znvc=0000
        jmpr    alw,badaddr+1   # badaddr
        add     r0,r0,r0

badshift:
        add     r0,$overset,r22
        st      r22,ntrap
        putpsw  0x0E0(r0)       # cwp=00, swp=01, isp=110, znvc=0000
        srl     r0,$4,r22       # badshift
        add     r0,r0,r0

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

overset:
        putpsw  0x1E0(r0)       # cwp=00, swp=11, isp=110, znvc=0000
        callr   over,r26        # overflow
        add     r0,r0,r0

over:
underset:
        putpsw  0x0E0(r0)       # cwp=00, swp=01, isp=110, znvc=0000
        ret     alw,8(r26)      # underflow
        add     r0,r0,r0

under:
intset:
        putpsw  0x0E0(r0)       # cwp=00, swp=01, isp=110, znvc=0000
        add     r0,r0,r0        # assert interrupt line
        add     r0,r0,r0

wait:
        add     r0,$0xF00,r22   # assert wait
        st      r22,mem         # assert wait
        ld      mem,r21         # assert wait

end:
        add     r0,r0,r0
        add     r0,r0,r0
        add     r0,r0,r0
```

```
mem:
        .word   0x00000000
```

test.ras

```
#
#       test test instructions, error detection, rollbacks, and state repair
#
#       clrrbm, clrrbs, jmprbm, jmprbs
#       addbpm, addbps
#       ldrbpm, ldrbps
#       strbdm, strbds
#


reset:
        jmpr    alw,init2
        add     r0,r0,r1
        .space  8

int:
        .space  16

over:
        .space  16

under:
        .space  16

shutdown:
        add     r0,r0,r0
        add     r0,r0,r0
        add     r0,r0,r0
        add     r0,r0,r0

init2:
        add     r0,r0,r2
        add     r0,r0,r3
        add     r0,r0,r4
        add     r0,r0,r5
        add     r0,r0,r6
        add     r0,r0,r7
        add     r0,r0,r8
        add     r0,r0,r9
        add     r0,r0,r10
        add     r0,r0,r11
        add     r0,r0,r12
        add     r0,r0,r13
        add     r0,r0,r14
        add     r0,r0,r15
        add     r0,r0,r16
        add     r0,r0,r17
        add     r0,r0,r18
        add     r0,r0,r19
        add     r0,r0,r20
        add     r0,r0,r21
        add     r0,r0,r22
        add     r0,r0,r23
        add     r0,r0,r24
        add     r0,r0,r25
        add     r0,r0,r26
        add     r0,r0,r27
        add     r0,r0,r28
        add     r0,r0,r29
        add     r0,r0,r30
        add     r0,r0,r31

endinit:
        add     r0,r0,r0

master:
        clrrbm
        clrrbs

        ldrbpm  mem,r1
        add     r0,r0,r0
        add     r0,r0,r0

        clrrbm
        clrrbs
        addbpm  r1,$1,r2
        add     r2,$1,r3
        add     r0,r0,r0
        add     r0,r0,r0

        clrrbm
        clrrbs

        strbdm  r3,mem
        add     r0,r0,r0
        add     r0,r0,r0

        clrrbm
        clrrbs

        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     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

slave:
        clrrbm
        clrrbs

        ldrbps  mem,r1
        add     r0,r0,r0
        add     r0,r0,r0

        clrrbm
        clrrbs

        addbps  r1,$1,r2
        add     r2,$1,r3
        add     r0,r0,r0
        add     r0,r0,r0

        clrrbm
        clrrbs

        strbds  r3,mem
        add     r0,r0,r0
        add     r0,r0,r0

        clrrbm
        clrrbs

        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     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

jump:
        add     r0,$16,r1       # branch offset
        add     r0,$0xF00,r2

        clrrbm
        clrrbs

        jmprbm  r2,r1,r3
        add     r0,r0,r0

loopm:
        jmpr    alw,loopm
        add     r0,r0,r0
        clrrbm
        clrrbs

        jmprbs  r2,r1,r3
        add     r0,r0,r0

loops:
        jmpr    alw,loops
        add     r0,r0,r0

die:
        add     r0,$8,r1        # branch offset
        clrrbm
        clrrbs
        jmprbm  r2,r1,r0
        add     r0,r0,r0
        clrrbm
        clrrbs
        jmprbm  r2,r1,r0
        add     r0,r0,r0
        add     r0,r0,r0
        add     r0,r0,r0

mem:
        .word   0xDEC0ADDE      # 0xDEADC0DE
```

mirror.signals

```
#
#       mirror.signals - bdsim and endot signal names and active times
#
#       m> bdsim make vector
#       s> set signal
#       v> verify signal
#       c> charge sharing propagated signal

mv:
m>mv pad.AD pad.AD32:0
m>mv in.AD in.AD32:0
m>mv _busA _busA31:0
m>mv _busB _busB31:0
m>mv busB_SDEC busB4:0_SDEC
m>mv busD busD31:0
m>mv busIMM busIMM4:0
m>mv busIN busIN32:0
m>mv busIR busIR13:0
m>mv busOUT busOUT31:0
m>mv busR busR31:0
m>mv busS busS31:0
m>mv busT busT31:0
m>mv out.AD out.AD32:0
m>mv busCC busCC3:0
m>mv busCWP busCWP1:0
m>mv busRA busRA6:0
m>mv busRB busRB6:0
m>mv busRD busRD6:0
m>mv busSDEC busSDEC24 busSDEC16 busSDEC13 busSDEC8 busSDEC2 busSDEC1 busSDEC0
m>mv busBAR busBAR1:0
m>mv state.PSW state.PSW3:0
m>mv state.RFaddr state.RFaddr3:0
m>mv state.busD state.busD3:0
m>mv state.cc state.cc3:0
m>mv _ivec _ivec6:4
m>mv ctrl.ALU_op ctrl.ALU_op4:0
m>mv ctrl.CWP_inc ctrl.CWP_inc1:0
m>mv out.size out.size1:0
m>mv pad.size pad.size1:0
m>mv state.RB state.rb2:0
m>mv CUdata.in CUdata.in1:0
m>mv busCU3 busCU31 busCU30
m>mv enb.pad.RB enb.pad.RB2:0
m>mv error.state_ error.state3:0_
m>mv in.RB in.RB2:0
m>mv pad.RB pad.RB2:0
m>mv st st2:0
m>mv out.state out.state3:0
m>mv pad.state pad.state3:0

c->c:
c>busIR                 :busIR                  !p12 p22 p23 p33 p34 p44 p41
v>ALU.carry_car         :ALU.carry_car          !p23 p33 p34
```

```
v>ALU.carry_val              :ALU.carry_val              !p23 p33 p34
v>ALU.gateOUT                :ALU.gateOUT                !p34 p44 p41
v>ALU.gateOUT_cond           :ALU.gateOUT_cond           !p34 p44 p41
v>CUdata.in                  :cu:CUdata.in               !p23 p33 p34
v>INC.gate                   :INC.gate                   !p34 p44 p41
v>INC.gate_cond              :INC.gate_cond              !p34 p44 p41
v>IR.write                   :IR.write                   !p23 p33 p34 p44 p41
v>PAR.busA                   :PAR.busA                   !p23 p33 p34
v>PAR.busB                   :PAR.busB                   !p23 p33 p34
v>PAR.busIN                  :PAR.busIN                  !p41 p11 p12
v>PAR.busOUTA                :PAR.busOUTA                !p41 p11 p12
v>PAR.busOUTB                :PAR.busOUTB                !p41 p11 p12
v>PC.write                   :PC.write                   !p34 p44 p41
v>RF.write                   :RF.write                   !p34 p44 p41
v>busCU3                     :cu:busCU                   !p23 p33 p34
v>cond                       :cu:cond                    !p11 p12 p34 p44 p41
v>ctrl.CU_write              :cu:ctrl.CU_write           !p34 p44 p41
v>ctrl.RB_write              :ctrl.RB_write              !p23 p33 p34
v>ctrl.badop                 :ctrl.badop                 !p12 p22 p23
v>ctrl.clrrb                 :ctrl.clrrb                 !p41 p11 p12 p22 p23
v>ctrl.norepair              :ctrl.norepair              !p23 p33 p34
v>ctrl.over_under            :ctrl.over_under            !p23 p33 p34
v>ctrl.override_master_      :fsm:ctrl.override_master_        !p12 p22 p23
v>ctrl.override_slave_       :fsm:ctrl.override_slave_         !p12 p22 p23
v>ctrl.privop                :ctrl.privop                !p12 p22 p23
v>gate.IR                    :gate.IR                    !p23 p33 p34
v>gate.busOUT_padAD2         :gate.busOUT_padAD2         !p12 p22 p23
v>gate.busOUT_padAD4         :gate.busOUT_padAD4         !p34 p44 p41
v>load.PAR_busA              :load.PAR_busA              !p23 p33 p34
v>load.PAR_busB              :load.PAR_busB              !p23 p33 p34
v>load.PAR_busIN             :load.PAR_busIN             !p41 p11 p12
v>load.PAR_busOUTA           :load.PAR_busOUTA           !p41 p11 p12
v>load.PAR_busOUTB           :load.PAR_busOUTB           !p41 p11 p12
v>repairA                    :fsm:repairA                !p11 p12 p23 p33 p34 p44 p41
v>repairA_                   :fsm:repairA_               !p11 p12 p23 p33 p34 p44 p41
v>repairB                    :fsm:repairB                !p11 p12 p23 p33 p34 p44 p41
v>repairB_                   :fsm:repairB_               !p11 p12 p23 p33 p34 p44 p41
v>st                         :fsm:st                     !p34 p44 p41
v>state.IO                   :state.IO                   !p41 p11 p12
v>state.op_prev_             :fsm:state.op_prev_         !p12 p22 p23 p33 p34
v>state.repair1              :state.repair1              !p11 p12 p22 p23 p33 p34 p44 p41
v>state.repair2              :state.repair2              !p11 p12 p22 p23 p33 p34 p44 p41
v>state.shutdown             :state.shutdown             !p41 p11 p12 p22 p23 p33 p34
v>state.suspend              :state.suspend              !p12 p22 p23 p33 p34
v>state.wait                 :state.wait                 !p11 p12 p23 p33 p34 p44 p41

c->d:
v>_ivec                      :int:ivec                   !p34 p44 p41
v>ctrl.ALU_c                 :ctrl.ALU_c                 !p23 p33 p34
v>ctrl.ALU_op                :ctrl.ALU_op                !p22 p23 p33 p34
v>ctrl.CWP_inc               :ctrl.CWP_inc               !p12 p22 p23 p33 p34
v>ctrl.DIMM_sxt              :ctrl.DIMM_sxt              !p23 p33 p34
v>ctrl.IMM_write             :ctrl.IMM_write             !p23 p33 p34
v>ctrl.IR_write              :ctrl.IR_write              !p34 p44 p41
v>ctrl.MAR_write             :ctrl.MAR_write             !p41 p11 p12
v>ctrl.PC_select             :ctrl.PC_select             !p41 p11 p12
v>ctrl.PC_write              :ctrl.PC_write              !p34 p44 p41
v>ctrl.PSW_reti              :ctrl.PSW_reti              !p44 p41 p11 p12
v>ctrl.PSW_write             :ctrl.PSW_write             !p34 p44 p41
v>ctrl.RF_write              :ctrl.RF_write              !p34 p44 p41
v>ctrl.RFpar_invert          :ctrl.RFpar_invert          !p41 p11 p12
v>ctrl.SDR_write             :ctrl.SDR_write             !p34 p44 p41
v>ctrl.SHIFT_sxtS            :ctrl.SHIFT_sxtS            !p12 p22 p23 p34 p44 p41
v>ctrl.SHIFT_sxtT            :ctrl.SHIFT_sxtT            !p12 p22 p23 p34 p44 p41
v>ctrl.busINpar_invert       :ctrl.busINpar_invert       !p23 p33 p34
v>gate.ALU_busD              :gate.ALU_busD              !p34 p44 p41
v>gate.ALU_busOUT            :gate.ALU_busOUT            !p34 p44 p41
v>gate.DIMM_busT             :gate.DIMM_busT             !p23 p33 p34
v>gate.IMM13_busT            :gate.IMM13_busT            !p41 p11 p12
v>gate.IMM19_busT            :gate.IMM19_busT            !p41 p11 p12
v>gate.INC_busOUT            :gate.INC_busOUT            !p34 p44 p41
v>gate.MAR_busOUT2           :gate.MAR_busOUT2           !p12 p22 p23
v>gate.MAR_busOUT4           :gate.MAR_busOUT4           !p34 p44 p41
v>gate.PC_busD2              :gate.PC_busD2              !p12 p22 p23
v>gate.PC_busD4              :gate.PC_busD4              !p34 p44 p41
v>gate.PSW_busD              :gate.PSW_busD              !p34 p44 p41
v>gate.SDR_busOUT            :gate.SDR_busOUT            !p12 p22 p23
v>gate.SHIFT_busL            :gate.SHIFT_busL            !p34 p44 p41
v>gate.SHIFT_busR            :gate.SHIFT_busR            !p34 p44 p41
v>gate.busA_busD2            :gate.busA_busD2            !p12 p22 p23
v>gate.busA_busD4            :gate.busA_busD4            !p34 p44 p41
v>gate.busA_busS             :gate.busA_busS             !p11 p12 p22 p23 p33 p34 p44 p41
v>gate.busD_busOUT           :gate.busD_busOUT           !p12 p22 p23
v>gate.busIN_busT            :gate.busIN_busT            !p23 p33 p34
v>gate.padAD_busIN           :gate.padAD_busIN           !p23 p33 p34
v>load.BAR_busDR             :load.BAR_busDR             !p12 p22 p23 p33
v>load.PSW_busCC             :load.PSW_busCC             !p41 p11 p12
v>load.PSW_busD              :load.PSW_busD              !p41 p11 p12
```

```
v>load.RFTRAN_busIN      :load.RFTRAN_busIN       !p23 p33 p34
v>load.RFaddr_RA         :load.RFaddr_RA          !p34 p44 p41
v>load.RFaddr_RB         :load.RFaddr_RB          !p34 p44 p41
v>load.RFaddr_RD         :load.RFaddr_RD          !p34 p44 p41
v>load.SHIFT_IMM         :load.SHIFT_IMM          !p12 p22 p23
v>load.SHIFT_busL        :load.SHIFT_busL         !p34 p44 p41
v>load.SHIFT_busR        :load.SHIFT_busR         !p34 p44 p41
v>load.SHIFT_busT        :load.SHIFT_busT         !p34 p44 p41
v>load.SHam_0            :load.SHam_0             !p41 p11 p12 p22 p23
v>load.SHam_BAR          :load.SHam_BAR           !p23 p33 p34
v>load.SHam_IMM          :load.SHam_IMM           !p12 p22 p23
v>load.SHam_busB         :load.SHam_busB          !p12 p22 p23

c->p:
v>enb.AD_                :cmp:enb.AD_             !p12 p22 p23 p34 p44 p41
v>enb.pad.RB             :rb:enb.pad.RB           !p11 p12 p22 p23 p33 p34 p44 p41
v>enb.pad.rb             :rb:enb.pad.rb           !p11 p12 p22 p23 p33 p34 p44 p41
v>enb.pad.shutdown       :rb:enb.pad.shutdown     !p11 p12 p22 p23 p33 p34 p44 p41
v>out.enb.addr           :pad.enb.addr            !p11 p12 p22 p23 p33 p34 p44 p41
v>out.enb.data           :pad.enb.data            !p11 p12 p22 p23 p33 p34 p44 p41
v>out.id                 :pad.id                  !p11 p12 p22 p23 p33 p34 p44 p41
v>out.ira                :pad.ira                 !p11 p12 p22 p23 p33 p34 p44 p41
v>out.repairA_           :rb:out.repairA_         !p11 p12 p22 p23 p33 p34 p44 p41
v>out.repairB_           :rb:out.repairB_         !p11 p12 p22 p23 p33 p34 p44 p41
v>out.rw                 :pad.rw                  !p11 p12 p22 p23 p33 p34 p44 p41
v>out.state              :cmp:state.state         !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.rw                 :pad.rw                  !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.enb.addr           :pad.enb.addr            !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.enb.data           :pad.enb.data            !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.id                 :pad.id                  !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.ira                :pad.ira                 !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.state              :pad.state               !p11 p12 p22 p23 p33 p34 p44 p41

c->c,d:
v>state.int              :state.int               !p23 p33 p34 p44 p41 p11 p12
v>state.rb               :state.rb                !p11 p12 p23 p33 p34 p44 p41
v>state.RB               :state.RB                !p23 p33 p34
v>state.reset            :state.reset             !p11 p12 p22 p23 p33 p34 p44 p41

c->d,p:
v>out.size               :out.size                !p12 p22 p23 p33 p34
s>pad.size               :pad.size                !p11 p12 p22 p23 p33 p34 p44 p41

d->c:
v>busB.par               :par:busB.par            !p23 p33 p34 p44 p41
v>busD.par               :busD.par                !p23 p33 p34 p44 p41
v>busOUT.par             :busOUT.par              !p41 p11 p12 p23 p33 p34
v>ctrl.badshift          :ctrl.badshift           !p23 p33 p34
v>ctrl.int_enb           :ctrl.int_enb            !p12 p22 p23 p33 p34
v>error.busIN1           :par:error.busIN1        !p41 p11 p12
v>rfA.par                :rfA.par                 !p12 p22 p23 p33 p34 p44 p41
v>rfB.par                :rfB.par                 !p12 p22 p23 p33 p34 p44 p41
v>state.PSW              :state.PSW               !p41 p11 p12 p23 p33 p34
v>state.PSW_overflow     :state.PSW_overflow      !p12 p22 p23 p33 p34
v>state.RFaddr           :state.RFaddr            !p44 p41
v>state.busD             :state.busD              !p11 p12
v>state.cc               :state.cc                !p12 p22 p23

d->d:
v>_busA                  :busA                    !p41 p11 p12
v>_busB                  :busB                    !p41 p11 p12
v>busB_SDEC              :gate:busB_SDEC          !p12 p22 p23
v>busD                   :busD                    !p11 p12 p22 p23 p33 p34 p44 p41
v>busIMM                 :busIMM                  !p12 p22 p23
c>busOUT                 :busOUT                  !p11 p12 p22 p23 p33 p34 p44 p41
v>busR                   :busR                    !p12 p22 p23
v>busS                   :busS                    !p11 p12 p22 p23 p33 p34 p44 p41
v>busS31s                :gate:busS31s            !p11 p12 p22 p23 p33 p34 p44 p41
v>busT                   :busT                    !p11 p12 p22 p23 p33 p34 p44 p41
v>_busRA.par             :busRA.par               !p12 p22 p23 p33 p34 p44 p41
v>_busRB.par             :busRB.par               !p12 p22 p23 p33 p34 p44 p41
v>busCC                  :busCC                   !p41 p11 p12
v>busCWP                 :busCWP                  !p33 p34
v>busD.par_RF            :busD.par                !p11 p12
v>busRA                  :busRA                   !p44 p41 p11
v>busRB                  :busRB                   !p44 p41 p11
v>busRD                  :busRD                   !p44 p41 p11 p12
v>busSDEC                :busSDEC                 !p12 p22 p23 p34 p44 p41
v>rfD.par                :rf:rfD.par              !p11 p12

d->p:
v>out.AD                 :bus:busOUT33            !p11 p12 p22 p23 p33 p34 p44 p41

d->c,d:
c>busIN                  :bus:busIN33             !p33 p34 p44 p41
v>busBAR                 :busBAR                  !p22 p23 p33 p34

d->c,p:
```

```
v>out.sysmode          :out.sysmode            !p12 p22 p23
s>pad.sysmode          :out.sysmode            !p11 p12 p22 p23 p33 p34 p44 p41

p->c:
v>error.AD0_           :cmp:error.AD0_         !p41 p11 p12 p23 p33 p34
v>error.AD1_           :cmp:error.AD1_         !p41 p11 p12 p23 p33 p34
v>error.ADp_           :cmp:error.ADp_         !p41 p11 p12 p23 p33 p34
v>error.CMP_           :cmp:error.CMP_         !p41 p11 p12
v>error.busA           :error.busA             !p34 p44 p41
v>error.busB           :error.busB             !p34 p44 p41
v>error.busOUT_        :par:error.busOUT_      !p41 p11 p12
v>error.enb.addr_      :cmp:error.enb.addr_    !p41 p11 p12
v>error.enb.data_      :cmp:error.enb.data_    !p41 p11 p12
v>error.id_            :cmp:error.id_          !p41 p11 p12
v>error.ira_           :cmp:error.ira_         !p41 p11 p12
v>error.rw_            :cmp:error.rw_          !p41 p11 p12
v>error.size_          :cmp:error.size_        !p41 p11 p12
v>error.state_         :cmp:error.state_       !p23 p33 p34
v>error.sysmode_       :cmp:error.sysmode_     !p23 p33 p34
v>in.RB                :rb:pad.RB              !p12 p22 p23
v>in.irr               :pad.irr                !p12 p22 p23
v>in.rb                :pad.rb                 !p12 p22 p23
v>in.repairAm          :pad.repairAm           !p12 p22 p23
v>in.repairAs          :pad.repairAs           !p12 p22 p23
v>in.repairBm          :pad.repairBm           !p12 p22 p23
v>in.repairBs          :pad.repairBs           !p12 p22 p23
v>in.reset             :pad.reset              !p12 p22 p23
v>in.shutdown          :pad.shutdown           !p12 p22 p23
v>in.wait              :pad.wait               !p12 p22 p23
s>pad.wait             :pad.wait               !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.reset            :pad.reset              !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.irr              :pad.irr                !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.rb               :pad.rb                 !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.RB               :rb:pad.RB              !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.repairAm         :pad.repairAm           !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.repairAs         :pad.repairAs           !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.repairBm         :pad.repairBm           !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.repairBs         :pad.repairBs           !p11 p12 p22 p23 p33 p34 p44 p41
s>pad.shutdown         :pad.shutdown           !p11 p12 p22 p23 p33 p34 p44 p41

p->d:
s>pad.AD               :pad.AD                 !p11 p12 p22 p23 p33 p34 p44 p41
v>in.AD                :pad.AD                 !p33 p34
```

7.3. Endot Code

This section contains the Endot code used to simulate the Mirror Processor at the register-transfer level. Console.isp is used to send signals into the system and read values out of the system (because some lines are active low). Memory.isp is the memory system the processor pair interfaces with. The rest of the modules form one processor. Two instances of this processor are connected together in the topology file, with pad.ms in the master processor set to 0 and pad.ms in the slave processor set to 1.

Most of the names match with the datapath modules and are self explanatory. Cmp.isp contains the pad comparators and part of the controller. Cu.isp, fsm.isp, int.isp, and rb.isp make up the datapath controller. The parity generators are described in par.isp, and the bus drivers to gate between busses are described in gate.isp. Endot only allows wired-OR or wired-AND signals; once the driver is removed, the signal goes back to 0 or 1. Bus.isp is a hack to make the interconnecting wires behave like VLSI capacitive wires that are capable of holding charge for short periods of time.

console.isp

```
/*
 *      console.isp - Control and Examine the System
 */

state
        in.reset,
        in.irr,
        in.rb,
        in.RB<3>,
        in.shutdown,
        out.reset,
        out.irr,
        out.wait,
        out.rb,
        out.RB<3>,
        enb.RB2,
        enb.RB1,
        enb.RB0,
        out.RB2,
        out.RB1,
        out.RB0,
        out.shutdown;
 port
         phi1 'output,
         phi2 'output,
         phi3 'output,
         phi4 'output,

         pullup.reset 'output:and,
         pullup.irr 'output:and,
         pullup.rb 'output:and,
         pullup.RB2 'output:and,
         pullup.RB1 'output:and,
         pullup.RB0 'output:and,
         pullup.shutdown 'output:and,

         pad.reset 'bidirectional:and:disconnect,
         pad.irr 'bidirectional:and:disconnect,
         pad.rb 'bidirectional:and:disconnect,
         pad.wait 'output,
         pad.RB2 'bidirectional:and:disconnect,
         pad.RB1 'bidirectional:and:disconnect,
         pad.RB0 'bidirectional:and:disconnect,
         pad.shutdown 'bidirectional:and:disconnect;

main:=(
         pullup.reset=1;
         pullup.irr=1;
         pullup.rb=1;
         pullup.RB2=1;
         pullup.RB1=1;
         pullup.RB0=1;
         pullup.shutdown=1;

         pad.reset=0;
         pad.irr=0;
         pad.wait=0;
         pad.rb=0;
         pad.RB2=0;
         pad.RB1=0;
         pad.RB0=0;
         pad.shutdown=0;
         next;
         terminate;
)

when (pad.reset):=(
         in.reset=pad.reset;
)

when (pad.irr):=(
         in.irr=pad.irr;
)

when (pad.rb):=(
         in.rb=not pad.rb;
)

when (pad.RB2, pad.RB1, pad.RB0):=(
         in.RB<2>=not pad.RB2;
         in.RB<1>=not pad.RB1;
         in.RB<0>=not pad.RB0;
)

when (pad.shutdown):=(
         in.shutdown=pad.shutdown;
)

when (out.reset):=(
         if (out.reset eql 1)
                 connect(pad.reset)
         else
                 disconnect(pad.reset);
)

when (out.irr):=(
         if (out.irr eql 1)
                 connect(pad.irr)
         else
                 disconnect(pad.irr);
)

when (out.wait):=(
         pad.wait=out.wait;
)

when (out.rb):=(
         if (out.rb eql 1)
                 connect(pad.rb)
         else
```

```
                disconnect(pad.rb);
)

when  (out.RB):=(
        out.RB2=out.RB<2>;
        out.RB1=out.RB<1>;
        out.RB0=out.RB<0>;
)

when  (enb.RB2:change, out.RB2:change, pad.RB2:change):=(
        delay(1);       /* Insert propagation delay */
        if (enb.RB2 and out.RB2)
                connect(pad.RB2)
        else
                disconnect(pad.RB2);
        enb.RB1=enb.RB2 and (out.RB2 or pad.RB2);
)
when  (enb.RB1:change, out.RB1:change, pad.RB1:change):=(
        delay(1);       /* Insert propagation delay */
        if (enb.RB1 and out.RB1)
                connect(pad.RB1)
        else
                disconnect(pad.RB1);
        enb.RB0=enb.RB1 and (out.RB1 or pad.RB1);
)
when  (enb.RB0:change, out.RB0:change, pad.RB0:change):=(
        delay(1);       /* Insert propagation delay */
        if (enb.RB0 and out.RB0)
                connect(pad.RB0)
        else
                disconnect(pad.RB0);
)

when  (out.shutdown):=(
        if (out.shutdown eql 1)
                connect(pad.shutdown)
        else
                disconnect(pad.shutdown);
)
```

memory.isp

```
/*
 *      memory.isp - Memory
 */

macro
        WORD    :=      32&;

state
        address<30>,
        bar<2>,
        rw,
        data_latch<WORD>;

port
        pad.AD<WORD+1> 'bidirectional:disconnect,       /* address/data bus */
        pad.size<2> 'input,                             /* width code */

        pad.enb.addr 'input,    /* address enable */
        pad.enb.data 'input,    /* data enable */
        pad.rw 'input,          /* read/write */
        pad.rb 'input:and;      /* rollback signal */


memory
        inst[0:0]<WORD>;

when LATCH (pad.enb.addr:lead):=( /*addr must be on bus before enable */
        delay(1);       /* latch and decode address */
        address=pad.AD<31:2>;
        bar=pad.AD<1:0>;
        rw=pad.rw;
)

when READ (pad.enb.addr:trail, rw
             ((pad.enb.addr eql 0) and (rw eql 0))):=(
        delay(2);       /* simulate read access delay */
        connect(pad.AD)
        pad.AD=parity(inst[address]) concat inst[address];
        wait(pad.enb.data:trail);
        disconnect(pad.AD)
 )

 when WRITE (pad.enb.data:lead, rw
            ((pad.enb.data eql 1) and (rw eql 1))):=(
        delay(1);       /* latch data */
        data_latch=pad.AD;      /* data should be on bus by now */
        next;
        if (pad.rb eql 1) (
                case (pad.size concat bar)
                8:      /* |W|W|W|W| */
                        inst[address]=data_latch
                4:      /* | | |W|W| */
                        inst[address]<15:0>=data_latch<15:0>
                6:      /* |W|W| | | */
                        inst[address]<31:16>=data_latch<31:16>
                0:      /* | | | |W| */
                        inst[address]<7:0>=data_latch<7:0>
                1:      /* | | |W| | */
                        inst[address]<15:8>=data_latch<15:8>
                2:      /* | |W| | | */
                        inst[address]<23:16>=data_latch<23:16>
                3:      /* |W| | | | */
                        inst[address]<31:24>=data_latch<31:24>
                esac;
        );
)
```

alu.isp

```
/*
 *      alu.isp - Arithmetic/Logic Unit
 */

macro
        WORD    :=      32&;

state
        AI<WORD>,
        BI<WORD>,
        A<WORD+1>,
        B<WORD+1>,
        AxorB<WORD+1>,
        AandB<WORD+1>,
        AorB<WORD+1>,
        L<WORD+1>,
        C<WORD+1>,
        D<WORD+1>;

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busD<WORD> 'bidirectional:disconnect,
        busR<WORD> 'input,
        busOUT<WORD> 'output:disconnect,
        busCC<4> 'output,

        ctrl.ALU_op<5> 'input,  ! 4: 1=subtraction
                                ! 3: 1=XOR
                                ! 2: 1=AND
                                ! 1: 1=OR
                                ! 0: 1=sum  0=logic
        ctrl.ALU_c 'input,
        gate.ALU_busD 'input,
        gate.ALU_busOUT 'input;

when (phi2, busD, busR
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        AI=busD;
        BI=busR;
)

when (AI, BI, ctrl.ALU_op):=(
        delay(1);       /* Insert propagation delay */
        A=AI ext 33;
        if (ctrl.ALU_op<4> eql 1)
                B=(not BI) ext 33
        else
                B=BI ext 33;
        next;
        AxorB=A xor B;
        AandB=A and B;
        AorB=A or B;
)

when (phi3:trail):=(
        delay(1);       /* Insert propagation delay */
        L=(AxorB and (ctrl.ALU_op<3> sxt 33)) or
          (AandB and (ctrl.ALU_op<2> sxt 33)) or
          (AorB and (ctrl.ALU_op<1> sxt 33));
        C=(A+B+(ctrl.ALU_c ext 33)) xor (A xor B);
        next;
        if (ctrl.ALU_op<0> eql 1)
                D=L xor C
        else
                D=L;
)

when (phi4:lead):=(
        busCC<1>=((A<31> and B<31> and (not D<31>)) or
                  ((not A<31>) and (not B<31>) and D<31>)) and
                 ctrl.ALU_op<0>;
        busCC<0>=D<32> and ctrl.ALU_op<0>;
)

when (D):=(
        delay(1);       /* Insert propagation delay */
        busD=D<31:0>;
        busOUT=D<31:0>;
)

when (phi1, busD
      (phi1 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        busCC<2>=busD<31> eql 1;
)

when (phi1:lead):=(
        delay(1);       /* Insert propagation delay */
        busCC<3>=busD eql 0;
)

when (phi4, gate.ALU_busD
      ((phi4 eql 1) and (gate.ALU_busD eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busD);
        wait(phi4:trail, gate.ALU_busD:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busD);
)

when (phi4, gate.ALU_busOUT
      ((phi4 eql 1) and (gate.ALU_busOUT eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busOUT);
        wait(phi4:trail, gate.ALU_busOUT:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busOUT);
)
```

bar.isp

```
/*
 *      bar.isp - Byte Address Register
 */

macro
        WORD    :=      32&;

state
        phi23;

port
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busD<WORD> 'input,
        busR<WORD> 'input,
        busOUT<WORD> 'input,
        busBAR<2> 'output:disconnect,
        busMAR<2> 'output:disconnect,

        load.BAR_busDR 'input,
        state.rb 'input;

when (phi23, load.BAR_busDR
      ((phi23 eql 1) and (load.BAR_busDR eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busBAR);
        wait(phi23:trail, load.BAR_busDR:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busBAR);
)

when (phi2:lead):=(
        delay(1);       /* Insert gate opening delay */
        phi23=1;
)

when (phi3:lead):=(
        delay(1);       /* Insert gate opening delay */
        phi23=0;
)

when (phi4, state.rb
      ((phi4 eql 1) and (state.rb eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busMAR);
        wait(phi4:trail, state.rb:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busMAR);
)

when (busD, busR):=(
        delay(1);       /* Insert propagation delay */
        busBAR<0>=busD<0> xor busR<0>;
        busBAR<1>=busD<1> xor busR<1> xor (busD<0> and busR<0>);
)

when (busOUT):=(
        delay(1);       /* Insert propagation delay */
        busMAR=busOUT<1:0>;
)
```

bus.isp

```
/*
 *      bus.isp - Retain data on busses
 */

macro
        WORD    :=      32&;

state
        busIN33<33>,    /* BDSIM */
        busOUT33<33>;   /* BDSIM */

port
        busIN<WORD> 'bidirectional:connect,
        busIN.par 'bidirectional:connect,
        busIR<14> 'bidirectional:connect,
        busCU<2> 'bidirectional:connect,
        busT<WORD> 'bidirectional:connect,
        busD<WORD> 'bidirectional:connect,
        busD.par 'bidirectional:connect,
        busBAR<2> 'bidirectional:connect,
        busOUT<WORD> 'bidirectional:connect,
        busOUT.par 'input;      /* BDSIM */

when (busIN
      ((drivers(busIN) gtr 1) and (portstable(busIN) gtr 0))):=(
        disconnect(busIN);
        delay(0);
        busIN=busIN;
        next;
        connect(busIN);
)
```

```
when (busIN.par
      ((drivers(busIN.par) gtr 1) and (portstable(busIN.par) gtr 0))):=(
        disconnect(busIN.par);
        delay(0);
        busIN.par=busIN.par;
        next;
        connect(busIN.par);
)

when (busIN, busIN.par):=(
        busIN33=busIN.par concat busIN;
)

when (busIR
      ((drivers(busIR) gtr 1) and (portstable(busIR) gtr 0))):=(
        disconnect(busIR);
        delay(0);
        busIR=busIR;
        next;
        connect(busIR);
)

when (busCU
      ((drivers(busCU) gtr 1) and (portstable(busCU) gtr 0))):=(
        disconnect(busCU);
        delay(0);
        busCU=busCU;
        next;
        connect(busCU);
)

when (busT
      ((drivers(busT) gtr 1) and (portstable(busT) gtr 0))):=(
        disconnect(busT);
        delay(0);
        busT=busT;
        next;
        connect(busT);
)

when (busD
      ((drivers(busD) gtr 1) and (portstable(busD) gtr 0))):=(
        disconnect(busD);
        delay(0);
        busD=busD;
        next;
        connect(busD);
)

when (busD.par
      ((drivers(busD.par) gtr 1) and (portstable(busD.par) gtr 0))):=(
        disconnect(busD.par);
        delay(0);
        busD.par=busD.par;
        next;
        connect(busD.par);
)

when (busBAR
      ((drivers(busBAR) gtr 1) and (portstable(busBAR) gtr 0))):=(
        disconnect(busBAR);
        delay(0);
        busBAR=busBAR;
        next;
        connect(busBAR);
)

when (busOUT
      ((drivers(busOUT) gtr 1) and (portstable(busOUT) gtr 0))):=(
        disconnect(busOUT);
        delay(0);
        busOUT=busOUT;
        next;
        connect(busOUT);
)

when (busOUT, busOUT.par):=(
        busOUT33=busOUT.par concat busOUT;
)
```

cmp.isp

```
/*
 *      cmp.isp - Compare on-chip outputs with off-chip outputs for errors
 */

macro
        WORD    :=      32&;

state
        error.enb.addr,
        error.enb.data,
        error.rw,
        error.size,
        error.id,
        error.sysmode,
        error.ira,
        error.A,
        error.D,
        error.CMP1,
        error.state,
        state.state<4>,
        save.shutdown,
        save.rb,
        save.repair1,
        enb.AD_,                /* BDSIM */
        error.AD1_,             /* BDSIM */
        error.AD0_,             /* BDSIM */
        error.ADp_,             /* BDSIM */
        error.CMP_,             /* BDSIM */
        error.enb.addr_,        /* BDSIM */
        error.enb.data_,        /* BDSIM */
        error.id_,              /* BDSIM */
        error.ira_,             /* BDSIM */
        error.rw_,              /* BDSIM */
        error.size_,            /* BDSIM */
        error.state_<4>,        /* BDSIM */
        error.sysmode_;         /* BDSIM */

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        out.enb.addr 'input,
        pad.enb.addr 'bidirectional:disconnect,

        out.enb.data 'input,
        pad.enb.data 'bidirectional:disconnect,

        out.rw 'input,
        pad.rw 'bidirectional:disconnect,

        out.size<2> 'input,
        pad.size<2> 'bidirectional:disconnect,

        out.id 'input,
        pad.id 'bidirectional:disconnect,

        out.sysmode 'input,
        pad.sysmode 'bidirectional:disconnect,

        out.ira 'input,
        pad.ira 'bidirectional:disconnect,

        state.busD<4> 'input,
        state.PSW<4> 'input,
        state.RFaddr<4> 'input,
        state.suspend 'input,
        state.op_prev 'input,
        pad.state<4> 'bidirectional:disconnect,

        busOUT<32> 'input,
        busOUT.par 'input,
        pad.AD<33> 'bidirectional:disconnect,

        error.CMP 'bidirectional,

        state.rb 'input,
        state.repair1 'input,
        state.repair2 'input,
        state.shutdown 'input,

        gate.busOUT_padAD2 'input,
          gate.busOUT_padAD4 'input,
          pad.ms 'input,
          ctrl.override_master 'input,
          ctrl.override_slave 'input;

 main:=(
          if (pad.ms eql 0) (
                  connect(pad.enb.addr);
                  connect(pad.enb.data);
                  connect(pad.rw);
                  connect(pad.size);
                  connect(pad.id);
                  connect(pad.sysmode);
                  connect(pad.ira);
                  connect(pad.state);
          );
          enb.AD_=1;
          error.AD1_=1;
          error.AD0_=1;
          error.ADp_=1;
          error.CMP_=1;
          error.enb.addr_=1;
          error.enb.data_=1;
          error.id_=1;
          error.ira_=1;
          error.rw_=1;
          error.size_=1;
          error.state_=1111;
          error.sysmode_=1;
          next;
          terminate;
 )

when  (out.enb.addr, pad.enb.addr):=(
          delay(1);        /* Insert propagation delay */
          error.enb.addr=out.enb.addr neq pad.enb.addr;
          next;
          error.enb.addr_=not error.enb.addr;
)
when  (out.enb.addr):=(
          delay(1);        /* Insert propagation delay */
          pad.enb.addr=out.enb.addr;
)

when  (out.enb.data, pad.enb.data):=(
          delay(1);        /* Insert propagation delay */
          error.enb.data=out.enb.data neq pad.enb.data;
          next;
          error.enb.data_=not error.enb.data;
)
when  (out.enb.data):=(
          delay(1);        /* Insert propagation delay */
          pad.enb.data=out.enb.data;
)

when  (out.rw, pad.rw):=(
          delay(1);        /* Insert propagation delay */
          error.rw=out.rw neq pad.rw;
          next;
          error.rw_=not error.rw;
)
when  (out.rw):=(
          delay(1);        /* Insert propagation delay */
          pad.rw=out.rw;
)

when  (out.size, pad.size):=(
          delay(1);        /* Insert propagation delay */
          error.size=out.size neq pad.size;
          next;
          error.size_=not error.size;
)
when  (out.size):=(
          delay(1);        /* Insert propagation delay */
          pad.size=out.size;
)

when  (out.id, pad.id):=(
          delay(1);        /* Insert propagation delay */
          error.id=out.id neq pad.id;
          next;
          error.id_=not error.id;
)
when  (out.id):=(
          delay(1);        /* Insert propagation delay */
          pad.id=out.id;
)
```

```
when (out.sysmode, pad.sysmode):=(
        delay(1);       /* Insert propagation delay */
        error.sysmode=out.sysmode neq pad.sysmode;
        next;
        error.sysmode_=not error.sysmode;
)
when (out.sysmode):=(
        delay(1);       /* Insert propagation delay */
        pad.sysmode=out.sysmode;
)

when (out.ira, pad.ira):=(
        delay(1);       /* Insert propagation delay */
        error.ira=out.ira neq pad.ira;
        next;
        error.ira_=not error.ira;
)
when (out.ira):=(
        delay(1);       /* Insert propagation delay */
        pad.ira=out.ira;
)

when (phil, state.shutdown, state.rb, state.repair1
      (phil eql 1)):=(
        delay(1);       /* Insert propagation delay */
        save.shutdown=state.shutdown;
        save.rb=state.rb;
        save.repair1=state.repair1;
)

when (state.busD, state.PSW, state.RFaddr, state.suspend, state.op_prev,
      state.repair1, state.repair2, state.rb, save.rb, save.repair1):=(
        delay(1);       /* Insert propagation delay */
        state.state=(state.busD and
                     ((not (save.rb or save.repair1)) sxt 4)) xor
                    (state.PSW and ((not state.rb) sxt 4)) xor
                    (state.RFaddr and ((not save.rb) sxt 4)) xor
                    (state.suspend concat state.op_prev concat
                     state.repair1 concat state.repair2);
)

when (state.state, pad.state):=(
        delay(1);       /* Insert propagation delay */
        error.state=(state.state neq pad.state) and (save.shutdown eql 0);
        error.state_=not (state.state xor pad.state);
)

when (state.state):=(
        delay(1);       /* Insert propagation delay */
        pad.state=state.state;
)

when (busOUT, busOUT.par, pad.AD):=(    /* BDSIM */
        delay(1);       /* Insert propagation delay */
        error.AD1_=busOUT<31:16> eql pad.AD<31:16>;
        error.AD0_=busOUT<15:0> eql pad.AD<15:0>;
        error.ADp_=busOUT.par eql pad.AD<32>;
)
when (phil, busOUT, busOUT.par, pad.AD
      (phil eql 1)):=(
        delay(1);       /* Insert propagation delay */
        error.A=((busOUT.par concat busOUT) neq pad.AD) and gate.busOUT_padAD4;
)
when (phi3, busOUT, busOUT.par, pad.AD
      (phi3 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        error.D=((busOUT.par concat busOUT) neq pad.AD) and gate.busOUT_padAD2;
)

when (phi2, gate.busOUT_padAD2,
      pad.ms, ctrl.override_master, ctrl.override_slave
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        if ((gate.busOUT_padAD2 eql 1) and
            (((pad.ms eql 0) or (ctrl.override_master eql 1)) and
             (ctrl.override_slave eql 0))) (
                connect(pad.AD); enb.AD_=0;
        ) else (
                disconnect(pad.AD); enb.AD_=1;
        );
)
when (phi4, gate.busOUT_padAD4, pad.ms
      (phi4 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        if ((gate.busOUT_padAD4 eql 1) and (pad.ms eql 0)) (
                connect(pad.AD); enb.AD_=0;
        ) else (
                disconnect(pad.AD); enb.AD_=1;
        );
)
when (busOUT, busOUT.par):=(
        delay(1);       /* Insert propagation delay */
        pad.AD=busOUT.par concat busOUT;
)

when (phi1, state.rb, state.shutdown,
      error.enb.addr, error.enb.data, error.rw, error.size,
      error.id, error.ira, error.A, error.D
      (phi1 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        error.CMP1=error.enb.addr or error.enb.data or error.rw or
                   error.size or error.id or error.ira or
                   (error.D and (state.rb eql 0) and (state.shutdown eql 0));
)

when (phi3, state.rb, state.shutdown,
      error.CMP1, error.state, error.sysmode, error.A
      (phi3 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        error.CMP=(error.CMP1 or error.state or error.sysmode or error.A) and
                  (state.rb eql 0) and (state.shutdown eql 0);
)
when (error.CMP):=(
        error.CMP_=not error.CMP;
)
```

cu.isp

```
/*
 *      cu.isp - Control Unit
 */

macro
        WORD    :=      32&,
        RB      :=      4&;

state
        CUdata.in<2>,
        CUdata1[0:RB]<16>,
        CUdata2[0:RB]<16>,
        CUvalid1[0:RB],
        CUvalid2[0:RB],
        cond,
        cond.2<4>,
        ctrl.CU_write,
        cc<4>,
        CUerror,
        busIRerror,
        op_prev;

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        pad.wait 'input,

        busIN<WORD> 'input,
        busIR<14> 'bidirectional:disconnect,
        busCU<2> 'bidirectional:disconnect,
        busIR3<14> 'bidirectional:disconnect,
        busCU3<2> 'bidirectional:disconnect,

        state.cc<4> 'input,
        state.repair2 'input,
        state.repair1 'input,
        state.suspend 'input,
        st.repair2 'input,
        st.repair1 'input,
        state.int 'input,
        state.int3 'input,
        state.IO 'input,
        state.wait 'bidirectional,
        state.rb 'input,
        state.RB<3> 'input,
        state.reset 'input,
        state.shutdown 'input,
        ctrl.badop 'input,

        ALU.carry_car 'input,
        ALU.carry_val 'input,
        ALU.gateOUT 'input,
        ALU.gateOUT_cond 'input,

        ENB 'input,

        PAR.busIN 'input,
        PAR.busA 'input,
        PAR.busB 'input,
        PAR.busOUTA 'input,
        PAR.busOUTB 'input,

        INC.gate 'input,
        INC.gate_cond 'input,

        SHIFT.sxtT 'input,
        gate.busA_busS4 'input,

        IR.write 'input,
        PC.write 'input,
        RF.write 'input,

        ctrl.ALU_c 'output,
        gate.ALU_busD 'output,
        gate.ALU_busOUT 'output,

        gate.padAD_busIN 'output,

        gate.INC_busOUT 'output,

        gate.IR 'input,

        out.enb.addr 'output,
        out.enb.data 'output,

        gate.busA_busS 'output,

        load.PAR_busIN 'output,
        load.PAR_busA 'output,
        load.PAR_busB 'output,
        load.PAR_busOUTA 'output,
        load.PAR_busOUTB 'output,

        load.RFTRAN_busIN 'output,

        ctrl.SHIFT_sxtT 'output,

        ctrl.IMM_write 'output,
        ctrl.IR_write 'output,
        ctrl.MAR_write 'output,
        ctrl.PC_write 'output,
        ctrl.PSW_write 'output,
        ctrl.RB_write 'output,
        ctrl.RF_write 'output,
        ctrl.SDR_write 'output;

format
        op      :=      busIN<31:25>,
        scc     :=      busIN<24>,
        rd      :=      busIN<23:19>,
        imm     :=      busIN<13>,

        cc_z    :=      cc<3>,
        cc_n    :=      cc<2>,
        cc_v    :=      cc<1>,
        cc_c    :=      cc<0>;

main:=(
        CUvalid1[RB]=1;
        CUvalid2[RB]=1;
        next;
        terminate;
)

when (phi4:trail):=(
        state
                cu<32>;

        CUdata1[0]=CUdata.in concat busIR;
        CUvalid1[0]=ctrl.CU_write;
        cu=1;
        next;
        while (cu lss RB) (
                CUdata1[cu]=CUdata2[cu-1];
                CUvalid1[cu]=CUvalid2[cu-1];
```

```
                        cu=cu+1;
                );
                if (CUvalid2[RB-1] eql 1) (
                        CUdata1[RB]=CUdata2[RB-1];
                        CUdata2[RB]=CUdata2[RB-1];
                );
        )

        when (phi2:trail):=(
                state
                        cu<32>;

                cu=0;
                next;
                while (cu lss RB) (
                        CUdata2[cu]=CUdata1[cu];
                        CUvalid2[cu]=CUvalid1[cu];
                        cu=cu+1;
                );
        )

        when (phi3, state.rb, state.RB
              ((phi3 eql 1) and (state.rb eql 1))):=(
                state
                        cu<32>;

                delay(1);       /* Insert propagation delay */
                cu=0;
                next;
                while ((cu lss (state.RB ext 32)) and (cu lss RB)) (
                        CUvalid1[cu]=0;
                        CUvalid2[cu]=0;
                        cu=cu+1;
                );

                while (CUvalid2[cu] eql 0)
                        cu=cu+1;
                busCU3=CUdata2[cu]<15:14>;
                busIR3=CUdata2[cu]<13:0>;
        )

        when (phi3, state.rb
              ((phi3 eql 1) and (state.rb eql 1))):=(
                delay(1);       /* Insert gate opening delay */
                connect(busIR3);
                connect(busCU3);
                wait(phi3:trail, state.rb:trail);
                delay(1);       /* Insert gate closing delay */
                disconnect(busIR3);
                disconnect(busCU3);
        )

        when (phi2, pad.wait, state.rb
              (phi2 eql 1)):=(
                delay(1);       /* Insert propagation delay */
                state.wait=pad.wait and (not state.rb);
        )

        when (phi3, IR.write, state.int, state.wait, busIN
              ((phi3 eql 1) and
               (((IR.write eql 1) and (state.wait eql 0)) or
                (state.int eql 1) or (state.rb eql 1)))):=(
                delay(1);       /* Insert propagation delay */
                busIR=op concat scc concat rd concat imm;
        )

        when (phi3, gate.IR, state.int, state.wait, state.rb
              ((phi3 eql 1) and
               (((gate.IR eql 1) and (state.wait eql 0) and (state.rb eql 0)) or
                ((state.int eql 1) and (state.rb eql 0))))):=(
                delay(1);       /* Insert propagation delay */
                connect(busIR);
                wait(phi3:trail, state.rb:lead);
                delay(1);       /* Insert propagation delay */
                disconnect(busIR);
        )

        when (phi4, busIR, st.repair2, st.repair1
              (phi4 eql 1)):=(
                if ((st.repair1 eql 1) or (st.repair2 eql 1))
                        op_prev=busCU<0>
                else
                        op_prev=busIR<12>;
        )

        when (state.suspend, op_prev):=(
                busCU=state.suspend concat op_prev;
        )
```

```
when (phi3, state.suspend
      (phi3 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        CUdata.in=state.suspend concat op_prev;
)

when (state.rb, st.repair1, st.repair2
      ((state.rb eql 0) and (st.repair1 eql 0) and (st.repair2 eql 0))):=(
        delay(1);        /* Insert propagation delay */
        connect(busCU);
        wait(state.rb:lead);
        delay(1);        /* Insert propagation delay */
        disconnect(busCU);
)

when (phi2, busIR
      ((phi2 eql 1))):=(
        cond.2=busIR<4:1>;
)

when (phi2, state.cc
      (phi2 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        cc=state.cc;
)

when (phi3, cond.2, cc
      (phi3 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        case (cond.2)
        0:      /* nev */
                cond=0
        1:      /* gt */
                cond=not ((cc_n xor cc_v) or cc_z)
        2:      /* le */
                cond=(cc_n xor cc_v) or cc_z
        3:      /* ge */
                cond=not (cc_n xor cc_v)
        4:      /* lt */
                cond=cc_n xor cc_v
        5:      /* hi */
                cond=not ((not cc_c) or cc_z)
        6:      /* los */
                cond=(not cc_c) or cc_z
        7:      /* lo || nc */
                cond=not cc_c
        8:      /* his || c */
                cond=cc_c
        9:      /* pl */
                cond=not cc_n
        10:     /* mi */
                cond=cc_n
        11:     /* ne */
                cond=not cc_z
        12:     /* eq */
                cond=cc_z
        13:     /* nv */
                cond=not cc_v
        14:     /* v */
                cond=cc_v
        15:     /* alw */
                cond=1
        esac;
)

when ALU.post (ALU.carry_car, ALU.carry_val, state.cc, state.int, state.rb,
               state.repair2, ALU.gateOUT, ALU.gateOUT_cond, cond):=(
        delay(1);        /* Insert propagation delay */
        ctrl.ALU_c=(ALU.carry_car and cc_c) or
                   ((not ALU.carry_car) and ALU.carry_val);

        gate.ALU_busOUT=(ALU.gateOUT or (ALU.gateOUT_cond and cond)) and
                        (state.int eql 0) and (state.rb eql 0) and
                        (state.repair2 eql 0);
)

when busIN.post (state.int, state.wait, state.rb):=(
        delay(1);        /* Insert propagation delay */
        gate.padAD_busIN=(state.int eql 0) and (state.wait eql 0) and
                         (state.rb eql 0);
)

/*
when MEMORY.post1 (phi1, ENB):=(
*/
when MEMORY.post1 (phi4:trail):=(
        delay(1);        /* Insert propagation delay */
        out.enb.addr=ENB and (state.wait eql 0) and (st.repair1 eql 0)
```

```
/*
                        and (st.repair2 eql 0) and phi1;
*/
                        and (st.repair2 eql 0);
)
when (phi1:trail):=(
        out.enb.addr=0;
)

/*
when MEMORY.post3 (phi3):=(
*/
when MEMORY.post3 (phi2:trail):=(
        delay(1);       /* Insert propagation delay */
        out.enb.data=(state.wait eql 0) and (state.repair1 eql 0) and
/*
                        (state.repair2 eql 0) and phi3;
*/
                        (state.repair2 eql 0);
)
when (phi3:trail):=(
        out.enb.data=0;
)

when PAR.post (PAR.busIN, PAR.busA, PAR.busB,
             PAR.busOUTA, PAR.busOUTB, state.rb, ctrl.badop):=(
        delay(1);       /* Insert propagation delay */
        load.PAR_busIN=PAR.busIN and
                        (state.shutdown eql 0) and (ctrl.badop eql 0);
        load.PAR_busA=PAR.busA and (state.rb eql 0) and
                      (state.shutdown eql 0) and (ctrl.badop eql 0);
        load.PAR_busB=PAR.busB and (state.rb eql 0) and
                      (state.shutdown eql 0) and (ctrl.badop eql 0);
        load.PAR_busOUTA=PAR.busOUTA and (state.rb eql 0) and
                      (ctrl.badop eql 0);
        load.PAR_busOUTB=PAR.busOUTB and (state.rb eql 0) and
                      (ctrl.badop eql 0);
)

when PC.post (INC.gate, INC.gate_cond, cond, state.int, state.rb):=(
        delay(1);       /* Insert propagation delay */
        gate.INC_busOUT=(INC.gate or (INC.gate_cond and (not cond))) and
                        (state.int eql 0) and (state.rb eql 0);
)

when RFTRAN.post (IR.write, state.wait, state.int):=(
        delay(1);       /* Insert propagation delay */
        load.RFTRAN_busIN=(IR.write and (state.wait eql 0)) or state.rb;
)

when SHIFT.post (phi1, SHIFT.sxtT, gate.busA_busS4
                (phi1 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        ctrl.SHIFT_sxtT=SHIFT.sxtT;
        gate.busA_busS=gate.busA_busS4;
)

when WRITE.post (IR.write, PC.write, RF.write, state.int, state.int3,
                state.IO, state.wait, state.rb, state.repair1, state.repair2,
                state.reset, state.shutdown):=(
        ctrl.CU_write=((not state.wait) and (not state.rb) and
                       (not state.repair1) and (not state.repair2) and
                       (not state.int3)) or
                      (state.int and (state.IO or (not state.int3))) or
                      state.reset or state.shutdown;
        ctrl.IMM_write=(IR.write and (not state.wait)) or state.rb;
        ctrl.IR_write=(IR.write and (not state.wait) and (not state.rb) and
                       (not state.repair1) and (not state.repair2) and
                       (not state.int3)) or
                      (state.int and (state.IO or (not state.int3))) or
                      state.reset or state.shutdown;
        ctrl.MAR_write=((not state.wait) and (not state.rb) and
                        (not state.repair1) and (not state.repair2) and
                        (not state.int3)) or
                       (state.int and (state.IO or (not state.int3))) or
                       state.reset or state.shutdown;
        ctrl.PC_write=(PC.write and (not state.wait) and (not state.rb) and
                       (not state.repair1) and (not state.repair2) and
                       (not state.int3)) or
                      (state.int and (state.IO or (not state.int3))) or
                      state.reset or state.shutdown;
        ctrl.PSW_write=((not state.wait) and (not state.rb) and
                        (not state.repair1) and (not state.repair2) and
                        (not state.int3)) or
                       (state.int and (state.IO or (not state.int3))) or
                       state.reset or state.shutdown;
        ctrl.RB_write=((not state.wait) and (not state.rb) and
                       (not state.repair1) and (not state.repair2) and
```

```
                        (not state.int3)) or
                        (state.int and (state.IO or (not state.int3))) or
                        state.reset or state.shutdown;
        ctrl.RF_write=RF.write and (not state.int) and
                        (not state.wait) and (not state.rb) and (not state.int3);
        ctrl.SDR_write=((not state.wait) and (not state.rb) and
                         (not state.repair1) and (not state.repair2) and
                         (not state.int3)) or
                        (state.int and (state.IO or (not state.int3))) or
                        state.reset or state.shutdown;
)
```

dimm.isp

```
/*
 *      dimm.isp - Data Immediate Register and Sign Extender
 */

macro
        WORD    :=      32&;

port
        phi3 'input,

        busIN<WORD> 'input,
        busT32<WORD> 'output:disconnect,
        busT24<WORD> 'output:disconnect,
        busT16<WORD> 'output:disconnect,
        busT8<WORD> 'output:disconnect,

        out.size<2> 'input,
        busBAR<2> 'input,
        ctrl.DIMM_sxt 'input,
        gate.DIMM_busT 'input,
        gate.busIN_busT 'input;

when (phi3, gate.DIMM_busT, gate.busIN_busT, out.size, busBAR):=(
        if ((phi3 eql 1) and
            (((gate.DIMM_busT eql 1) and
              ((out.size<1> eql 0b1) or
               (busBAR eql 0b11) or
               ((out.size<0> concat busBAR<1>) eql 0b11))) or
             (gate.busIN_busT eql 1))) (
                delay(1);       /* Insert gate opening delay */
                connect(busT32);
        ) else (
                delay(1);       /* Insert gate closing delay */
                disconnect(busT32);
        );
)

when (phi3, gate.DIMM_busT, out.size, busBAR):=(
        if ((phi3 eql 1) and (gate.DIMM_busT eql 1) and
            ((out.size concat busBAR) eql 0b0010)) (
                delay(1);       /* Insert gate opening delay */
                connect(busT24);
        ) else (
                delay(1);       /* Insert gate closing delay */
                disconnect(busT24);
        );
)

when (phi3, gate.DIMM_busT, out.size, busBAR):=(
        if ((phi3 eql 1) and (gate.DIMM_busT eql 1) and
            (((out.size concat busBAR<1>) eql 0b010) or
             ((out.size<1> concat busBAR) eql 0b001))) (
                delay(1);       /* Insert gate opening delay */
                connect(busT16);
        ) else (
                delay(1);       /* Insert gate closing delay */
                disconnect(busT16);
        );
)

when (phi3, gate.DIMM_busT, out.size, busBAR):=(
        if ((phi3 eql 1) and (gate.DIMM_busT eql 1) and
            ((out.size concat busBAR) eql 0b0000)) (
                delay(1);       /* Insert gate opening delay */
                connect(busT8);
        ) else (
                delay(1);       /* Insert gate closing delay */
                disconnect(busT8);
        );
}

when (busIN, ctrl.DIMM_sxt):=(
        busT32:=busIN<31:0>;
        if (ctrl.DIMM_sxt eql 1) {
                busT24=busIN<23:0> sxt 32;
                busT16=busIN<15:0> sxt 32;
                busT8=busIN<7:0> sxt 32;
        } else {
                busT24=busIN<23:0> ext 32;
                busT16=busIN<15:0> ext 32;
                busT8=busIN<7:0> ext 32;
        );
}
```

fsm.isp

```
/*
 *      fsm.isp - Finite State Machine
 */

macro
        WORD            :=      32&,
        st_normal       :=      (st eql 0b000)&,        /* normal i-decode */
        st_suspend      :=      (st eql 0b001)&,        /* suspend i-decode */
        st_repair1      :=      (st<1> eql 1)&,
        st_repair2      :=      (st<2> eql 1)&,

        opx00000x       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<3> eql 0) and (op<2> eql 0) and
                                 (op<1> eql 0))&,
        opxx000x0       :=      ((op<4> eql 0) and (op<3> eql 0) and
                                 (op<2> eql 0) and (op<0> eql 0))&,
        opx000011       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<3> eql 0) and (op<2> eql 0) and
                                 (op<1> eql 1) and (op<0> eql 1))&,
        opxx0010x       :=      ((op<4> eql 0) and (op<3> eql 0) and
                                 (op<2> eql 1) and (op<1> eql 0))&,
        opx000110       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<3> eql 0) and (op<2> eql 1) and
                                 (op<1> eql 1) and (op<0> eql 0))&,
        opx0001x1       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<3> eql 0) and (op<2> eql 1) and
                                 (op<0> eql 1))&,
        opx00x000       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<2> eql 0) and (op<1> eql 0) and
                                 (op<0> eql 0))&,
        opx001001       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<3> eql 1) and (op<2> eql 0) and
                                 (op<1> eql 0) and (op<0> eql 1))&,
        opx001010       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<3> eql 1) and (op<2> eql 0) and
                                 (op<1> eql 1) and (op<0> eql 0))&,
        opx001011       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<3> eql 1) and (op<2> eql 0) and
                                 (op<1> eql 1) and (op<0> eql 1))&,
        opx001100       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<3> eql 1) and (op<2> eql 1) and
                                 (op<1> eql 0) and (op<0> eql 0))&,
        opx00x101       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                 (op<2> eql 1) and (op<1> eql 0) and
                                 (op<0> eql 1))&,
        opx0x1110       :=      ((op<5> eql 0) and (op<3> eql 1) and
                                 (op<2> eql 1) and (op<1> eql 1) and
                                 (op<0> eql 0))&,
        opx0x1111       :=      ((op<5> eql 0) and (op<3> eql 1) and
                                 (op<2> eql 1) and (op<1> eql 1) and
                                 (op<0> eql 1))&,
        opx01000x       :=      ((op<5> eql 0) and (op<4> eql 1) and
                                 (op<3> eql 0) and (op<2> eql 0) and
                                 (op<1> eql 0))&,
        opxx100x0       :=      ((op<4> eql 1) and (op<3> eql 0) and
                                 (op<2> eql 0) and (op<0> eql 0))&,
        opx010011       :=      ((op<5> eql 0) and (op<4> eql 1) and
                                 (op<3> eql 0) and (op<2> eql 0) and
                                 (op<1> eql 1) and (op<0> eql 1))&,
        opxx10x00       :=      ((op<4> eql 1) and (op<3> eql 0) and
                                 (op<1> eql 0) and (op<0> eql 0))&,
        opxxx0101       :=      ((op<3> eql 0) and (op<2> eql 1) and
                                (op<1> eql 0) and (op<0> eql 1))&,
opx01x110           :=         ((op<5> eql 0) and (op<4> eql 1) and
                                (op<2> eql 1) and (op<1> eql 1) and
                                (op<0> eql 0))&,
opx01x111           :=         ((op<5> eql 0) and (op<4> eql 1) and
                                (op<2> eql 1) and (op<1> eql 1) and
                                (op<0> eql 1))&,
opxx1x000           :=         ((op<4> eql 1) and (op<2> eql 0) and
                                (op<1> eql 0) and (op<0> eql 0))&,
opxx11001           :=         ((op<4> eql 1) and (op<3> eql 1) and
                                (op<2> eql 0) and (op<1> eql 0) and
                                (op<0> eql 1))&,
opx011010           :=         ((op<5> eql 0) and (op<4> eql 1) and
                                (op<3> eql 1) and (op<2> eql 0) and
                                (op<1> eql 1) and (op<0> eql 0))&,
opx011011           :=         ((op<5> eql 0) and (op<4> eql 1) and
                                (op<3> eql 1) and (op<2> eql 0) and
                                (op<1> eql 1) and (op<0> eql 1))&,
opxx11100           :=         ((op<4> eql 1) and (op<3> eql 1) and
                                (op<2> eql 1) and (op<1> eql 0) and
                                (op<0> eql 0))&,
opx0111x1           :=         ((op<5> eql 0) and (op<4> eql 1) and
                                (op<3> eql 1) and (op<2> eql 1) and
                                (op<0> eql 1))&,
opx100x0x           :=         ((op<5> eql 1) and (op<4> eql 0) and
                                (op<3> eql 0) and (op<1> eql 0))&,
opx10001x           :=         ((op<5> eql 1) and (op<4> eql 0) and
                                (op<3> eql 0) and (op<2> eql 0) and
                                (op<1> eql 1))&,
opx100x10           :=         ((op<5> eql 1) and (op<4> eql 0) and
                                (op<3> eql 0) and (op<1> eql 1) and
                                (op<0> eql 0))&,
opx1001x1           :=         ((op<5> eql 1) and (op<4> eql 0) and
                                (op<3> eql 0) and (op<2> eql 1) and
                                (op<0> eql 1))&,
opx1xx000           :=         ((op<5> eql 1) and (op<2> eql 0) and
                                (op<1> eql 0) and (op<0> eql 0))&,
opx1x1001           :=         ((op<5> eql 1) and (op<3> eql 1) and
                                (op<2> eql 0) and (op<1> eql 0) and
                                (op<0> eql 1))&,
opx10x010           :=         ((op<5> eql 1) and (op<4> eql 0) and
                                (op<2> eql 0) and (op<1> eql 1) and
                                (op<0> eql 0))&,
opx101011           :=         ((op<5> eql 1) and (op<4> eql 0) and
                                (op<3> eql 1) and (op<2> eql 0) and
                                (op<1> eql 1) and (op<0> eql 1))&,
opx1xx100           :=         ((op<5> eql 1) and (op<2> eql 1) and
                                (op<1> eql 0) and (op<0> eql 0))&,
opx1xx101           :=         ((op<5> eql 1) and (op<2> eql 1) and
                                (op<1> eql 0) and (op<0> eql 1))&,
opx101110           :=         ((op<5> eql 1) and (op<4> eql 0) and
                                (op<3> eql 1) and (op<2> eql 1) and
                                (op<1> eql 1) and (op<0> eql 0))&,
opx101111           :=         ((op<5> eql 1) and (op<4> eql 0) and
                                (op<3> eql 1) and (op<2> eql 1) and
                                (op<1> eql 1) and (op<0> eql 1))&,
opx11xx0x           :=         ((op<5> eql 1) and (op<4> eql 1) and
                                (op<1> eql 0))&,
opx11001x           :=         ((op<5> eql 1) and (op<4> eql 1) and
                                (op<3> eql 0) and (op<2> eql 0) and
                                (op<1> eql 1))&,
opx110xx0           :=         ((op<5> eql 1) and (op<4> eql 1) and
                                (op<3> eql 0) and (op<0> eql 0))&,
opx110xx1           :=         ((op<5> eql 1) and (op<4> eql 1) and
                                (op<3> eql 0) and (op<0> eql 1))&,
opx1110x0           :=         ((op<5> eql 1) and (op<4> eql 1) and
                                (op<3> eql 1) and (op<2> eql 0) and
                                (op<0> eql 0))&,
opx1110x1           :=         ((op<5> eql 1) and (op<4> eql 1) and
                                (op<3> eql 1) and (op<2> eql 0) and
                                (op<0> eql 1))&,
opx1111x0           :=         ((op<5> eql 1) and (op<4> eql 1) and
                                (op<3> eql 1) and (op<2> eql 1) and
                                (op<0> eql 0))&,
opx1111x1           :=         ((op<5> eql 1) and (op<4> eql 1) and
                                (op<3> eql 1) and (op<2> eql 1) and
                                (op<0> eql 1))&,
op1xxxxxx           :=         (op<6> eql 1)&,
opxxx0000           :=         ((op<3> eql 0) and (op<2> eql 0) and
                                (op<1> eql 0) and (op<0> eql 0))&,
opxx00101           :=         ((op<4> eql 0) and (op<3> eql 0) and
                                (op<2> eql 1) and (op<1> eql 0) and
                                (op<0> eql 1))&,
opx01111x           :=         ((op<5> eql 0) and (op<4> eql 1) and
                                (op<3> eql 1) and (op<2> eql 1) and
                                (op<1> eql 1))&,
opx1x00x0           :=         ((op<5> eql 1) and (op<3> eql 0) and
                                (op<2> eql 0) and (op<0> eql 0))&,
            opx1x010x       :=      ((op<5> eql 1) and (op<3> eql 0) and
                                     (op<2> eql 1) and (op<1> eql 0))&,
            opx111x0x       :=      ((op<5> eql 1) and (op<4> eql 1) and
                                     (op<3> eql 1) and (op<1> eql 0))&,
            opx001xxx       :=      ((op<5> eql 0) and (op<4> eql 0) and
                                     (op<3> eql 1))&,
            opx1x0xxx       :=      ((op<5> eql 1) and (op<3> eql 0))&,
            opx1x10xx       :=      ((op<5> eql 1) and (op<3> eql 1) and
                                     (op<2> eql 0))&,
            opx1xxxxx       :=      (op<5> eql 1)&,
            opx11xxxx       :=      ((op<5> eql 1) and (op<4> eql 1))&,

            op_calli        :=      opx00000x&,
            op_getpsw       :=      opxx000x0&,
            op_getlpc       :=      opx000011&,
            op_putpsw       :=      opxx0010x&,
            op_clrrbm       :=      opx000110&,
            op_clrrbs       :=      opx0001x1&,
            op_callx        :=      opx00x000&,
            op_callr        :=      opx001001&,
            op_jmprbm       :=      opx001010&,
            op_jmprbs       :=      opx001011&,
            op_jmpx         :=      opx001100&,
            op_jmpr         :=      opx00x101&,
            op_ret          :=      opx0x1110&,
            op_reti         :=      opx0x1111&,
            op_sll          :=      opx01000x&,
            op_sra          :=      opxx100x0&,
            op_srl          :=      opx010011&,
            op_ldhi         :=      opxx10x00&,
            op_and          :=      opxxx0101&,
            op_or           :=      opx01x110&,
            op_xor          :=      opx01x111&,
            op_add          :=      opxx1x000&,
            op_addc         :=      opxx11001&,
            op_addbpm       :=      opx011010&,
            op_addbps       :=      opx011011&,
            op_sub          :=      opxx11100&,
            op_subc         :=      opx0111x1&,
            op_ldrbpm       :=      opx100x0x&,
            op_ldrbps       :=      opx10001x&,
            op_ldxw         :=      opx100x10&,
            op_ldrw         :=      opx1001x1&,
            op_ldxhu        :=      opx1xx000&,
            op_ldrhu        :=      opx1x1001&,
            op_ldxhs        :=      opx10x010&,
            op_ldrhs        :=      opx101011&,
            op_ldxbu        :=      opx1xx100&,
            op_ldrbu        :=      opx1xx101&,
            op_ldxbs        :=      opx101110&,
            op_ldrbs        :=      opx101111&,
            op_strbdm       :=      opx11xx0x&,
            op_strbds       :=      opx11001x&,
            op_stxw         :=      opx110xx0&,
            op_strw         :=      opx110xx1&,
            op_stxh         :=      opx1110x0&,
            op_strh         :=      opx1110x1&,
            op_stxb         :=      opx1111x0&,
            op_strb         :=      opx1111x1&,
            op_alu          :=      (op_and or op_or or op_xor or
                                     op_add or op_addc or
                                     op_addbpm or op_addbps or
                                     op_sub or op_subc)&,
            op_clrrb        :=      (op_clrrbm or op_clrrbs)&,
            op_jmprb        :=      (op_jmprbm or op_jmprbs)&,
            op_ldrbp        :=      (op_ldrbpm or op_ldrbps)&,
            op_ldx          :=      (op_ldxw or op_ldxhu or op_ldxhs or
                                     op_ldxbu or op_ldxbs)&,
            op_ldr          :=      (op_ldrw or op_ldrhu or op_ldrhs or
                                     op_ldrbu or op_ldrbs or op_ldrbp)&,
            op_ld           :=      (op_ldx or op_ldr)&,
            op_strbd        :=      (op_strbdm or op_strbds)&,
            op_stx          :=      (op_stxw or op_stxh or op_stxb)&,
            op_str          :=      (op_strw or op_strh or op_strb or op_strbd)&,
            op_st           :=      (op_stx or op_str)&,
            op_badop        :=      (op1xxxxxx or opxxx0000 or opxx00101 or
                                     opx01111x or opx1x00x0 or opx1x010x or
                                     opx111x0x)&,
            mem_op_branch   :=      opx001xxx&,
            mem_op_lsw      :=      opx1x0xxx&,
            mem_op_lsh      :=      opx1x10xx&,
            mem_op_ls       :=      opx1xxxxx&,
            mem_op_st       :=      opx11xxxx&;

state
            st_prev<3>,
            st<3>,
            op<7>,
        op_prev,
        imm,
        scc,
        int,
        ws,
        rb,
        repairA,
        repairA_,
        repairB,
        repairB_,
        ctrl.override_master_,  /* BDSIM */
        ctrl.override_slave_,   /* BDSIM */
        state.op_prev_;         /* BDSIM */

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busIR<14> 'input,
        busCU<2> 'input,
        state.int 'input,
        state.wait 'input,
        state.rb 'input,
        ctrl.repairA 'input,
        ctrl.repairA_ 'input,
        ctrl.repairB 'input,
        ctrl.repairB_ 'input,
        ctrl.norepair 'input,
        state.rb_bit 'input,
        pad.ms 'input,

        state.repair1 'bidirectional,
        state.repair2 'bidirectional,
        state.suspend 'bidirectional,
        state.op_prev 'output,
        st.repair2 'output,
        st.repair1 'output,

        ctrl.badop 'output,
        ctrl.privop 'output,
        ctrl.over_under 'output,

        ctrl.ALU_op<5> 'output,
        ALU.carry_car 'output,
        ALU.carry_val 'output,
        gate.ALU_busD 'output,
        ALU.gateOUT 'output,
        ALU.gateOUT_cond 'output,

        load.BAR_busDR 'output,

        ctrl.DIMM_sxt 'output,
        gate.DIMM_busT 'output,

        gate.busIN_busT 'output,
        gate.busA_busD2 'output,
        gate.busA_busD4 'output,
        gate.busD_busOUT 'output,

        gate.IMM13_busT 'output,
        gate.IMM19_busT 'output,

        IR.write 'output,
        gate.IR 'output,

        gate.MAR_busOUT2 'output,
        gate.MAR_busOUT4 'output,

        ctrl.override_master 'bidirectional,
        ctrl.override_slave 'bidirectional,
        gate.busOUT_padAD2 'output,
        gate.busOUT_padAD4 'output,
        out.id 'output,
        out.rw 'output,
        ENB 'output,
        out.size<2> 'output,

        PAR.busIN 'output,
        PAR.busA 'output,
        PAR.busB 'output,
        PAR.busOUTA 'output,
        PAR.busOUTB 'output,
        ctrl.RFpar_invert 'output,
        ctrl.busINpar_invert 'output,

        PC.write 'output,
         ctrl.PC_select 'output,
         gate.PC_busD2 'output,
         INC.gate 'output,
         INC.gate_cond 'output,
         gate.PC_busD4 'output,

         ctrl.CWP_inc<2> 'output,
         load.PSW_busCC 'output,
         ctrl.PSW_reti 'output,
         load.PSW_busD 'output,
         gate.PSW_busD 'output,

         ctrl.clrrb 'output,

         RF.write 'output,
         load.RFaddr_RA 'output,
         load.RFaddr_RB 'output,
         load.RFaddr_RD 'output,

         load.SHam_0 'output,
         load.SHam_BAR 'output,
         load.SHam_IMM 'output,
         load.SHam_busB 'output,

         gate.SDR_busOUT 'output,

         load.SHIFT_IMM 'output,
         load.SHIFT_busT 'output,
         load.SHIFT_busR 'output,
         load.SHIFT_busL 'output,
         ctrl.SHIFT_sxtS 'output,
         SHIFT.sxtT 'output,
         gate.SHIFT_busR 'output,
         gate.SHIFT_busL 'output,

         gate.busA_busS4 'output;

when  (busIR):=(
         delay(1);         /* Insert propagation delay */
         op=busIR<13:7>;
         scc=busIR<6>;
         imm=busIR<0>;
)

when  (state.repair2:change, state.repair1:change, busCU):=(
         delay(1);         /* Insert propagation delay */
         st_prev=state.repair2 concat state.repair1 concat busCU<1>;
         op_prev=busCU<0>;
)

when  (phi4, op_prev
        (phi4 eql 1)):=(
         delay(1);         /* Insert propagation delay */
         state.op_prev=op_prev;
         state.op_prev_=not op_prev;
)

when  (state.int:change):=(
         delay(1);         /* Insert propagation delay */
         int=state.int;
)

when  (state.wait:change):=(
         delay(1);         /* Insert propagation delay */
         ws=state.wait;
)

when  (state.rb:change):=(
         delay(1);         /* Insert propagation delay */
         rb=state.rb;
)

when  (phi3, state.rb, state.repair2, state.repair1,
       ctrl.repairA, ctrl.repairA_, ctrl.repairB, ctrl.repairB_
        ((phi3 eql 1) and
         ((state.rb eql 1) or
          ((state.repair1 eql 0) and (state.repair2 eql 0))))):=(
         repairA=ctrl.repairA;
         repairA_=ctrl.repairA_;
         repairB=ctrl.repairB;
         repairB_=ctrl.repairB_;
)

when STATE.3 (phi3, st_prev, int, rb, repairA, repairB, ws, op_prev,
              ctrl.norepair
              (phi3 eql 1)):=(
        if (int eql 1)
              st=0b000                                 /* normal */
        else if ((rb eql 1) and (ctrl.norepair eql 0) and
                 ((repairA eql 1) or (repairB eql 1)))
                st=0b01 concat st_prev<0>                   /* repair1 */
        else if (st_prev<1> eql 1)                          /* repair1 */
                st=0b10 concat st_prev<0>                   /* repair2 */
        else if (ws eql 1)
                st=st_prev
        else if (st_prev<0> eql 0) (                        /* normal */
                if (op_prev eql 1)
                        st=0b001                            /* suspend */
                else
                        st=0b000;                           /* normal */
        ) else if (st_prev<0> eql 1)                        /* suspend */
                st=0b000;                                   /* normal */
)

when (st):=(
        st.repair2=st<2>;
        st.repair1=st<1>;
)

when STATE.2 (phi2, st
             (phi2 eql 1)):=(
        state.repair2=st<2>;
        state.repair1=st<1>;
        state.suspend=st<0>;
)

when (phi1, op
      (phi1 eql 1)):=(
        ctrl.badop=op_badop;

        ctrl.privop=op_calli or op_getlpc or op_putpsw or op_reti or
                    op_clrrb or op_addbpm or op_addbps or op_jmprb or
                    op_ldrbp or op_strbd;

        ctrl.over_under=op_calli or op_callx or op_callr;
)

when ALU.1 (phi1, st, op, state.rb_bit
            (phi1 eql 1)):=(
        ctrl.ALU_op<4>=op_sub or op_subc;
        ctrl.ALU_op<3>=op_putpsw or op_callx or op_callr or
                       op_jmpx or op_jmpr or op_ret or op_reti or
                       op_xor or op_add or op_addc or op_sub or op_subc or
                       op_addbpm or op_addbps or op_jmprbm or op_jmprbs or
                       ((op_ld or op_st) and st_normal);
        ctrl.ALU_op<2>=op_and;
        ctrl.ALU_op<1>=op_or;
        ctrl.ALU_op<0>=op_putpsw or op_callx or op_callr or
                       op_jmpx or op_jmpr or op_ret or op_reti or
                       op_add or op_addc or op_sub or op_subc or
                       op_jmprbm or op_jmprbs or op_addbpm or op_addbps or
                       ((op_ld or op_st) and st_normal);
        ALU.carry_car=op_addc or op_subc;
        ALU.carry_val=op_sub;
        gate.ALU_busD=(op_putpsw or op_alu) and
                      (not st_repair1) and (not st_repair2);
        ALU.gateOUT=(op_callx or op_callr or op_ret or op_reti or
                    (op_jmprb and (state.rb_bit eql 1)) or
                    ((op_ld or op_st) and st_normal)) and
                   (not st_repair2);
        ALU.gateOUT_cond=(op_jmpx or op_jmpr) and (not st_repair2);
)

when BAR.1 (phi1, st
            (phi1 eql 1)):=(
        load.BAR_busDR=not ((op_ld or op_st) and st_suspend);
)

when DIMM.1 (phi1, st, op
             (phi1 eql 1)):=(
        ctrl.DIMM_sxt=op_ldxhs or op_ldrhs or op_ldxbs or op_ldrbs;
        gate.DIMM_busT=op_ld and (not st_repair2);
)

when GATE.4 (phi4, st, op, repairA
           (phi4 eql 1)):=(
        gate.busA_busS4=op_sll or op_sra or op_srl or
                        ((repairA eql 1) and st_repair1);
)

when GATE.1 (phi1, st, op, pad.ms, state.rb_bit
             (phi1 eql 1)):=(
        /* gate shifts for parity checking */
        gate.busA_busD2=((op_putpsw or op_callx or
                          op_jmpx or op_ret or op_reti or
                          op_sll or op_sra or op_srl or op_alu) and
                        (not st_repair2)) or
                        ((op_ldx or op_stx) and st_normal);
        gate.busA_busD4=(op_jmprbm and (pad.ms eql 0)) or
                        (op_jmprbs and (pad.ms eql 1)) or
                        (op_jmprb and (state.rb_bit eql 1));
        gate.busIN_busT=st_repair2;      /* Same as gate.MAR_busOUT4 */
        gate.busD_busOUT=st_repair2;     /* Same as gate.MAR_busOUT4 */
 )

 when IMM.4 (phi4, op
          (phi4 eql 1)):=(
        gate.IMM13_busT=op_putpsw or op_callx or op_jmpx or
                        op_ret or op_reti or op_alu or op_ldx or op_stx;
        gate.IMM19_busT=op_callr or op_jmpr or op_ldhi or op_ldr or op_str;
 )

 when IR.1 (phi1, st
          (phi1 eql 1)):=(
        IR.write=st_normal;
        gate.IR=(st_normal and (not (op_ld or op_st))) or st_suspend;
 )

 when MAR.1 (phi1, op, st, pad.ms, state.rb_bit
          (phi1 eql 1)):=(
        gate.MAR_busOUT2=((op_strbdm and (pad.ms eql 0)) or
                         (op_strbds and (pad.ms eql 1))) and
                        (state.rb_bit eql 0) and st_suspend;
        gate.MAR_busOUT4=st_repair2;
 )

 when MEMORY.4 (phi4, st, op, repairA, repairA_, repairB, repairB_
            (phi4 eql 1)):=(
        gate.busOUT_padAD2=(op_st and st_suspend) or
                          ((((repairA eql 1) and (repairA_ eql 0)) or
                            ((repairB eql 1) and (repairB_ eql 0) and
                             (repairA eql 0))) and
                           st_repair2);
        ctrl.override_master=(((repairA eql 1) and (repairA_ eql 0)) or
                             ((repairB eql 1) and (repairB_ eql 0) and
                              (repairA eql 0))) and
                            st_repair2;
        ctrl.override_slave=(((repairA eql 1) and (repairA_ eql 1)) or
                            ((repairB eql 1) and (repairB_ eql 1) and
                             (repairA eql 0))) and
                           st_repair2;
        ENB=st_normal or st_suspend;
        /* fast */
        gate.busOUT_padAD4=(st_normal or st_suspend) and (ws eql 0);
        /* fast */
        out.ld=st_suspend and mem_op_ls;
        out.rw=mem_op_st and st_suspend;
        out.size<1>=mem_op_branch or mem_op_lsw;
        out.size<0>=mem_op_lsh;
 )

 when (ctrl.override_master, ctrl.override_slave):=(
        ctrl.override_master_=not ctrl.override_master;
        ctrl.override_slave_=not ctrl.override_slave;
 )

 when PAR.1 (phi1, repairA_, repairB_, repairA, st, op, state.rb_bit
         (phi1 eql 1)):=(
        PAR.busIN=not st_repair1;
        /* gate.busA_busD */
        PAR.busA=((op_putpsw or op_callx or op_jmpx or op_ret or
                  op_reti or op_sll or op_sra or op_srl or op_alu) and
                 (not (st_repair1 and repairA_)) and (not st_repair2)) or
                ((op_ldx or op_stx) and st_normal);
        PAR.busB=((imm eql 0) and
                 (op_putpsw or op_callx or op_jmpx or
                  op_ret or op_reti or op_sll or
                  op_sra or op_srl or op_alu or
                  (op_ldx and st_normal)) and
                 (not (st_repair1 and repairB_)) and (not st_repair2)) or
                (op_st and st_normal);
        PAR.busOUTA=st_repair2 and repairA and (not repairA_);
        PAR.busOUTB=st_repair2 and repairB and (not repairB_) and (not repairA);
        ctrl.busINpar_Invert=st_suspend and (state.rb_bit eql 0) and
                            ((op_ldrbpm and (pad.ms eql 0)) or
                             (op_ldrbps and (pad.ms eql 1)));
)

when PAR.2 (phi2, repairA_, repairB_, repairA, pad.ms
         (phi2 eql 1)):=(
        ctrl.RFpar_invert=(op_addbpm and (pad.ms eql 0)) or
                         (op_addbps and (pad.ms eql 1));
)
```

```
when PC.4 (phi4, st, op
          (phi4 eql 1)):=(
        ctrl.PC_select=op_calli or op_getlpc;
)

when PC.1 (phil, st, op, state.rb_bit
          (phil eql 1)):=(
        PC.write=(st_normal and (not (op_ld or op_st))) or st_suspend;
        INC.gate=(not (op_callx or op_callr or op_jmpx or
                       op_jmpr or op_ret or op_reti or
                       (op_jmprb and (state.rb_bit eql 1)) or
                       ((op_ld or op_st) and st_normal))) and
                  (not st_repair2);
        INC.gate_cond=(op_jmpx or op_jmpr) and (not st_repair2);
        gate.PC_busD2=op_callr or op_jmpr or op_jmprb or
                      ((op_ldr or op_str) and st_normal);
        gate.PC_busD4=op_calli or op_getlpc or op_callx or op_callr;
)

when PSW.4 (phi4, op
          (phi4 eql 1)):=(
        ctrl.CWP_inc<1>=op_calli or op_callx or op_callr;
        ctrl.CWP_inc<0>=op_calli or op_callx or op_callr or op_ret or op_reti;
)

when PSW.1 (phil, op
          (phil eql 1)):=(
        gate.PSW_busD=op_getpsw;
)

when PSW.2 (phi2, op, scc
          (phi2 eql 1)):=(
        load.PSW_busCC=scc eql 1;
        ctrl.PSW_reti=op_reti;
        load.PSW_busD=op_putpsw;
)

when RB.1 (phil, op, pad.ms
          (phil eql 1)):=(
        ctrl.clrrb=(op_clrrbm and (pad.ms eql 0)) or
                   (op_clrrbs and (pad.ms eql 1));
)

when RF.1 (phil, st, op, repairA, repairA_, repairB, repairB_
          (phil eql 1)):=(
        RF.write=((op_calli or op_getpsw or op_getlpc or
                   op_callx or op_callr or op_sll or op_sra or
                   op_srl or op_ldhi or op_alu or op_jmprb) and
                  st_normal) or (op_ld and st_suspend) or st_repair2;
        load.RFaddr_RA=(repairA eql 1) and st_repair2;
        load.RFaddr_RB=(repairB eql 1) and (repairA eql 0) and st_repair2;
        load.RFaddr_RD=not st_repair2;
)

when SDEC.4 (phi4, st, op
          (phi4 eql 1)):=(
        load.SHam_0=op_ldhi or st_repair1 or st_repair2;
)

when SDEC.1 (phil, st, op
          (phil eql 1)):=(
        load.SHam_BAR=(not (op_ldhi or op_sll or op_sra or op_srl or
                       (op_ld and st_normal) or (op_st and st_suspend))) and
                      (not st_repair1) and (not st_repair2);
        load.SHam_IMM=(op_sll or op_sra or op_srl) and (imm eql 1) and
                      (not st_repair1) and (not st_repair2);
        load.SHam_busB=(op_sll or op_sra or op_srl) and (imm eql 0) and
                       (not st_repair1) and (not st_repair2);
)

when SDR.1 (phil, st
          (phil eql 1)):=(
        gate.SDR_busOUT=((op_st and (not op_strbd)) or
                         (op_strbdm and (pad.ms eql 1)) or
                         (op_strbds and (pad.ms eql 0)) or
                         (op_strbd and (state.rb_bit eql 1))) and
                        st_suspend;
)

when SHIFT.4 (phi4, st, op
          (phi4 eql 1)):=(
        load.SHIFT_IMM=(imm eql 1) or op_callr or op_jmpr or
                       ((op_ldr or op_str) and st_normal);
        SHIFT.sxtT=not (op_sll or op_srl or op_sra or
                        op_ldxhu or op_ldrhu or op_ldxbu or op_ldrbu);
)

when SHIFT.1 (phil, st, op, pad.ms
                (phil eql 1)):=(
        ctrl.SHIFT_sxtS=op_sra;
        load.SHIFT_busT=op_ldhi or (op_ld and st_suspend) or st_repair2;
        load.SHIFT_busR=op_sll or (op_st and st_normal) or st_repairl;
        load.SHIFT_busL=op_sra or op_srl or op_jmprb;
        gate.SHIFT_busL=op_sll or (op_st and st_normal) or st_repairl;
        gate.SHIFT_busR=op_sra or op_srl or op_ldhi or
                        (op_ld and st_suspend) or st_repair2 or
                        (((op_jmprbm and (pad.ms eql 1)) or
                          (op_jmprbs and (pad.ms eql 0))) and
                         (state.rb_bit eql 0));
)
```

gate.isp

```
/*
 *      gate.isp - Gate one bus onto another
 */

macro
        WORD    :=      32&;

state
        busB_SDEC<5>,   /* BDSIM */
        busS31s;        /* BDSIM */

port
        phi2 'input,
        phi3 'input,
        phi4 'input,

        pad.AD<WORD+1> 'bidirectional,

        busIN<WORD> 'output:disconnect,
        busIN.par 'output:disconnect,
        busA<WORD> 'input,      /* inverted */
        busB<WORD> 'input,      /* inverted */
        busS<WORD> 'output,
        busD<WORD> 'bidirectional:disconnect,
        busOUT<WORD> 'output:disconnect,

        ctrl.busINpar_invert 'input,
        gate.padAD_busIN 'input,
        gate.busA_busD2 'input,
        gate.busA_busD4 'input,
        gate.busA_busS 'input,
        gate.busD_busOUT 'input;

main:=(
        pad.AD=0;       /* Endot sillyness for BDSIM */
        next;
        terminate;
)

when (phi2, gate.busA_busD2
      ((phi2 eql 1) and (gate.busA_busD2 eql 1)))=(
        delay(1);       /* Insert gate opening delay */
        connect(busD);
        wait(phi2:trail, gate.busA_busD2:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busD);
)
when (phi4, gate.busA_busD4
      ((phi4 eql 1) and (gate.busA_busD4 eql 1)))=(
        delay(1);       /* Insert gate opening delay */
        connect(busD);
        wait(phi4:trail, gate.busA_busD4:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busD);
)
when (busA):=(
        busD=not busA;
)

when (phi2, gate.busD_busOUT
      ((phi2 eql 1) and (gate.busD_busOUT eql 1)))=(
        delay(1);       /* Insert gate opening delay */
        connect(busOUT);
        wait(phi2:trail, gate.busD_busOUT:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busOUT);
)
when  (busD):=(
        busOUT=busD;
)

when  (phi3, gate.padAD_busIN
       ((phi3 eql 1) and (gate.padAD_busIN eql 1))):=(
        delay(2);        /* Insert gate opening delay */
        connect(busIN);
        connect(busIN.par);
        wait(phi3:trail, gate.padAD_busIN:trail);
        delay(1);        /* Insert gate closing delay */
        disconnect(busIN);
        disconnect(busIN.par);
)
when  (pad.AD):=(
        busIN=pad.AD<31:0>;
        busIN.par=pad.AD<32> xor ctrl.busINpar_invert;
)

when  (gate.busA_busS, busA, busB):=(
        delay(1);        /* Insert propagation delay */
        if (gate.busA_busS eql 1) (
                busS=not busA;
                busS31s=not busA<31>;
        ) else (
                busS=not busB;
                busS31s=not busB<31>;
        );
)

when  (busB):=(
        busB_SDEC=not busB<4:0>;
)
```

imm.isp

```
/*
 *      imm.isp - Immediate Register
 */

macro
        WORD    :=      32&;

port
        phi1 'input,
        phi3 'input,

        busIN<WORD> 'input,
        busT13<WORD> 'output:disconnect,
        busT19<WORD> 'output:disconnect,
        busIMM<5> 'output,

        ctrl.IMM_write 'input,
        gate.IMM13_busT 'input,
        gate.IMM19_busT 'input;

when  (phi1, gate.IMM13_busT
       ((phi1 eql 1) and (gate.IMM13_busT eql 1))):=(
        delay(1);        /* Insert gate opening delay */
        connect(busT13);
        wait(phi1:trail, gate.IMM13_busT:trail);
        delay(1);        /* Insert gate closing delay */
        disconnect(busT13);
)

when  (phi1, gate.IMM19_busT
       ((phi1 eql 1) and (gate.IMM19_busT eql 1))):=(
        delay(1);        /* Insert gate opening delay */
        connect(busT19);
        wait(phi1:trail, gate.IMM19_busT:trail);
        delay(1);        /* Insert gate closing delay */
        disconnect(busT19);
)

when  (phi3, ctrl.IMM_write, busIN
       ((phi3 eql 1) and (ctrl.IMM_write eql 1))):=(
        delay(1);        /* Insert propagation delay */
        busT13=(busIN<12:0> sxt 32) *:logical 13;
        busT19=(busIN<18:0> sxt 32) *:logical 13;
        busIMM=busIN<4:0>;
)
```

int.isp

```
/*
 *      int.isp - Interrupt Logic
 */

macro
        WORD    :=      32&;

state
        ivec<6:4>,      /* inverted, BDSIM */
        I0, I1, I2, I3, I4, privop, badaddr, int4;

port
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busIN<WORD> 'output:disconnect,
        busIN.par 'output:disconnect,
        busOUT<WORD> 'output:disconnect,

        state.repair1 'input,
        state.repair2 'input,

        pad.reset 'input:and,
        pad.irr 'input:and,
        pad.shutdown 'input:and,
        state.PSW_overflow 'input,
        ctrl.over_under 'input,
        ctrl.badop 'input,
        ctrl.privop 'input,
        ctrl.int_enb 'input,
        out.sysmode 'input,
        out.size<2> 'input,
        busBAR<2> 'input,
        ctrl.badshift 'input,

        state.int 'bidirectional,
        state.int3 'bidirectional,
        state.I0 'output,
        out.ira 'output,
        state.reset 'bidirectional,
        state.shutdown 'output,
        state.rb 'input;

when (phi2, pad.reset
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        state.reset=not pad.reset;
)

when (phi2, pad.shutdown
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        state.shutdown=not pad.shutdown;
)

when (phi2, I0, I1, I2, I3, I4, ctrl.int_enb, state.rb,
      state.repair1, state.repair2, state.reset
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        state.int=state.reset or I4 or
                  ((I0 or I1 or I2 or I3) and
                   ctrl.int_enb and (not state.rb) and
                   (not state.repair1) and (not state.repair2));
)

when (phi2, I0
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        state.I0=I0;
)

when (phi3, state.int
      ((phi3 eql 1) and (state.int eql 1))):=(
        busIN=0b00000010110010000000000000000000;
        busIN.par=0b0;
        delay(1);       /* Insert gate opening delay */
        connect(busIN);
        connect(busIN.par);
        wait(phi3:trail, state.int:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busIN);
        disconnect(busIN.par);
)

when (phi3, I1, state.int, state.int3
      (phi3 eql 1)):=(
        delay(1);       /* Insert gate opening delay */
        out.ira=I1 and (not I0) and state.int and (not state.int3);
)

when (phi4, state.int
      ((phi4 eql 1) and (state.int eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busOUT);
        wait(phi4:trail, state.int:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busOUT);
)

when (phi3, I0, I1, I2, I3, I4, state.reset
      (phi3 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        ivec<6>=not (I4 and (not state.reset));
        ivec<5>=not ((not I4) and (not I0) and (I2 or I3));
        ivec<4>=not ((not I4) and (not I0) and (not I2));
        next;
        busOUT=((not ivec<6>) concat (not ivec<5>) concat
                (not ivec<4>) concat (0 ext 4)) ext 32;
)

when (state.reset, ctrl.badop, privop, int4):=(
        delay(1);       /* Insert propagation delay */
        I0=state.reset or ctrl.badop or privop or int4;
)

when (pad.irr):=(
        delay(1);       /* Insert propagation delay */
        I1=not pad.irr;
)

when (state.PSW_overflow, ctrl.over_under):=(
        delay(1);       /* Insert propagation delay */
        I2=state.PSW_overflow and ctrl.over_under;
        I3=state.PSW_overflow and (not ctrl.over_under);
)

when (pad.shutdown):=(
        delay(1);       /* Insert propagation delay */
        I4=not pad.shutdown;
)

when (ctrl.privop, out.sysmode):=(
        delay(1);       /* Insert propagation delay */
        privop=ctrl.privop and out.sysmode;
)

when (out.size, busBAR):=(
        delay(1);       /* Insert propagation delay */
        badaddr=((out.size eql 0b10) and (busBAR neq 0)) or
                ((out.size eql 0b01) and (busBAR<0> neq 0));
)

when (phi3, badaddr, ctrl.badshift
      (phi3 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        state.int3:=(badaddr or ctrl.badshift) and
                    ctrl.int_enb and (not state.rb) and
                    (not state.repair1) and (not state.repair2);
)

when (phi4, state.int3
      (phi4 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        int4=state.int3;
)
```

ir.isp

```
/*
 *      ir.isp - Instruction Register
 */

macro
        WORD    :=      32&,
        RB      :=      4&;

state
        IRdata1[0:RB]<WORD+1>,
        IRdata2[0:RB]<WORD+1>,
        IRvalid1[0:RB],
        IRvalid2[0:RB];

port
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busIN<WORD> 'bidirectional:disconnect,
        busIN.par 'bidirectional:disconnect,

        state.rb 'input,
        state.RB<3> 'input,
        ctrl.IR_write 'input;

main:=(
        IRvalid1[RB]=1;
        IRvalid2[RB]=1;
        next;
        terminate;
)

when (phi4:trail):=(
        state
                ir<32>;

        IRdata1[0]=busIN.par concat busIN;
        IRvalid1[0]=ctrl.IR_write;
        ir=1;
        next;
        while (ir lss RB) (
                IRdata1[ir]=IRdata2[ir-1];
                IRvalid1[ir]=IRvalid2[ir-1];
                ir=ir+1;
        );
        if (IRvalid2[RB-1] eql 1) (
                IRdata1[RB]=IRdata2[RB-1];
                IRdata2[RB]=IRdata2[RB-1];
        );
)

when (phi2:trail):=(
        state
                ir<32>;

        ir=0;
        next;
        while (ir lss RB) (
                IRdata2[ir]=IRdata1[ir];
                IRvalid2[ir]=IRvalid1[ir];
                ir=ir+1;
        );
)

when (phi3, state.rb, state.RB
      (phi3 eql 1)):=(
        state
                ir<32>;

        delay(1);       /* Insert propagation delay */
        if (state.rb eql 1) (
                ir=0;
                next;
                while ((ir lss (state.RB ext 32)) and (ir lss RB)) (
                        IRvalid1[ir]=0;
                        IRvalid2[ir]=0;
                        ir=ir+1;
                );
        );

        ir=0;
        next;
        while (IRvalid2[ir] eql 0)
                ir=ir+1;
        busIN=IRdata2[ir]<31:0>;
        busIN.par=IRdata2[ir]<32>;
)

when (phi3, state.rb
      ((phi3 eql 1) and (state.rb eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busIN);
        connect(busIN.par);
        wait(phi3:trail, state.rb:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busIN);
        disconnect(busIN.par);
)
```

mar.isp

```
/*
 *      mar.isp - Memory Address Register
 */

macro
        WORD    :=      32&,
        RB      :=      4&;

state
        MARdata1[0:RB]<WORD>,
        MARdata2[0:RB]<WORD>,
        MARvalid1[0:RB],
        MARvalid2[0:RB];

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busOUT<WORD> 'bidirectional:disconnect,

        state.rb 'input,
        state.RB<3> 'input,
        ctrl.MAR_write 'input,
        gate.MAR_busOUT2 'input,
        gate.MAR_busOUT4 'input;

main:=(
        MARvalid1[RB]=1;
        MARvalid2[RB]=1;
        next;
        terminate;
)

when (phi1, busOUT, ctrl.MAR_write
      (phi1 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        MARdata1[0]=busOUT;
        MARvalid1[0]=ctrl.MAR_write;
)

when (phi1:trail):=(
        state
                mar<32>;

        mar=1;
        next;
        while (mar lss RB) (
                MARdata1[mar]=MARdata2[mar-1];
                MARvalid1[mar]=MARvalid2[mar-1];
                mar=mar+1;
        );
        if (MARvalid2[RB-1] eql 1) (
                MARdata1[RB]=MARdata2[RB-1];
                MARdata2[RB]=MARdata2[RB-1];
        );
)

when (phi2, gate.MAR_busOUT2
      ((phi2 eql 1) and (gate.MAR_busOUT2 eql 1))):=(
        delay(1);       /* Insert gate opening delay */
```

```
          connect(busOUT);
          wait(phi2:trail, gate.MAR_busOUT2:trail);
          delay(1);        /* Insert gate closing delay */
          disconnect(busOUT);
 )

 when (phi2:trail):=(
          state
                   mar<32>;

          mar=0;
          next;
          while (mar lss RB) (
                   MARvalid2[mar]=MARvalid1[mar];
                   mar=mar+1;
          );
 )

 when (phi3, state.rb, state.RB
       ((phi3 eql 1) and (state.rb eql 1))):=(
          state
                   mar<32>;

          delay(1);        /* Insert propagation delay */
          mar=0;
          next;
          while ((mar lss (state.RB ext 32)) and (mar lss RB)) (
                   MARvalid1[mar]=0;
                   MARvalid2[mar]=0;
                   mar=mar+1;
          );
 )

 when (phi3:trail):=(
          state
                   mar<32>;

          mar=0;
          next;
          while (mar lss RB) (
                   MARdata2[mar]=MARdata1[mar];
                   mar=mar+1;
          );
          mar=0;
          next;
          while (MARvalid2[mar] eql 0)
                   mar=mar+1;
          busOUT=MARdata2[mar];
 )

 when (phi4, gate.MAR_busOUT4, state.rb
       ((phi4 eql 1) and ((gate.MAR_busOUT4 eql 1) or (state.rb eql 1)))):=(
          delay(1);        /* Insert gate opening delay */
          connect(busOUT);
          wait(phi4:trail, gate.MAR_busOUT4:trail, state.rb:trail);
          delay(1);        /* Insert gate closing delay */
          disconnect(busOUT);
 )
```

par.isp

```
/*
 *       par.isp - Parity Generators
 */

macro
         WORD     :=      32&;

state
         busB.par,
         error.busIN1,
         rfA.par4,
         rfB.par4,
         error.busOUT_;  /* BDSIM */

port
         phi1 'input,
         phi2 'input,
         phi3 'input,
         phi4 'input,
        busIN<WORD> 'input,
        busIN.par 'input,
        rfA.par 'input,
        busB<WORD> 'input,
        rfB.par 'input,
        busD<WORD> 'input,
        state.busD<4> 'output,
        busD.par 'bidirectional,
        busOUT<WORD> 'input,
        busOUT.par 'bidirectional,

        error.busIN 'output,
        error.busA 'output,
        error.busB 'output,
        error.busOUT 'bidirectional,

        load.PAR_busIN 'input,
        load.PAR_busA 'input,
        load.PAR_busB 'input,
        load.PAR_busOUTA 'input,
        load.PAR_busOUTB 'input;

main:=(
        error.busOUT_=1;
        next;
        terminate;
)

when (phi1, busIN, busIN.par
      (phi1 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        error.busIN1=busIN.par neq parity(busIN);
)

when (phi3, error.busIN1, load.PAR_busIN
      (phi3 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        error.busIN=error.busIN1 and load.PAR_busIN;
)

when (phi4, rfA.par, rfB.par
      (phi4 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        rfA.par4=rfA.par;
        rfB.par4=rfB.par;
)

when (phi1:lead):=(
        delay(1);        /* Insert propagation delay */
        state.busD=busD<31:28> xor busD<27:24> xor
                   busD<23:20> xor busD<19:16> xor
                   busD<15:12> xor busD<11:8> xor
                   busD<7:4> xor busD<3:0>;
)

when (phi1:lead, phi3:lead):=(
        delay(1);        /* Insert propagation delay */
        busD.par=parity(busD);
)

when (phi3, busD.par, rfA.par, load.PAR_busA
      (phi3 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        error.busA=(rfA.par xor busD.par) and load.PAR_busA;
)

when (busB):=(
        delay(1);        /* Insert propagation delay */
        busB.par=parity(busB);
)

when (phi3, busB, rfB.par, load.PAR_busB
      (phi3 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        error.busB=(rfB.par xor busB.par) and load.PAR_busB;
)

when (busOUT):=(
        delay(1);        /* Insert propagation delay */
        busOUT.par=parity(busOUT);
)

when (phi3, busOUT.par, rfA.par4, rfB.par4,
      load.PAR_busOUTA, load.PAR_busOUTB
      (phi3 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        error.busOUT=((busOUT.par xor rfA.par4) and load.PAR_busOUTA) or
                     ((busOUT.par xor rfB.par4) and load.PAR_busOUTB);
        next;
        error.busOUT_=not error.busOUT;
)
```

pc.isp

```
/*
 *      pc.isp - Program Counter
 */

macro
        WORD    :=      32&,
        RB      :=      4&;

state
        PCdata1[0:RB+2]<WORD>,
        PCdata2[0:RB+2]<WORD>,
        PCvalid1[0:RB+2],
        PCvalid2[0:RB+2],
        PCvalid;

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busD<WORD> 'output:disconnect,
        busOUT<WORD> 'bidirectional:disconnect,

        ctrl.PC_write 'input,
        state.rb 'input,
        state.RB<3> 'input,
        ctrl.PC_select 'input,
        gate.INC_busOUT 'input,
        gate.PC_busD2 'input,
        gate.PC_busD4 'input;

main:=(
        PCvalid1[RB]=1;
        PCvalid2[RB]=1;
        PCvalid1[RB+1]=1;
        PCvalid2[RB+1]=1;
        PCvalid1[RB+2]=1;
        PCvalid2[RB+2]=1;
        next;
        terminate;
)

when (phi3, state.rb, state.RB
      (phi3 eql 1)):=(
        state
                pc<32>;

        delay(1);       /* Insert propagation delay */
        pc=0;
        next;
        while ((pc lss (state.RB ext 32)) and (pc lss RB)) (
                PCvalid1[pc]=0;
                PCvalid2[pc]=0;
                pc=pc+1;
        );
)

when (phi3:trail):=(
        state
                pc<32>;

        pc=1;
        next;
        while (pc lss RB) (
                PCdata1[pc]=PCdata2[pc-1];
                PCvalid1[pc]=PCvalid2[pc-1];
                pc=pc+1;
        );
        PCvalid=PCvalid2[RB-1];
        if (PCvalid2[RB-1] eql 1) (
                PCdata1[RB]=PCdata2[RB-1];
                PCdata1[RB+1]=PCdata2[RB];
                PCdata1[RB+2]=PCdata2[RB+1];
                PCdata2[RB+2]=PCdata2[RB+1];
        );
)

when (phi4:trail):=(
        state
                pc<32>;

        PCvalid1[0]=ctrl.PC_write;
        PCvalid2[0]=ctrl.PC_write;
        pc=1;
        next;
        while (pc lss RB) (
                PCvalid2[pc]=PCvalid1[pc];
                pc=pc+1;
        );
)

when (phi1, busOUT
      (phi1 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        PCdata1[0]=busOUT;
        PCdata2[0]=busOUT;
)

when (phi1:trail):=(
        state
                nxtpc<32>, pc<32>, lstpc<32>;

        pc=1;
        next;
        while (pc lss RB) (
                PCdata2[pc]=PCdata1[pc];
                pc=pc+1;
        );
        if (PCvalid eql 1) (
                PCdata2[RB]=PCdata1[RB];
                PCdata2[RB+1]=PCdata1[RB+1];
        );
        nxtpc=0;
        next;
        while (PCvalid2[nxtpc] eql 0)
                nxtpc=nxtpc+1;
        pc=nxtpc+1;
        next;
        while (PCvalid2[pc] eql 0)
                pc=pc+1;
        lstpc=pc+1;
        next;
        while (PCvalid2[lstpc] eql 0)
                lstpc=lstpc+1;

        busOUT=(PCdata2[nxtpc]+4) and (not (0b11 ext 32));
        if (ctrl.PC_select eql 1)
                busD=PCdata2[lstpc]
        else
                busD=PCdata2[pc];
)

when (phi2, gate.PC_busD2
      ((phi2 eql 1) and (gate.PC_busD2 eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busD);
        wait(phi2:trail, gate.PC_busD2:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busD);
)

when (phi4, gate.PC_busD4
      ((phi4 eql 1) and (gate.PC_busD4 eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busD);
        wait(phi4:trail, gate.PC_busD4:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busD);
)

when (phi4, gate.INC_busOUT
      ((phi4 eql 1) and (gate.INC_busOUT eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busOUT);
        wait(phi4:trail, gate.INC_busOUT:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busOUT);
)
```

psw.isp

```
/*
 *      psw.isp - Processor Status Word
 */

macro
        WORD    :=      32&,
        RB      :=      4&,
        PASS    :=      0b00&,
        INC     :=      0b01&,
        DEC     :=      0b11&;

state
        ctrl.CWP_inc2<2>,
        PSWdata1[0:RB]<11>,
        PSWdata2[0:RB]<11>,
        PSWvalid1[0:RB],
        PSWvalid2[0:RB],
        PSW.data<11>,
        PSW<11>,
        NXTPSW<11>;

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busD<WORD> 'bidirectional:disconnect,
        busCWP<2> 'output,
        busCC<4> 'input,
        ctrl.int_enb 'output,
        out.sysmode 'output,
        state.PSW_overflow 'output,
        state.cc<4> 'output,
        state.PSW<4> 'output,

        state.int 'input,
        state.rb 'input,
        state.RB<3> 'input,
        state.reset 'input,
        ctrl.PSW_write 'input,
        ctrl.CWP_inc<2> 'input,
        gate.PSW_busD 'input,
        load.PSW_busCC 'input,
        ctrl.PSW_reti 'input,
        load.PSW_busD 'input;

format
        PSW.data_CWP    :=      PSW.data<10:9>,
        PSW.data_SWP    :=      PSW.data<8:7>,
        PSW.data_CC     :=      PSW.data<3:0>,
        PSW_CWP         :=      PSW<10:9>,
        PSW_SWP         :=      PSW<8:7>,
        PSW_ISP         :=      PSW<6:4>,
        PSW_I           :=      PSW<6>,
        PSW_S           :=      PSW<5>,
        PSW_P           :=      PSW<4>,
        PSW_CC          :=      PSW<3:0>,
        busD_CWP        :=      busD<10:9>,
        busD_SWP        :=      busD<8:7>,
        busD_ISP        :=      busD<6:4>,
        busD_CC         :=      busD<3:0>,
        NXTPSW_CWP      :=      NXTPSW<10:9>,
        NXTPSW_SWP      :=      NXTPSW<8:7>,
        NXTPSW_ISP      :=      NXTPSW<6:4>,
        NXTPSW_CC       :=      NXTPSW<3:0>;

main:=(
        PSWvalid1[RB]=1;
        PSWvalid2[RB]=1;
        next;
        terminate;
)

when (phi4:trail):=(
        state
                psw<32>;

        PSWvalid1[0]=ctrl.PSW_write;
        psw=1;
        next;
        while (psw lss RB) (
                PSWdata1[psw]=PSWdata2[psw-1];
                PSWvalid1[psw]=PSWvalid2[psw-1];
                psw=psw+1;
        );
        if (PSWvalid2[RB-1] eql 1) (
                PSWdata1[RB]=PSWdata2[RB-1];
                PSWdata2[RB]=PSWdata2[RB-1];
        );
)

when (phi1, NXTPSW
      (phi1 eql 1)):=(
        state
                psw<32>;

        delay(1);       /* Insert propagation delay */
        PSWdata1[0]=NXTPSW;
        PSWdata2[0]=NXTPSW;
        PSWvalid2[0]=PSWvalid1[0];
        psw=1;
        next;
        while (psw lss RB) (
                PSWdata2[psw]=PSWdata1[psw];
                PSWvalid2[psw]=PSWvalid1[psw];
                psw=psw+1;
        );

        psw=0;
        next;
        while (PSWvalid2[psw] eql 0)
                psw=psw+1;
        PSW.data=PSWdata2[psw];
)

when (phi2, phi3, PSW.data, state.rb
      ((phi2 eql 1) or ((phi3 eql 1) and (state.rb eql 1)))):=(
        delay(1);       /* Insert propagation delay */
        if (state.reset eql 1)
                PSW=0b00000000000
        else
                PSW=PSW.data;
)

when (phi2, ctrl.CWP_inc
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        ctrl.CWP_inc2=ctrl.CWP_inc;
)

when (phi3, state.rb, state.RB
      ((phi3 eql 1) and (state.rb eql 1))):=(
        state
                psw<32>;

        delay(1);       /* Insert propagation delay */
        psw=0;
        next;
        while ((psw lss (state.RB ext 32)) and (psw lss RB)) (
                PSWvalid1[psw]=0;
                PSWvalid2[psw]=0;
                psw=psw+1;
        );

        while (PSWvalid2[psw] eql 0)
                psw=psw+1;
        PSW.data=PSWdata2[psw];
)

when (phi4, gate.PSW_busD
      ((phi4 eql 1) and (gate.PSW_busD eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busD);
        wait(phi4:trail, gate.PSW_busD:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busD);
)

when (PSW.data):=(
        delay(1);       /* Insert propagation delay */
        busCWP=PSW.data_CWP;
        state.cc=PSW.data_CC;
        state.PSW=((0b0 concat PSW.data<9> concat
                    PSW.data<10> concat PSW.data<8>) xor
                   PSW.data<7:4>) xor PSW.data<3:0>;
)

when (PSW) :=(
        delay(1);       /* Insert propagation delay */
        busD=PSW ext 32;
        out.sysmode=PSW_S;
        ctrl.int_enb=PSW_I;
  }

  when (load.PSW_busD, load.PSW_busCC, ctrl.PSW_reti, ctrl.CWP_inc2,
        busD, busCC, PSW, state.int:change):=(
        delay(1);        /* Insert propagation delay */
        if (load.PSW_busCC and (not state.int))
                NXTPSW_CC=busCC
        else if (load.PSW_busD and (not state.int))
                NXTPSW_CC=busD_CC
        else
                NXTPSW_CC=PSW_CC;

        if (state.int)
                NXTPSW_ISP=0 concat 0 concat PSW_S
        else if (load.PSW_busD and (not state.int))
                NXTPSW_ISP=busD_ISP
        else if (ctrl.PSW_reti and (not state.int))
                NXTPSW_ISP=1 concat PSW_P concat PSW_P
        else
                NXTPSW_ISP=PSW_ISP;

        if (load.PSW_busD and (not state.int))
                NXTPSW_SWP=busD_SWP
        else
                NXTPSW_SWP=PSW_SWP;

        if (load.PSW_busD and (not state.int))
                NXTPSW_CWP=busD_CWP
        else if (state.int)
                NXTPSW_CWP=PSW_CWP
        else
                NXTPSW_CWP=PSW_CWP+ctrl.CWP_inc2;
  }

  when (PSW.data, ctrl.CWP_inc):=(
        delay(1);        /* Insert propagation delay */
        state.PSW_overflow=((PSW.data_CWP+ctrl.CWP_inc) eql PSW.data_SWP);
  }
```

rb.isp

```
/*
 *      rb.isp - Rollback and State Repair Logic
 */

macro
        RB      :=      4&;

state
        RBvalid1[0:RB+2],
        RBvalid2[0:RB+2],
        rbcount<32>,
        rb1<32>,
        rb2<32>,
        rb3<32>,
        enb.RB2,
        enb.RB1,
        enb.RB0,
        out.RB2,
        out.RB1,
        out.RB0,
        ctrl.shutdown,
        out.repairA_,
        out.repairB_,
        ctrl.RB_enable,
        enable_count<3>,
        shutdown_count1<4>,
        shutdown_count2<2>,
        rbshutdown,
        enb.pad.RB0,            /* BDSIM */
        enb.pad.RB1,            /* BDSIM */
        enb.pad.RB2,            /* BDSIM */
        enb.pad.RB<3>,          /* BDSIM */
        enb.pad.rb,             /* BDSIM */
        enb.pad.shutdown,       /* BDSIM */
        pad.RB<3>;              /* BDSIM */

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        pad.rb 'bidirectional:and:disconnect,
        pad.RB2 'bidirectional:and:disconnect,
        pad.RB1 'bidirectional:and:disconnect,
        pad.RB0 'bidirectional:and:disconnect,
        error.busA 'input,
        error.busB 'input,
        error.busIN 'input,
        error.CMP 'input,
        error.busOUT 'input,

        state.rb 'bidirectional,
        state.RB<3> 'bidirectional,
        pad.ms 'input,
        pad.repairAm 'bidirectional:and:disconnect,
        pad.repairAs 'bidirectional:and:disconnect,
        pad.repairBm 'bidirectional:and:disconnect,
        pad.repairBs 'bidirectional:and:disconnect,
        pad.shutdown 'bidirectional:and:disconnect,
        state.reset 'input,
        state.shutdown 'input,
        ctrl.RB_write 'input,
        ctrl.repairA_ 'bidirectional,
        ctrl.repairA 'output,
        ctrl.repairB_ 'bidirectional,
        ctrl.repairB 'output,
        ctrl.clrrb 'input,
        ctrl.norepair 'output,
        state.rb_bit 'output;

main:=(
        RBvalid1[RB]=1;
        RBvalid2[RB]=1;
        RBvalid1[RB+1]=1;
        RBvalid2[RB+1]=1;
        RBvalid1[RB+2]=1;
        RBvalid2[RB+2]=1;
        pad.rb=0;
        pad.RB2=0;
        pad.RB1=0;
        pad.RB0=0;
        pad.shutdown=0;
        rbcount=0;
        out.repairA_=1;
        out.repairB_=1;
        if (pad.ms eql 0) (
                connect(pad.repairAm);
                connect(pad.repairBm);
        ) else (
                connect(pad.repairAs);
                connect(pad.repairBs);
        );
        enb.pad.RB0=0;
        enb.pad.RB1=0;
        enb.pad.RB2=0;
        enb.pad.RB=0;
        enb.pad.rb=0;
        enb.pad.shutdown=0;
        pad.RB=0;
        next;
        terminate;
)

when (phi3, state.rb, state.RB, ctrl.RB_write
      (phi3 eql 1)):=(
        state
                rb<32>;

        delay(1);       /* Insert propagation delay */
        if (state.rb eql 1) (
                rb=0;
                next;
                while ((rb lss (state.RB ext 32)) and (rb lss RB)) (
                        RBvalid1[rb]=0;
                        RBvalid2[rb]=0;
                        rb=rb+1;
                );
                rbcount=1;
        );

        RBvalid1[0]=ctrl.RB_write;      /* should be 0 on rb */
)

when (phi3:trail):=(
        state
```

```
                        rb<32>;

                rb=1;
                next;
                while (rb lss RB) (
                        RBvalid1[rb]=RBvalid2[rb-1];
                        rb=rb+1;
                );
        )

        when (phi4:trail):=(
                state
                        rb<32>;

                rb=0;
                next;
                while (rb lss RB) (
                        RBvalid2[rb]=RBvalid1[rb];
                        rb=rb+1;
                );

                if ((state.rb eql 0) and (rbcount gtr 0))
                        rbcount=(rbcount+1) mod 4;
        )

        when (phi1
              (phi1 eql 1)):=(
                state
                        rb<32>;

                delay(1);       /* Insert propagation delay */
                rb1=0;
                next;
                while (RBvalid2[rb1] eql 0)
                        rb1=rb1+1;
                rb2=rb1+1;
                next;
                while (RBvalid2[rb2] eql 0)
                        rb2=rb2+1;
                rb3=rb2+1;
                next;
                while (RBvalid2[rb3] eql 0)
                        rb3=rb3+1;
                next;
                rb1=rb1+1;
                rb2=rb2+1;
                rb3=rb3+1;
                next;
        )

        when (phi2, pad.rb, state.reset, state.shutdown, pad.RB2, pad.RB1, pad.RB0,
              pad.repairAm, pad.repairAs, pad.repairBm, pad.repairBs
              (phi2 eql 1)):=(
                delay(1);       /* Insert propagation delay */
                state.rb=((not pad.rb) and (not state.reset) and (not state.shutdown));
                state.RB<2>=not pad.RB2;
                state.RB<1>=not pad.RB1;
                state.RB<0>=not pad.RB0;
                pad.RB=pad.RB2 concat pad.RB1 concat pad.RB0;
                ctrl.repairA=(not pad.repairAm) or (not pad.repairAs);
                ctrl.repairB=(not pad.repairBm) or (not pad.repairBs);
        )

        when (phi2, state.rb, ctrl.clrrb
              (phi2 eql 1)):=(
                if (state.rb eql 1)
                        state.rb_bit=1
                else if (ctrl.clrrb eql 1)
                        state.rb_bit=0;
        )

        when (phi4, error.busA, error.busB
              (phi4 eql 1)):=(
                delay(1);       /* Insert propagation delay */
                ctrl.repairA_=error.busA;
                ctrl.repairB_=error.busB;
        )

        when (phi4, state.reset, state.shutdown
              (phi4 eql 1)):=(
                delay(1);       /* Insert propagation delay */
                if ((state.shutdown eql 1) or (state.reset eql 1))
                        enable_count=4
                else if (enable_count neq 0)
                        enable_count=enable_count-1;
        )

        when (enable_count):=(
        delay(1);        /* Insert propagation delay */
        ctrl.RB_enable=(enable_count eql 0);
)

when (phi1, error.busIN, ctrl.repairA_, ctrl.repairB_, error.CMP,
      error.busOUT, ctrl.RB_enable, rb1, rb2, rb3
      ((phi1 eql 1) and (ctrl.RB_enable eql 1) and
       ((error.busIN eql 1) or (error.CMP eql 1) or (error.busOUT eql 1) or
        (ctrl.repairA_ eql 1) or (ctrl.repairB_ eql 1))))):=(
        delay(1);        /* Insert propagation delay */
        connect(pad.rb); enb.pad.rb=1;
        if ((rbcount eql 3) and ((error.busIN eql 1) or (error.CMP eql 1))) (
                if (rb3 leq 4) (
                        out.RB2=rb3<2>;
                        out.RB1=rb3<1>;
                        out.RB0=rb3<0>;
                ) else (
                        out.RB2=1;
                        out.RB1=0;
                        out.RB0=0;
                );
        ) else if (rbcount eql 2) (
                if (rb2 leq 4) (
                        out.RB2=rb2<2>;
                        out.RB1=rb2<1>;
                        out.RB0=rb2<0>;
                ) else (
                        out.RB2=1;
                        out.RB1=0;
                        out.RB0=0;
                );
        ) else if ((error.busIN eql 1) or (error.CMP eql 1)) (
                out.RB2=0;
                out.RB1=1;
                out.RB0=0;
        ) else (
                out.RB2=0;
                out.RB1=0;
                out.RB0=1;
        );
        enb.RB2=1;
)

/* Titus's ugly endot hack */
when (phi1, ctrl.RB_enable, error.busIN, error.CMP,
      error.busOUT, ctrl.repairA_, ctrl.repairB_
      ((phi1 eql 1) and
       ((ctrl.RB_enable eql 0) or
        ((error.busIN eql 0) and (error.CMP eql 0) and
         (error.busOUT eql 0) and (ctrl.repairA_ eql 0) and
         (ctrl.repairB_ eql 0)))))):=(
        delay(1);        /* Insert propagation delay */
        disconnect(pad.rb); enb.pad.rb=0;
        enb.RB2=0;
)

when (phi1, ctrl.repairA_
      ((phi1 eql 1) and (ctrl.repairA_ eql 1))):=(
        delay(1);        /* Insert propagation delay */
        out.repairA_=0;
)

when (phi1, ctrl.repairB_
      ((phi1 eql 1) and (ctrl.repairB_ eql 1))):=(
        delay(1);        /* Insert propagation delay */
        out.repairB_=0;
)

when (phi4
      (phi4 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        disconnect(pad.rb); enb.pad.rb=0;
        enb.RB2=0;
        out.repairA_=1;
        out.repairB_=1;
        disconnect(pad.shutdown); enb.pad.shutdown=0;
)

when (phi4, state.rb, state.reset, state.shutdown, ctrl.clrrb
      (phi4 eql 1)):=(
        delay(1);        /* Insert propagation delay */
        if ((state.reset eql 1) or (state.shutdown)) (
                rbshutdown=0;
                shutdown_count1=0;
                shutdown_count2=0;
        ) else (
                if (state.rb eql 1) (
                        rbshutdown=(shutdown_count2 eql 3);
```

```
                                next;
                                shutdown_count2=shutdown_count2+1;
                                next;
                        ) else if (ctrl.clrrb eql 1) (
                                rbshutdown=0;
                                shutdown_count1=0;
                        ) else
                                rbshutdown=0;
                        next;
                        if ((ctrl.clrrb eql 0) or (state.rb eql 1)) (
                                shutdown_count1=(shutdown_count1+1) mod 16;
                                next;
                                if (shutdown_count1 eql 0)
                                        shutdown_count2=0;
                        )
                )
        )

when (phi4, state.rb, state.RB, rbshutdown
      (phi4 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        ctrl.shutdown=((state.rb eql 1) and ((state.RB ext 32) gtr RB)) or
                      (rbshutdown eql 1);
)

when (phi2, pad.repairAm, pad.repairAs, pad.repairBm, pad.repairBs
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        ctrl.norepair=((pad.repairAm eql 0) and (pad.repairAs eql 0)) or
                      ((pad.repairBm eql 0) and (pad.repairBs eql 0));
)

when (phi1, ctrl.shutdown
      ((phi1 eql 1) and (ctrl.shutdown eql 1))):=(
        delay(1);       /* Insert propagation delay */
        connect(pad.shutdown); enb.pad.shutdown=1;
)

when (enb.RB2:change, out.RB2:change, pad.RB2:change):=(
        delay(1);       /* Insert propagation delay */
        if (enb.RB2 and out.RB2) (
                connect(pad.RB2); enb.pad.RB2=1;
        ) else (
                disconnect(pad.RB2); enb.pad.RB2=0;
        );
        enb.RB1=enb.RB2 and (out.RB2 or pad.RB2);
)
when (enb.RB1:change, out.RB1:change, pad.RB1:change):=(
        delay(1);       /* Insert propagation delay */
        if (enb.RB1 and out.RB1) (
                connect(pad.RB1); enb.pad.RB1=1;
        ) else (
                disconnect(pad.RB1); enb.pad.RB1=0;
        );
        enb.RB0=enb.RB1 and (out.RB1 or pad.RB1);
)
when (enb.RB0:change, out.RB0:change, pad.RB0:change):=(
        delay(1);       /* Insert propagation delay */
        if (enb.RB0 and out.RB0) (
                connect(pad.RB0); enb.pad.RB0=1;
        ) else (
                disconnect(pad.RB0); enb.pad.RB0=0;
        );
)
when (enb.pad.RB0, enb.pad.RB1, enb.pad.RB2):=(
        delay(1);       /* Insert propagation delay */
        enb.pad.RB=enb.pad.RB2 concat enb.pad.RB1 concat enb.pad.RB0;
)

when (out.repairA_):=(
        delay(1);       /* Insert propagation delay */
        pad.repairAm=out.repairA_;
        pad.repairAs=out.repairA_;
)

when (out.repairB_):=(
        delay(1);       /* Insert propagation delay */
        pad.repairBm=out.repairB_;
        pad.repairBs=out.repairB_;
)
```

rf.isp

```
/*
 *      rf.isp - Register File
 */

macro
        WORD    :=      32&,
        RB      :=      4&;

state
        busA.par,       /* inverted */
        busB.par,       /* inverted */
        rfD.par,        /* BDSIM */
        RFdata1[0:RB-1]<WORD+1>,
        RFdata2[0:RB-1]<WORD+1>,
        RFaddr1[0:RB-1]<7>,
        RFaddr2[0:RB-1]<7>,
        RFvalid,
        RFvalid1[0:RB-1],
        RFvalid2[0:RB-1],
        RFglobal[0:9]<WORD+1>,
        RFlocal[0:63]<WORD+1>;

port
        phi1 'input,
        phi3 'input,
        phi4 'input,

        busRA<7> 'input,
        busRA.par 'input,       /* inverted */
        busRB<7> 'input,
        busRB.par 'input,       /* inverted */
        busRD<7> 'input,
        busD<WORD> 'input,
        busD.par 'input,
        busA<WORD> 'output,     /* inverted */
        rfA.par 'output,
        busB<WORD> 'output,     /* inverted */
        rfB.par 'output,
        state.RFaddr<4> 'bidirectional,

        state.rb 'input,
        state.RB<3> 'input,
        ctrl.RF_write 'input,
        ctrl.RFpar_invert 'input;

main:=(
        RFglobal[0]=0b000000000000000000000000000000000;
        next;
        terminate;
)

when (phi3, state.rb, state.RB
       (phi3 eql 1)):=(
        state
                rf<32>;

        delay(1);       /* Insert propagation delay */
        if (state.rb eql 1) (
                rf=0;
                next;
                while ((rf lss (state.RB ext 32)) and (rf lss RB)) (
                        RFvalid1[rf]=0;
                        RFvalid2[rf]=0;
                        rf=rf+1;
                );
        );
)

when (phi3:trail):=(
        state
                rf<32>;

        rf=1;
        next;
        while (rf lss RB) (
                RFdata1[rf]=RFdata2[rf-1];
                RFaddr1[rf]=RFaddr2[rf-1];
                RFvalid1[rf]=RFvalid2[rf-1];
                rf=rf+1;
        );
        if (RFvalid2[RB-1] eql 1) (
                if (RFaddr2[RB-1]<6> eql 0)
                        RFglobal[RFaddr2[RB-1] ext 8]=RFdata2[RB-1]
                else
                        RFlocal[RFaddr2[RB-1]<5:0> ext 8]=RFdata2[RB-1];
        );
)

when (phi4, ctrl.RF_write, busRD
      (phi4 eql 1)):=(
        RFvalid=ctrl.RF_write and (busRD neq 0);
        next;
        state.RFaddr=((not RFvalid) concat busRD<6:4>) xor busRD<3:0>;
        RFaddr1[0]=busRD;
        RFaddr2[0]=busRD;
        RFvalid1[0]=RFvalid;
        RFvalid2[0]=RFvalid;
)

when (phi4:trail):=(
        state
                rf<32>;

        rf=1;
        next;
        while (rf lss RB) (
                RFaddr2[rf]=RFaddr1[rf];
                RFvalid2[rf]=RFvalid1[rf];
                rf=rf+1;
        );
)

when (phi1, busD, busD.par, state.RFaddr, ctrl.RFpar_invert
      (phi1 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        RFdata1[0]=(ctrl.RFpar_invert xor (parity(state.RFaddr)) xor
                        busD.par) concat busD;
        RFdata2[0]=(ctrl.RFpar_invert xor (parity(state.RFaddr)) xor
                        busD.par) concat busD;
        rfD.par=ctrl.RFpar_invert xor (parity(state.RFaddr)) xor busD.par;
)

when (phi1:lead):=(
        state
                rf<32>, rfa<32>, rfb<32>;

        delay(50);      /* Silly Endot hack for BDSIM */

        rf=1;
        next;
        while (rf lss RB) (
                RFdata2[rf]=RFdata1[rf];
                rf=rf+1;
        );

        rfa=0;
        next;
        while (((RFvalid2[rfa] eql 0) or (RFaddr2[rfa] neq busRA)) and
               (rfa leq RB))
                rfa=rfa+1;
        if (rfa leq RB) (
                busA=not RFdata2[rfa]<31:0>;
                busA.par=not RFdata2[rfa]<32>;
        ) else (
                if (busRA<6> eql 0) (
                        busA=not RFglobal[busRA ext 8]<31:0>;
                        busA.par=not RFglobal[busRA ext 8]<32>;
                ) else (
                        busA=not RFlocal[busRA<5:0> ext 8]<31:0>;
                        busA.par=not RFlocal[busRA<5:0> ext 8]<32>;
                )
        );

        rfb=0;
        next;
        while (((RFvalid2[rfb] eql 0) or (RFaddr2[rfb] neq busRB)) and
               (rfb leq RB))
                rfb=rfb+1;
        if (rfb leq RB) (
                busB=not RFdata2[rfb]<31:0>;
                busB.par=not RFdata2[rfb]<32>;
        ) else (
                if (busRB<6> eql 0) (
                        busB=not RFglobal[busRB ext 8]<31:0>;
                        busB.par=not RFglobal[busRB ext 8]<32>;
                ) else (
                        busB=not RFlocal[busRB<5:0> ext 8]<31:0>;
                        busB.par=not RFlocal[busRB<5:0> ext 8]<32>;
                );
        );
)
```

```
when (busA.par, busRA.par):=(
        rfA.par=busA.par xor busRA.par;
)

when (busB.par, busRB.par):=(
        rfB.par=busB.par xor busRB.par;
)
```

rftran.isp

```
/*
 *      rftran.isp - Register File Translator
 */

macro
        WORD    :=      32&,
        RB      :=      4&;

state
        ra.in<5>,
        rb.in<5>,
        rd.in<5>,
        ra.out<7>,
        rb.out<7>,
        rd.out<7>,
        CWP<2>;

port
        phi1 'input,
        phi2 'input,
        phi3 'input,

        busIN<WORD> 'input,
        busIR<14> 'input,
        busCWP<2> 'input,
        ctrl.CWP_inc<2> 'input,
        state.int 'input,
        state.rb 'input,

        busRA<7> 'output,
        busRA.par 'output,      /* inverted */
        busRB<7> 'output,
        busRB.par 'output,      /* inverted */
        busRD<7> 'output,

        load.RFTRAN_busIN 'input,
        load.RFaddr_RA 'input,
        load.RFaddr_RB 'input,
        load.RFaddr_RD 'input;

when (phi1, ra.out, rb.out
      (phi1 eql 1)):=(
        busRA.par=not parity(ra.out);
        busRB.par=not parity(rb.out);
)

when (phi2
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        rd.in=busIR<5:1>;
)

when (phi3, busCWP, state.int, state.rb, ctrl.CWP_inc
      (phi3 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        if ((state.int eql 0) and (state.rb eql 0))
                CWP=busCWP+ctrl.CWP_inc
        else
                CWP=busCWP;
)

when (phi3, load.RFTRAN_busIN, busIN
      ((phi3 eql 1) and (load.RFTRAN_busIN eql 1))):=(
        delay(1);       /* Insert propagation delay */
        ra.in=busIN<18:14>;
        if (busIN<30:29> eql 0b11)
                rb.in=busIN<23:19>
        else
                rb.in=busIN<4:0>;
)
```

```
when (ra.in, CWP):=(
        state   ra<7>;

        delay(1);       /* Insert propagation delay */
        ra<3:0>=ra.in<3:0>;
        ra<5:4>=CWP-((not ra.in<4>) ext 2);
        ra<6>=not ((ra.in leq 9) and (ra.in geq 0));
        next;
        if (ra<6> eql 0)
                ra.out=ra<3:0> ext 7
        else
                ra.out=ra;
        next;
        busRA=ra.out;
)

when (rb.in, CWP):=(
        state   rb<7>;

        delay(1);       /* Insert propagation delay */
        rb<3:0>=rb.in<3:0>;
        rb<5:4>=CWP-((not rb.in<4>) ext 2);
        rb<6>=not ((rb.in leq 9) and (rb.in geq 0));
        next;
        if (rb<6> eql 0)
                rb.out=rb<3:0> ext 7
        else
                rb.out=rb;
        next;
        busRB=rb.out;
)

when (rd.in, CWP):=(
        state
                rd<7>;

        delay(1);       /* Insert propagation delay */
        rd<3:0>=rd.in<3:0>;
        rd<5:4>=CWP-((not rd.in<4>) ext 2);
        rd<6>=not ((rd.in<4:0> leq 9) and (rd.in<4:0> geq 0));
        next;
        if (rd<6> eql 0)
                rd.out=rd<3:0> ext 7
        else
                rd.out=rd;
)

when (load.RFaddr_RA, load.RFaddr_RB, load.RFaddr_RD, ra.out, rb.out, rd.out
       ((load.RFaddr_RA eql 1) or (load.RFaddr_RB eql 1) or
        (load.RFaddr_RD eql 1))):=(
        delay(1);       /* Insert propagation delay */
        busRD=(ra.out and (load.RFaddr_RA sxt 7)) or
              (rb.out and (load.RFaddr_RB sxt 7)) or
              (rd.out and (load.RFaddr_RD sxt 7));
)
```

sdec.isp

```
/*
 *      sdec.isp - Shifter Decoder
 */

macro
        WORD    :=      32&;

state
        SHam<5>;

port
        phi1 'input,
        phi2 'input,
        phi3 'input,

        busIMM<5> 'input,
        busB<WORD> 'input,      /* inverted (but actually true) */
        busBAR<2> 'input,
        busSDEC<7> 'bidirectional,
        ctrl.badshift 'output,

        load.SHam_IMM 'input,
        load.SHam_busB 'input,
        load.SHam_0 'input,
        load.SHam_BAR 'input;

when  (phi1, load.SHam_0
      (phi1 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        busSDEC<0>=load.SHam_0;         /* 0 */
        busSDEC<1>=0;                   /* 1 */
        busSDEC<2>=0;                   /* 2 */
        busSDEC<3>=0;                   /* 8 */
        busSDEC<4>=not load.SHam_0;     /* 13 */
        busSDEC<5>=0;                   /* 16 */
        busSDEC<6>=0;                   /* 24 */
)

when  (phi2, load.SHam_IMM, load.SHam_busB, load.SHam_0, busIMM, busB
      (((load.SHam_IMM eql 1) or (load.SHam_busB eql 1) or
        (load.SHam_0 eql 1)) and (phi2 eql 1))):=(
        delay(1);       /* Insert propagation delay */
        SHam=(busIMM and (load.SHam_IMM sxt 5)) or
             ((not busB<4:0>) and (load.SHam_busB sxt 5)) or
             (0 and (load.SHam_0 sxt 5));
)

when  (phi3, load.SHam_BAR, busBAR
      ((load.SHam_BAR eql 1) and (phi3 eql 1))):=(
        delay(1);       /* Insert propagation delay */
        SHam=(busBAR ext 5)*8;
)

when  (SHam):=(
        delay(1);       /* Insert propagation delay */
        ctrl.badshift=(SHam neq 0) and (SHam neq 1) and (SHam neq 2) and
                      (SHam neq 8) and (SHam neq 16) and (SHam neq 24);
)

when  (phi3, SHam
      (phi3 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        busSDEC<0>=(SHam and 0b11011) eql 0;    /* 0 */
        busSDEC<1>=(SHam and 0b11011) eql 1;    /* 1 */
        busSDEC<2>=(SHam and 0b11011) eql 2;    /* 2 */
        busSDEC<3>=(SHam and 0b11011) eql 8;    /* 8 */
        busSDEC<4>=0;                           /* 13 */
        busSDEC<5>=(SHam and 0b11011) eql 16;   /* 16 */
        busSDEC<6>=(SHam and 0b11011) eql 24;   /* 24 */
        delay(1);       /* Insert propagation delay */
)
```

sdr.isp

```
/*
 *      sdr.isp - Store Data Register
 */

macro
        WORD    :=      32&,
        RB      :=      4&;

state
        SDRdata1[0:RB]<WORD>,
        SDRdata2[0:RB]<WORD>,
        SDRvalid1[0:RB],
        SDRvalid2[0:RB];

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busD<WORD> 'input,
        busOUT<WORD> 'output:disconnect,

        state.rb 'input,
        state.RB<3> 'input,
        ctrl.SDR_write 'input,
        gate.SDR_busOUT 'input;

main:=(
        SDRvalid1[RB]=1;
```

```
          SDRvalid2[RB]=1;
          next;
          terminate;
  )

  when (phi3, state.rb, state.RB
        ((phi3 eql 1) and (state.rb eql 1))):=(
          state
                  sdr<32>;

          delay(1);        /* Insert propagation delay */
          sdr=0;
          next;
          while ((sdr lss (state.RB ext 32)) and (sdr lss RB)) (
                  SDRvalid1[sdr]=0;
                  SDRvalid2[sdr]=0;
                  sdr=sdr+1;
          );
  )

  when (phi3:trail):=(
          state
                  sdr<32>;

          sdr=1;
          next;
          while (sdr lss RB) (
                  SDRdata1[sdr]=SDRdata2[sdr-1];
                  SDRvalid1[sdr]=SDRvalid2[sdr-1];
                  sdr=sdr+1;
          );
          if (SDRvalid2[RB-1] eql 1) (
                  SDRdata1[RB]=SDRdata2[RB-1];
                  SDRdata2[RB]=SDRdata2[RB-1];
          );
  )

  when (phi4:trail):=(
          state
                  sdr<32>;

          SDRvalid1[0]=ctrl.SDR_write;
          SDRvalid2[0]=ctrl.SDR_write;
          sdr=1;
          next;
          while (sdr lss RB) (
                  SDRvalid2[sdr]=SDRvalid1[sdr];
                  sdr=sdr+1;
          );
  )

  when (phi1, busD
        (phi1 eql 1)):=(
          delay(1);        /* Insert propagation delay */
          SDRdata1[0]=busD;
          SDRdata2[0]=busD;
  )

  when (phi1:trail):=(
          state
                  sdr<32>;

          sdr=1;
          next;
          while (sdr lss RB) (
                  SDRdata2[sdr]=SDRdata1[sdr];
                  sdr=sdr+1;
          );
          sdr=0;
          next;
          while (SDRvalid2[sdr] eql 0)
                  sdr=sdr+1;
          busOUT=SDRdata2[sdr];
  )

  when (phi2, gate.SDR_busOUT
        ((phi2 eql 1) and (gate.SDR_busOUT eql 1))):=(
          delay(1);        /* Insert gate opening delay */
          connect(busOUT);
          wait(phi2:trail, gate.SDR_busOUT:trail);
          delay(1);        /* Insert gate closing delay */
          disconnect(busOUT);
  )
```

shift.isp

```
/*
 *      shift.isp - Shifter
 */

macro
        WORD    :=      32&;

state
        ctrl.SHIFT_fill,
        SHIFT.right<WORD>,
        SHIFT.left<WORD>,
        SHIFT.fill<WORD>;

port
        phi1 'input,
        phi2 'input,
        phi3 'input,
        phi4 'input,

        busS<WORD> 'input,
        busT<WORD> 'input,
        busSDEC<7> 'input,
        busDR<WORD> 'output:disconnect,
        busDL<WORD> 'output:disconnect,
        busR<WORD> 'output,

        load.SHIFT_busT 'input,
        load.SHIFT_busR 'input,
        load.SHIFT_busL 'input,
        load.SHIFT_IMM 'input,
        ctrl.SHIFT_sxtS 'input,
        ctrl.SHIFT_sxtT 'input,
        gate.SHIFT_busR 'input,
        gate.SHIFT_busL 'input;

when (phi1:lead, phi3:lead):=(
        delay(1);       /* Insert propagation delay */
        SHIFT.right=not 0;
        SHIFT.left=not 0;
        SHIFT.fill=not 0;
)

when (phi2, load.SHIFT_IMM, ctrl.SHIFT_fill, busS, busT
      (phi2 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        if (load.SHIFT_IMM eql 1)
                SHIFT.left=SHIFT.left and (not busT)
        else
                SHIFT.right=SHIFT.right and (not busS);
        if ((ctrl.SHIFT_fill eql 1) and (load.SHIFT_IMM eql 1))
                SHIFT.fill=0;
)

when (phi4, load.SHIFT_busT, load.SHIFT_busR, load.SHIFT_busL,
      ctrl.SHIFT_fill, busS, busT
      (phi4 eql 1)):=(
        delay(1);       /* Insert propagation delay */
        if (load.SHIFT_busR eql 1)
                SHIFT.right=SHIFT.right and (not busS);
        if ((load.SHIFT_busL eql 1) and (load.SHIFT_busT eql 1))
                SHIFT.left=SHIFT.left and (not busT) and (not busS)
        else if (load.SHIFT_busL eql 1)
                SHIFT.left=SHIFT.left and (not busS)
        else if (load.SHIFT_busT eql 1)
                SHIFT.left=SHIFT.left and (not busT);
        if (ctrl.SHIFT_fill eql 1)
                SHIFT.fill=0;
)

when (phi4, gate.SHIFT_busR
      ((phi4 eql 1) and (gate.SHIFT_busR eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busDR);
        wait(phi4:trail, gate.SHIFT_busR:trail);
        delay(1);       /* Insert gate closing delay */
        disconnect(busDR);
)

when (phi4, gate.SHIFT_busL
      ((phi4 eql 1) and (gate.SHIFT_busL eql 1))):=(
        delay(1);       /* Insert gate opening delay */
        connect(busDL);
        wait(phi4:trail, gate.SHIFT_busL:trail);
```

```
		delay(1);	/* Insert gate closing delay */
		disconnect(busDL);
	}

	when (busSDEC, SHIFT.right, SHIFT.left, SHIFT.fill):=(
		delay(1);	/* Insert propagation delay */
		if (busSDEC<0> eql 1) ( /* Shift 0 */
			SHIFT.right=SHIFT.right and SHIFT.left;
			next;
			SHIFT.left=SHIFT.right;
			next;
		);
		if (busSDEC<1> eql 1) ( /* Shift 1 */
			SHIFT.right=SHIFT.right and
				(SHIFT.fill<0:0> concat SHIFT.left<31:1>);
			next;
			SHIFT.left<31:1>=SHIFT.right;
			SHIFT.fill<0:0>=SHIFT.right<31:31>;
			next;
		);
		if (busSDEC<2> eql 1) ( /* Shift 2 */
			SHIFT.right=SHIFT.right and
				(SHIFT.fill<1:0> concat SHIFT.left<31:2>);
			next;
			SHIFT.left<31:2>=SHIFT.right;
			SHIFT.fill<1:0>=SHIFT.right<31:30>;
			next;
		);
		if (busSDEC<3> eql 1) ( /* Shift 8 */
			SHIFT.right=SHIFT.right and
				(SHIFT.fill<7:0> concat SHIFT.left<31:8>);
			next;
			SHIFT.left<31:8>=SHIFT.right;
			SHIFT.fill<7:0>=SHIFT.right<31:24>;
			next;
		);
		if (busSDEC<4> eql 1) ( /* Shift 13 */
			SHIFT.right=SHIFT.right and
				(SHIFT.fill<12:0> concat SHIFT.left<31:13>);
			next;
			SHIFT.left<31:13>=SHIFT.right;
			SHIFT.fill<12:0>=SHIFT.right<31:19>;
			next;
		);
		if (busSDEC<5> eql 1) ( /* Shift 16 */
			SHIFT.right=SHIFT.right and
				(SHIFT.fill<15:0> concat SHIFT.left<31:16>);
			next;
			SHIFT.left<31:16>=SHIFT.right;
			SHIFT.fill<15:0>=SHIFT.right<31:16>;
			next;
		);
		if (busSDEC<6> eql 1) ( /* Shift 24 */
			SHIFT.right=SHIFT.right and
				(SHIFT.fill<23:0> concat SHIFT.left<31:24>);
			next;
			SHIFT.left<31:24>=SHIFT.right;
			SHIFT.fill<23:0>=SHIFT.right<31:8>;
			next;
		);
}

when (SHIFT.right):=(
	delay(1);	/* Insert propagation delay */
	busDR=not SHIFT.right;
	busR=not SHIFT.right;
)

when (SHIFT.left):=(
	delay(1);	/* Insert propagation delay */
	busDL=not SHIFT.left;
)

when (ctrl.SHIFT_sxtS, busS, ctrl.SHIFT_sxtT, busT):=(
	delay(1);	/* Insert propagation delay */
	ctrl.SHIFT_fill=(ctrl.SHIFT_sxtS and busS<31>) or
			(ctrl.SHIFT_sxtT and busT<31>);
)
```