Chapter Four
Behavioral Modeling

In order to verify that the AMPIRE design is architecturally correct, a behavioral model of the processor has been built with the Cadence Verilog hardware description language. Some selected modules and submodules have also been designed and simulated at the gate level to show how they may be implemented in VLSI, and they will be discussed in Chapter 6.

The complete Verilog code can be found in Appendix A. Verilog resembles procedural programming languages such as C and Pascal, with extensions for hardware simulation. Some frequently used statements and expressions are listed in Table 4.1 for reference. For detailed information, please see [Cade91].

Keyword	Explanation
#	Delay execution for a specified time.
@	Delay execution until an event occurs (edge sensitive).
always	Statement is executed repeatedly until simulation is terminated.
disable	Abruptly terminate a block of statements.
fork-join	Statements within the structure are executed in parallel.
initial	Statement is executed once when simulation is started.
reg	A register variable that holds value.
wait	Delay execution until expression becomes true (level sensitive).
wire	A net for signal declaration and connection.

Table 4.1: Frequently Used Verilog Keywords

4.1. Building Blocks

The basic code elements and techniques which are widely used in the various processor modules will be presented in this section.

4.1.1. Tri-State Bus

An output can be tri-stated by assigning a high-impedance value *z* to the output variable, and the following is the syntax for conditional assignment:

```
variable = expression ? value if true : value if false;
```

Therefore, a tri-state bus connection (assuming 33-bit wide) can be specified in Verilog *continuous assignment* as:

```
assign bus_variable = enable_signal ? output_value : 33'b z;
```

This way, multiple sources can be connected to the same bus, with the restriction that at most one may be enabled at any time.

4.1.2. Muller C-Element

The following code simulates a 2-input C-element:

```
always @(input_1 or input_2)
      if (input_1 & input_2)
            output = 1;
      else if (~input_1 & ~input_2)
            output = 0;
```

The first line activates the *always* block only when either input changes its value. The rest of the code simply follows the behavior of the C-element. When both inputs are high, the output is high; when both inputs are low, the output becomes low. The code segment can be easily expanded to handle C-elements with more than two inputs. An example is the *bigc.v* that has 14 input ports. Since the C-element contains state information, its state (output) has to be initialized in the reset routine.

4.1.3. Buffer

Many buffers are used in AMPIRE to improve concurrency because each module runs at its own speed. Buffers are also needed to store uncommitted instructions and data. The basic buffer structure is shown below, divided into input and output sections.

```
always wait (req_in & ~valid)
begin :input_cycle
      #1;
      data_out = data_in;
      ack_in = 1;
      valid = 1;
      wait (~req_in);
      #1;
      ack_in = 0;
end

always wait (valid)
begin :output_cycle
      #1;
      req_out = 1;
      wait (ack_out);
      #1;
      valid = 0;
      req_out = 0;
      wait (~ack_out);
end
```

Each input and output cycle follow the 4-phase handshake convention discussed in section 2.1, and each one can be considered as a process executed in parallel. The input cycle is started by receiving an input request REQ_{in}. When data is latched, the input is acknowledged, and the output cycle is initiated by setting the *valid* flag. The input cycle is also guarded by $\overline{valid}$ so that a new input would not be accepted until the previous data has been sent to another unit. The #1 statements simulate the delays between the input and output signals, and they allow the Verilog simulation clock to advance so that the sequence of events can be observed.

A simple asynchronous queue can be built by stacking the buffer elements. A piece of data added to the queue would ripple toward the other end until it hits another

data entry. The queues used in AMPIRE require additional controls because of the need to support rollback and other functions, and they will be discussed in separate sections. As with the C-element, the internal variable *valid* and the external output signals have to be initialized when the processor is reset.

4.1.4. Reset

Just like most digital systems, some state elements have to be initialized when the system is reset. The following reset routine provides the necessary initialization steps for both the C-element and the buffer discussed in the previous two sections:

```
always wait (reset)
begin
      disable input_cycle;
      disable output_cycle;
      valid = 0;
      ack_in = 0;
      req_out = 0;
      output = 0;        // for C-element
      wait (~reset);
end
```

Disable is a convenient way for an interrupt handler, such as this reset routine, to terminate other concurrent procedures. These *disable* commands are needed because a reset may occur in the middle of a data transaction, not just when "powered-on". Since the various modules may receive the *reset* signal at different times, and the time required to complete the reset process may be different, the disabled routines must not continue until the *reset* signal is withdrawn. This can be done by adding $\overline{reset}$ as one of the enabling conditions:

```
always wait (req_in & ~valid & ~reset)
always wait (valid & ~reset)
```

Since no handshake is involved in the reset process, the *reset* signal must be applied long enough for *all* modules to recognize it and complete the initialization. For

convenience, all reset routines in the behavioral level are executed in zero simulation time. However, gate-level modules do have minimum reset pulse-width requirements, which will be discussed in Chapter 6.

4.1.5. Arbiter

An arbiter is needed whenever two or more devices may want to access the same resource at the same time, which can happen because of the asynchronous nature of AMPIRE. As described before, the two bus arbiters ARBK and ARBR control K_bus and R_bus, respectively. Some modules, such as LOG and MEMDWB, also require internal arbiters for sequencing control.

Arbiters can be divided into two categories: prioritized and non-prioritized. Non-prioritized arbiter is fair and operates in a round-robin fashion. It can be represented in the following code:

```
always wait (req1 | req2 | req3)
begin
      found = 0;
      while (~found)
            begin
                  grant = grant + 1;
                  if (grant >= 4)
                        grant = 1;
                  case (grant)
                        1: if (req1)  found = 1;
                        2: if (req2)  found = 1;
                        3: if (req3)  found = 1;
                  endcase
            end
      [process grant handshake]
end
```

If there are only two request lines, the code can be simplified as shown in the *arbr.v* listing in Appendix A. Sometimes a prioritized arbiter should be used because of system requirements, or just for better system performance, as done in *log.v* and *memdwb.v*. Some simple nested if-else statements can be used to describe its behavior:

```
always wait (req1 | req2 | req3)
begin
      if (req1)
                  [process request 1]
      else
            if (req2)
                  [process request 2]
            else
                  [process request 3]
end
```

4.1.6. Rollback

The rollback scheme was introduced in the last chapter, but some details were left off because looking at the behavioral code should provide more insight on how the rollback works. The following is the basic structure:

```
always wait (halt_req)
begin :rollback_cycle
      #1;
      halt_ack = 1;
      wait (rollback_req);
      #1;
      disable input_cycle;
      disable output_cycle;
      ack_in = 0;
      req_out = 0;

      diff = sequence_num - sequence_error;
      if (~diff [high_bit])
            valid = 0;

      rollback_ack = 1;
      [finish halt and rollback transactions]
end
```

The rollback cycle follows the sequence of events shown in Figure 3.7. Each module (and sub-module) that needs to roll back must provide its own rollback handler that supports this protocol. After the halt request is received, the current data transactions are stopped by disabling the handshake outputs. The halt acknowledgement is sent without any feedback from the I/O routine/circuitry because local signal propagation can be easily controlled. The actual instruction invalidation is

not performed until the rollback request is given, which means all modules in the processor have acknowledged the halt request issued by the instruction log.

Whether a data entry should be deleted is determined by the sequence number comparison, as discussed in sections 3.4.1 and 3.4.4. If the high bit of the result is zero, then the entry is invalidated. After the rollback of this module is completed, an acknowledgement is sent, and the rest of the code just finishes the 4-phase halt and rollback handshake cycles.

Suspending input and output cycles as part of the rollback process can be accomplished by adding *wait* ($\overline{halt_req}$) statements before each group of output commands, as shown below:

```
always wait (req_in & ~valid & ~halt_req)
begin :input_cycle
      #1;
      wait (~halt_req);
      data_out = data_in;
      ack_in = 1;
      valid = 1;
      wait (~req_in);
      #1;
      wait (~halt_req);
      ack_in = 0;
end

always wait (valid & ~halt_req)
begin :output_cycle
      #1;
      wait (~halt_req);
      req_out = 1;
      wait (ack_out);
      #1;
      valid = 0;
      wait (~halt_req);
      req_out = 0;
      wait (~ack_out);
end
```

As can be seen in the code or in the 4-phase handshake diagram shown in Figure 2.2, the input cycle alternates between waiting for request and sending

acknowledgement, and vice versa for the output cycle. Once the output cycle issues a request, it cannot drop the request abruptly without receiving an acknowledgement first, or the handshake protocol would be violated. The only exceptions are: (1) when both parties of the transaction are notified before the cancellation, as in the case of rollback, or (2) when a local or global reset occurs, as for the instruction fetch modules (PC, IMEM, and IQ) during a branch.

The rollback process deletes all the data associated with the erroneous and more recent instructions. There is no problem with that because every data entry is tagged with a sequence number. However, care also must be taken to ensure that the processor is in a correct state for the data *not* invalidated in the rollback. Specifically, there must not be any deadlock, data loss, or data duplication. The three cases are discussed below:

- There is no deadlock because *all* transactions are cleared during rollback and restarted afterwards if necessary. Therefore, no module is left in the middle of a handshake cycle waiting for a signal that will never come.
- There is no data loss because all data transfers follow the sequence shown below. The possibility of duplication in the second phase is eliminated by the next step.

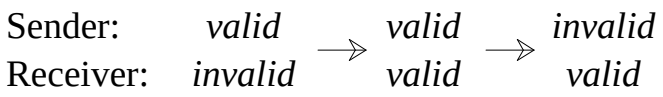

Figure 4.1: Data Transfer Sequence

- In the receiving end, the valid bit is set when acknowledgement is sent, as can be seen in the code for the input cycle. For the sender, the valid bit is reset when acknowledgement is received. In order to avoid duplication, the sender must wait for and respond to the acknowledgement signal after the transfer request is sent.

This is consistent with the 4-phase handshake protocol. Note that there is no *wait* ($\overline{halt_req}$) blocking the statements between *req_out=1* and *valid=0* in the output cycle. If the receiver detects the halt request before acknowledgement is sent, the sender still has the valid bit set, and the transaction will be restarted after the rollback cycle, if data is not invalidated. Otherwise, the transfer is treated as completed, even if the 4-phase handshake protocol is not used to release the request and acknowledgement signals.

4.2. Processor Modules

Now that we know how the basic elements work, the complete modules will be discussed by their functional groups. Only code segments will be shown here as needed. Please see Appendix A for the full Verilog model.

4.2.1. Queues

There are two dedicated queues: instruction queue (IQ) and data queue (DQ). Other internal queues are very similar. The IQ contains only two buffers, but it also needs to support local reset to invalidate pre-fetched instructions whenever a branch occurs. The cancellation cycle is similar to the reset routine, except that the cancellation is performed with the 4-phase handshake protocol to notify the IIU that invalidation is finished. A C-element is used to combine the two cancellation acknowledgement signals from the buffers, as can be seen in *iq.v*, at line 35. When a rollback occurs, the IQ is simply cleared with the same signal, so a separate rollback routine is not needed.

The DQ is a combination of a queue and a bus-access handler. The queue buffer includes a rollback routine to selectively invalidate data entries. The bus handler sends

a request to the ARBR arbiter and then proceeds with the actual transfer after grant is given. The bus handler also needs to support rollback because data is latched before the bus cycle is started.

4.2.2. Checkers

All four checkers are functionally and structurely identical, and they differ only in the number of data bits being checked and in their identification numbers. Each checker has an internal queue with depth of two, and of course, all components of a checker must support rollback. Errors are detected by XORing all the bits in a word along with its parity bit, as in the following code:

```
parity = 0;
for (loop=0; loop<1+DATA_WIDTH; loop=loop+1)
      parity = parity ^ data_buf[loop];
```

If the resulting parity is one, then an error has occurred because even parity is used. For checkers with two data words, the two parity bits are generated individually. After a bus grant is given by ARBK, the checker sends the result to LOG for validation or invalidation.

4.2.3. Memories

The three memory modules are: IMEM, DMEM, and REGFILE, which all use the Verilog memory structure. The IMEM is read-only and pipelined so that the PC can start its next cycle when the instruction is sent to IQ. The pipelined handshake is handled here by a *fork-join* structure rather than separate input and output cycles as in the buffer:

```
always wait (in_req)
begin
      data = imemory [addr[ADDR_WIDTH-1:ADDR_IGNORE]];
      fork
            begin
                  in_ack = 1;
                  wait (~in_req);
                  in_ack = 0;
            end
            begin
                  out_req = 1;
                  wait (out_ack);
                  out_req = 0;
                  wait (~out_ack);
            end
      join
end
```

Because the the unit of address is byte, but the the memory is organized as 32-bit words, the `ADDR_IGNORE` parameter is used to eliminate the two lower address bits.

DMEM is *not* pipelined in the read mode because the destination register and the sequence number from MEMDWB are routed directly to DQ and not through DMEM. This way, the data from MEMDWB are not released until they are latched in DQ, along with the data read from DMEM. The following is the non-pipelined handshake, in contrast to the pipelined version used by IMEM:

```
always wait (in_req)
begin
      data_out = dmemory [addr[ADDR_WIDTH-1:ADDR_IGNORE]];
      out_req = 1;
      wait (out_ack);
      in_ack = 1;
      out_req = ZZZ;    // release request
      wait (~in_req);
      in_ack = 0;
      wait (~out_ack);
end
```

The input is acknowledged *after* the output acknowledgement is received. Note that the request signal is released by tri-stating it because it is shared with MEMDWB. That line is pulled down by an external source.

Since the IMEM is read-only, the program is loaded from the instruction file only when the simulation is started, in an *initial* block. The same file is also loaded into DMEM so that memory data defined in the program can be accessed. However, the loading is done in the reset routine because DMEM has to be initialized when a reset occurs. Both IMEM and DMEM have "error correction" capability to re-calculate the parity bit after a rollback, to simulate transient memory faults. The correction is triggered by the *retry* flag from IIU.

While IMEM and DMEM are single-ported memory, the REGFILE has three ports, supporting two reads and one write simultaneously. Even though the register *R0* is a zero constant, operations that write to it are carried through as usual, but reading that register always results in zeroes, as determined by these two statements:

```
rd_data1 = rd_reg1 ? rfile [rd_reg1] : 0;
rd_data2 = rd_reg2 ? rfile [rd_reg2] : 0;
```

4.2.4. Program Counter

The PC module is a loadable incrementing counter. During normal operations, the counter is incremented by four after each instruction fetch, and a new parity bit is generated. When a branch occurs, the load cycle interrupts the increment cycle and obtains the new address from IIU.

4.2.5. Arithmetic Logic Unit

The ALU itself is very simple because Verilog operators are directly used for all arithmetic and logic functions except SRA (shift right arithmetic), which is implemented by:

```
for (loop=0; loop<(data2 % DATA_WIDTH); loop=loop+1)
    out_data = {data1[DATA_WIDTH-1], data1[DATA_WIDTH-1:1]};
```

The whole word is shifted to the right, but the highest bit, or sign bit, is kept the same. After each computation, a new parity bit is generated to accompany the data to its destination. For fault simulation, if the requested function is ADDUF (add unsigned with fault), then the parity bit is initialized to 1, which would result in an incorrect parity bit:

```
parity = (func == FUNC_ADDUF);
```

The rest of the ALU module contains an input queue and routines for handshake and rollback.

4.2.6. Controller

4.2.6.1. Instruction Issuing Unit

The AMPIRE controller is consisted of the instruction issuing unit (IIU) and the reservation table (RESTABLE). In addition to performing the steps between instruction fetch and execution, the IIU also handles branches and jumps. Within IIU, the general sequence is: receive the instruction, decode, log, notify CKI, check reservation, read registers, notify CKF, and send request to an execution unit.

The "fetch cycle" actually just accepts the instruction from the IQ. The instruction and its address from the internal PC are latched in the two bus output buffers before the internal PC is incremented. Then parity bit for the new PC is calculated, and the *go_decode* flag is set to start the decoding routine.

One of the steps in the decoding cycle is determining the check vector so that instruction can be logged. This is done with the *case* structure at line 163 of *iiu.v*. As

discussed in section 3.4.3, the instruction must be accepted by LOG before it can be sent to the CKI checker. Therefore, the IIU waits for the log acknowledgement at line 186 before starting the CKI handshake in the *fork-join* block.

Depending on the opcode, different *tasks*, or subroutines, are called to perform the appropriate functions. All register-to-register ALU operations are assigned a single opcode number (0), and they have to be decoded further using the function field, at line 244. Note that NOP (binary 0) is also decoded in this group because its opcode field is the same as the *special* opcode. All ALU instructions are processed by the ALU control task, which reserves and retrieves the necessary registers. Before these two steps can be taken, the three output variables *reg_rs*, *reg_rt*, and *reg_rd* must be setup properly with data from the instruction word. Any unused field is assigned a zero since *R0* is always available for the reservation process. The two source operands are latched in the bus output buffers by these two statements:

```
abus_out = abus_in;
bbus_out = (opcode == OP_SPECIAL) ? bbus_in :
      {{(DATA_WIDTH-IMM_WIDTH){imm[IMM_WIDTH-1]}}, imm};
```

Abus_in and *bbus_in* are the bus input registers containing values read from the register file. The first operand (abus_out) is always a register value as determined by the instruction set. The second operand (bbus_out) is either a register value or a sign-extended immediate data from the instruction word. A conditional assignment based on the opcode is used to determine the proper selection. Finally, the ALU handler is in charge of generating the parity bits and carrying out communication with the ALU module.

The two tasks for memory and branch operations should be pretty straight forward. For jumps, the JAL and JALR instructions also need to have the previous PC stored in *R31*. Therefore, PC is stored in the output buffer at line 341 before the new PC is

calculated. After the new internal PC is loaded into the PC module, the previous PC is sent to ALU and then passed to the register file.

The PC handler updates the PC module and invalidates the pre-fetched instructions in IQ. When the *inst_en* line is pulled low, the output buffer of IQ is disabled, and the internal PC is gated to I_bus. The request signal that initiates the PC module loading cycle also cancels all current transactions in IMEM and IQ. The acknowledgement signals from all three modules have to be received by the IIU before the next handshake step is taken. This is simply another C-element.

The rollback routine for IIU is slightly different from other ones because it has to initiate recovery after rollback. After the halt request is issued, it has to wait for any current PC loading cycle to finish before it can proceed, which is accomplished by line 94:

```
wait (~newpc_req & ~pc_ack & ~imem_ack & ~iq_ack);
```

The PC handler must be allowed to finish its 4-phase handshake cycle because the same interface is to be used for loading the address being rolled back, and the three instruction fetch modules must be left in a usable state. After all modules have rolled back, as indicated by the LOG releasing the *roll_req* signal, IIU updates the sequence number and the internal PC from data stored in the LOG:

```
seq_num = val_seq;
int_pc = a_bus;
```

Going back to the fetch cycle at line 140, there is an "extra" enabling condition *inst_en*. That is necessary to prevent the fetch cycle from being activated before a valid instruction is available from IQ after rollback. The *inst_en* signal is reset in the rollback cycle and enabled in the PC handler, after the rollback address is loaded in the PC module.

A few fault simulation instructions have been added to test the fault-tolerant portions of the processor. However, since the processor is rolled back to the instruction where error occurred, simply executing it again would cause the same error and result in an infinite loop. A *retry* flag is set in the rollback cycle to indicate that the next instruction to be executed is the second attempt. That flag is then used to "correct" the error so that rollback would not be triggered again for the same instruction. The code can be found at lines 275, 306, and 352.

Another instruction used for simulation is TRAP (as is done in the DLX simulator) that is supported by the task at line 375. It is normally used to call exception handling routines in the DLX, but AMPIRE does not support interrupt/exception. Traps from 0 to 31 cause the simulator to print the register values in decimal, hexadecimal, and binary formats. Trap *BLANK* (90) simply prints a blank line for output formatting. Finally, trap *STOP* (100) is used to signal the end of program. No new instruction will be issued afterwards, but the ones being executed are allowed to finish within the time pre-determined in the test setup.

Note that the sequence of executing multiple handshake cycles in the IIU is *not* optimal for concurrency. For example, the reservation cycle is completed before the register file can be accessed because these two routines are in separate tasks. In reality, the register file request can be sent as soon as the reservation is acknowledged, and the rest of the handshake cycle can be finished in parallel. However, doing so in the already complex IIU module would make the code less readable and less modular. Higher concurrency can be more easily explored in the actual implementation, or in terms of signal transition specifications [Meng89].

4.2.6.2. Reservation Table

The need for register reservation was discussed in section 3.3.2. The reservation table uses two Verilog memory structures, one for the reservation bits, and another one for the storage of sequence numbers, both addressed by the register number. When a reservation request is issued, the first step is to wait until all three reservation bits are cleared:

```
wait (~res_table [reg_w] & ~res_table [reg_r1] &
      ~res_table [reg_r2]);
```

Even though any of the reservation bits may be cleared at any time, there is no problem with glitches because any change must be from 1 (reserved) to 0 (not reserved). Furthermore, only one bit can be changed at any time because there is only one clearance port. Then the destination register is reserved by setting its reservation bit and storing the sequence number in the corresponding slot:

```
res_table [reg_w] = 1 & (reg_w != 0);
seq_table [reg_w] = seq_num;
```

The first line above also makes sure that *R0* is never reserved because an instruction writing to *R0* does not have any effect on the register file. When a rollback occurs, each sequence number is compared with the error sequence number, just like other storage elements. A *for-loop* cycles through all of the slots sequentially, but in reality, associative comparators should be used. Invalidation is accomplished by clearing the appropriate reservation bits.

4.2.7. Delayed Write Buffers

A DWB provides a temporary storage for data not yet committed to its permanent target so that if an error occurs, rollback can be made by deleting the appropriate data from DWB. The two DWBs are the two largest modules in AMPIRE in terms of code size, but a lot of the code is related to submodule declarations and connections.

4.2.7.1. REGDWB

In the REGDWB controller module, there are five main functional cycles in addition to the standard reset and rollback routines: input, check, clear, commit, and read. The input cycle directs the queue to accept data and then enables the check and clear cycles, which requests CKR to check the data and clears the reservation at RESTABLE, respectively. The input routine is guarded with the $\overline{go_check}$ and $\overline{go_clear}$ signals to prevent it from starting another cycle before the previous one is completed.

When the oldest entry in the queue reaches the last buffer, its output request would start the commit cycle. However, it does not mean that data is ready to be written to the register file unless the *wait* bit has been cleared by LOG. After that bit is cleared, a write request is sent to the register file. This step is *not* pipelined because data in the queue cannot be deleted until it is written to the register file. This is done so that a read operation always has a valid data to be read. If the commit cycle is pipelined, then the most recent data may be held in a buffer that cannot be searched, or data is just being written to the register file and not stable enough to be read properly. Also, the two *com_ack* transitions are guarded with the $\overline{rd_req}$ signal so that data cannot be dropped in the middle of a read operation.

The read request from IIU goes directly to the REGDWB queue and REGFILE,

but the two acknowledgement signals do not return to IIU but are used to activate the read cycle in the controller instead. If there is a match in the buffer, then that data is gated to the bus. Otherwise, data from the register file is selected. Multiple matches in the queue are resolved by a priority circuit, but that is transparent to the controller. The read cycle also supports fault simulation by inverting the parity bit if the *sim_f* signal from IIU is active.

The *regdwbq* module connects the four buffers in a usable fashion with four C-elements to handle four sets of acknowledgement signals and a priority routine to deliver correct data to the REGDWB controller. When multiple matches are found, the *casex* structures select the latest entries based on the match results. The two source registers are handled separately.

In an individual buffer, the validate cycle clears the appropriate *wait* bit when its corresponding instruction is validated by the LOG. The find cycle is responsible for checking if its buffer contains the data for the register being read. Note that a match signal for *R0* must be suppressed because that register cannot be overwritten, although any instruction may *try* to do so. Otherwise, a non-zero value may be returned for *R0* because data in the buffer has higher priority than the register file.

The input and output cycles are the main handshake routines for data transfers, similar to the ones in section 4.1.6. The major difference is that the DWB buffer has to pause when the validate and find operations are active. Both of these two functions are associative searches, and the validate cycle also modifies a bit in the buffer. Before a comparison can be made, the input and output routines must be stopped to keep the data stationary. Therefore, each output group is guarded by this statement:

```
wait (~val_req & ~val_ack & ~read_req & ~halt_req);
```

Because of this requirement, each search may slow down the normal flow of data in the queue. This penalty was discussed in section 3.5.1.

4.2.7.2. MEMDWB

The queue for MEMDWB is basically the same as the one for REGDWB, except that the register numbers are substituted with memory addresses for data tracking and searching. The controller has to be modified to handle a single-ported memory rather than a three-ported register file.

Each memory transaction can be either a read or write, and the input cycle has to process the data accordingly. Memory writes are queued in DWB, but memory reads are not buffered in order to reduce reservation waiting time, as explained in section 3.3.3.2. For a read operation, the read routine is enabled by setting the *go_read* flag, and then a check request is sent to CKM. For a write operation, data is sent to the DWB queue.

After the read cycle is activated, the first step is to search DWB for any data destined to the same memory location. If there is a match, then data is retrieved from the buffer and sent to DQ directly, and a memory access is saved. If there is no match, then DMEM has to be read. Since DMEM is a single-ported memory, only one read *or* write operation can be done at once. Therefore, an internal arbiter is used for mutual exclusion, with priority given to the read request for higher performance, as registers are reserved for loading from memory. Even though both MEMDWB and DMEM may send data to DQ, there is at most one outstanding memory read operation at any time, so no arbitration is required for accessing the DQ.

Interactions between the read and commit cycles need some explanation. The read cycle issues a search request and then waits for arbitration. At the same time, the

commit cycle may be writing to DMEM and will then acknowledge the queue to delete the committed entry. However, as explained in the last section with REGDWB, the input and output routines must be suspended when the queue is searched. The question is whether there is a possibility of deadlock since the commit cycle is not able to delete the entry from the queue. The following code is taken from the *commit_cycle* in *memdwb.v* , but not as a continuous block:

```
wait (out_ack);
com_ack = 1;
out_req = 0;
arbwr_req = 0;
wait (~com_req);
```

After DMEM has completed the write cycle (*out_ack* becomes 1), an acknowledgement *com_ack* is sent back to the queue for deletion. At this time, both of the DMEM and arbitration requests are released, so the read cycle is able to continue after grant is given by the arbiter. The commit routine cannot finish its cycle until the queue search is completed, but that does not interfere with the read operation.

4.2.8. Instruction Log

The instruction log stores every instruction issued by the IIU until each one is completely checked and validated. If an error occurs, the LOG contains information necessary to roll back the processor and re-execute the offending instruction. The LOG can be split into a controller and an actual storage log that is organized as a queue. The IIU logs an instruction by communicating directly with the log, not through the controller.

The controller has two main functions: validation and rollback. Each checker reports to LOG with the checker ID, the sequence number, and the pass/fail signal. If there is no error, then the appropriate check bit in the log is cleared. If the check failed,

a rollback cycle is activated, using the sequence discussed before. Each module that needs to roll back must monitor these signals and respond accordingly. An arbiter is needed because the sequence number output port is shared between the validation and rollback routines.

When the check bits of the oldest entry in the log are all cleared, then that instruction can be validated. Instruction validation involves notifying one of the DWBs and deleting its entry from the log. Data from the check vector, format shown in Figure 3.4, is used to determine which DWB contains the waiting data. Since there are two DWBs in AMPIRE, the CKR and CKM check bits are stored separately as *DWB bits*, which are *not* reset by the check bit clearance process. The four possible combinations are shown in Table 4.2.

<CKR,CKM>	**Target DWB**	**Instruction Examples**
00	None	NOP, Conditional Branches
01	MEMDWB	SW (store word)
10	REGDWB	LHI, All ALU Instructions
11	REGDWB	LW (load word)

Table 4.2: DWB Bits in the Instruction Log

Cases 01 and 10 should be obvious. For the instruction LW, it has to go through both checkers CKM and CKR, but the data must be in REGDWB when the instruction is validated. By sending the validation signal to only one DWB, as indicated by the DWB bits, unnecessary queue searches are avoided. If the DWB bits are 00, that instruction does not cause any changes to the permanent storage modules, and it can be simply deleted from the log.

When the processor is rolled back, the instruction address matching the error sequence number is retrieved from the log and sent to IIU via A_bus. That is done in

the rollback cycle of the log buffer, at line 354 of *log.v* :

```
if (valid & (out_s == clr_s))
      tri_addr = out_d;
```

There is no conflict with multiple matching because each valid log entry has a unique sequence number within the current "window". The *tri_addr* register is put back to high-impedance state after the rollback cycle.

4.3. Putting the Processor Together

4.3.1. Parameters and Timing

The *parameter* file declares the constants used by all processor modules, including the opcode assignments, instruction field widths, module delay factors, etc. The DLX instruction opcodes are taken from the documentation in [Host91] so that the same instructions would have identical binary code. Most importantly, the behavior of the processor can be modified by adjusting the delay parameters. For example, the order of instruction completion can be changed by increasing the access time of the DMEM module, which will be shown in the next chapter with simulation results.

Whether the processor is delay-insensitive or just self-timed depends on the actual implementation at the transistor level. The behavioral model of AMPIRE is self-timed in the sense that the delay values in all modules may be changed without affecting the processor's functionality and correctness, but it is not delay-insensitive because single-rail data and requests are used, as discussed in section 2.1. Of course, delays cannot be placed *anywhere* because logical grouping of some statements must be observed. Most delays are set at 1 to minimize idle simulation time while allowing the Verilog clock to advance.

4.3.2. Top-Level Wiring and Testing Module

Ampire.v is the file that connects all processor modules together, with a lot of wires. The buses are declared to be *trireg* to simulate the capacitance in wires by keeping the signals at the last driven values when all drivers are at high impedance state. This only reduces the number of signal transitions and is *not* required to satisfy any timing conditions in AMPIRE. Passive *pulldowns* are used to keep the shared handshaking lines from floating because all handshake transitions are significant in a real asynchronous system, and noise should be minimized.

The test setup routine appears toward the end of the *ampire.v* listing. A large *initial* block organizes the signals for generating simulation waveforms that will be shown in the next chapter. The other *initial* block starts the processor by issuing a global reset. After that, the processor runs at its own pace until the *stop* instruction is encountered or until the preset simulation time is exceeded. The #100 delay before stopping Verilog allows instructions still running in the processor to be completed. The high delay value for abnormal termination is mainly used for catching infinite loops created by software and "hardware" bugs.