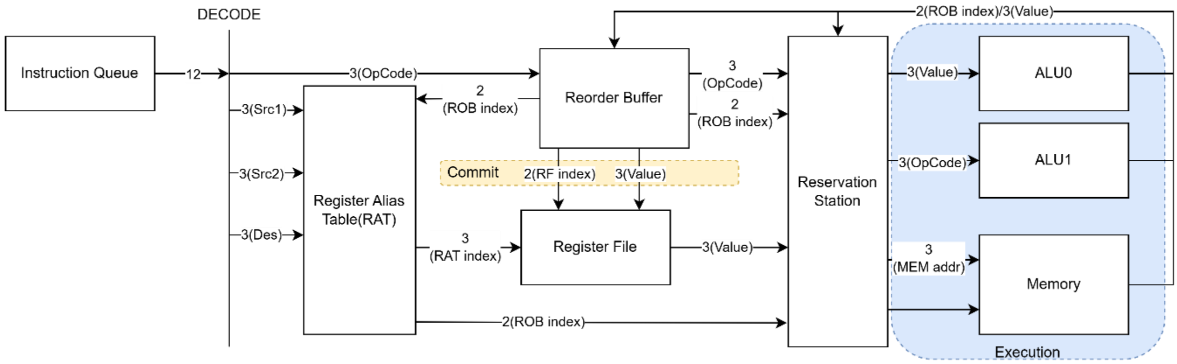
Project Milestone(Tiny OoO CPU)-Amin Hong

Data Schematics



Trying to stay in the 2000 cell limitation requirement, I set every module tiny that could at least show some OoO execution.

The first step in this design is that as an instruction is received from the input, it is placed into the Instruction Queue. When the instruction reaches the head of the queue, it is decoded, and the system checks whether there is available space in the Reorder Buffer (ROB) and the Reservation Station. If both have available entries, the instruction is issued to the ROB and Reservation Station.

In the Reservation Station, the system determines whether the source operands are immediately available or if they are still waiting due to data dependencies. Once all required operands are ready, execution is performed in an ALU or memory unit during a designated clock cycle.

After execution completes, the results are updated in both the Reservation Station and the ROB. When the instruction at the head of the ROB is ready to commit, its value is written back to the Register File. This ensures correct out-of-order execution (OoO) while maintaining in-order commitment of results.

Modules are set as follows:

Instruction Queue : 8 entry

Opcode(3bits)	Dst(3bits)	Src1(3bits)	Src2(3bits)

Opcode

Opcode	3bit
ADD	000
SUB	001
AND	010
OR	011
XOR	100
LD	101
	111

ADD/SUB/AND/OR/XOR(2cycle)

Opcode	Dst	Src1	Src2

LD(5cycle)

Opcode	Reg	Reg	Mem address

Register Alias Table(RAT) : 8 entry

Q-ROB index(2bit)	RF valid(1bit)

Reorder Buffer(ROB) : 4 entry

Opcode(3bit)	Value(3bit)	Dst(3bit)	Status(1bit)

Register File(RF) : 8 entry

Value(3bit)

Memory (RF) : 4 entry

Value(3bit)

Reservation Station :

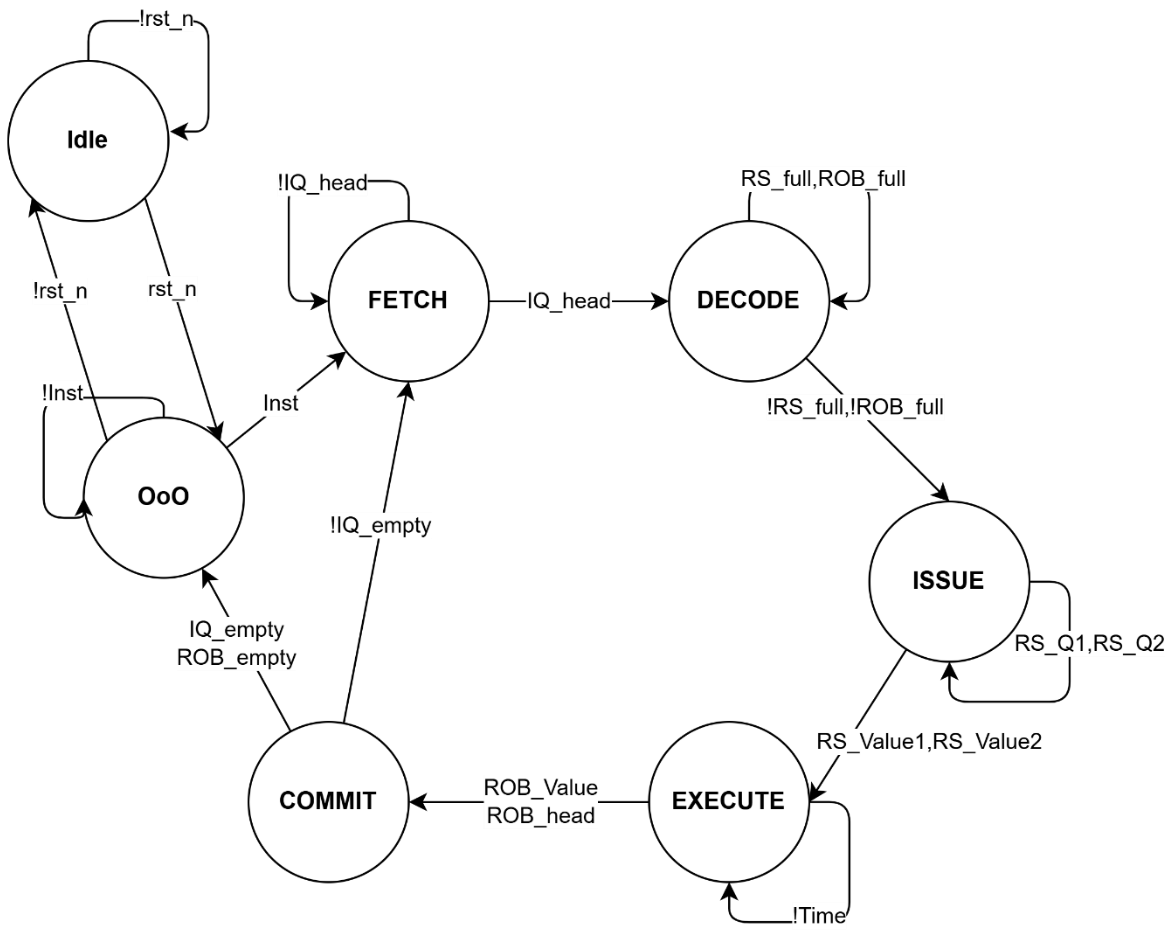
ALU reservation station 2 entry

ROB index (2bit)	Time (2bit)	Busy (1bit)	Opcode (3bit)	Value1 (3bit)	Value2 (3bit)	Q1- ROB index (2bit)	Q2- ROB index (2bit)

LD/STR reservation station 1 entry

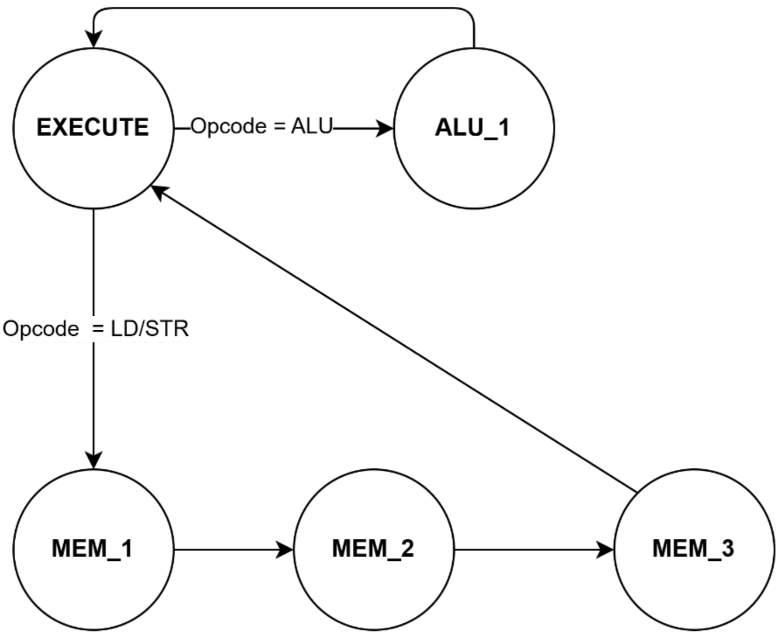
ROB index (2bit)	Time (2bit)	Busy (1bit)	Opcode (3bit)	Value1 (3bit)	Q1- ROB index (2bit)

FSM



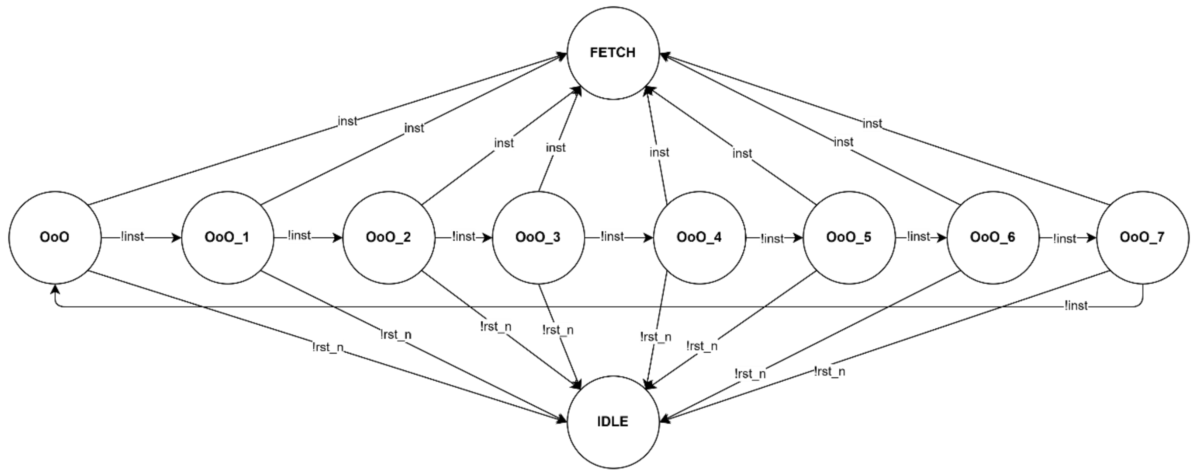
- OoO reads the instructions and fetch in instruction queue
- Instructions move from FETCH to DECODE when available in the Instruction Queue (IQ_head)
- DECODE waits if Reservation Station or Reorder Buffer is full (RS_full, ROB_full)
- ISSUE waits for source operands to be ready (RS_Q1, RS_Q2)
- EXECUTE waits for operation completion (!Time)
- COMMIT updates architectural state and determines whether to start a new instruction cycle

EXECUTE



- ALU operations require 2 cycles to complete (including return to EXECUTE)
- Memory operations require 4 cycles to complete (including return to EXECUTE)

OoO



- Check the register value during OoO state
- Each state prints each register value (OoO_1 prints r1 index, r1 value)
- Clk cycle is also printed in each state.

Testbench

#1 LD R1 MEM[0]	**-**
#2 ADD R2 R1 R3	**RAW(R1)**
#3 SUB R5 R6 R4	**-**
#4 OR R5 R7 R8	**WAW(R5)**
#5 LD R7 MEM[2]	**WAR(R7)**
#6 XOR R3 R4 R7	**RAW(R7)**
#7 AND R4 R8 R7	**WAR(R4), RAW(R7)**

- Able to check all possible cases in data dependency(RAW,WAW,WAR)
- Limits the number of instruction for test bench due to number of entries in IQ
- In order to check the result set register, memory value as index in idle state(not appropriate in real CPU)
- In OoO state, output prints the # of clk cycle, reg index, reg value

# of clk cycle(6bit)	Reg index(3bit)	Reg value(3bit)

tb.sv(some part)

```
initial begin
  $dumpfile("wave.vcd");
  $dumpvars(0, ooo_cpu_issue_tb);
  $dumpvars(0, uut);

  clk = 0;
  reset = 1;

  instr_mem[0] = {3'b101, 3'd0, 3'd0, 3'd0}; // LD R0, MEM[0]
  instr_mem[1] = {3'b000, 3'd1, 3'd0, 3'd2}; // ADD R1, R0, R2
  instr_mem[2] = {3'b001, 3'd4, 3'd5, 3'd3}; // SUB R4, R5, R3
  instr_mem[3] = {3'b011, 3'd4, 3'd6, 3'd7}; // OR   R4, R6, R7
  instr_mem[4] = {3'b101, 3'd6, 3'd6, 3'd2}; // LD   R6, MEM[2]
  instr_mem[5] = {3'b100, 3'd2, 3'd3, 3'd6}; // XOR R2, R3, R6
  instr_mem[6] = {3'b010, 3'd3, 3'd7, 3'd6}; // AND R3, R7, R6
```

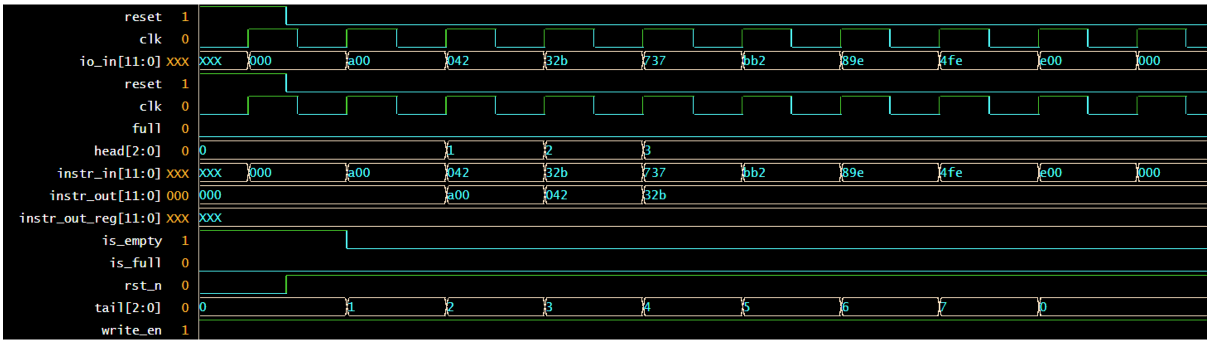
```
    instr_mem[7] = {3'b111, 9'd0};                  // HALT

    #30 reset = 0; // deassert reset after a few cycles
  end

  always_ff @(posedge clk) begin
    if (reset) begin
      idx <= 0;
      io_in <= 12'd0;
    end else if (idx < 8) begin
      io_in <= instr_mem[idx];
      idx <= idx + 1;
    end else begin
      io_in <= 12'd0;
    end
  end
```

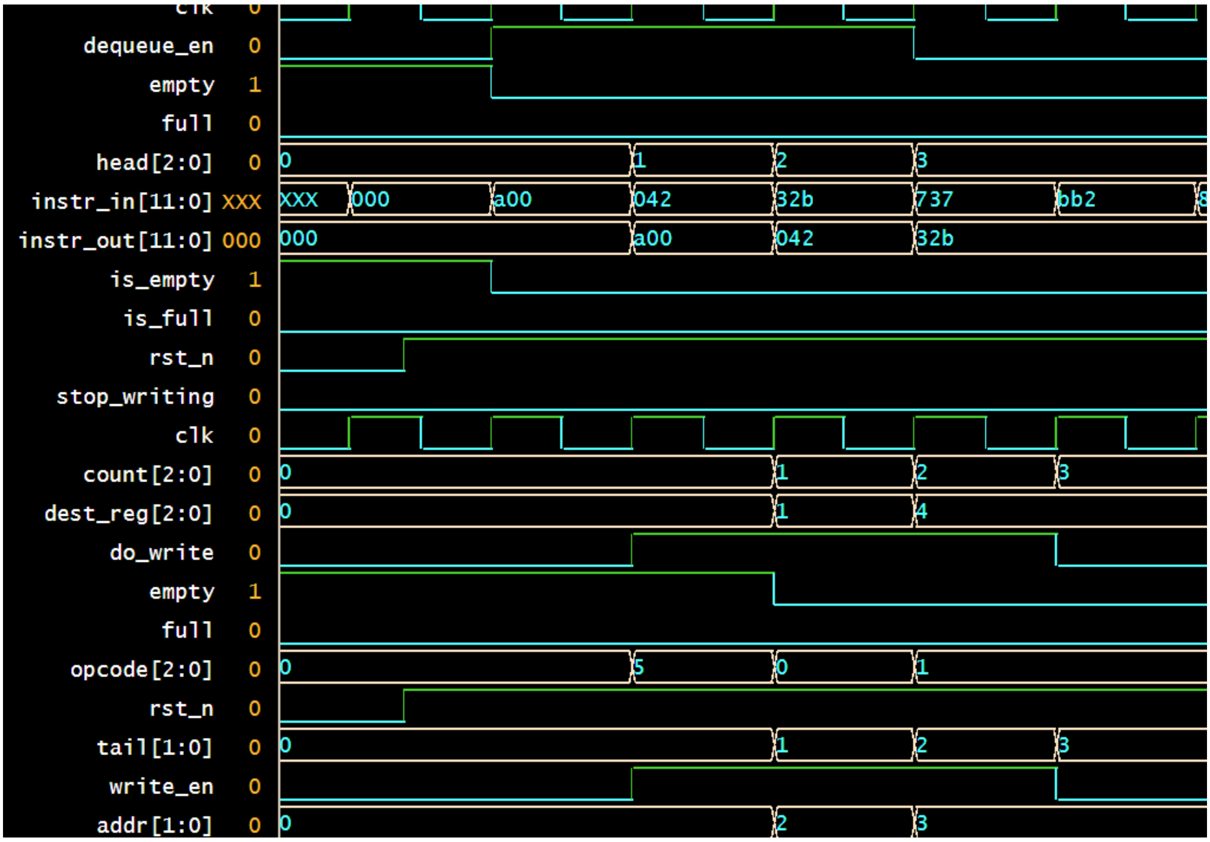
Test Case Analysis

Fetch



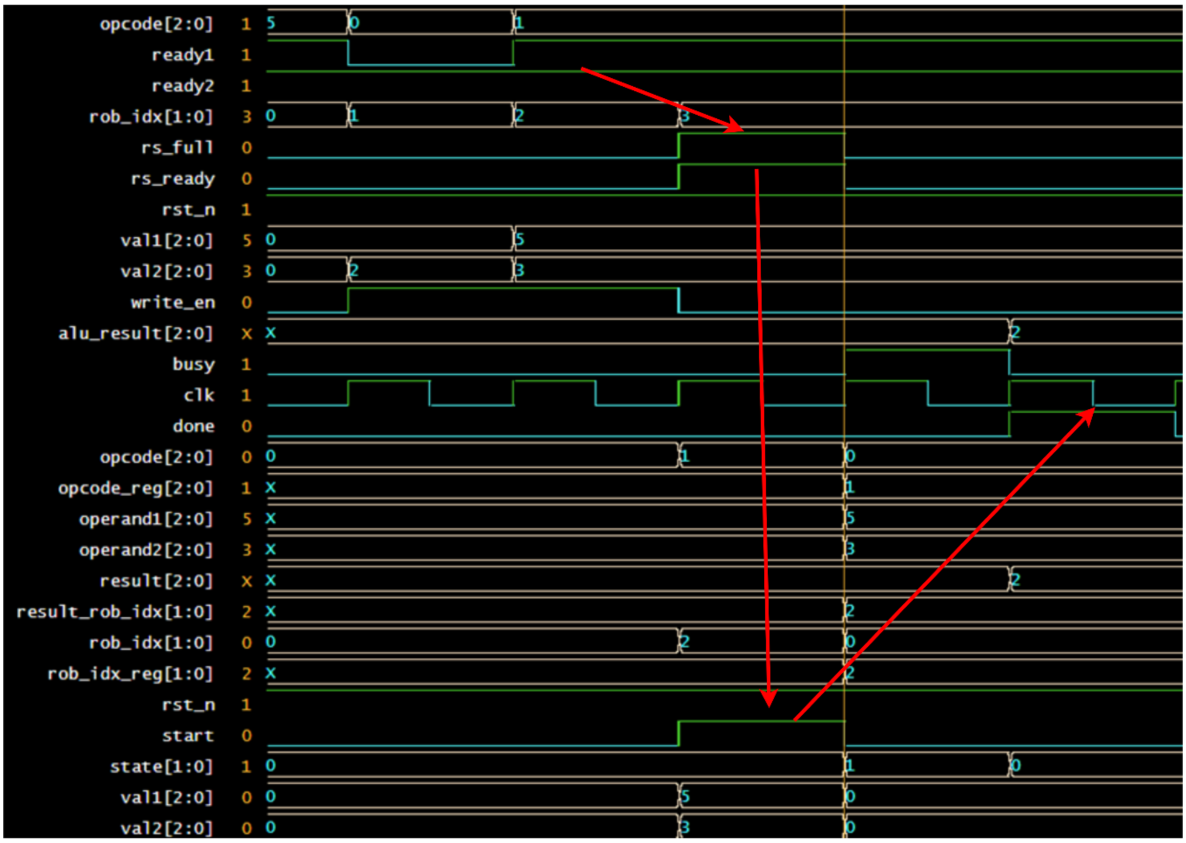
As write_en becomes high, the 12-bit instruction from io_in is fetched into the instruction queue at the tail position, and the tail pointer increments to the next slot.

Issue



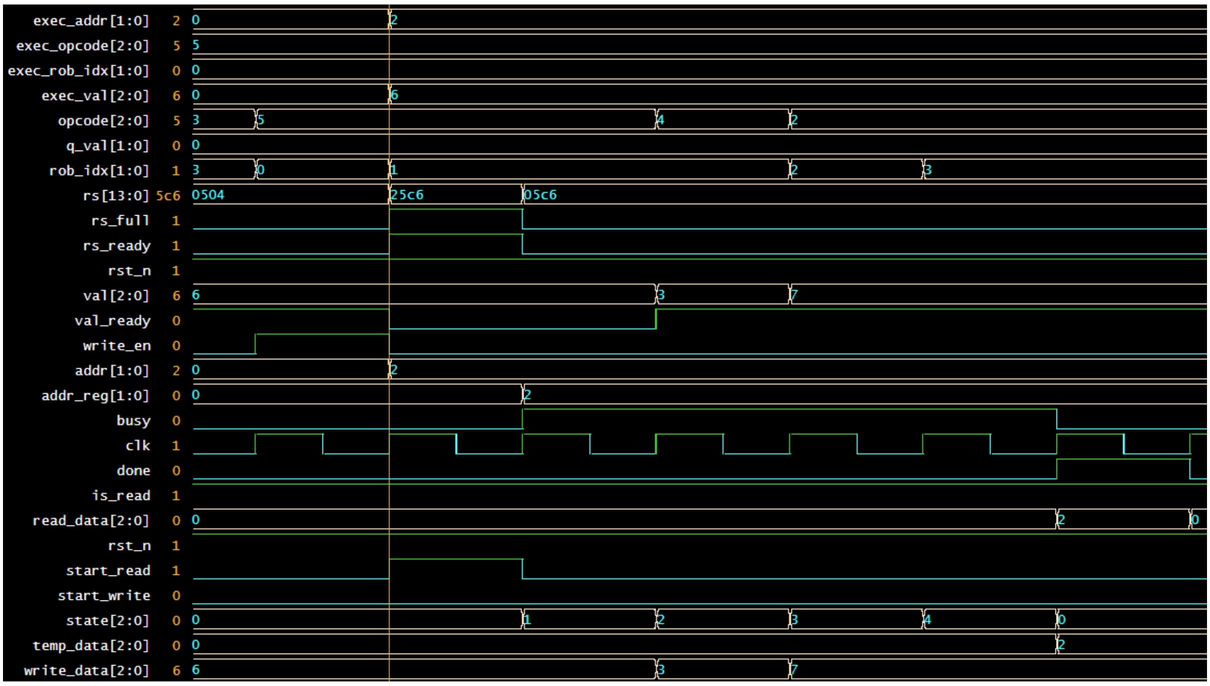
When the ROB and either the ALU or memory reservation station have available space (depending on the opcode), dequeue_en is asserted, and the instruction at the head of the instruction queue is decoded. Then, the corresponding write_en signals for the ROB and either the ALU or memory reservation station are activated based on the instruction type. Rob index get placed on the Alu and memory reservation station.

ALU Execution



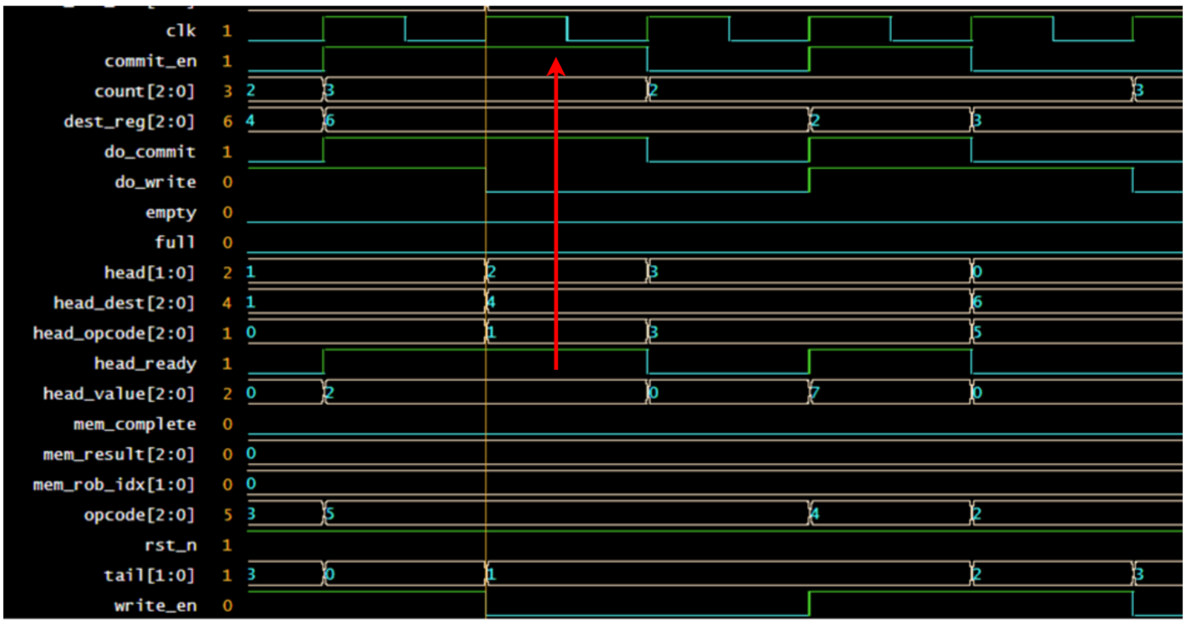
When both ready1 and ready2 are set in the ALU reservation station, rs_ready is asserted, indicating that the instruction is ready to be issued to the ALU. Once the ALU starts execution, it takes two cycles to complete, after which the result is produced (e.g., if operand1 = 5 and operand2 = 3, and the operation is SUB, then the result is 2 and is forwarded with its corresponding result_rob_idx).

Mem Execution



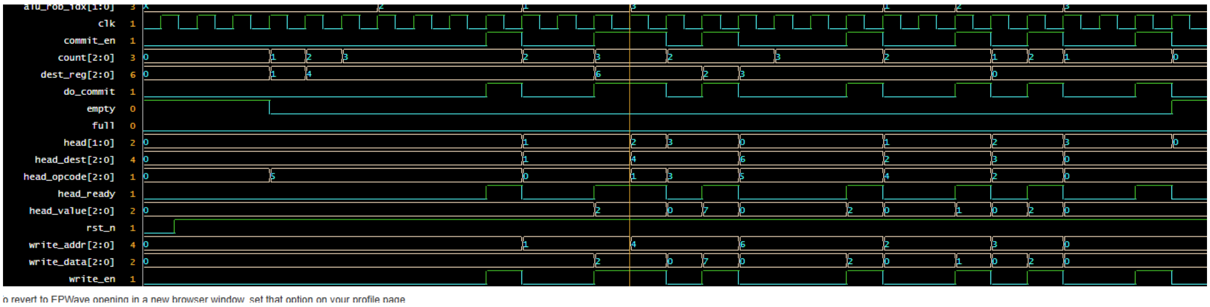
When the memory reservation station becomes ready, rs_ready is asserted and memory access begins. After 5 cycles of execution, the data is read from memory, temp_data is updated, and done is asserted.

Commit



0

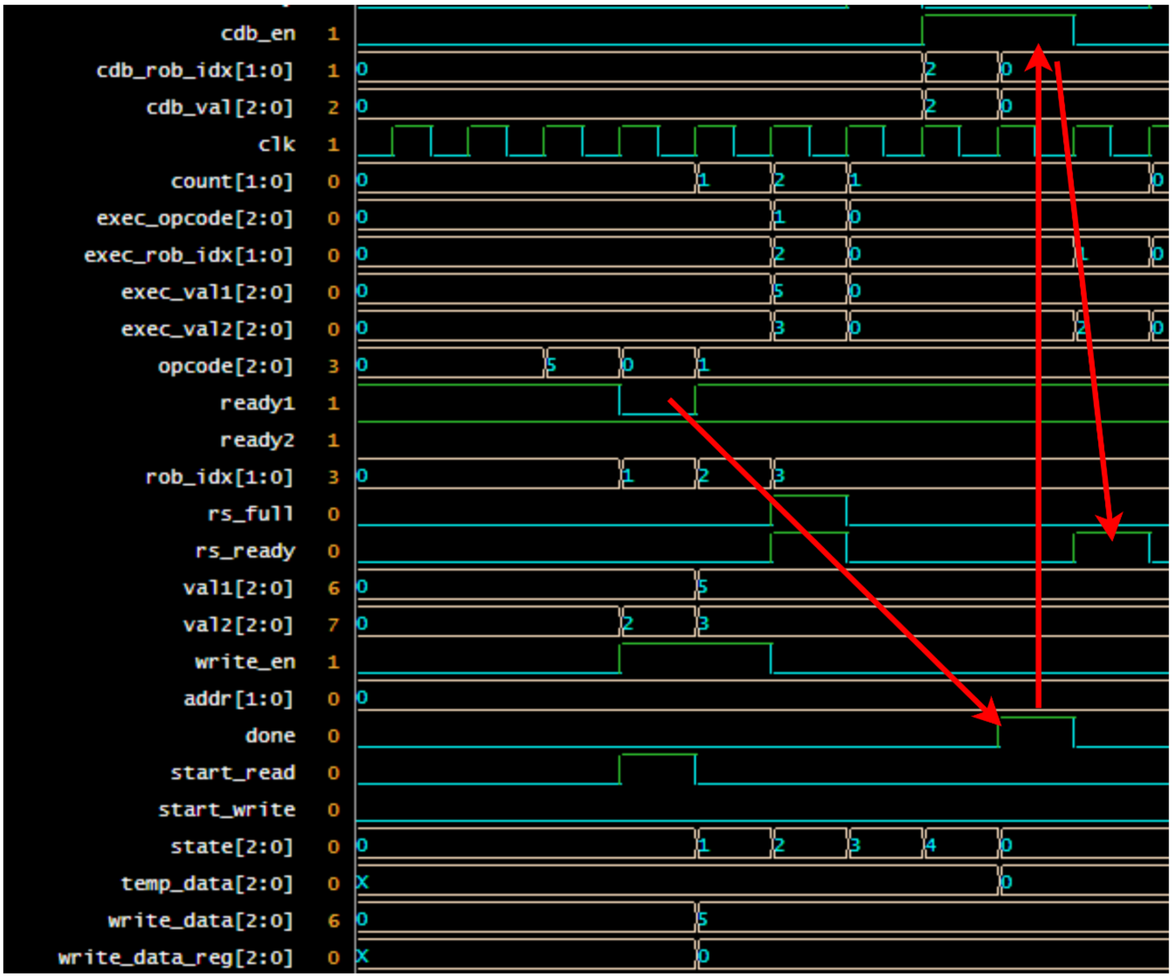
When the value at the head of the reorder buffer becomes ready, commit_en is asserted and the instruction is committed.



After committing, the values are simultaneously written back to the register file.

Data Dependency

RAW



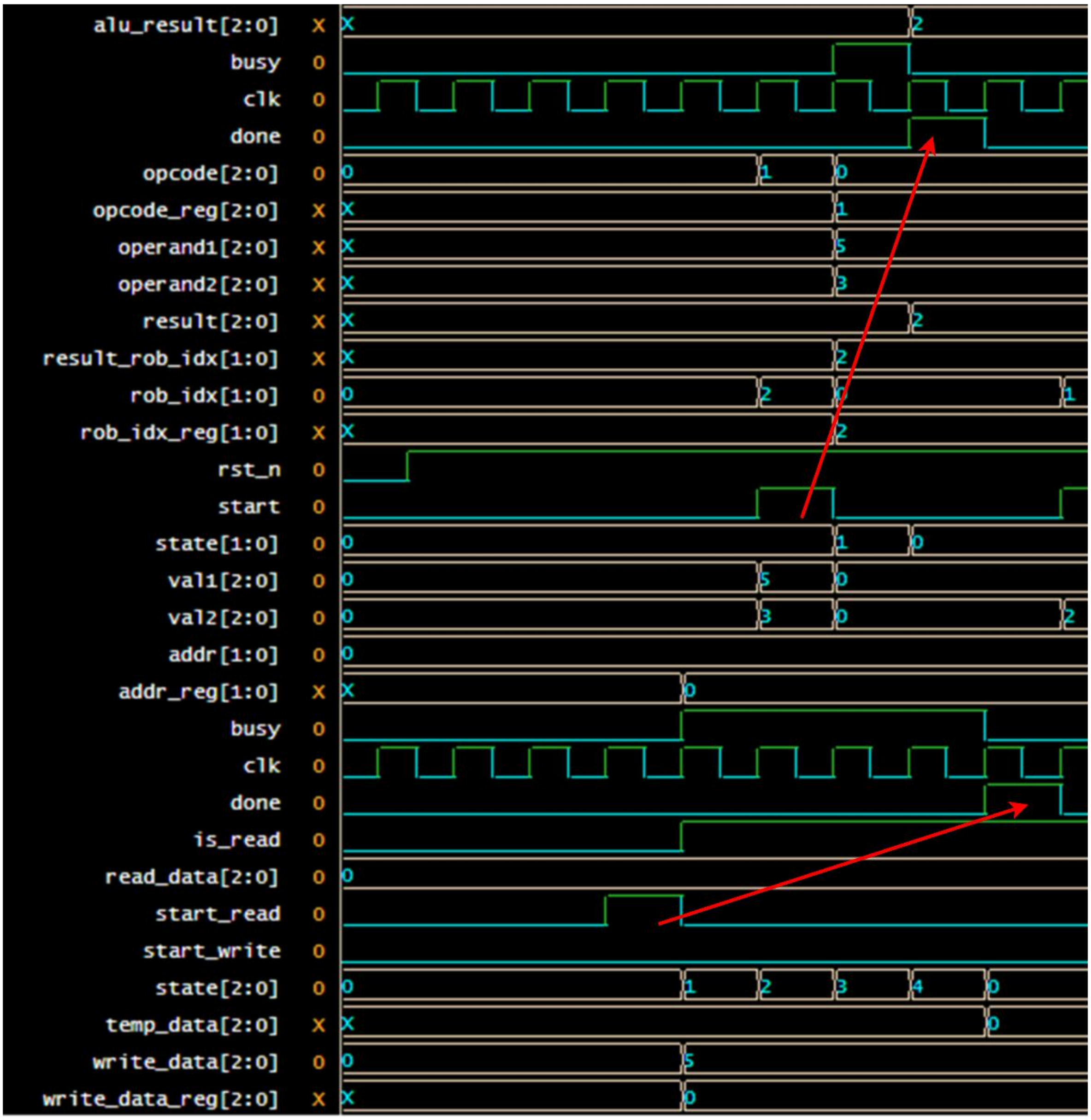
0

As the mem access is done it updates the rob and rs by cdb_en and as the alu get the register value rs_ready turns high

WAW /WAR

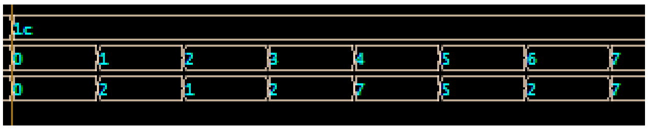
False dependencies such as WAW and WAR are eliminated by using ROB index-based register renaming. Each instruction gets a unique ROB index, so writes and reads refer to different versions of the same logical register.

Out of order execution



This waveform demonstrates that instructions are executed out-of-order because ALU and memory operations take different numbers of clock cycles. While a memory access is in progress, an ALU operation may finish earlier and update the ROB accordingly. We could also implement the ILP.

RESULT



7 instruction took 28 cycles to finish executing and the result is shown as follows.

R1 = 0(R0) + 2(R2)

R3 = R7(111) AND R6(010) = 2

R6 = MEM[2] = 2

R4 = R6(110) or R7(111) = 111 = 7