Chapter Five
Behavioral Simulation

5.1. Instruction Fetch/Issue and Queue Operations

In this chapter, several test programs and their simulation waveforms will be presented. Figures 5.1 and 5.2 show the effects of running the following program with slow and fast instruction memories, respectively. Figures 5.3 displays only the first 85 time steps of Figure 5.2 so that the sequence of events can be easily observed.

```
          //    ; test1.a --
          //    ;    jumps, IF stage, reservation, ALU & REGDWB queues
          //
          //    ; DLY_ALU_ADD=20, DLY_ALU_OR=30, DLY_ALU_XOR=10
          //    ; DLY_ALU_PASS=5, DLY_IMEM_RD=1 and 10
          //
000000000 // 00         nop
00C000018 // 04         jal     sub
          //
034000001 // 08         ori     r0, r0, 1       ; write R0
03401000F // 0C         ori     r1, r0, 15
038020005 // 10         xori    r2, r0, 5
000221821 // 14         addu    r3, r1, r2      ; read after write
050230004 // 18         slli    r3, r1, 4       ; write after write
144000064 // 1C         trap    100             ; stop
          //
14BE00000 // 20 sub:    jr      r31             ; return
```

Im_req is the request signal from PC to IMEM to start a memory access cycle, and *im_ack* is the corresponding acknowledgement. *Iq_req* from IMEM notifies IQ that an instruction has been retrieved, and then IQ sends *iiu_req* to IIU when the instruction is ready at the output of the queue. *Pc_load* is active when a branch is taken, and the two *pc_load* pulses represent the two jumps in the test program. Please see Figure 3.2 for direction of data flow.

Since the instruction cycle continues until IQ fills up or a branch occurs, the

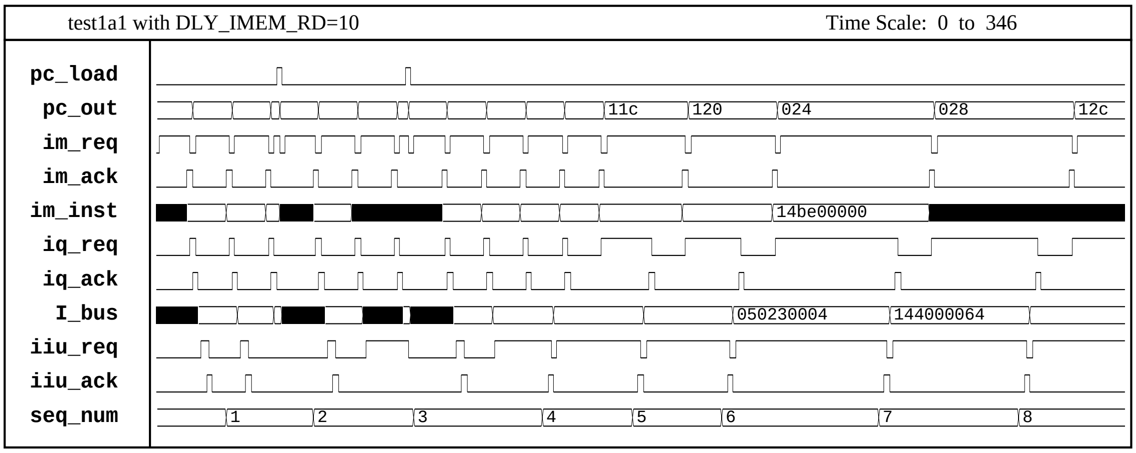

Figure 5.1: Test 1 with Slow IMEM

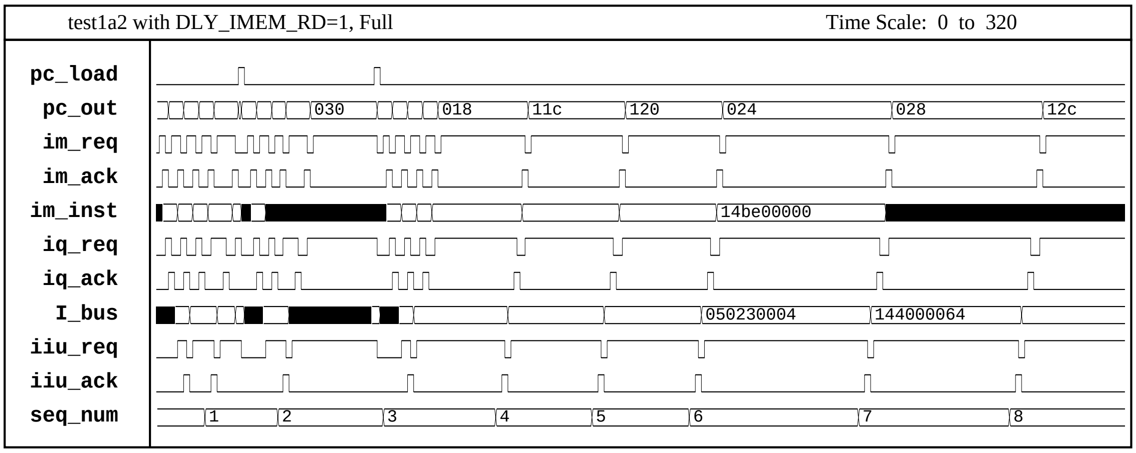

Figure 5.2: Test 1 with Fast IMEM

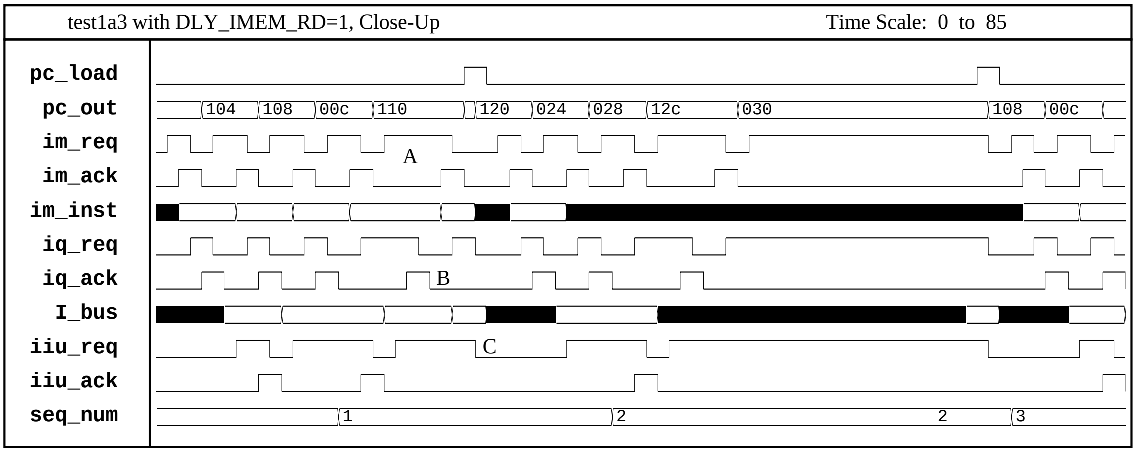

Figure 5.3: Test 1 with Fast IMEM, Close-Up

number of pre-fetched instructions depends on the speed and depth of IQ and the access time of IMEM. Looking at the *im_ack* signal before the first *pc_load* pulse, 3 memory cycles completed in Figure 5.1, compared with 5 cycles in Figure 5.2. Furthermore, looking at point A in Figure 5.3, the response time of *im_ack* is significantly slower than the previous four. At that time, the IQ is full and cannot accept more data until an entry is removed from its output at point B.

As discussed in section 2.1, the 4-phase handshake protocol requires that an activated request signal stays high until its acknowledgement is received. Checking these waveforms would reveal many violations; point C in Figure 5.3 is an example. However, these are caused by local resets of the IF stage as initiated by *pc_load* and, therefore, are not real violations of the protocol.

Figure 5.4 shows the simulation result of the same test program, displaying signals from other parts of the processor. The JAL instruction needs to store the current PC in R31, and that register is reserved at point A. The next instruction JR R31 checks for register availability at point B, but it is held until R31 is cleared at point J. Note that R31 is *not* written to the register file until point N, after that instruction is validated at point R. This means that R31 is read from REGDWB instead of the actual register file.

The two ORI instructions demonstrate that "writing" R0 does not block other instructions from using it, since it is a zero constant. R0 with the first ORI is not cleared until point K, but reservation clearance for the second ORI is already completed at point C. It may be argued that the register clearance at point K and the write operation at point O are not necessary, since R0 cannot be written anyway. However, eliminating these steps would need at least an additional zero detector to handle this special case; another performance and cost trade-off.

The waveforms at points (D,L) and (E,M) indicate the dependencies for read-

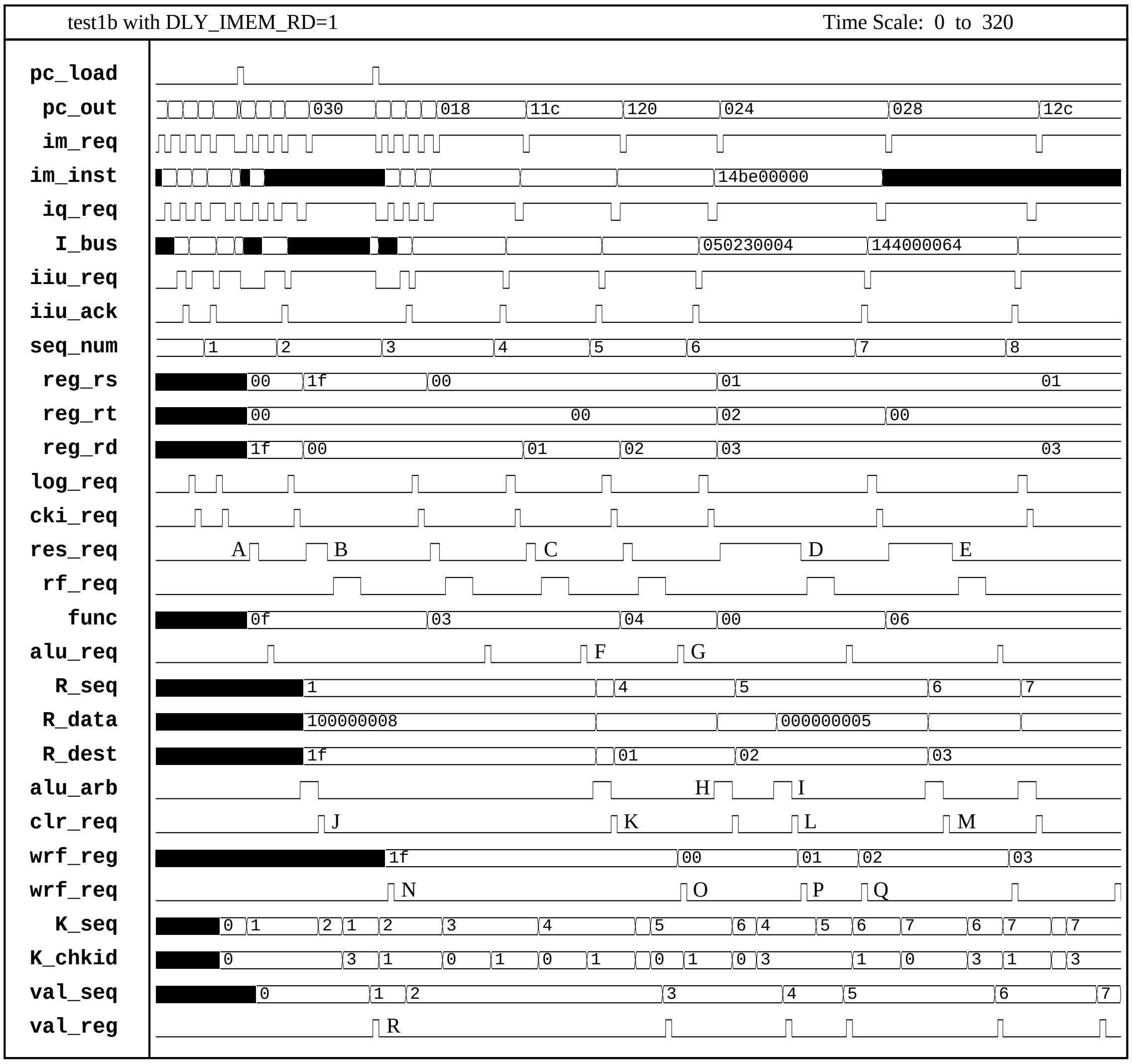

Figure 5.4: Test 1, Detailed Activities

after-write and write-after-write, respectively. Instructions are held by the reservation table until the required registers are available. The signals at points (F,H) and (G,I) show the input-output timing for the ALU. The ALU is able to accept a new request (pulse G) before finishing its current operation (pulse H) because of its internal buffer. Similarly, (H,P) and (I,Q) are the input-output pairs for REGDWB, indicating queuing inside that block.

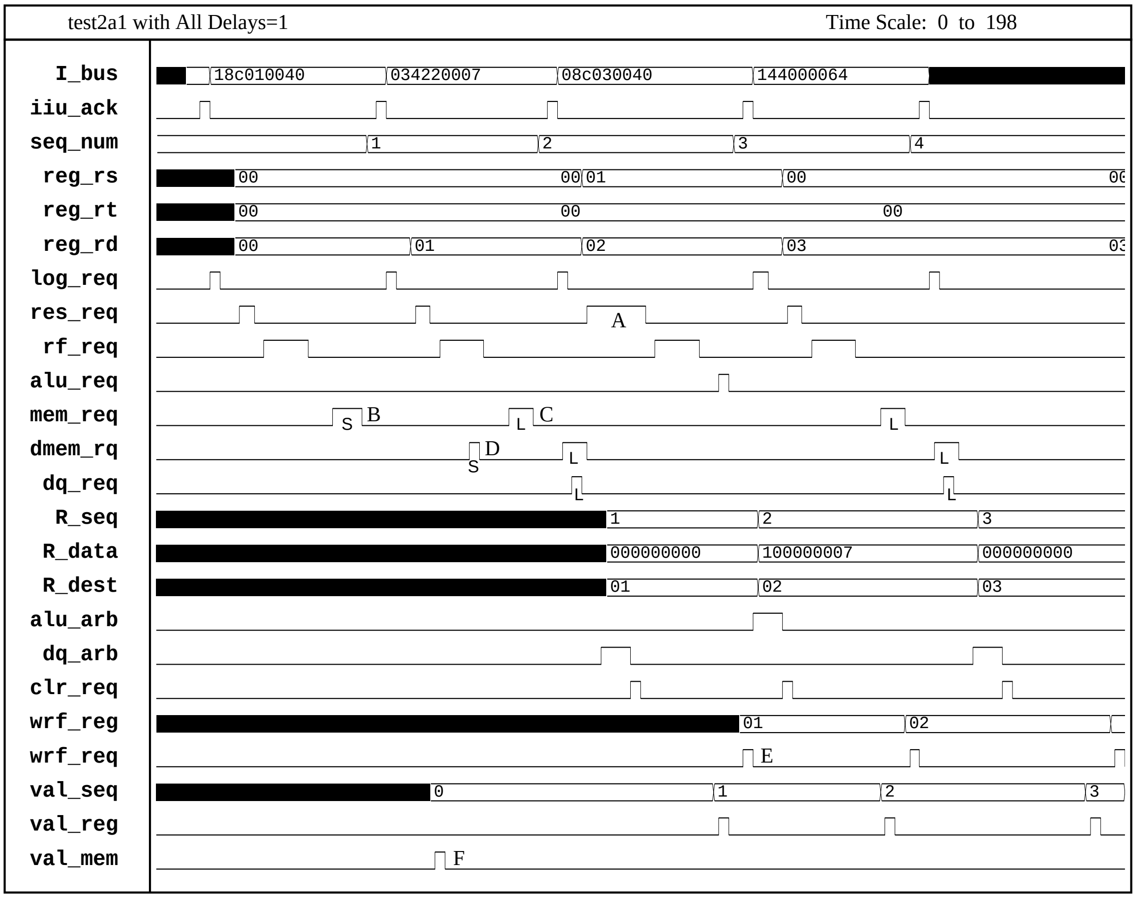

Figure 5.5: Test 2 with All Delays=1

5.2. Delayed Write Buffers and Checker Arbitration

Figure 5.5 is the simulation output of test program #2 with all delays=1. *Mem_req* is the DMEM request signal from IIU to MEMDWB, and then MEMDWB communicates with DMEM through *dmem_rq* as needed. *Dq_req* is driven either by DMEM or MEMDWB, depending on where the data is actually retrieved. These three waveforms are labeled *S* and *L* for store and load, for ease of reference. The ORI instruction is another example of reading data from REGDWB, as indicated by the timing of pulse A occurring before pulse E.

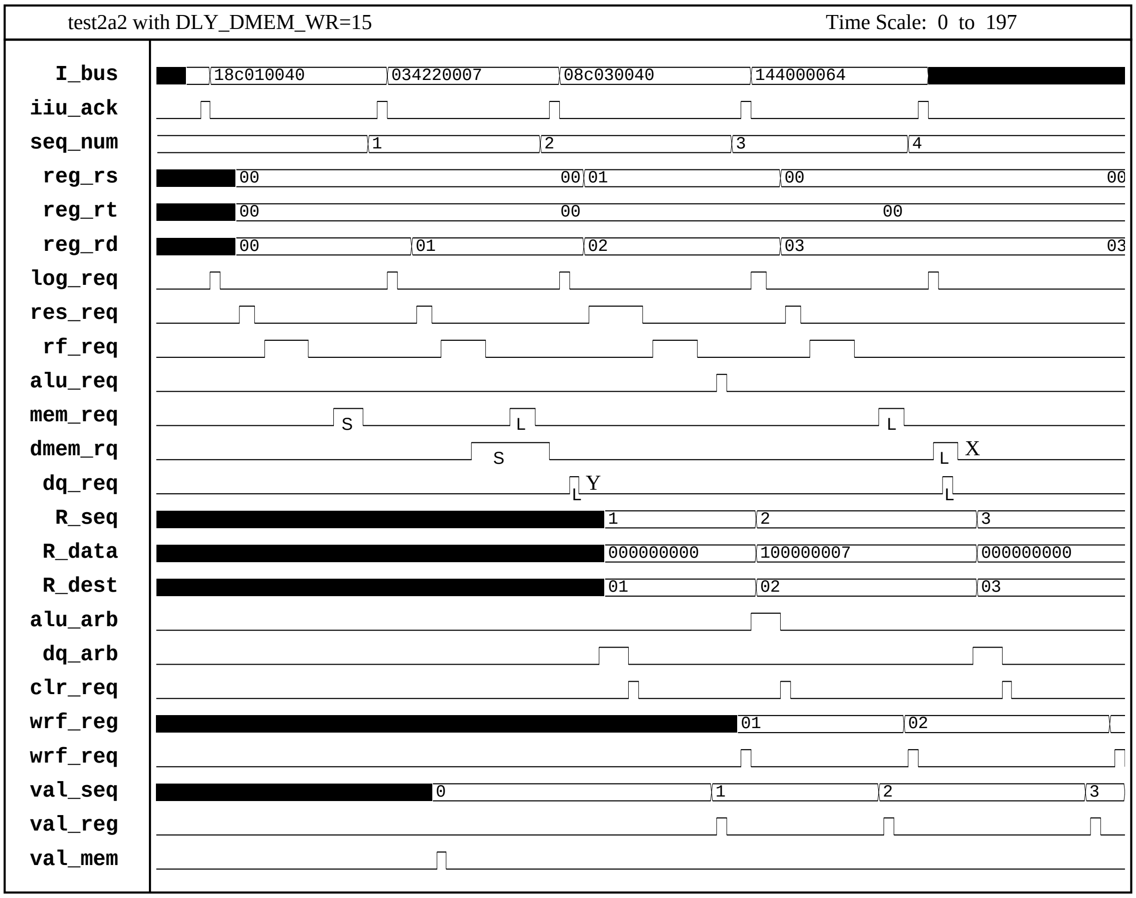

Figure 5.6: Test 2 with Long DMEM Write Cycle

```
          //    ; test2.a -- MEMDWB, REGDWB, checker arbitration
          //
          //    ; one at a time: DLY_DMEM_WR=15, DLY_CKM_CHK=10
          //    ;                DLY_CKF_CHK=10
          //
1AC000040 // 00 sw      40h(r0), r0
18C010040 // 04 lw      r1, 40h(r0)     ; read from MEMDWB
034220007 // 08 ori     r2, r1, 7       ; read from REGDWB
08C030040 // 0C lw      r3, 40h(r0)     ; read from memory
144000064 // 10 trap    100
```

The SW instruction is sent to MEMDWB at point B, but the memory write is not done until point D because it has to be validated at point F first. When the write is completed, that entry is deleted from MEMDWB, and the LW instruction at point C has to retrieve the data from DMEM. With a long memory write cycle, as shown in Figure

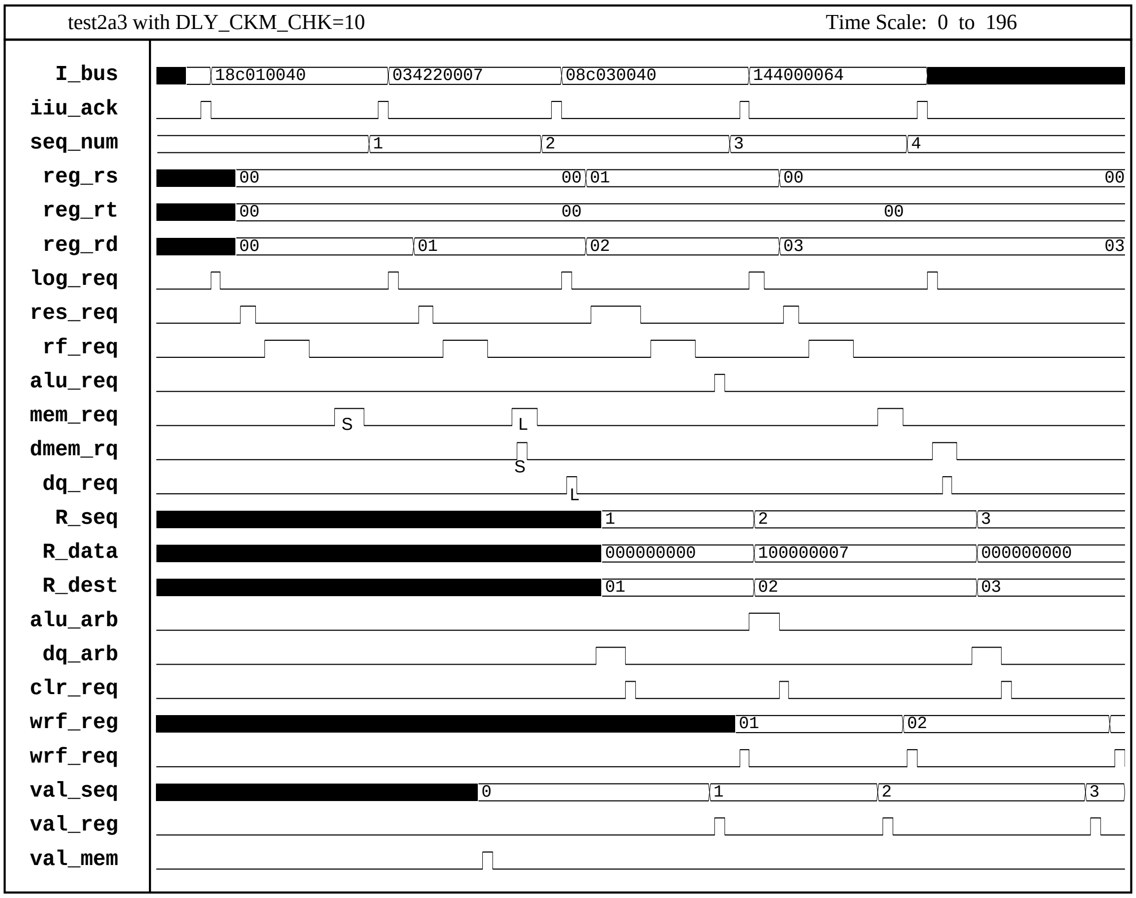

Figure 5.7: Test 2 with Slow Memory Checker

5.6, the load request may arrive before the write is finished. Data is transferred directly from MEMDWB to DQ at point Y, saving a DMEM read cycle. For the second LW instruction, data is still read from DMEM at point X, as expected.

Because a memory store has to be validated before initiating the write cycle, a long delay through the memory checker CKM may have a similar effect as the previous case. In Figure 5.7, the read/search cycle for MEMDWB is already started when the write operation takes place. Even though read has higher priority than write, the external write cycle must be allowed to complete. The internal queue is already halted by the read request, preventing data from being deleted due to write completion. Therefore, data is retrieved from the buffer and sent to DQ.

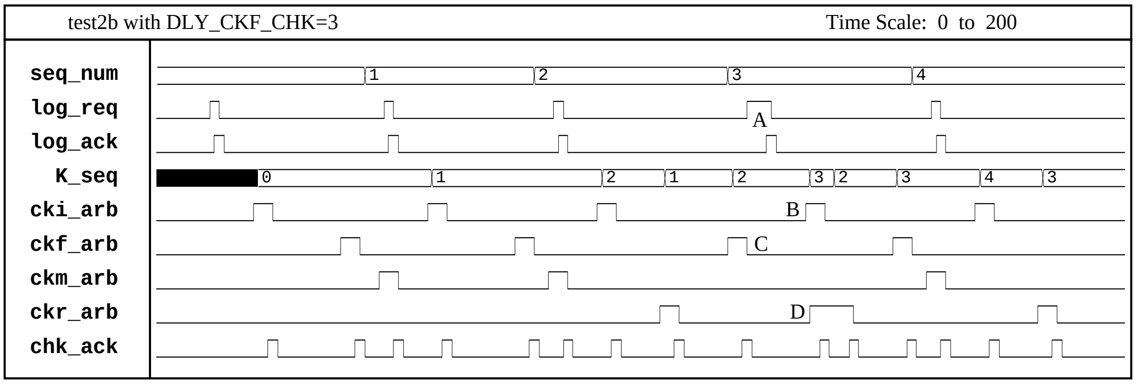

Figure 5.8: Test 2, Showing Arbitration of Checkers

Figure 5.8 shows the activities between the four checkers and the instruction log. The *K_seq* bus is shared by all checkers to send the sequence number being verified to LOG. The single *chk_ack* line from LOG is routed to all checkers, but it is meaningful only to the checker that has been granted access to the K_bus. Two arbitration requests are overlapped at points B and D. Pulse B is actually active one time step ahead of pulse D, and therefore, CKI is given access while CKR must wait.

This figure also indicates that the *log_ack* at point A has a longer delay than other cycles. That is caused by communication with CKF at point C because the queue in LOG must be kept from advancing while the appropriate check bit is searched and updated in the queue.

5.3. Arbitration for R_bus and Out-of-Order Completion

Just like the K_bus, an arbiter is also needed for the R_bus to control data flow from ALU and DQ. The following program is run with different DMEM delays, and the results are shown in Figures 5.9 and 5.10.

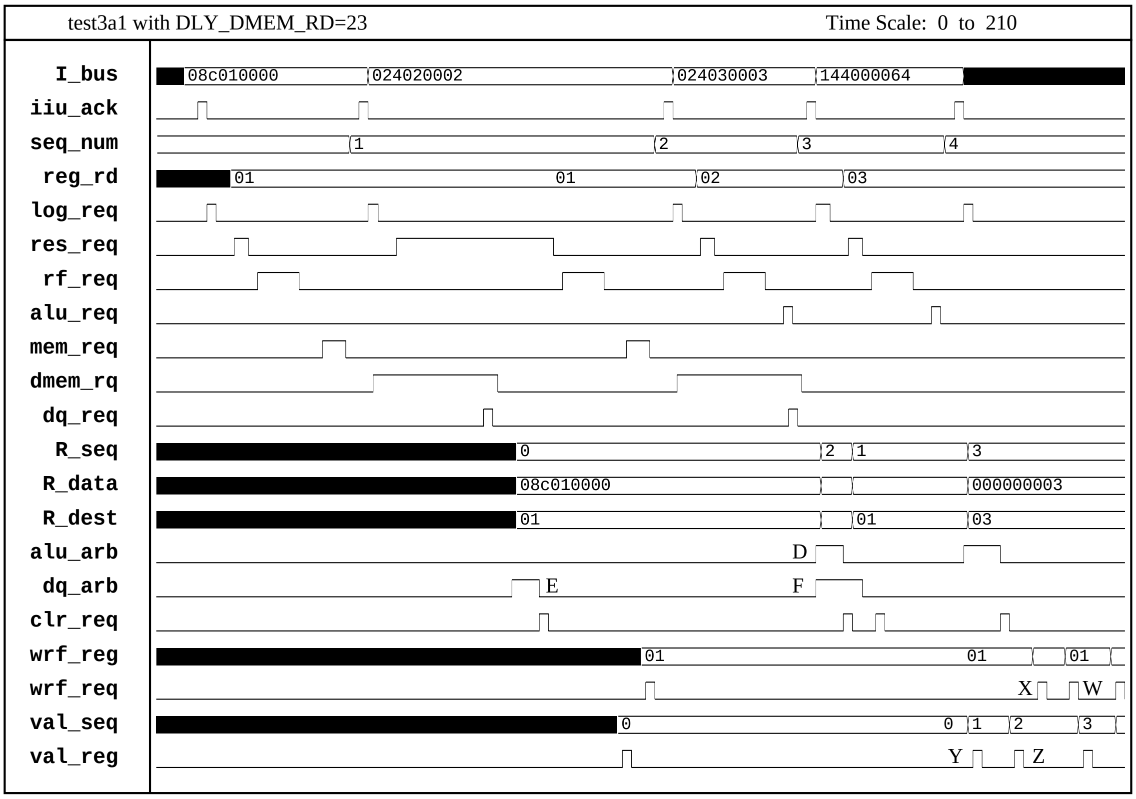

Figure 5.9: Test 3 with Slow DMEM

```
          //    ; test3.a -- R_bus arbitration
          //
          //    ; DLY_DMEM_RD=1 and 23
          //
08C010000 // 00 lw      r1, 0(r0)
08C010000 // 04 lw      r1, 0(r0)    ;completed after next instruction
024020002 // 08 addui   r2, r0, 2
024030003 // 0C addui   r3, r0, 3
144000064 // 10 trap    100
```

In Figure 5.9, arbitration requests D and F occur simultaneously. Access is granted to ALU because the previous bus cycle was taken by DQ at point E. As a consequence, result for the third instruction (ADDUI) is written to REGDWB before the second instruction (LW). Since REGDWB is a FIFO queue, data are also committed to the register file in reverse order, as can be seen by the W and X pulses in the two figures. Even though instructions may *complete* out of order, they are always

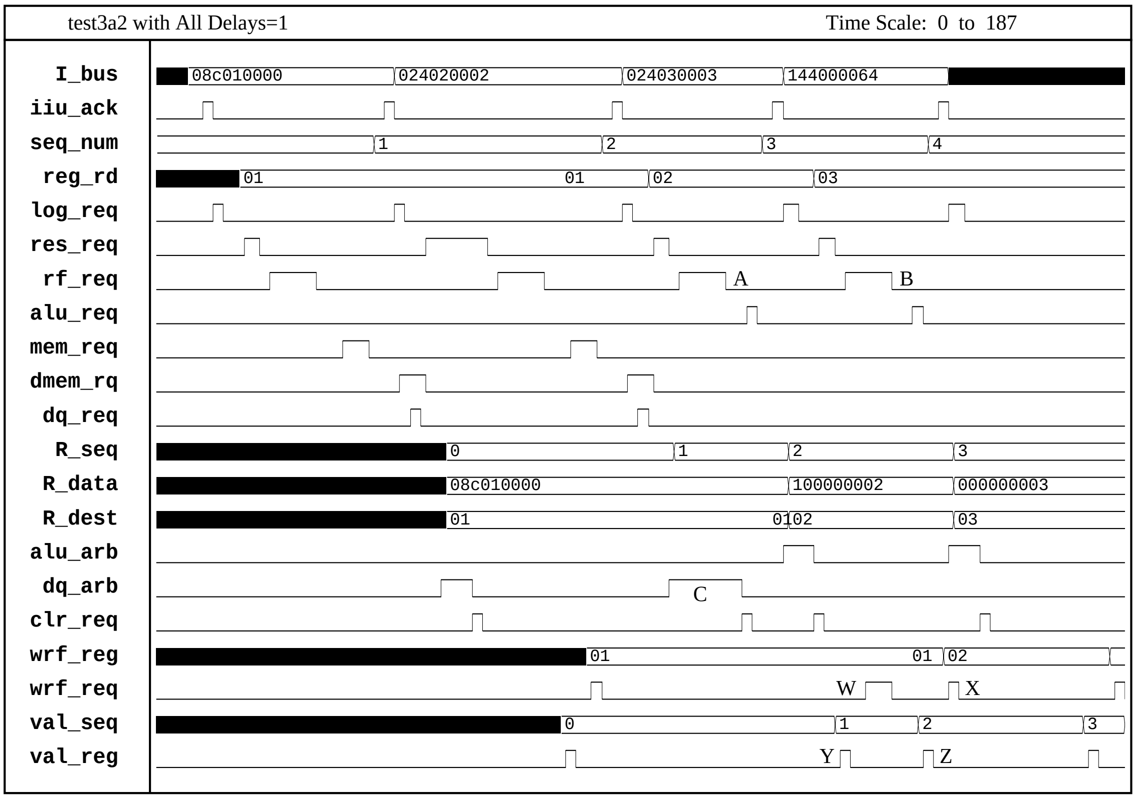

Figure 5.10: Test 3 with Fast DMEM

validated in the same sequence as issued, at points Y and Z.

Figure 5.10 also has two other results worth noting. The DQ-to-REGDWB cycle at C is delayed because of the REGFILE access at A. At that time, the queue in REGDWB is stopped so that uncommitted register data may be searched and retrieved. For the same reason, the write cycle at W takes longer to complete because of the *rf_req* signal at B.

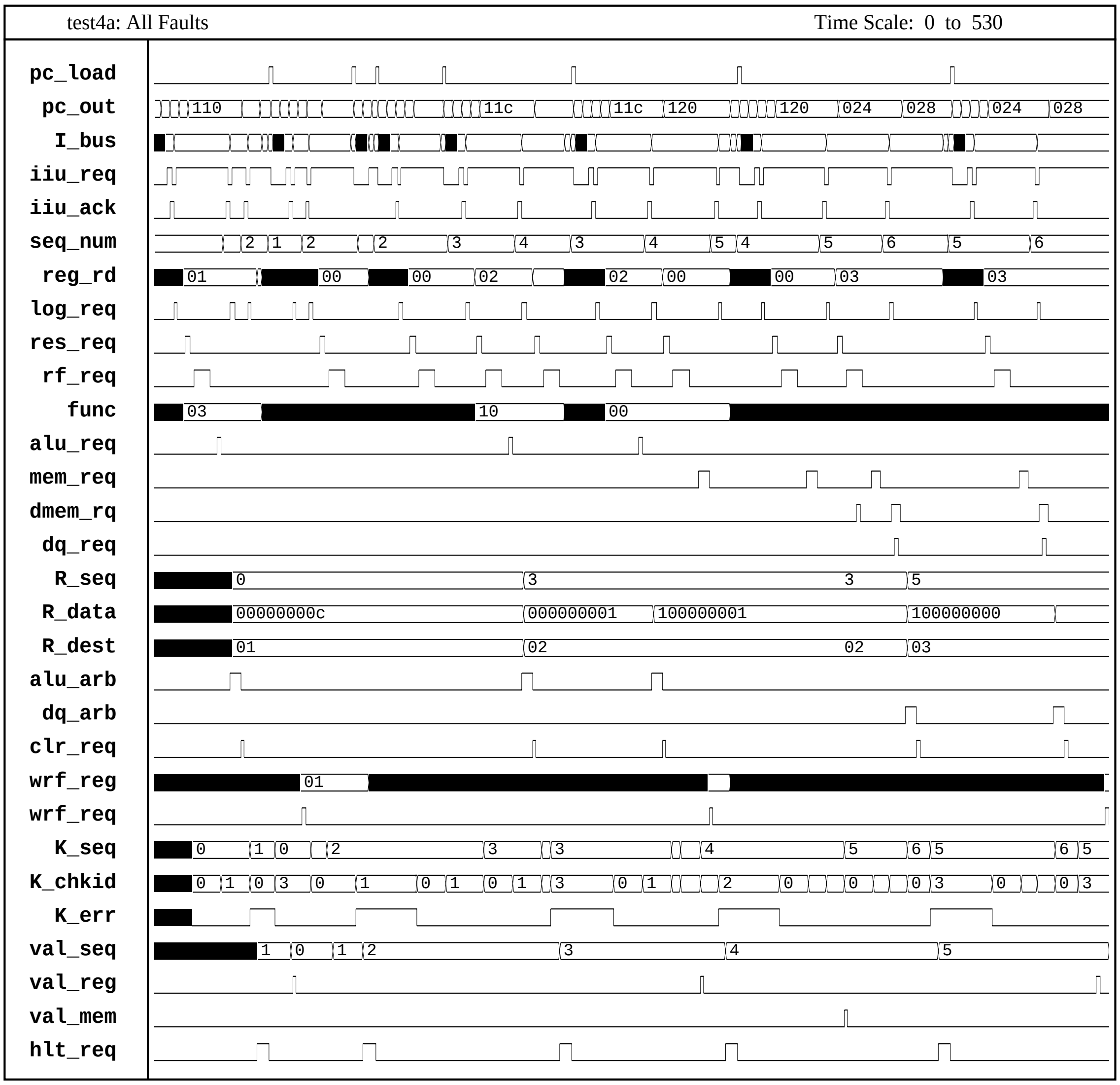

Figure 5.11: Test 4, All Faults

5.4. Fault Detection and Rollback

All of the simulation results presented thus far are free of faults (at least from the processor's point of view). In this section, we will see how the rollback process works when faults are detected. Figure 5.11 is an overview of running test program #4 with all faults shown. The five halt request *hlt_req* pulses at the bottom correspond to the five faults in the program. Each one will be displayed and discussed in greater detail.

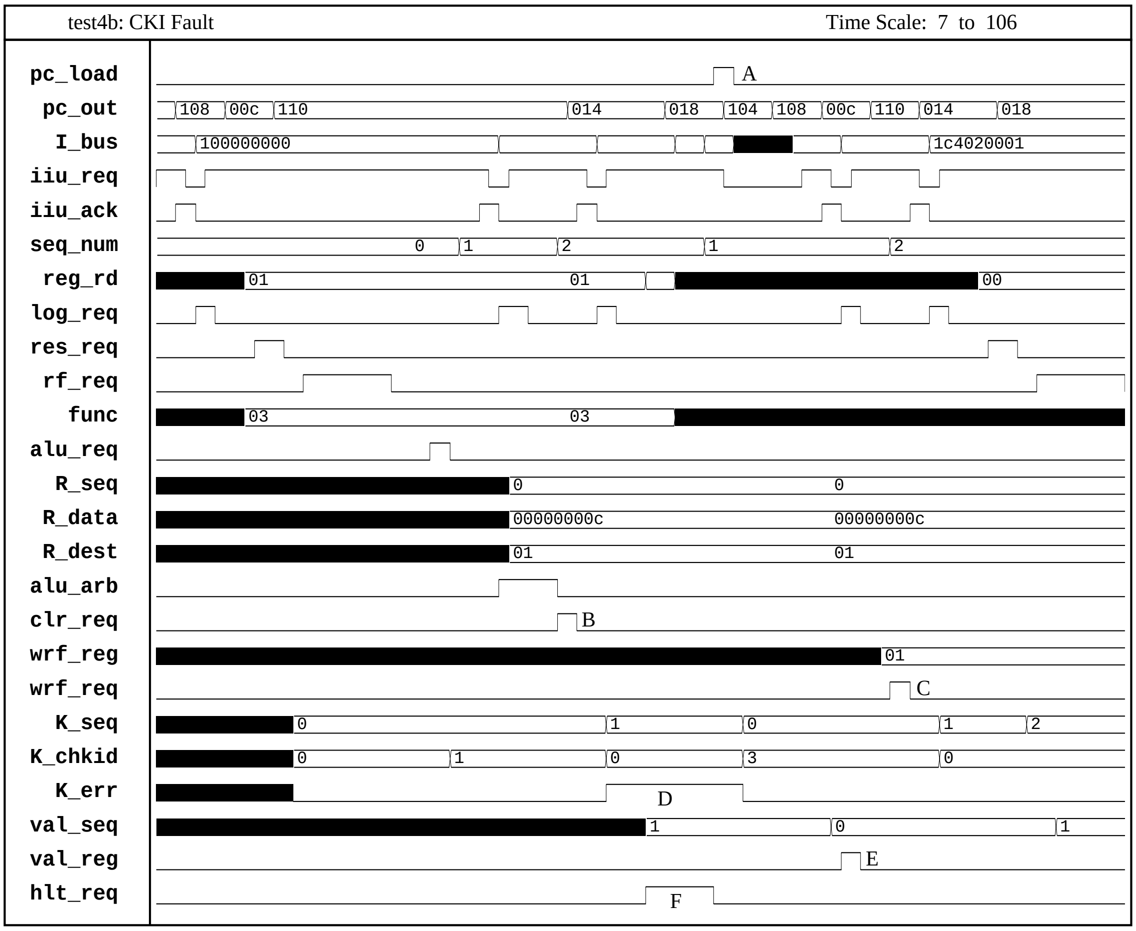

Figure 5.12: Test 4, CKI Fault

```
          //    ; test4.a -- faults, rollback
          //
03401000C // 00         ori     r1, r0, here
100000000 // 04         *nop                        ; cki fault
0C8200000 // 08         jrf     r1                  ; ckf fault
1C4020001 // 0C here:   adduif  r2, r0, 1           ; ckr fault
1CC000040 // 10         swf     40h(r0), r0         ; ckm fault
08C03001C // 14         lw      r3, badata(r0)      ; ckr fault
144000064 // 18         trap    100
          //
100000000 // 1C badata: *.word  0
```

The first fault is caused by the instruction NOP retrieved from IMEM, and it is detected at point D in Figure 5.12. The three K_bus components indicate that there is an error for sequence number 1, and it is detected by checker 0 (CKI). The instruction log controller then initiates a rollback cycle by activating *hlt_req* at point F, and the program counter is updated at point A. Before the rollback starts, the result for the first instruction ORI is already written to REGDWB at point B. Since it is not dependent on the erroneous instruction, data is kept in the REGDWB, and after it is validated at point E, it is written to the REGFILE at point C.

The second fault is from the JRF instruction. Register R1 written by ORI is correct, but JRF forces a bad parity bit when R1 is read, and the fault is detected by the register file checker CKF. The *pc_load* signal at point G in Figure 5.13 is the first JRF attempt. A new address is loaded at point H for rollback, and the final branch occurs at point I.

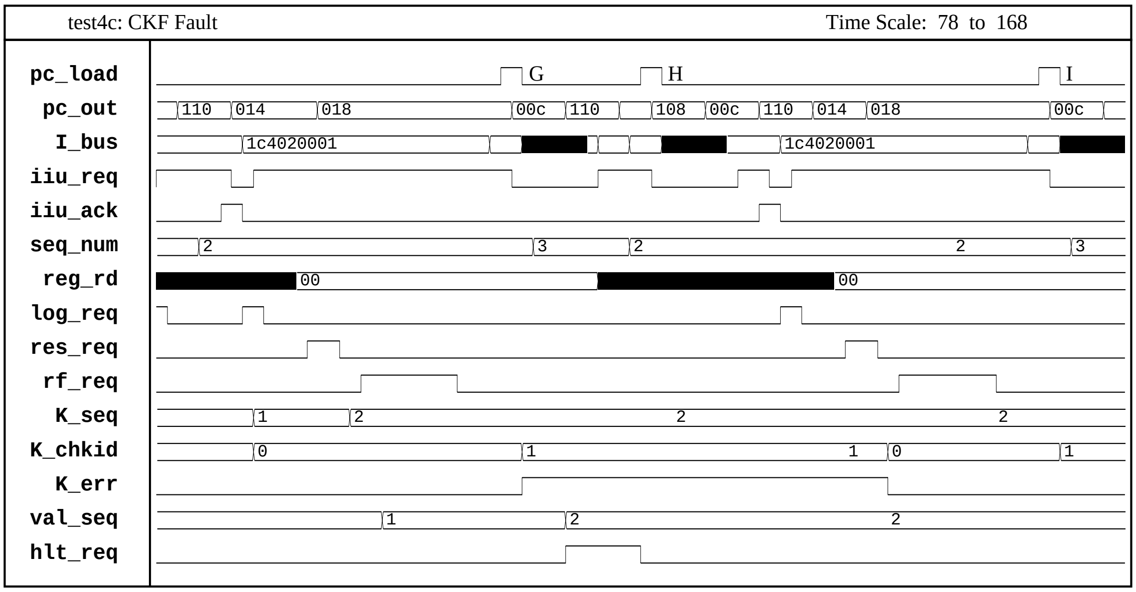

Figure 5.13: Test 4, CKF Fault

The next instruction ADDUIF tells the ALU to perform an addition with bad parity. The results can be seen on the R_data bus in Figure 5.14, before the rollback at point J, and after the rollback at point K. Both data words are written to REGDWB, but the incorrect one is invalidated in the rollback process, and only one word is actually committed to the register file at point L. Similarly, the SWF instruction causes an erroneous data to be written to the MEMDWB at point M in Figure 5.15. The rollback cycle makes the correction, and the data is sent to DMEM at point N.

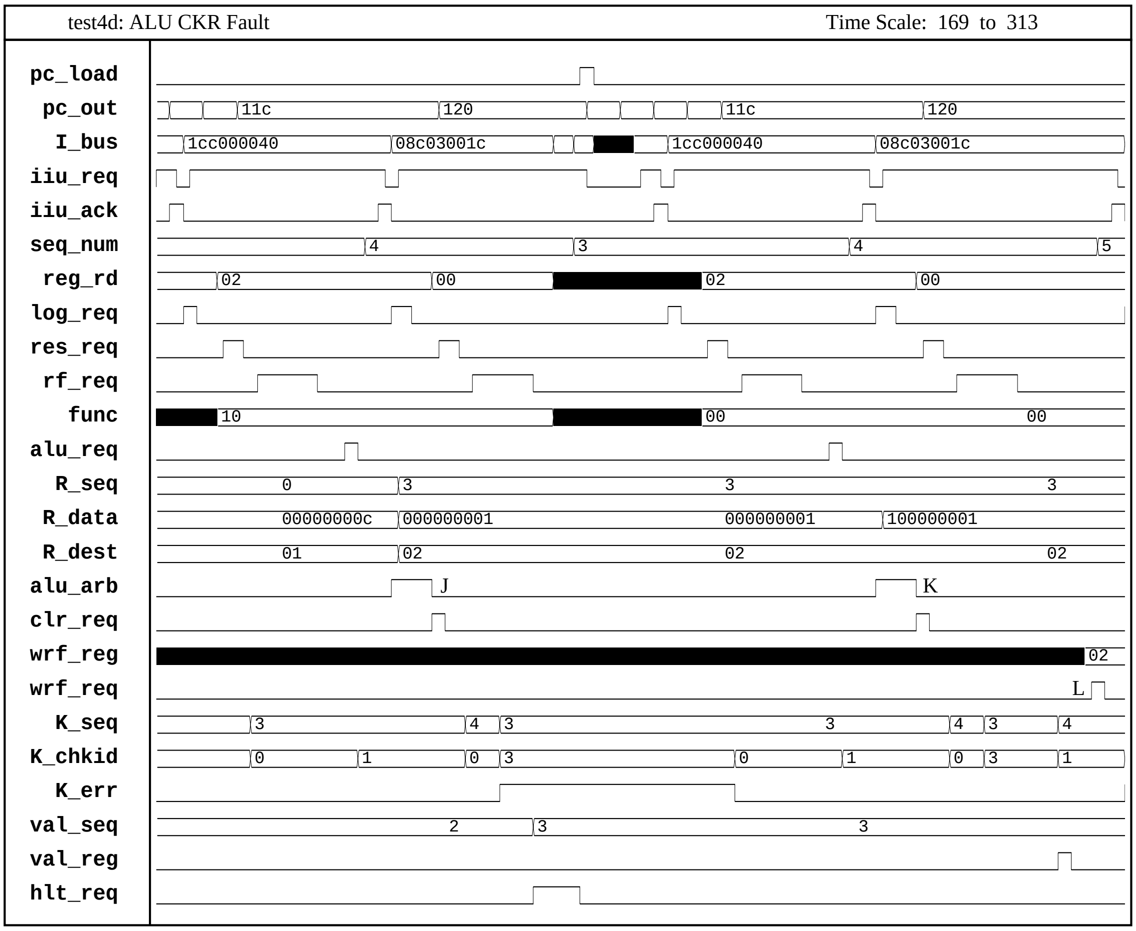

Figure 5.14: Test 4, ALU CKR Fault

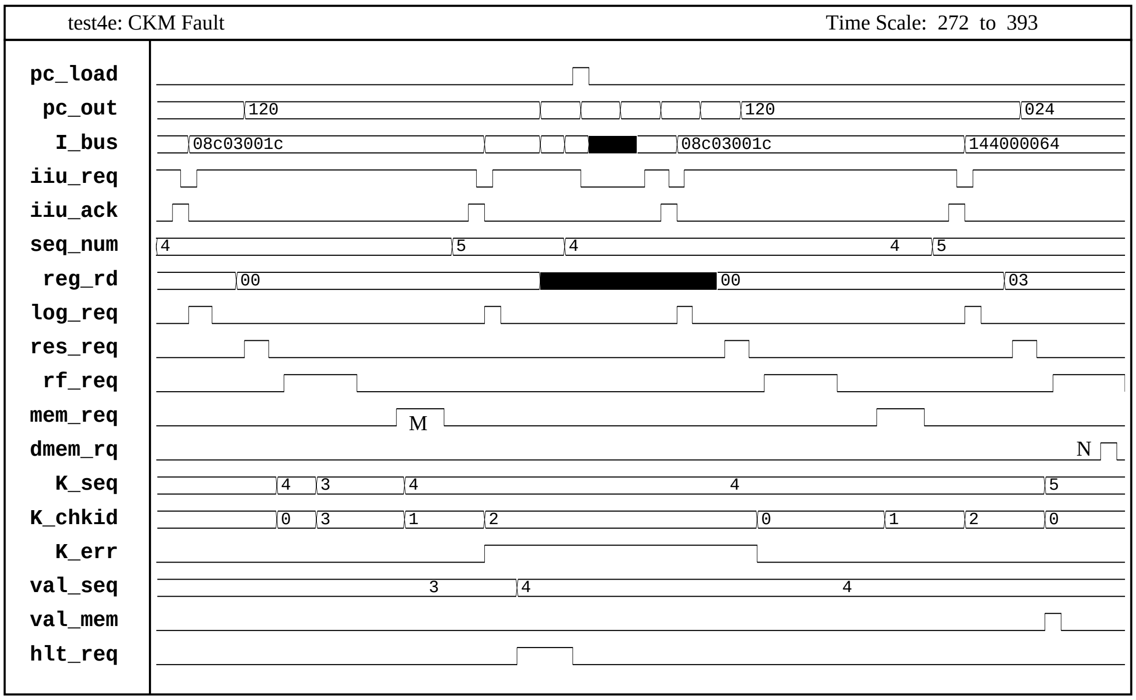

Figure 5.15: Test 4, CKM Fault

Another CKR fault takes place when the LW instruction tries to load a bad word from DMEM. Since the error is in DMEM and not in MEMDWB, two DMEM read cycles occur in Figure 5.16, at points O and P. The rest of the events are similar to the CKR fault caused by ADDUIF because both instructions modify the register file. Note that the pulse at N is from the previous SWF instruction.

Figure 5.17 takes the first fault at NOP as an example to show the detailed interactions between IIU and LOG. The four halt and rollback signals are discussed in section 3.5.2 and also shown in Figure 3.7. At point Y, the IIU sends the sequence number on *seq_num*, the instruction address on *A_bus_L*, and the instruction on *B_bus* to the LOG. *A_bus_L* is just the lower bits of *A_bus* that are significant for address formation. Before the *rol_req* signal is released at point X, the rollback address is retrieved from LOG and sent to IIU on *A_bus_L*. The IIU then updates its internal program counter and loads it into PC at point Z.

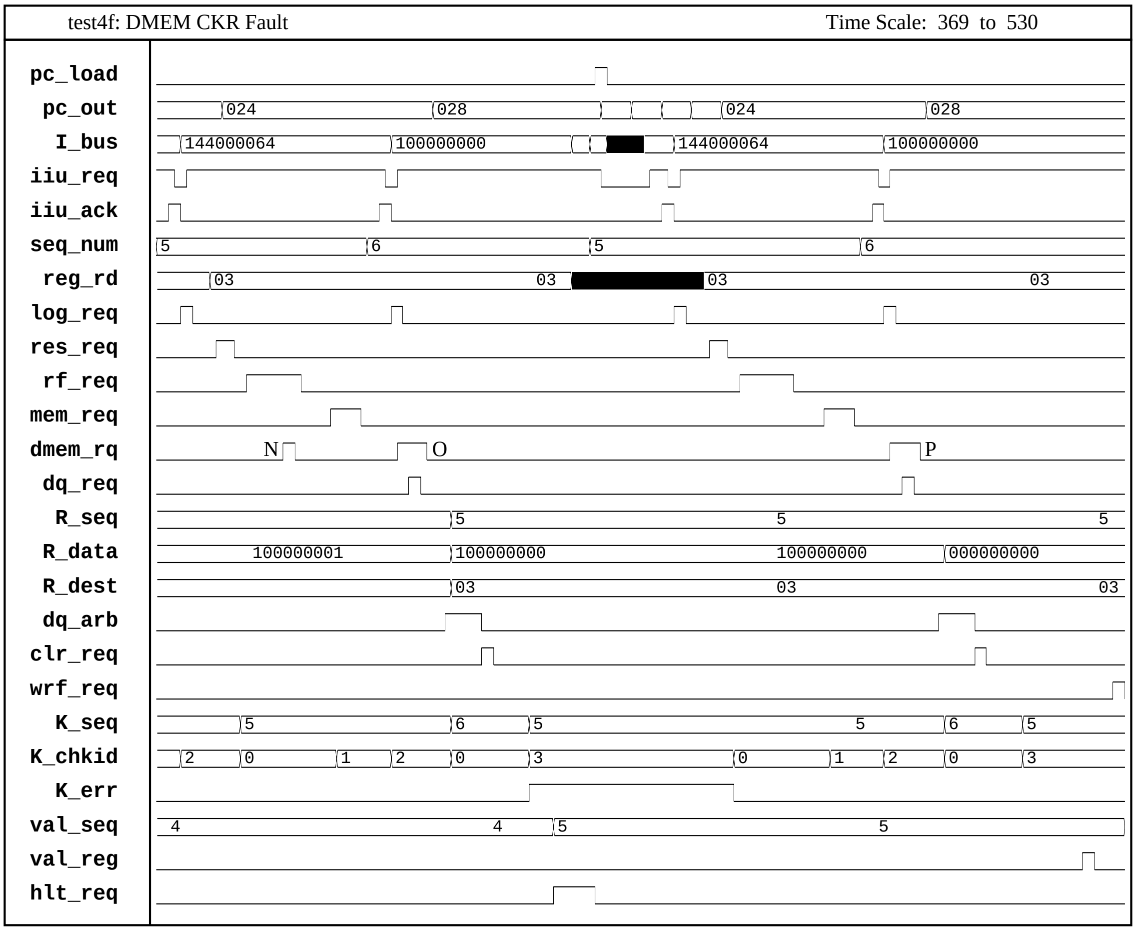

Figure 5.16: Test 4, DMEM CKR Fault

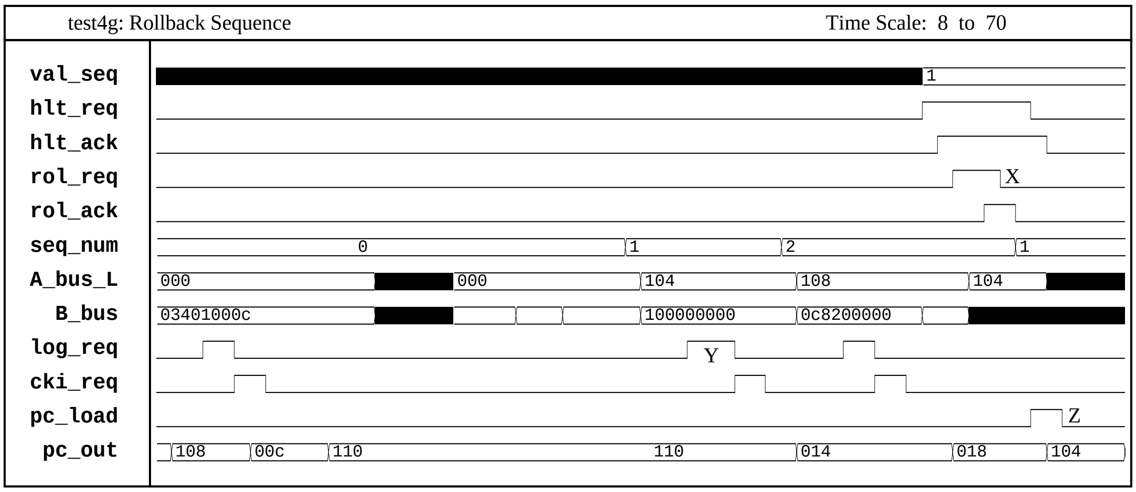

Figure 5.17: Test 4, Rollback Sequence

5.5. Running a Real Program

A simple, *useful* program is shown below to exercise various parts of the processor while causing three different faults. The program should be self-explanatory, and the simulation waveforms are displayed in Figure 5.18. Even though the details are too small to be traced, some properties, such as the variations in register reservation delays *res_req* and occurrences of rollback *hlt_req*, can still be observed.

```
          //    ; test5.a -- find the largest number
          //
13401003C // 00         *ori    r1, r0, first
02402004C // 04         addui   r2, r0, last
100001825 // 08         or      r3, r0, r0     ;current largest number
          //
18C240000 // 0C loop:   lw      r4, 0(r1)       ; load from memory
10083282B // 10         sgt     r5, r4, r3      ; r5 = 1 if r4 > r3
010A00004 // 14         beqz    r5, smallr
000041825 // 18         or      r3, r0, r4      ; found larger number
000223028 // 1C smallr: seq     r6, r1, r2      ; check if done
114C00008 // 20         bnez    r6, done
124210004 // 24         addui   r1, r1, 4       ; increment pointer
00BFFFFE0 // 28         j       loop
          //
034070050 // 2C done:   ori     r7, r0, largst
1CCE30000 // 30         swf     0(r7), r3       ; store in memory
044000003 // 34         trap    3               ; print the number
144000064 // 38         trap    100             ; stop
          //
00000000A // 3C first:  .word   10
100000005 // 40         *.word  5
10000000D // 44         .word   13
100000002 // 48         .word   2
000000009 // 4C last:   .word   9
          //
000000000 // 50 largst: .word   0
```

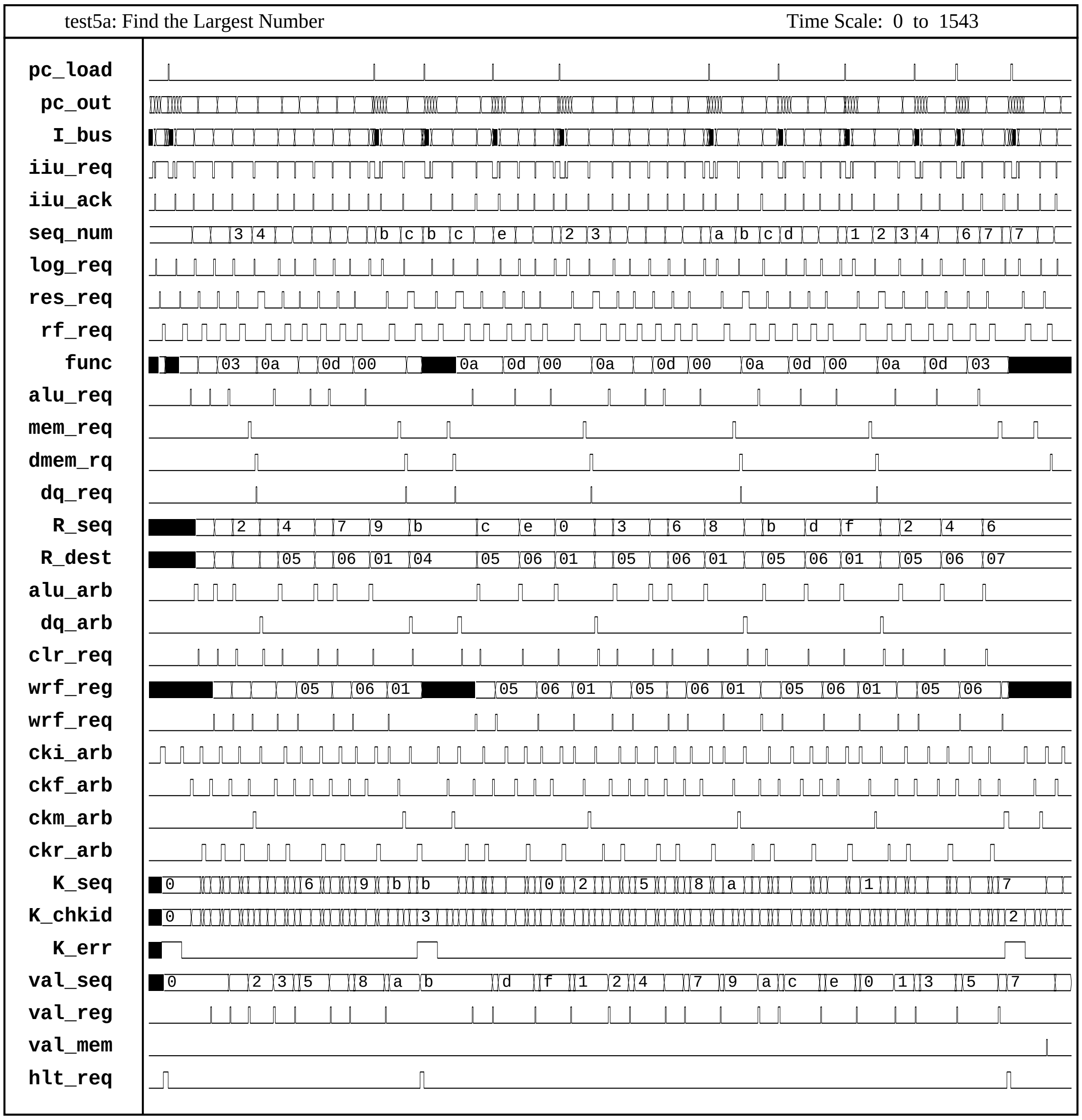

Figure 5.18: Test 5, Find the Largest Number