# *Chapter Six*

# Hardware Design and Gate-Level Simulation

*Gates.v* in Appendix A is a collection of gate-level components, from which more complex circuits are built in other Verilog modules. The main purpose of this chapter is to show how some structures used in the behavioral Verilog model may be built in hardware. Some circuit diagrams contain high level logic gates for simplicity and clarity, but more efficient implementations may be found at the full transistor level.

## 6.1. C-Elements

C-element is a very basic unit in a self-timed system, and a NMOS 2-input C-element appears on [Seit80, p. 255]. The number of inputs may be extended by adding transistors to the parallel and series structures, as shown in Figure 6.1. However, the number of transistors in series should be limited because of the body effect.

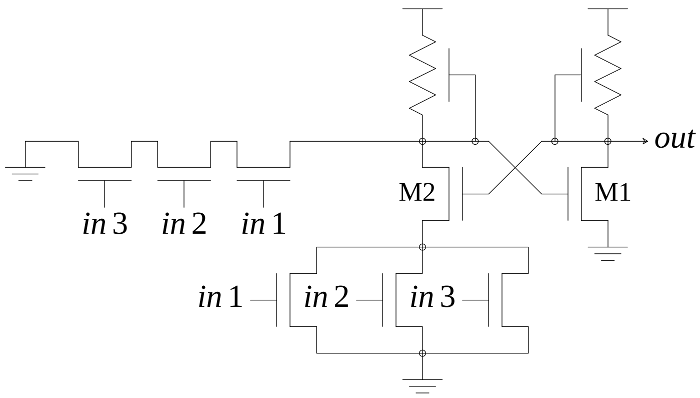


**Figure 6.1:** 3-Input C-Element

When a C-element with large number of inputs is required, it can be easily built from smaller C-elements because its function is associative. The 14-input C-element used in *bigc.v* is shown in Figure 6.2. Sometimes a C-element needs to be reset to clear its state, and that can be accomplished by forcing the inputs to be zeros. Figure 6.3 represents the *muller*2*c* module in *gates.v*. Other C-element implementations can be found in [Berk91, Jaco90, Suth89].

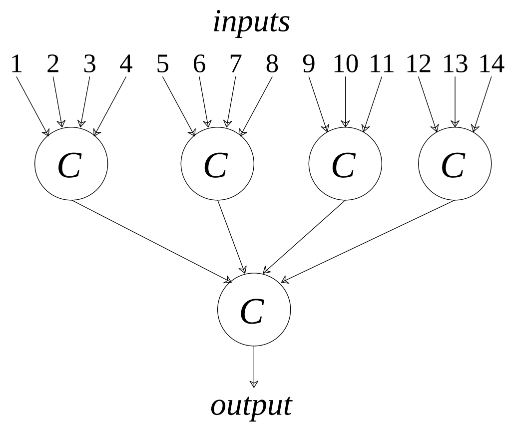


**Figure 6.2:** 14-Input C-Element

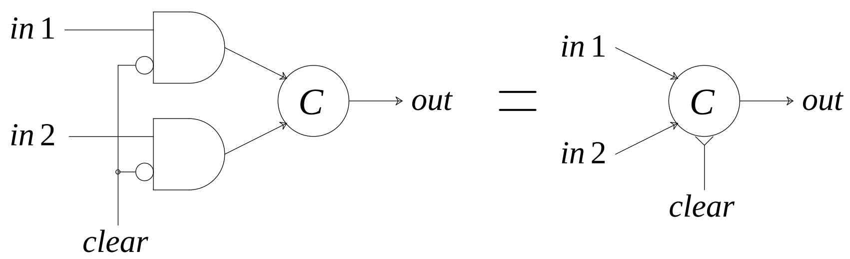


**Figure 6.3:** 2-Input C-Element with Clear

All transistor and gate delays in the Verilog model are set to 1 unless there are timing issues involved. This code segment comes from the *muller*3 module:

```
gates.v   29:     nmos    #2 m1 (out, gnd, b);
gates.v   30:     nmos    #1 m2 (b, a, out);
```

Transistors M1 and M2 are marked in Figure 6.1. The cross-coupled structure needs to be biased so that oscillation would not occur in simulation, due to *exactly* matched delays when two or more inputs switch at the same time in opposite directions. This kind of switching does not occur under the 4-phase handshake protocol, except during local reset when the transaction is canceled anyway. In a real circuit, the C-element should not oscillate.

### 6.2. Arbiter

Another circuit, an interlock element, is also presented on [Seit80, p. 261]. It is shown in Figure 6.4 along with logic for rollback signals, which make up the arbiter for R_bus. This arbiter is non-prioritized, and multiple modules can be connected in a binary-tree fashion to support more than two arbitration requests.

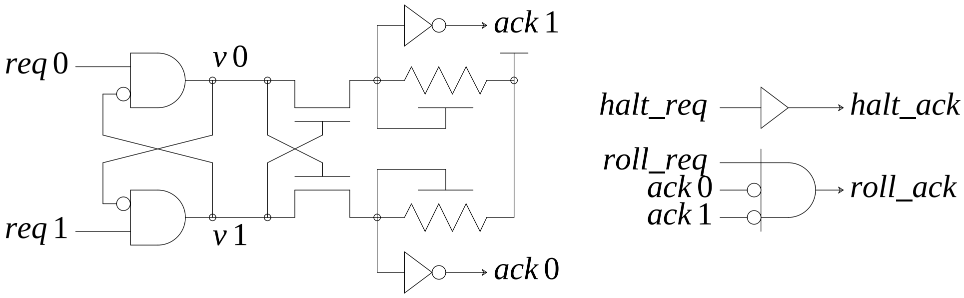


**Figure 6.4:** Arbiter for R_bus

Since the arbiter is not directly involved in transferring data, it is always reset when a rollback occurs. As long as the actual data transmitter and receiver handle the rollback sequence properly, the outputs of the arbiter do not have to be stopped before sending *halt_ack*. This simplification allows the arbiter to run at full speed in normal operations. The 3-input AND gate delays the generation of *roll_ack* until all arbitration grants (*ack*s) are released.

Figure 6.5 is the simulation result of running test program #3 (section 5.3) with the gate-level arbiter. *Alu_arb* is connected to *req*0, and *dq_arb* is connected to *req*1. When both requests become active at the same time, ALU wins because the circuit is

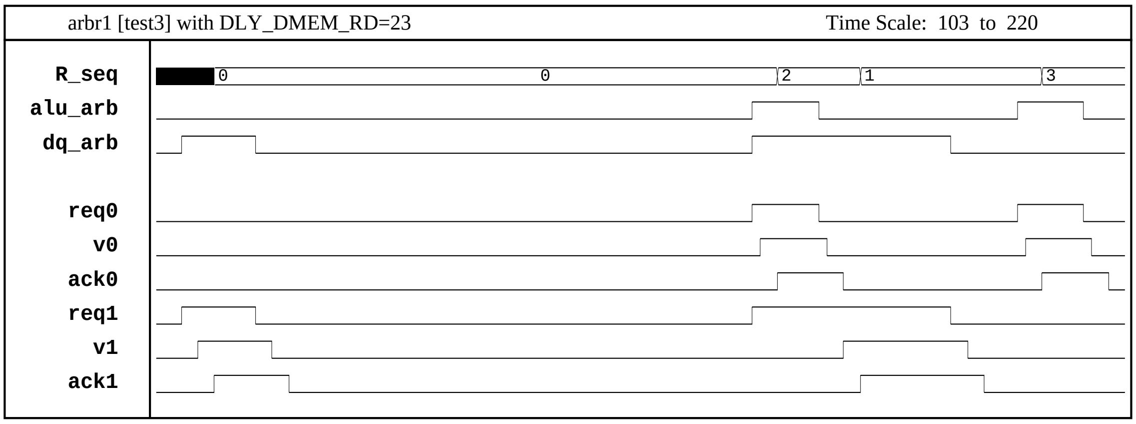


**Figure 6.5:** Gate-Level Arbiter

biased toward *req* 0 due to the cross-coupled AND gates. In actual implementation, slight mismatch between cross-coupled pairs would help combat the problem of metastability.

## 6.3. Register Completion Detector

Before the actual completion circuit is discussed, some latches need to be introduced first. Figure 6.6 is a simple latch made from inverters and pass gates. For actual circuit implementation, it may be more desirable to use full CMOS pass gates to reduce power consumption, but they do not make any difference in logic simulation. Figure 6.7 adds the capability to set or reset the latch by connecting one of the S/R transistors (but not both). Again, the NMOS pull-up is used only for simplicity.

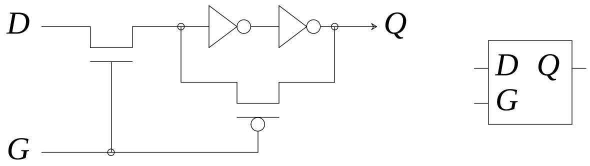


**Figure 6.6:** D-Latch

After data is latched in a register, a signal has to be generated to notify other

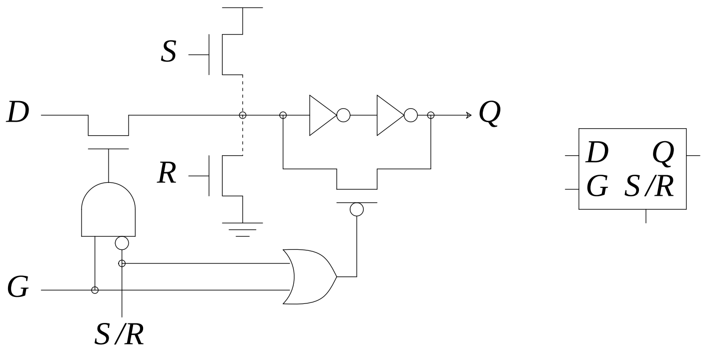


**Figure 6.7:** D-Latch with Set/Reset

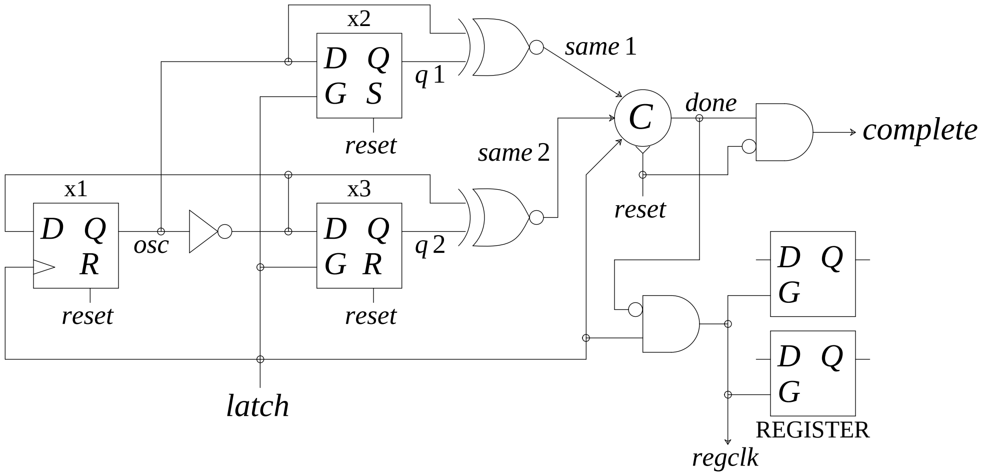


**Figure 6.8:** Register Completion Detector

circuits about its completion. A fully self-timed circuit requires a completion signal generator (XNOR) for *each* bit of the register and a large C-element to merge them. The overhead is very high, especially if the processor has a high proportion of register elements.

The completion scheme used in AMPIRE is shown in Figure 6.8. Component *x1* is a master-slave flip-flop made from two resettable D-latches. With an inverter connected from its output back to the input, the output is flipped every time a latch request is received. *X2* and *x3* model the delay of the actual register, and they are most likely to be slower than the real register on the same die (compare Figures 6.6 and 6.7). Together with the flip-flop "oscillator", the outputs *q1* and *q2* are switched every latch cycle, modeling both rise and fall delays.

For the register, the gate is opened when the *latch* signal becomes active. After the signal *done* becomes high, the gate is closed, protecting the register from further changes at the input. Therefore, the latches in the register behave like edge-triggered flip-flops without the additional hardware complexity. The register can be arbitrarily long, but the completion circuit must be placed properly to take the wire delays into account. Note that the *reset* signal is not necessarily the global processor reset. Simulation waveforms will be shown with the gate-level instruction queue.

### 6.4. Instruction Queue

The buffers in AMPIRE may be separated into two groups: the ones which are always cleared during rollback or other conditions, and the ones with selective invalidation. The IQ buffer falls into the former group and will be presented here, and the latter type will be discussed in the next section. The actual storage elements are simple latches that can be used with the register completion detector in the previous section. What we need is additional circuitry to communicate with other modules or sub-modules.

The IQ buffer control is shown in Figure 6.9. The two C-elements make up a 4-phase full-handshake circuit, and they represent the input and output cycles in the behavioral code. Please see [Meng89] for detailed description on this handshake circuit. *Cancel_r* is asserted when a rollback or branch occurs. The buffer for *cancel_a* has to provide enough delay to reset the control circuit, including the register completion unit. A better method should really be used to generate that

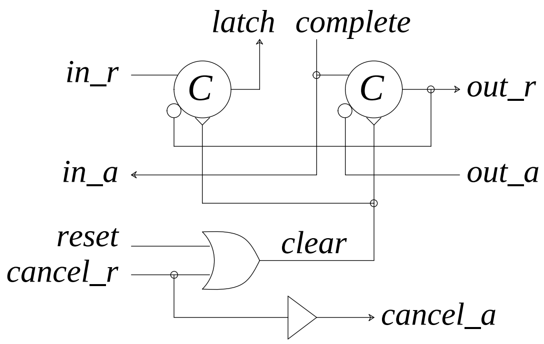


**Figure 6.9:** Control for IQ Buffer

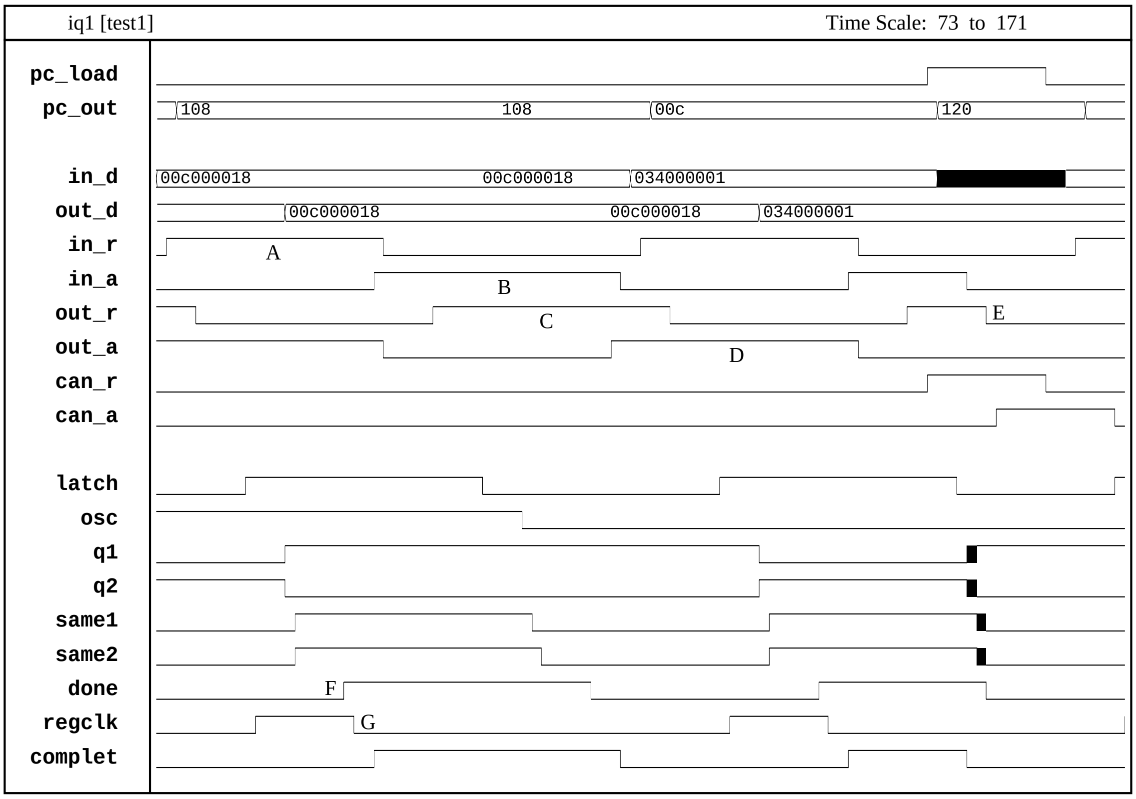


**Figure 6.10:** Gate-Level IQ

acknowledgment because matching the delays of a large circuit may be difficult and not always reliable.

Figure 6.10 is simulated with full gate-level buffers, running test program #1 (section 5.1). Pulses (A,B) and (C,D) form the 4-phase handshake pairs for input and output, respectively. The output request C can be started before the input acknowledgement B is finished because the data is already latched and ready for output.

Looking at the signals inside the register completion detector, the *done* signal at point F becomes high after the input data has propagated to the output of the latch, as checked by the two *same* signals. Then the register clock *regclk* is turned off at point G. When a branch is taken, the queue is cleared in response to the cancel request *can_r*, which is connected directly to *pc_load*. As a result, the signal *out_r* is dropped at point E without receiving a corresponding acknowledgement.

## 6.5. Data Queue

Unlike the IQ buffer, the buffers in DQ, DWBs, checkers, etc. have to be selectively invalidated in the event of a rollback. Since invalidation is based on sequence number comparison, the comparator hardware will be discussed next.

### 6.5.1. Sequence Number Comparator

The rollback code was introduced in section 4.1.6, and these lines describe what needs to be done to determine if data should be invalidated:

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

The subtractor can be simplified because only the highest bit of the difference is used. Four bits are allocated for the sequence number in AMPIRE, and therefore, we need three bits of borrow (as opposed to carry) and one bit of difference circuits, as shown in Figure 6.11. BRWEND is a borrow generator without borrow-in, for the least significant bit. The logic equations for these blocks are (for A–B):

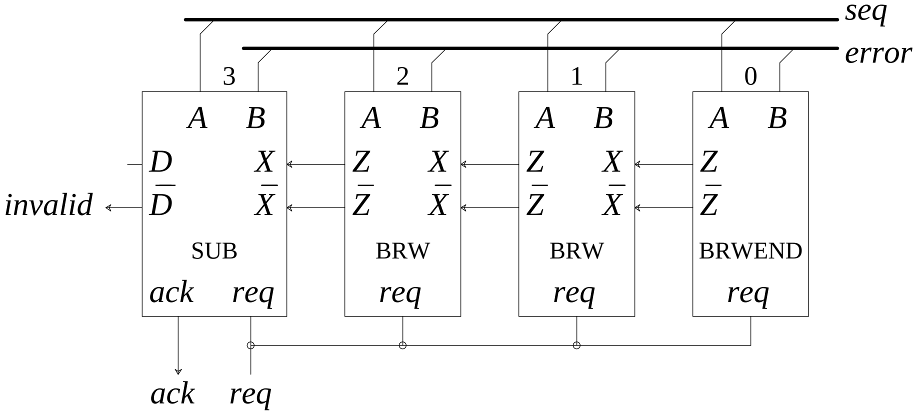


**Figure 6.11:** Sequence Number Comparator

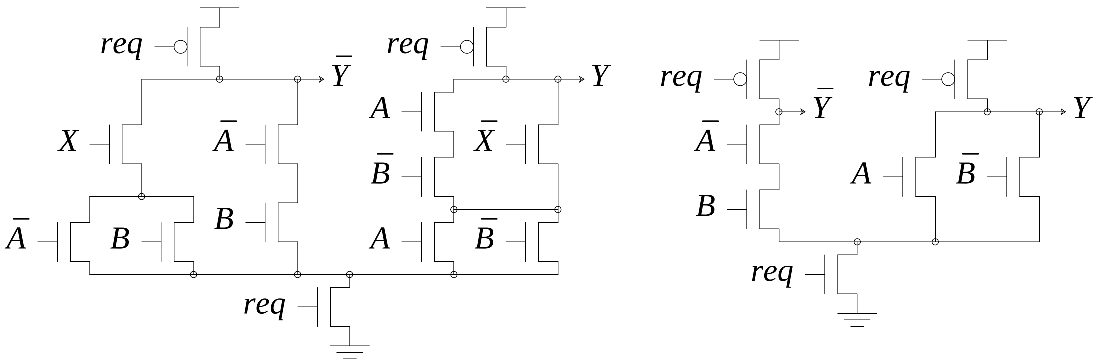


**Figure 6.12:** Borrow Circuits, Full and Half

BRWEND: $Z = \overline{A}B$

BRW: $Z = \overline{A}B + \overline{A}X + BX = \overline{A}B + X(\overline{A} + B)$

SUB: $D = A \text{ xor } B \text{ xor } X$

The self-timed components appear in Figures 6.12 through 6.14, which are designed using a technique called DCVSL, differential cascode voltage switch logic [Jaco90, Meng89]. The PMOS transistors precharge the $Y$ and $\overline{Y}$ nodes when *req* is low, and the NMOS transistor that is gated by *req* prevents premature discharge. When the request signal is activated, only one of the precharged nodes is pulled low because the two NMOS trees are complementary. The output inverters are added, as

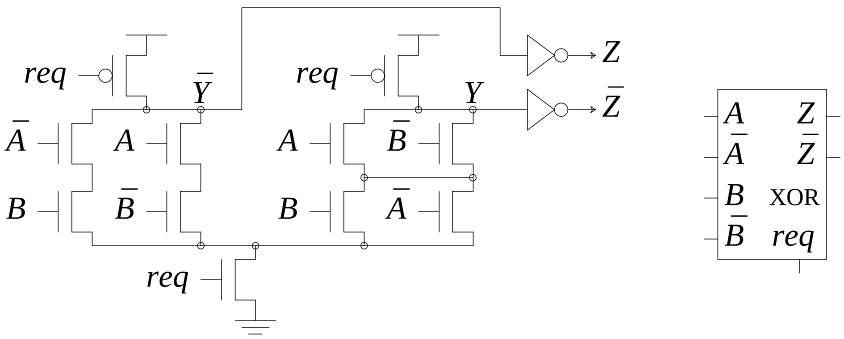


**Figure 6.13:** XOR Gate

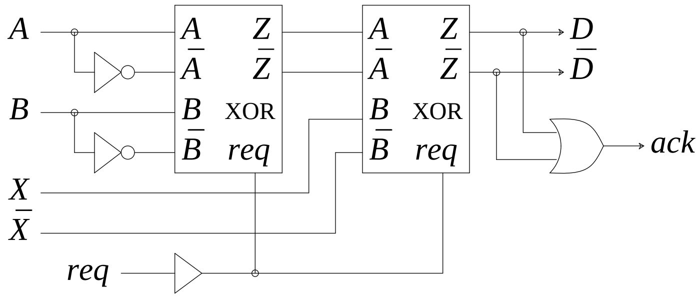


**Figure 6.14:** Difference (SUB) Circuit

done in Figure 6.13, so that the voltage level is low when the circuit is inactive or not ready. The differential outputs can then be fed directly into another DCVSL circuit. At the final stage of the chain (SUB), the outputs are ORed together to generate the completion/acknowledgement signal. Because the circuit is self-timed, completion time varies with the data being compared.

### 6.5.2. DQ Control

The control circuit for the DQ buffer is shown in Figure 6.15. The register completion detector is interfaced through *latch* and *complete*, like the IQ buffer. A sequence number comparator cycle is started with *roll_req*, and the acknowledgement *cmp_done* is raised when the result *invalid* is available.

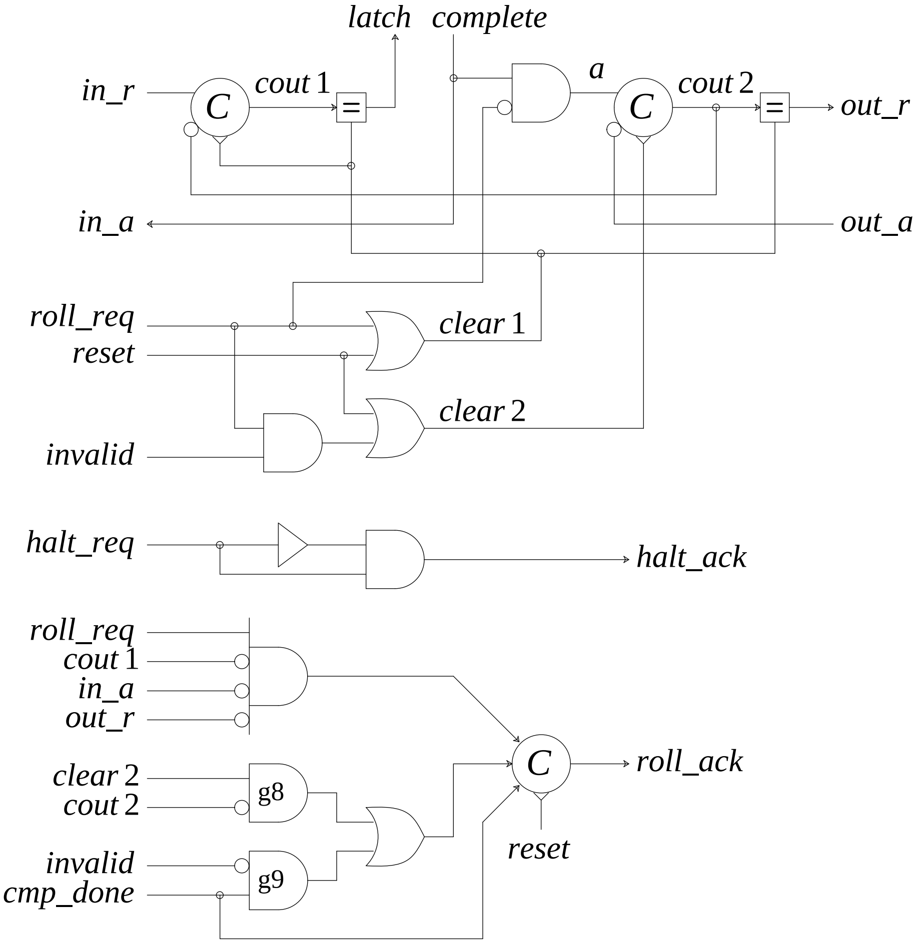


**Figure 6.15:** Control for DQ Buffer

The square part with equal sign is just a standard resettable D-latch, with its *gate* connected to $\overline{halt\_req}$ (not shown) and its *reset* connected to *clear1*. In normal operations, it is in the transparent mode (output = input). When a rollback cycle is started, the inputs are cut off because $\overline{halt\_req}$ becomes low, and thus accomplishing `wait (~halt_req)` in the behavioral code. However, *complete* cannot be interrupted because if data is already latched, the sender *must* be notified, or data duplication may occur when normal operation is resumed (see section 4.1.6). The delay buffer in the *halt_ack* circuit is provided for that purpose.

The *cout2* signal is equivalent to the *valid* variable in the behavioral code, and it is cleared in the rollback process only if the corresponding register is invalidated, and consequently activating *clear2*. The AND gates *g8* and *g9* partially determine when the invalidation process is finished. If invalidation is not necessary, then it can proceed once the comparison is done (*g9*). Otherwise, *roll_ack* cannot be sent until the *cout2* signal is reset (*g8*).

### 6.5.3. DQ Simulation

A couple of test programs will be run to show the internal signals of the DQ buffer and the sequence number comparator. Both programs read data from DMEM, which are sent to the REGDWB through DQ. The first program causes a CKI fault, but it does not invalidate LW.

```
          //    ; test6.a -- gate level DQ, no invalidation
          //
          //    ; DLY_DMEM_RD=50, DLY_CKI_CHK=48 and 52
          //
08C010000 // 00 lw      r1, 0(r0)
100000000 // 04 *nop
144000001 // 08 trap    1
144000064 // 0C trap    100
```

In Figure 6.16, the output request *out_r* is set at point A, but the halt request is also activated at the same time. *Out_r* is cleared by *clear1* at point C, but since *cout2* retains its value through the rollback process (not invalidated), another handshake cycle is started at point B.

The signals in the lower half of the figure belong to the sequence number comparator, showing 0–1. (_a,na) through (d,inv) are the differential output pairs of the four sub-blocks in Figure 6.11. The waveforms are staggered because each one is dependent on the result from the preceding stage.

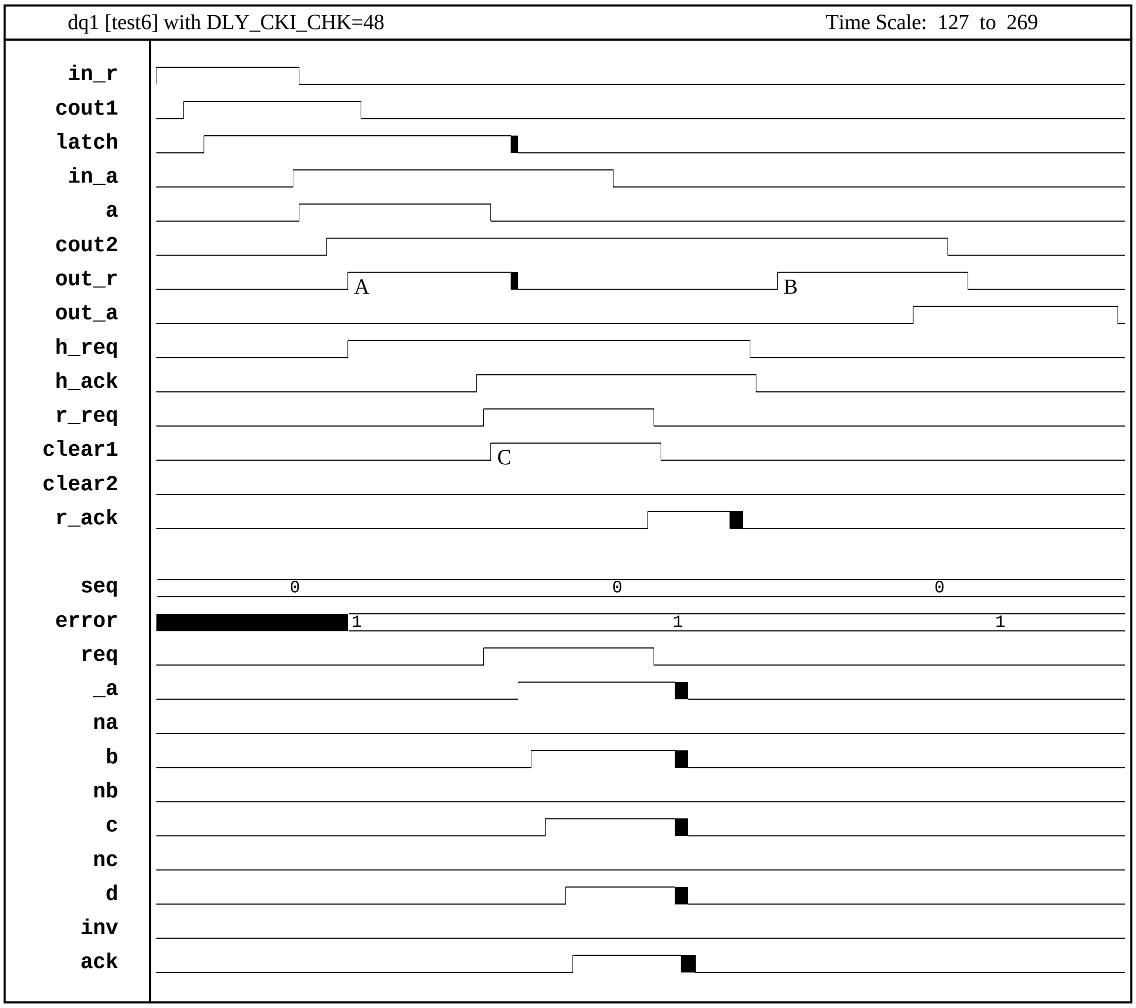


**Figure 6.16:** Test 6, Gate-Level DQ with DLY_CKI_CHK=48

By delaying the rollback cycle a little, we get Figure 6.17. By the time *h_req* becomes high at point G, the receiver has already started latching its register. That is completed at point F, and the the acknowledgment clears *cout2* at point D. The signal *out_r* is held constant by a closed latch, and therefore, it is not dropped until point E, after *clear1* is raised.

The next program causes LW to be invalidated because of an error in the preceding instruction.

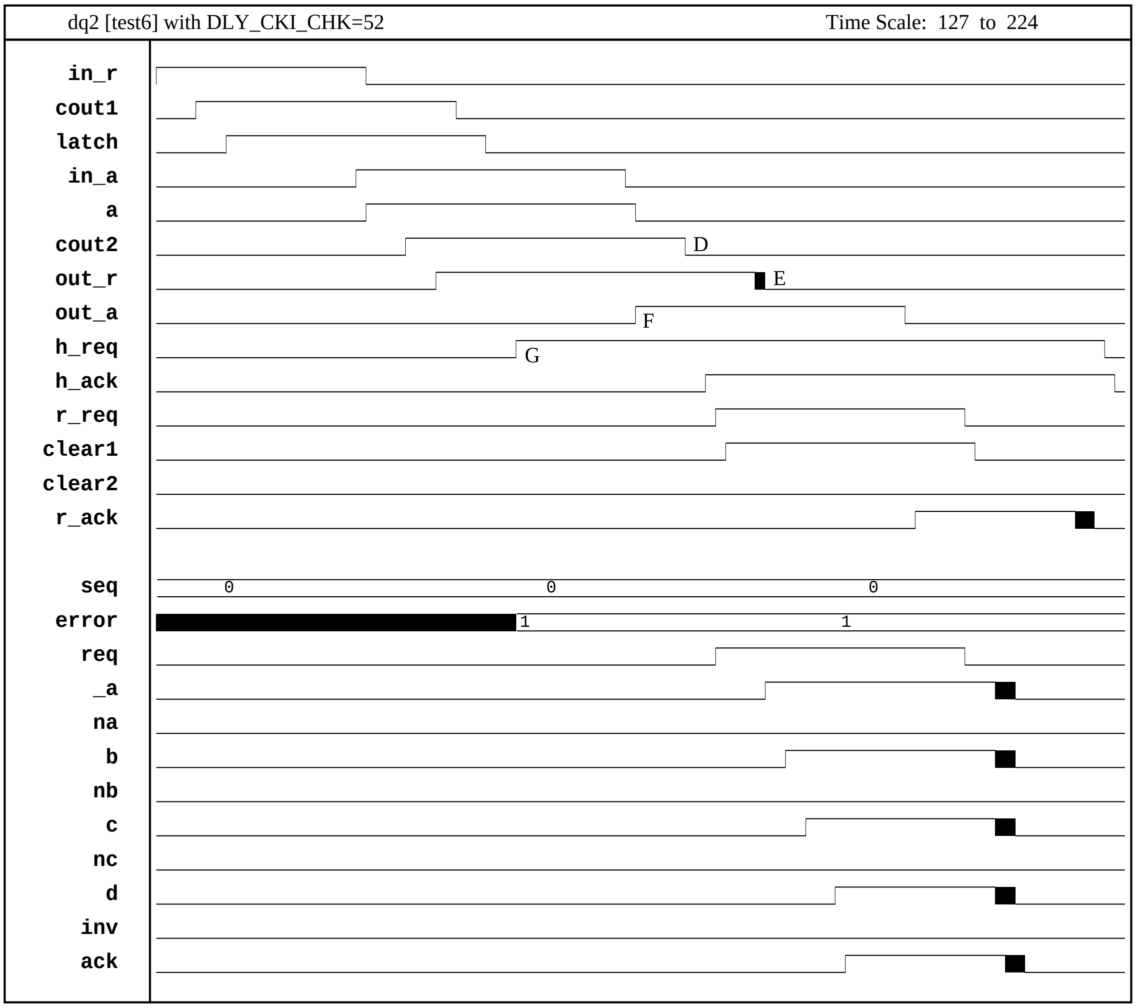


**Figure 6.17:** Test 6, Gate-Level DQ with DLY_CKI_CHK=52

```
          //    ; test7.a -- gate level DQ, invalidation
          //
          //    ; DLY_CKI_CHK=15, DLY_DMEM_RD=8 and 1
          //
000000000 // 00 nop
000000000 // 04 nop
000000000 // 08 nop
100000000 // 0C *nop
08C010000 // 10 lw      r1, 0(r0)
144000001 // 14 trap    1
144000064 // 18 trap    100
```

In Figure 6.18, an input request *in_r* is received and causes *cout1* to become high. However, that signal is not propagated to *latch* because the path is already closed by

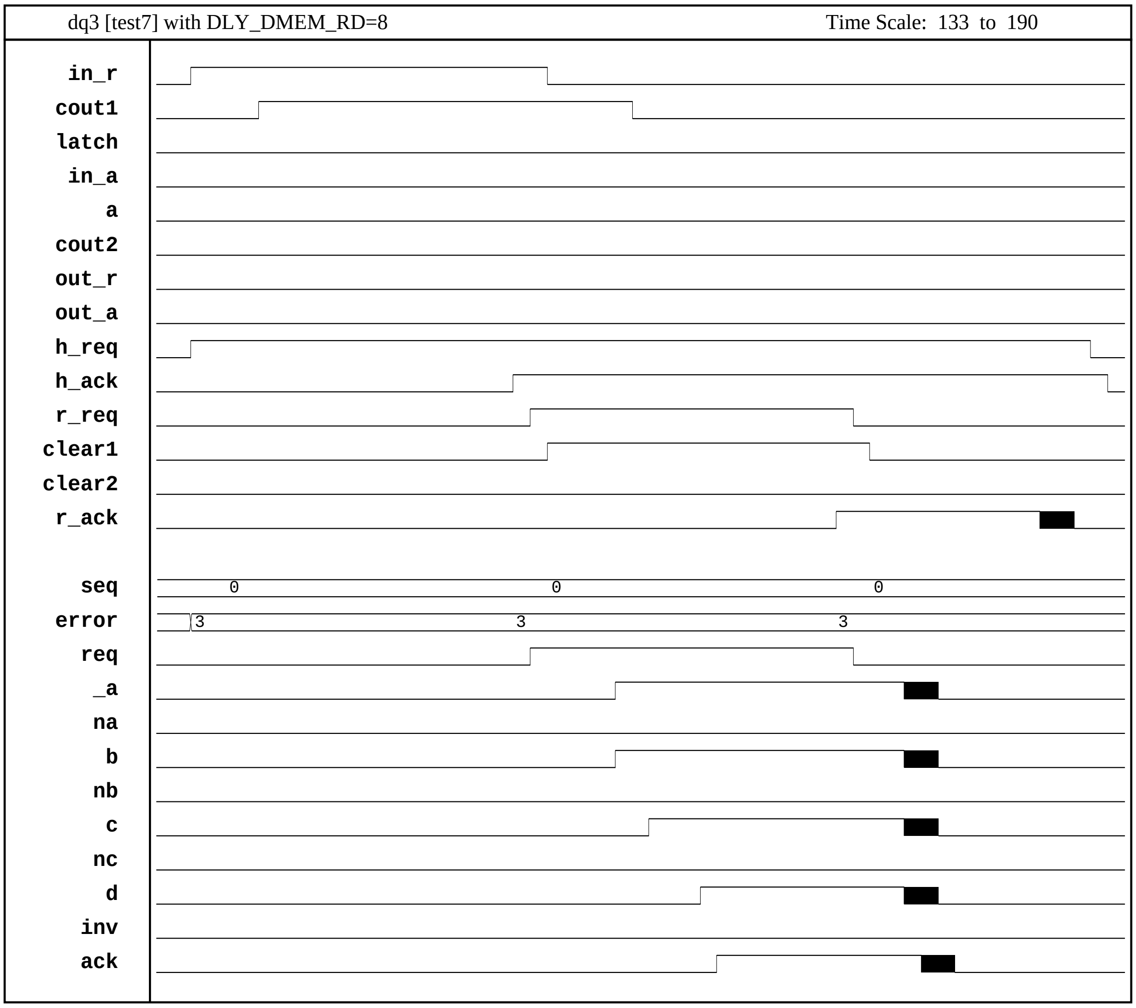


**Figure 6.18:** Test 7, Gate-Level DQ with DLY_DMEM_RD=8

$\overline{h\_req}$. Since the data is not received by this buffer, it is invalidated somewhere else. In Figure 6.19, the sequence number comparison results in the *inv* signal being high at point I. Then *clear2* invalidates the data by resetting *cout2* at point H. Notice that there is no activity on the *out_r* line during this period of time.

The sequence number comparisons performed in Figures 6.18 and 6.19 are: 0–3 and 4–3 (0000–0011 and 0100–0011 in binary). For both cases, the borrow generation for bit 1 is not dependent on the result from bit 0. Therefore, *_a* and *b* become high at the same time, reducing the time for borrow propagation.

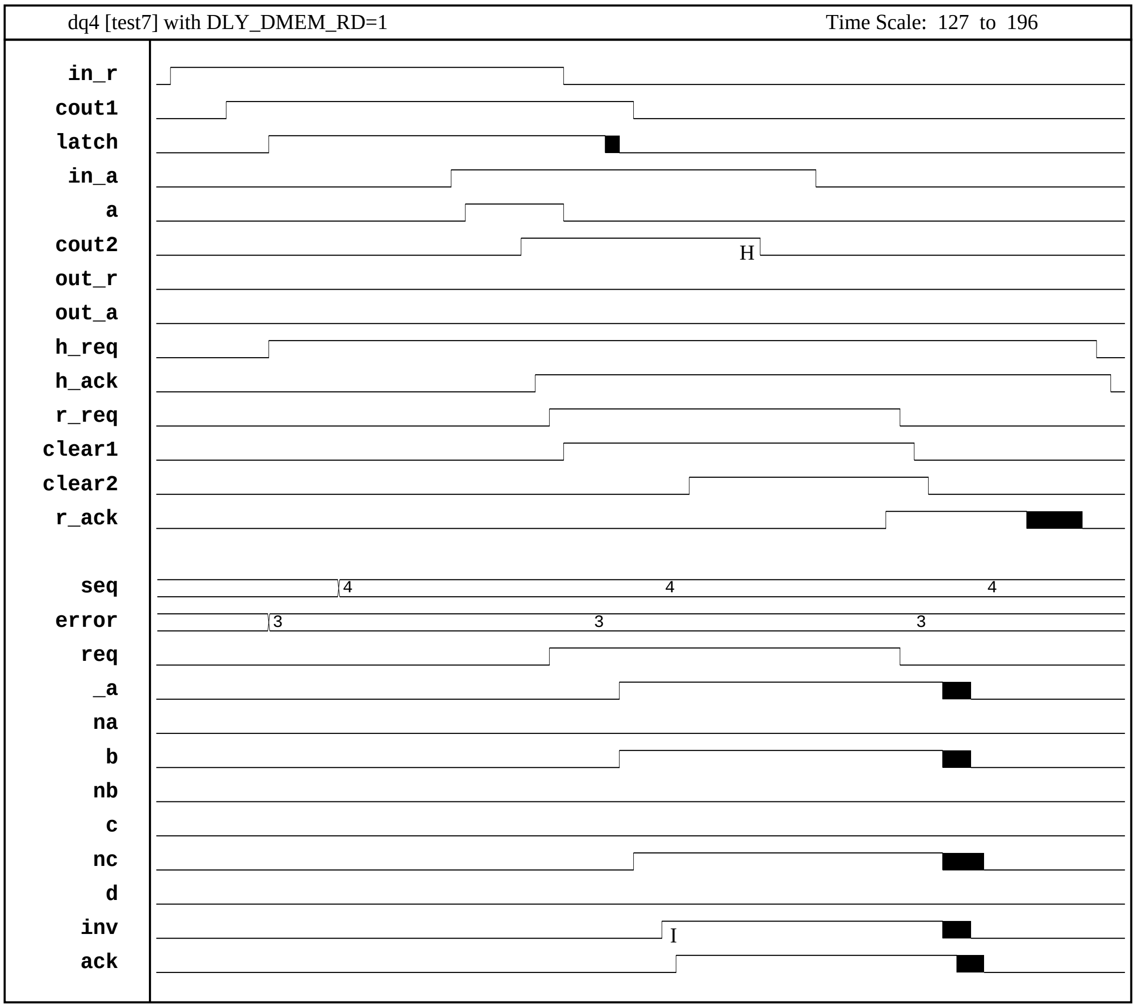


**Figure 6.19:** Test 7, Gate-Level DQ with DLY_DMEM_RD=1

## 6.6. Fault Simulation with Gate-Level Modules

Four modules in AMPIRE have gate-level models: BIGC, ARBR, IQ, and DQ. Each one can be switched through declarations in the *parameter* file. Figure 6.20 is the result of running test program #4 (section 5.4) with all gate-level models enabled. The waveforms are very similar to the ones in Figure 5.11, which is from the behavioral model. Figure 6.20 does have an extra REGFILE write request pulse at point A. The data is not actually written at that time because it is interrupted by the rollback cycle at

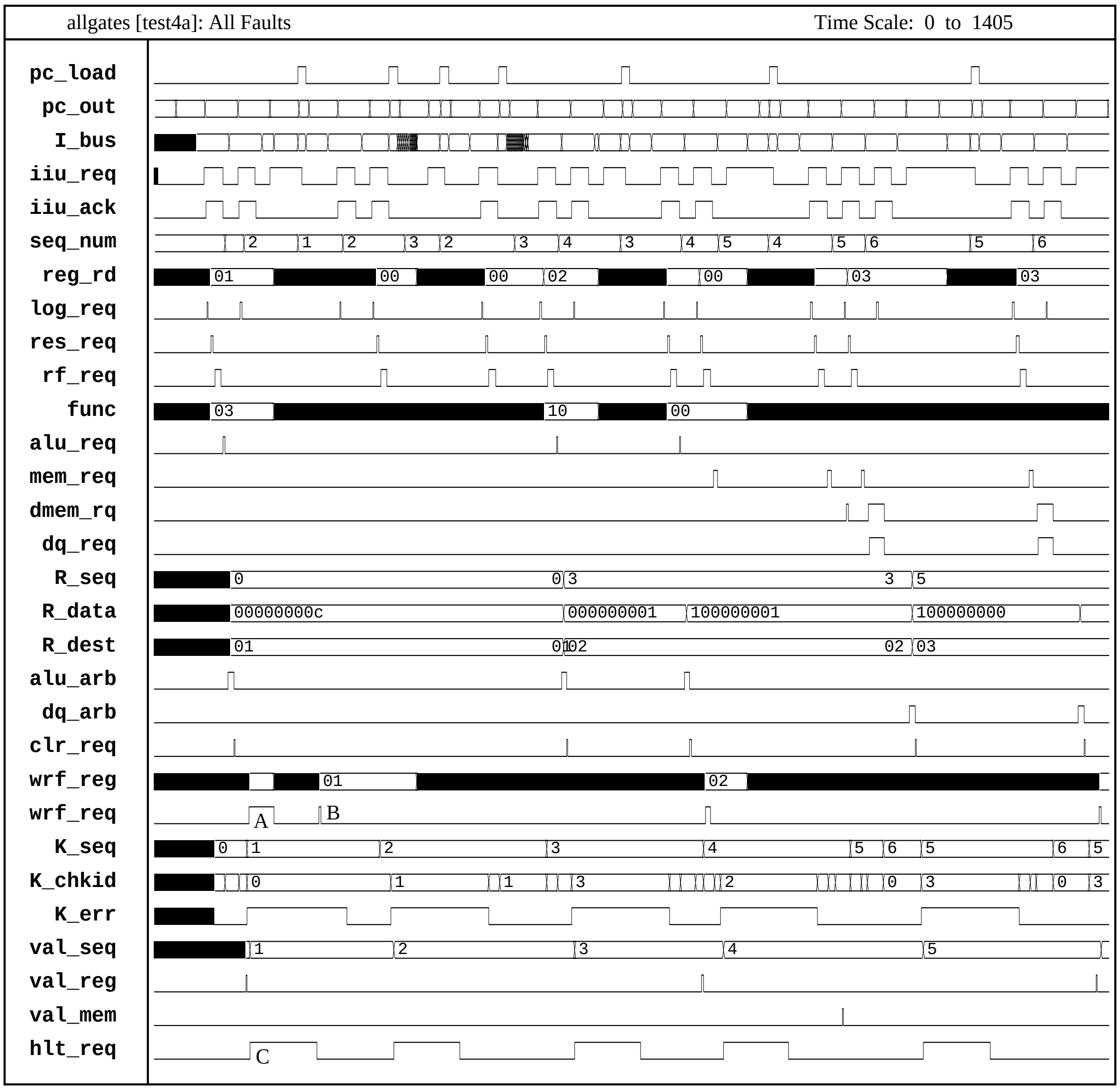


**Figure 6.20:** Fault Simulation with Gate-Level Modules

point C. After the processor is restarted, the transaction is repeated at point B.