Chapter Three
Processor Architecture

3.1. Instruction Set

A subset of the DLX instruction set as presented in [Henn90] has been chosen for AMPIRE. The DLX RISC architecture is now widely studied in computer architecture classes, and it allows ease of implementation. Verilog models of DLX have been built at CMU [Siew92], and a VLSI implementation has been designed at the Montana State University with the Berkeley OCT tools and fabricated through MOSIS [Wint92].

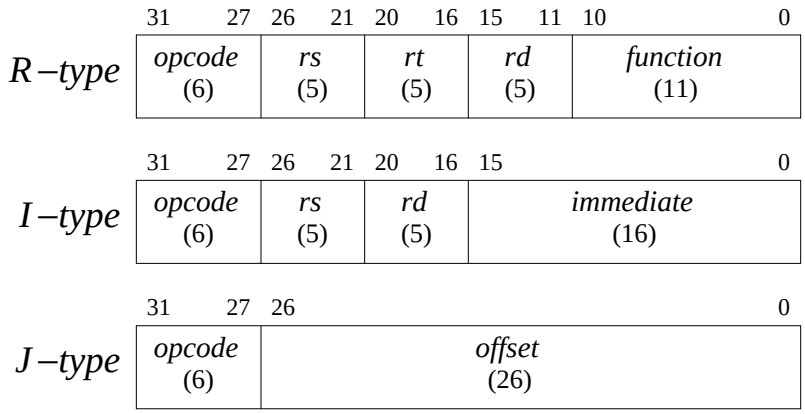

Figure 3.1: AMPIRE Instruction Format

The AMPIRE instruction format is shown in Figure 3.1, with minor notational changes from DLX [Henn90, p. 166]. All register-to-register ALU instructions share a single opcode number, and the specific ALU operations are encoded in the function field. The AMPIRE instruction set appears in Table 3.1, and their opcode/function code assignments can be found in the *parameter* listing in Appendix A. The opcodes for the DLX instructions were obtained from [Host91]. The new instructions for fault simulation are placed in slots unused by the DLX.

Data Transfer:		
LHI	rd, imm	Load high (upper half of register) with immediate
LW	rd, imm(rs)	Load word from Mem[rs+imm]
SW	imm(rs), rd	Store word to Mem[rs+imm]
Arithmetic/Logical (register):		
ADDU	rd, rs, rt	Add unsigned
SUBU	rd, rs, rt	Subtract unsigned (rs – rt)
AND	rd, rs, rt	Bitwise AND
OR	rd, rs, rt	Bitwise OR
XOR	rd, rs, rt	Bitwise XOR
SLL	rd, rs, rt	Shift left logical by (rt mod 32) bits
SRL	rd, rs, rt	Shift right logical by (rt mod 32) bits
SRA	rd, rs, rt	Shift right arithmetic by (rt mod 32) bits
SEQ	rd, rs, rt	Set if (rs == rt)
SNE	rd, rs, rt	Set if (rs != rt)
SLT	rd, rs, rt	Set if (rs < rt)
SGT	rd, rs, rt	Set if (rs > rt)
SLE	rd, rs, rt	Set if (rs <= rt)
SGE	rd, rs, rt	Set if (rs >= rt)
Arithmetic/Logical (immediate):		
ADDUI	rd, rs, imm	Add unsigned immediate
SUBUI	rd, rs, imm	Subtract unsigned immediate (rs – imm)
ANDI	rd, rs, imm	Bitwise AND immediate
ORI	rd, rs, imm	Bitwise OR immediate
XORI	rd, rs, imm	Bitwise XOR immediate
SLLI	rd, rs, imm	Shift left logical by (imm mod 32) bits
SRLI	rd, rs, imm	Shift right logical by (imm mod 32) bits
SRAI	rd, rs, imm	Shift right arithmetic by (imm mod 32) bits
SEQI	rd, rs, imm	Set if (rs == imm)
SNEI	rd, rs, imm	Set if (rs != imm)
SLTI	rd, rs, imm	Set if (rs < imm)
SGTI	rd, rs, imm	Set if (rs > imm)
SLEI	rd, rs, imm	Set if (rs <= imm)
SGEI	rd, rs, imm	Set if (rs >= imm)
Flow Control:		
BEQZ	rs, imm	Branch to (PC+4+imm) if (rs == 0)
BNEZ	rs, imm	Branch to (PC+4+imm) if (rs != 0)
J	offset	Jump to (PC+4+offset)
JR	rs	Jump to address in rs
JAL	offset	Jump to (PC+4+offset); store (PC+4) in R31
JALR	rs	Jump to address in rs; store (PC+4) in R31
Miscellaneous:		
NOP		No operation
ADDUF	rd, rs, rt	Add unsigned with fault (bad parity)
ADDUIF	rd, rs, imm	Add unsigned immediate with fault
JRF	rs	Jump to address in rs with fault
SWF	imm(rs), rd	Store word to Mem[rs+imm] with fault
TRAP	offset	Special simulation function

Table 3.1: AMPIRE Instruction Set

The immediate and offset values are sign-extended to 32 bits before the instructions are executed. The TRAP instruction is normally reserved for exception handling, but it is used by the AMPIRE simulator to print out register values and to terminate the simulation. Details will be discussed at the end of section 4.2.6.1.

3.2. Processor Overview

The block diagram of the processor is shown in Figure 3.2, with brief descriptions of the modules in Table 3.2. Delayed write buffers (DWBs) are used to postpone commitment to the register file and the data memory until all required checks are finished. Since each module has a different completion time based on the function being performed, queues are used to improve concurrency. Queues are transparent to the functional elements so that the number of buffers can be changed without modifying other parts of the processor.

3.3. Normal Operation

The asynchronous operations without fault tolerance features will be discussed first. The order of presentation will follow the flow of instructions through the various modules.

3.3.1. Instruction Fetch

When the processor is reset, the PC is set to 0, and an instruction fetch cycle is started. As soon as a memory cycle is completed (acknowledged), the PC is automatically incremented to start another cycle. This process continues until a branch occurs. Even though AMPIRE only supports 32-bit read/write, the unit of address is in bytes to be compatible with DLX. Therefore, the PC is incremented by four each time.

Module	Description
ALU	Arithmetic logic unit. Except for branch/jump computations, all other arithmetic and logic operations are handled by this module.
ARBK	Arbiter for accessing the K_bus (not shown). It controls outputs from the four checkers.
ARBR	Arbiter for accessing the R_bus.
BIGC	Big C-element (not shown) for rollback synchronization.
CKF	Checker for REGDWB outputs to A_bus and B_bus.
CKI	Checker for instructions executed by the IIU.
CKM	Checker for data written to MEMDWB.
CKR	Checker for data written to REGDWB.
DMEM	Data memory, single-port, organized as 32-bit words.
DQ	Data queue for memory-to-register transfers.
IIU	Instruction issuing unit. It is the main controller that decodes instructions, reads data from the register file, and dispatches operation requests to other modules.
IMEM	Instruction memory, read-only, organized as 32-bit words.
IQ	Instruction queue.
LOG	Instruction log, where uncommitted instructions are kept. All checkers send their results to the log, and the log issues validation signals or initiates rollback.
MEMDWB	Delayed write buffer for data memory operations.
PC	Program counter. It starts at 0 when the processor is reset, and it is automatically incremented after each instruction fetch.
REGDWB	Delayed write buffer for register file operations.
REGFILE	Register file, with two read ports and one write port. There are thirty two 32-bit general purpose registers, with R0 being a constant of zero.
RESTABLE	Register reservation table. All registers being read or written must be cleared by the reservation table first.

Table 3.2: Processor Modules

After an instruction is retrieved from memory, IMEM sends a write request to the instruction queue. The time required to read an instruction from memory does not change much, even for real self-timed memory, unless it actually contains multiple elements with different access times. On the other hand, the instruction cycles are variable, from very short NOP to long delay for reservation clearance (discussed in the next section). With the IQ, multiple instructions can be pre-fetched during long

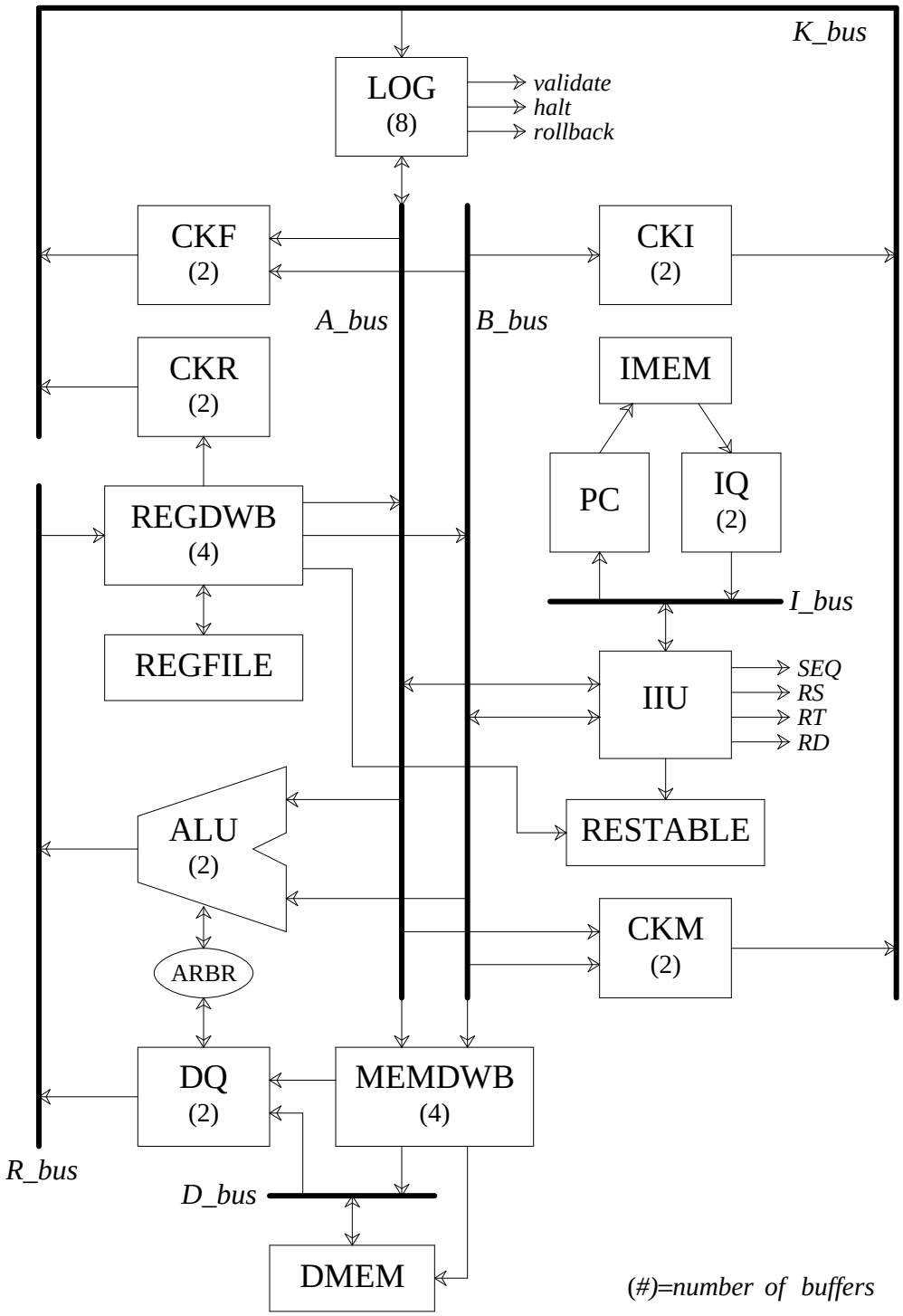

Figure 3.2: AMPIRE Block Diagram

instruction cycles, and new instructions can be made available quickly following short cycles.

When a branch occurs, the pre-fetched instructions have to be invalidated. The IIU disables the output buffer of the IQ and sends a new address to the PC via I_bus. The PC_load signal also causes IMEM and IQ to drop all current transactions, and the IQ is cleared. The IQ size has been chosen to be two so that the latency through IQ is not excessive after a branch is taken.

Since the speed of IMEM and the size of IQ are both unknown to the rest of the processor, the value of the PC module cannot be used to determine the address of an instruction being executed. One solution, as used in AMPIRE, is to add a separate program counter inside the IIU for instruction logging and branch computations. Whereas the PC module is incremented after every IMEM access, this internal counter is incremented when an instruction is read from the IQ, like a synchronous processor in every instruction fetch stage. In this respect, the PC module is very similar to the remote program counter as discussed in [Patt83]. When a branch occurs, the value of the internal program counter is used to calculate the destination address, and both program counters are loaded with the same new address.

An alternative to having a program counter in the IIU is to store the instructions and their addresses in IQ. This way, the IIU simply reads both the instruction and its address at the same time. However, since the IQ can be arbitrarily long, the costs of additional storage elements for the buffers and wires for routing 32-bit addresses can be quite high.

3.3.2. Instruction Issue

An instruction issue cycle is started when the IIU accepts an instruction presented by the IQ. Part of the instruction decoding process determines which registers need to be read or written, and then they are sent to the reservation table for clearance. Because the processor is asynchronous, the time required to execute an instruction cannot be pre-determined, and even the order of instruction completion is unknown. Therefore, a compiler may not be able to schedule instructions correctly, and data hazard avoidance has to be handled by the processor.

Before an instruction *IY* that writes to *R1* can be issued, there must not be another instruction *IX* that also modifies *R1* in the pipeline. Otherwise, *IY* may be completed before *IX* and causes a write-after-write error. Reading operation is similar; any previous instruction that writes to *R1* must be completed before *R1* can be accessed again. In fact, all instructions being executed at the same time must be independent of each other. Note that *completion* does not imply *commitment* to the register file.

When a reservation request is received at the RESTABLE, the reservation bits for all source and destination registers are checked. If all of them are clear, then the destination register is reserved, and the reservation request is acknowledged. Otherwise, the acknowledgement is delayed until the appropriate bits are cleared by the REGDWB. Since R0 is a zero constant, it is always available.

Even though PC and data memory are also state elements, reservations are not needed. Each PC operation is executed directly and immediately by the IIU, and since AMPIRE is a load-store machine, memory transfers occur between the data memory and the register file only. Memory write operations are carried out sequentially through the MEMDWB, and memory read operations have register destinations which are checked by the normal reservation process.

After reservation, the source registers need to be read from the register file. Even though the IIU actually communicates with the REGDWB, the existence of REGDWB is transparent to the IIU. The operations within REGDWB will be discussed later.

Instructions can be divided into three categories: ALU, data memory, and flow control. ALU and data memory instructions are sent to ALU and MEMDWB, respectively, and branches/jumps are executed by the IIU itself. Sign extension of all immediate and offset values are also performed by the IIU because they are required for branch and data memory address computations.

3.3.3. Instruction Execution

3.3.3.1. ALU

ALU operations are pretty straight forward. Two operands (except one for LHI and PASS) and a function code are used to produce a result that is to be written to a destination register. No condition code is used in DLX nor in AMPIRE.

Unlike an ALU for a synchronous processor, there is no pre-determined time limit for each operation. For example, a bitwise OR can be done very quickly, but a simple ripple-carry adder needs much longer time for carry propagation. Even though a synchronous processor can allocate multiple cycles for time consuming operations, other components in the pipeline must be designed to accommodate it.

3.3.3.2. Data Memory Load/Store

All data memory read and write operations are controlled by the MEMDWB. Address and data (for SW instructions) are sent from the IIU via A_bus and B_bus, respectively. Based on today's technology, a memory device is generally slower than the speed of a processor. After a data memory request is accepted by the MEMDWB, the IIU can start working on the next instruction instead of waiting for the DMEM. Therefore, MEMDWB is useful even if fault tolerance is not needed.

Memory writes are queued in the buffers (first-in, first-out) until DMEM is ready for the next transaction. Each MEMDWB entry contains the data to be written and its destination address. For read operations, the address to be read is compared with the ones in the buffers. If a match is found, data is retrieved from the appropriate buffer and sent to the DQ, without accessing DMEM. If there is more than one match, a priority decoder selects the most recent value. If there is no match, then data is read from DMEM in the next memory cycle, even if other write operations are waiting in the queue. Memory reads are given priority over memory writes so that the reserved destination registers are cleared as quickly as possible to improve concurrency. Each DQ buffer entry includes the memory data and its target register number. The register number is passed directly from the MEMDWB controller to the DQ, without going through the MEMDWB buffer elements and DMEM.

A drawback for this process is the need for associative comparators. With 32-bit addresses, the overhead can be quite high. Another penalty is that before comparisons can be made, the propagation of data in the queue buffers must be stopped, which may increase the latency through the queue. This side effect will be discussed in detail in section 3.5.

If associative comparators are not desirable, an alternative is to put both memory

read and write requests in the same queue, and process them in the same order as they are issued. Because there is no comparator to check whether the memory data is already in the MEMDWB, memory read cannot be assigned a higher priority than memory write. Since multiple read requests can be accumulated in the queue, each MEMDWB buffer also has to store two additional pieces of data: destination register number and a read/write mode bit. The IIU can still go to the next instruction, but memory read operations have to wait until all previous memory requests have been processed by DMEM. As will be discussed in the fault tolerance section, memory writes are held in the MEMDWB until the data have been validated, which means longer delay for *both* read and write operations. This method results in simpler hardware because address comparators are not used, but performance is lower because of longer reservation waiting period for memory reads, and higher memory traffic.

3.3.3.3. Flow Control

Flow control instructions include conditional branches and unconditional jumps. These are handled by the IIU directly so that the ALU can be dedicated to perform "real" computations. This is similar to how branches are handled in the Berkeley DSP [Jaco90]. Because branches are based on register comparisons instead of condition codes, they are independent of previous ALU operations except when the registers are reserved. The IIU does require an internal adder for address calculations.

It is not clear whether the DLX architecture has delayed branch, but for simplicity, AMPIRE does not support it. Delayed branch is generally used in a synchronous processor to reduce branch penalties by minimizing pipeline stalls. Although an asynchronous processor may also benefit from having delayed branch, it makes rollback more difficult because looking at the instruction in the delay slot alone does

not reveal any information about the branch instruction associated with it. Furthermore, statistics show that less than 50% of the delay slots are usefully filled [Henn90, p. 276].

However, since delayed branch usually improves performance [Henn90, p. 277], a method to support it in AMPIRE is briefly described here. The reader should go through section 3.4 on fault tolerance before reading this and the next paragraph. One way to handle both delayed branch and rollback is to maintain two sets of sequence number and check vector for each branch in the instruction log. For example, a branch $INST_b$ has sequence SEQ_b and check vector V_b, and the delay slot $INST_d$ has sequence SEQ_d and check vector V_d. The instruction log has to store both sequence numbers SEQ_b SEQ_d and both check vectors V_b V_d with the branch instruction $INST_b$. The entry for the delay slot $INST_d$ is not changed. When a checker clears a bit in vector V_d, two bits are cleared in the LOG, one for each instruction $INST_b$ and $INST_d$. This way, the branch instruction cannot be validated until all the check bits for its delay slot are also cleared. The delay slot is already prevented from being validated before its branch by the FIFO nature of the LOG.

When either instruction $INST_b$ or $INST_d$ is to be invalidated, the processor has to be rolled back to the branch $INST_b$. This means the instruction log has to include a priority circuit to send out the address for $INST_b$, not for $INST_d$. The LOG also needs to have more buffer elements and comparators to maintain the additional sequence number and check vector for *each* log entry. In addition, the IIU and/or the LOG control circuit becomes more complicated because the check vector for $INST_d$ is not available until that instruction is fetched and decoded, but it must be stored with $INST_b$ in the LOG.

3.3.4. Register File Access

After an ALU computation or memory retrieval is completed, the R_bus is used to transfer data to the register file. Since both ALU and DQ may have data ready at any time, the ARBR arbiter is needed to coordinate access to the R_bus. If only one request is detected, then that request is granted, and the module can start its transaction with the REGDWB. If both requests are received at the same time, the one which just had the bus access has to wait until the other module finishes one transaction.

The DQ helps to reduce the DMEM read cycle time by separating the R_bus arbitration stage from the data memory. Since the R_bus may be busy when the data read from DMEM is ready, the DQ buffer allows the read transaction to complete so that the DMEM can start the next memory operation. This is important because memory access is likely to be the slowest activity in a processor. The overall memory read time (from IIU to REGDWB) also depends on the depth of DQ. While the DQ reduces the DMEM read cycle time, it also adds propagation delay through the queue. The penalty is the highest for isolated single DMEM read operation, but for multiple DMEM reads in a row, the average performance may be improved due to concurrency between DMEM and DQ. The depth of DQ needs to be selected based on the expected memory activity, but in general it should be kept short so that the worst-case penalty is not excessive.

After a piece of data is written to the REGDWB, the corresponding register reservation is cleared by sending the register number to RESTABLE. This can be done before the actual register is updated because the REGDWB controls both read and write operations. When a read request is initiated by the IIU, the REGFILE is read while the REGDWB is searched for a match. This is different from MEMDWB because DMEM is read only if there is a miss in MEMDWB. The register file has two read ports and

one write port, and it is designed to be read and written every instruction cycle. This is not practical for a large and slow memory like DMEM, so MEMDWB is optimized to reduce memory traffic.

Since the same register in the REGFILE can be read and written at the same time, and these two operations occur asynchronously, care must be taken to prevent wrong (partially written) data from being presented to the requester. Because the data in the REGDWB is not erased until its write cycle is completed, reading the register still being written to the REGFILE will also result in a match in the REGDWB. By sorting all matches in REGDWB and REGFILE through a priority decoder, with REGFILE having the lowest priority, the correct and most recent data is given to the IIU. The data read from REGFILE is effectively ignored in this case.

3.4. Fault Tolerance

In this section, the instruction rollback process and the hardware features needed to support it will be described.

3.4.1. Sequence Number and Check Vector

Because each module in AMPIRE runs at its own rate, the sequence of events, such as the order of completion, is not predictable. Therefore, a sequence number has to be assigned to each instruction when it is issued by the IIU. All storage elements for uncommitted data, including intermediate buffers, the RESTABLE, and DWBs, must store the sequence numbers along with their data. For example, the instruction $R3{=}R1{+}R2$ has the sequence number SEQ_x. When the values of $R1$ and $R2$ are sent to the ALU, SEQ_x is sent there, too. When the result for $R3$ is delivered to REGDWB, the same sequence number SEQ_x is stored in its buffer, since that data is related to the

same instruction. When a rollback is to be done, the processor can determine which instructions have to be invalidated by checking their sequence numbers. The number of bits that needs to be assigned to the sequence number is determined by the maximum number of uncommitted instructions that is desired:

$$uncommitted_{\max} = 2^{(number\ of\ sequence\ bits\ -\ 1)}$$

An extra bit is used so that a quick comparison can be made to determine whether an instruction X comes before or after instruction Y. For example, if 3 bits are used for the sequence number, then there can be up to 4 outstanding instructions, or half of 3-bit combinations. Figure 3.3 contains graphical illustrations of the sequence number "windows".

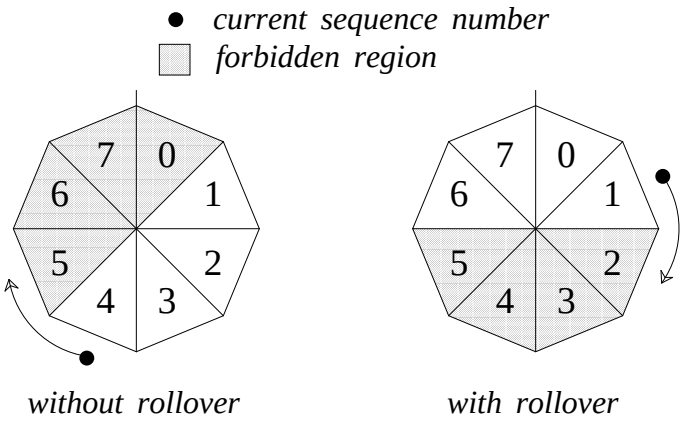

Figure 3.3: Sequence Number Windows

If n is the number of bits allocated for the sequence number, then there are $N=2^n$ number of possible sequence numbers, and the maximum number of uncommitted instructions is $\frac{N}{2}$. At any point in time, the sequence numbers of the most recent $\frac{N}{2}$ instructions are:

$$k,\ (k+1)\ mod\ N,\ (k+2)\ mod\ N,\ .....\ (k+\frac{N}{2}-1)\ mod\ N,\quad \text{where } 0 \le k < N.$$

An instruction $INST_x$ is the same as, or newer instruction than $INST_y$ if and only if:

$$SEQ_x = (SEQ_y + j)\ mod\ N, \quad \text{where } 0 \le j \le \frac{N}{2} - 1.$$

Hence, $j = (SEQ_x - SEQ_y)\ mod\ N$, which can be obtained using an n-bit subtractor. The condition $0 \le j \le \frac{N}{2} - 1$ holds true if and only if the most significant bit (MSb) of j is 0. Therefore, if the MSb of the subtraction result is 0, $INST_x$ is the same as or newer than $INST_y$. Otherwise, $INST_x$ precedes $INST_y$. This result is used to determine if an instruction should be validated or not in the rollback process.

Another piece of information that has to be determined when the instruction is issued is the verification steps it must go through before that instruction can be declared valid. For example, all instructions have to pass the instruction checker CKI, but only memory operations need to be checked by CKM. A check vector is the collection of check bits, with each bit identifying a particular checker. An example of a check vector for ALU operation is shown in Figure 3.4. When all the required checks are verified, that instruction can be validated. The check vectors are stored and updated in the instruction log.

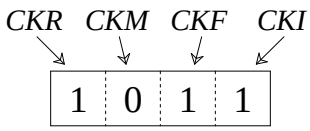

Figure 3.4: Check Vector for ALU Operation

3.4.2. Error Detection

Each data word and address in the AMPIRE is protected by an even parity bit. This simple mechanism is used only to show how rollback is done when an error is detected. If more protection is desired, better error detection or correction codes can be used instead. For storage elements such as memories and queue buffers, the parity bit is simply passed along with the data to the destination. It is up to the receiver to determine the data integrity. When an instruction or the register file is read, the IIU sends a request to the CKI or CKF checker, respectively. The two other checkers, CKM and CKR, are used by MEMDWB and REGDWB to verify the data requests received by them.

When a module performs a computation, a new parity bit is generated. This is done in the PC each time the address is incremented, in the IIU for address calculation and sign extension, and in the ALU before each result is sent to the register file. The checkers decide whether there is an error by XORing all the bits. Since even parity bit is used, a result of 1 would indicate an error. Again, more elaborate schemes can be integrated into the processor.

All checkers share the K_bus for reporting their results to the instruction log, and the ARBK arbiter is used to grant bus access to each checker in a round-robin fashion. The components of the K_bus are the sequence number, the checker identification number, and the pass/fail signal.

3.4.3. Instruction Log and Validation

Uncommitted instructions are stored here, and all instruction validations and rollbacks are determined by this module. The log contains 8 entries, so 4 bits have to be used for the sequence number, as discussed in section 3.4.1. Each log entry has a set of data for an instruction: sequence number, instruction address, and check vector. The instructions themselves are not stored in the log because they are read from IMEM again in the event of rollback.

When a checker reports a "pass", the appropriate check bit for an instruction is cleared. When all of the check bits for the *oldest* entry in the log become zeros, then that instruction can be validated and deleted from the log. The first-in first-out sequence must be maintained because once an instruction is validated, it cannot be *un*-validated. Because of the asynchrony, $INST_{x+1}$ may pass all its checkpoints before $INST_x$ does, but if an error occurs in $INST_x$, $INST_{x+1}$ must be undone.

An instruction is validated by sending the sequence number to the DWB(s) so that the data waiting in the buffer can be written to its permanent location. If there is a piece of data ahead of it in the queue, whether validated or not, then the commitment process has to be delayed so that write-after-write hazard would not occur. If an instruction does not modify a storage element, then no validation signal needs to be sent. In AMPIRE, an instruction can modify at most one register *or* memory location. Therefore, the LOG only needs to send the validation signal to one DWB and not disturb other modules. The added cost is an extra handshaking wire, but the other DWB not directly involved in the validation is not slowed down for unnecessary comparisons. This performance decrease will be addressed in section 3.5. However, if there are many DWBs in the processor, the trade-offs may have to be re-considered.

It is important to note that when the validation is issued by the LOG, the entry to be validated must already be in the appropriate DWB. This is required because buffering a validation request and waiting for the appropriate data make this problem more complicated than necessary. This sequence is ensured by sending the checker request *after* the data is placed in the DWB queue. Without a "pass" signal from the checker, the instruction cannot be validated. If there is no checker associated with a DWB, then a check bit is still used so that the LOG cannot validate an instruction without receiving an "arrived" signal from the DWB. The CKI checker is placed on the B_bus instead of the I_bus for the same reason. The IIU first logs an instruction before sending it to the CKI so that when the checker notifies the LOG, the entry is already in the LOG to be processed.

3.4.4. Rollback

If a checker reports an error, then that instruction, and *all* instructions issued after it, have to be invalidated. The instruction at which the error occurred is re-executed by loading the PC with its address from the LOG. The rollback request is sent by the LOG to all modules except PC, IMEM, and IQ, since they can be easily cleared by the IIU as if a branch is taken. The high-level signal flow diagram of validation and rollback is shown in Figure 3.5.

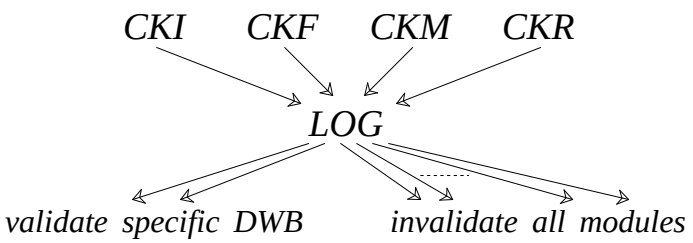

Figure 3.5: Validation and Rollback Signal Flow Diagram

The instructions and operations which have to be invalidated are determined by subtracting the error sequence number from the sequence numbers in all of the buffers. If the high bit of a result is 0, then that entry is either the erroneous one or a more recent one, and it is erased. If the high bit is 1, than that operation is not affected by the error. An invalidation request is sent to all buffer elements at the same time, and the comparison process is done in parallel. After each buffer is finished, the acknowledgement signals from the individual elements are grouped together by one or more C-elements, from which a module-level acknowledgement signal is generated and sent back to the LOG.

Since the IIU and all transactions with the IIU (such as reading the register file) are always working on the most recent instruction, they can be invalidated without any comparison with the error sequence number. The reservation table is a special case that deserves attention, since there is no DWB for it. When a reservation is made, the sequence number is stored in the corresponding entry in the table. Rolling back the RESTABLE simply means clearing the appropriate reservation bits based on the same sequence number comparisons.

Before the rollback can actually take place, all components in the processor must be synchronized. This is the topic of section 3.5. Further details of the rollback process will be addressed in section 4.1.6 when the Verilog code is discussed.

3.4.5. Delayed Write Buffer

Since the DWB is an important element that allows rollback to be done, its hardware structure is summarized here. The DWB for REGFILE is shown in Figure 3.6, with an expanded view of the DWB buffer element. It is an extension of the synchronous DWB shown in Figure 2.8, with two additional fields: sequence number

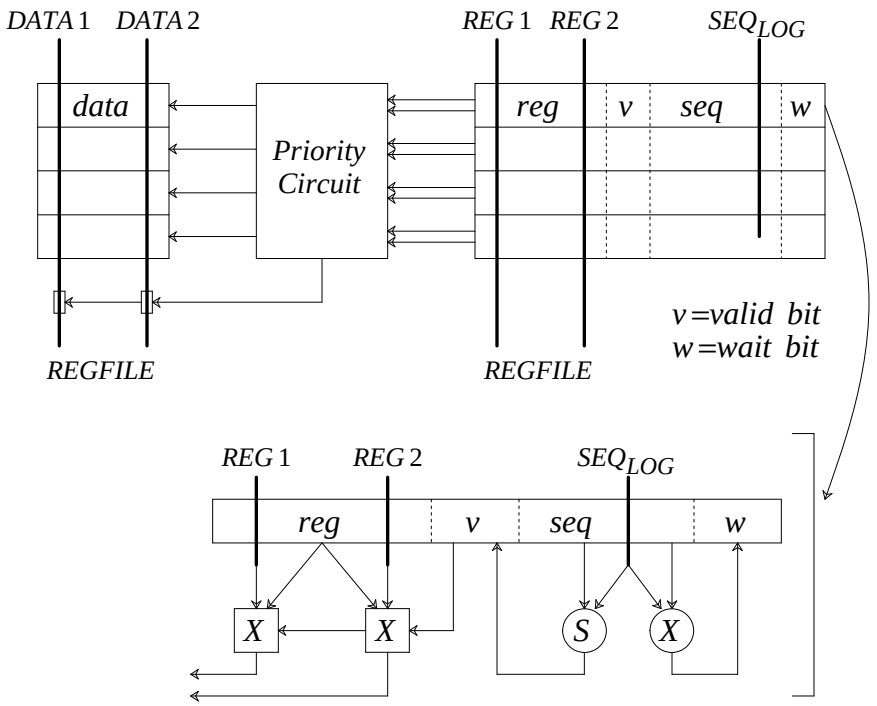

Figure 3.6: Asynchronous Delayed Write Buffer for Register File

and the wait bit.

The X elements are comparators, and the S element is a subtractor as described before. The bus SEQ_{LOG} is driven by the instruction log during validation and rollback cycles. Since these two activities are mutually exclusive, only one bus is needed. When an instruction is validated, if $seq = SEQ_{LOG}$, then the wait bit is cleared, and the data can be written to the register file if all the entries ahead of it are also cleared. During the rollback cycle, the subtractor S determines if that entry should be invalidated by controlling the valid bit.

The two X comparators in squares already exist in the synchronous DWB, in the CAM section shown in Figure 2.8. For MEMDWB, the two DATA buses are reduced to one, and the two REG buses are replaced by a single memory address bus.

3.5. Synchronization

Because all of the operations in AMPIRE are asynchronous, care must be taken when a transaction requires more than two components to cooperate.

3.5.1. Module Level

AMPIRE contains a lot of buffer elements to balance the different processing speeds of the various modules, and data can be transferred from one buffer to another at any time. Many operations involve searching all the buffers in a queue, such as matching a memory address in the MEMDWB. A comparison cannot be made when a piece of data is "moving", and this is true for both synchronous and asynchronous systems.

A comparison cycle is started by requesting all buffers in a queue to suspend their transactions. After all buffers have acknowledged, then another request is sent to collect the comparison results, and the interrupted transactions may continue. This additional delay is the reason that unnecessary search requests should be avoided, as done in the instruction validation scheme.

3.5.2. Processor Level

Global synchronization is required for the rollback process because all modules must invalidate the erroneous data. When an error is detected, the LOG first sends out a halt request to all the modules, except the ones in the instruction fetch stage (PC, IMEM, and IQ). Each module is responsible for making sure all of its data transactions are suspended before the acknowledgement is returned. After all modules have responded, another signal is sent by the LOG for invalidation, performing the real rollback. The processor is restarted only when the LOG explicitly releases the halt

signal, after all rollback steps are completed.

The sequence of halt and rollback events are shown in Figure 3.7. Each request signal is a single wire from the LOG routed to all modules. Each group of the acknowledgement signals is fed into the inputs of a large C-element (BIGC), and the single output goes back to the LOG. Therefore, the LOG has no knowledge of the number of modules in the processor. Since the C-element function is associative, the BIGC can be physically distributed as smaller C-elements.

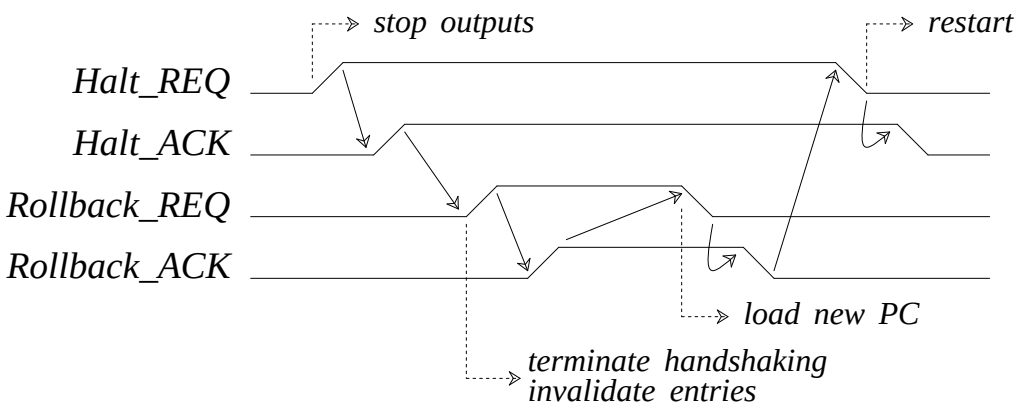

Figure 3.7: Halt and Rollback Sequence