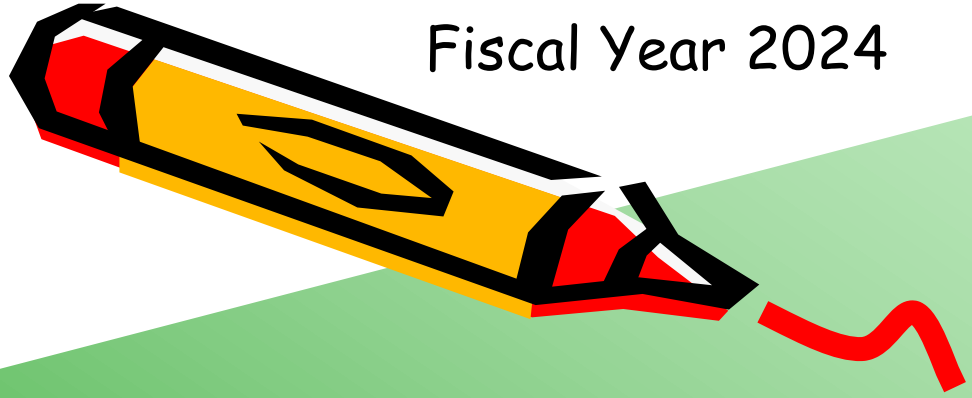


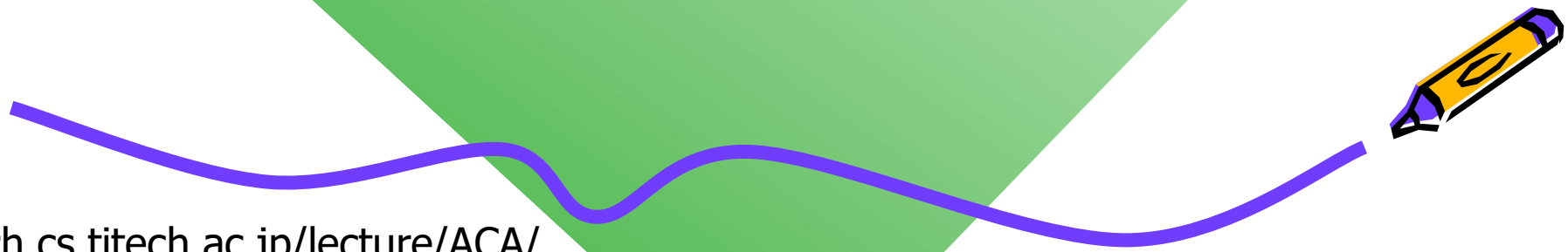
Fiscal Year 2024

Ver. 2024-12-18a

Course number: CSC.T433
School of Computing,
Graduate major in Computer Science

Advanced Computer Architecture

4. Pipelining

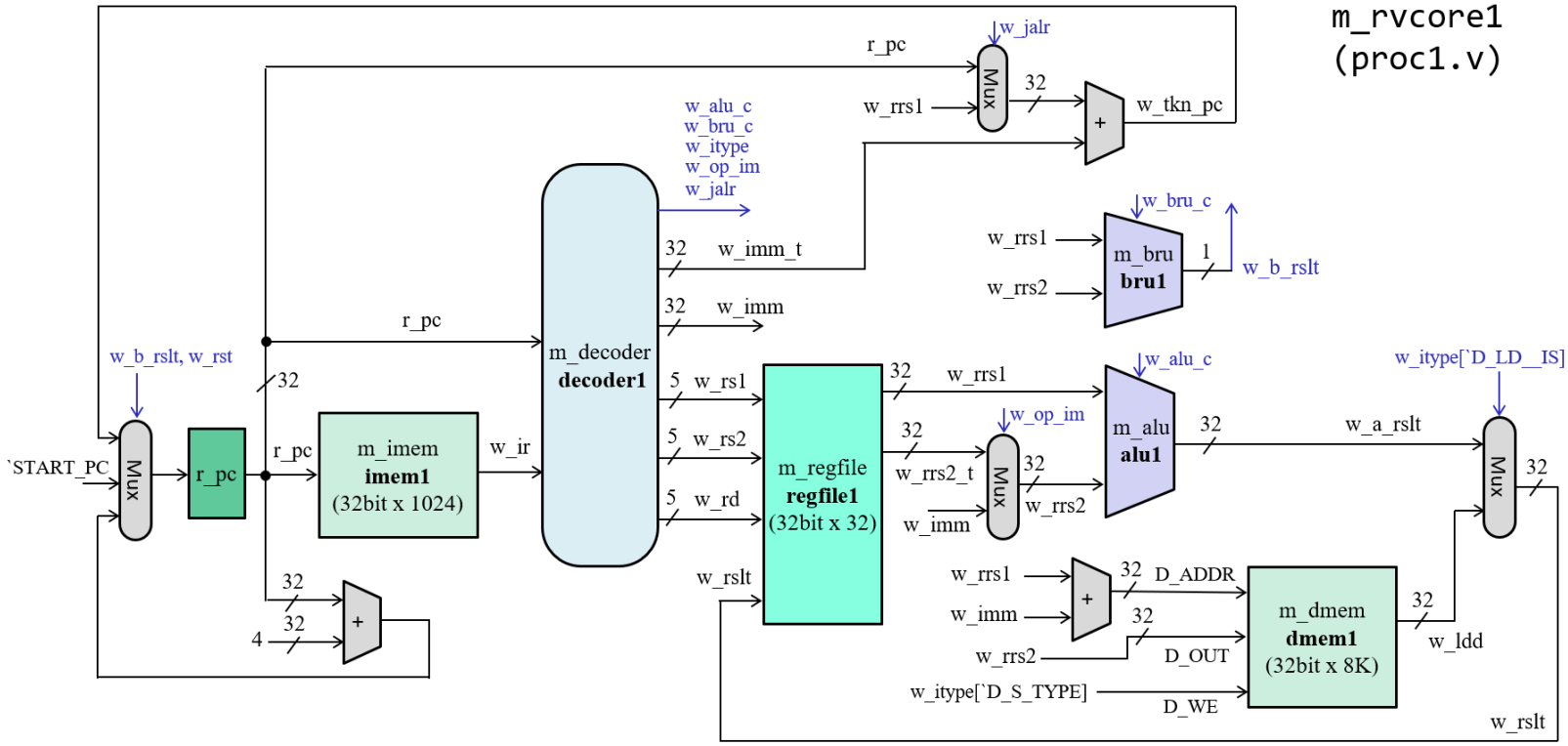


www.arch.cs.titech.ac.jp/lecture/ACA/
Room No. W8E-308, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

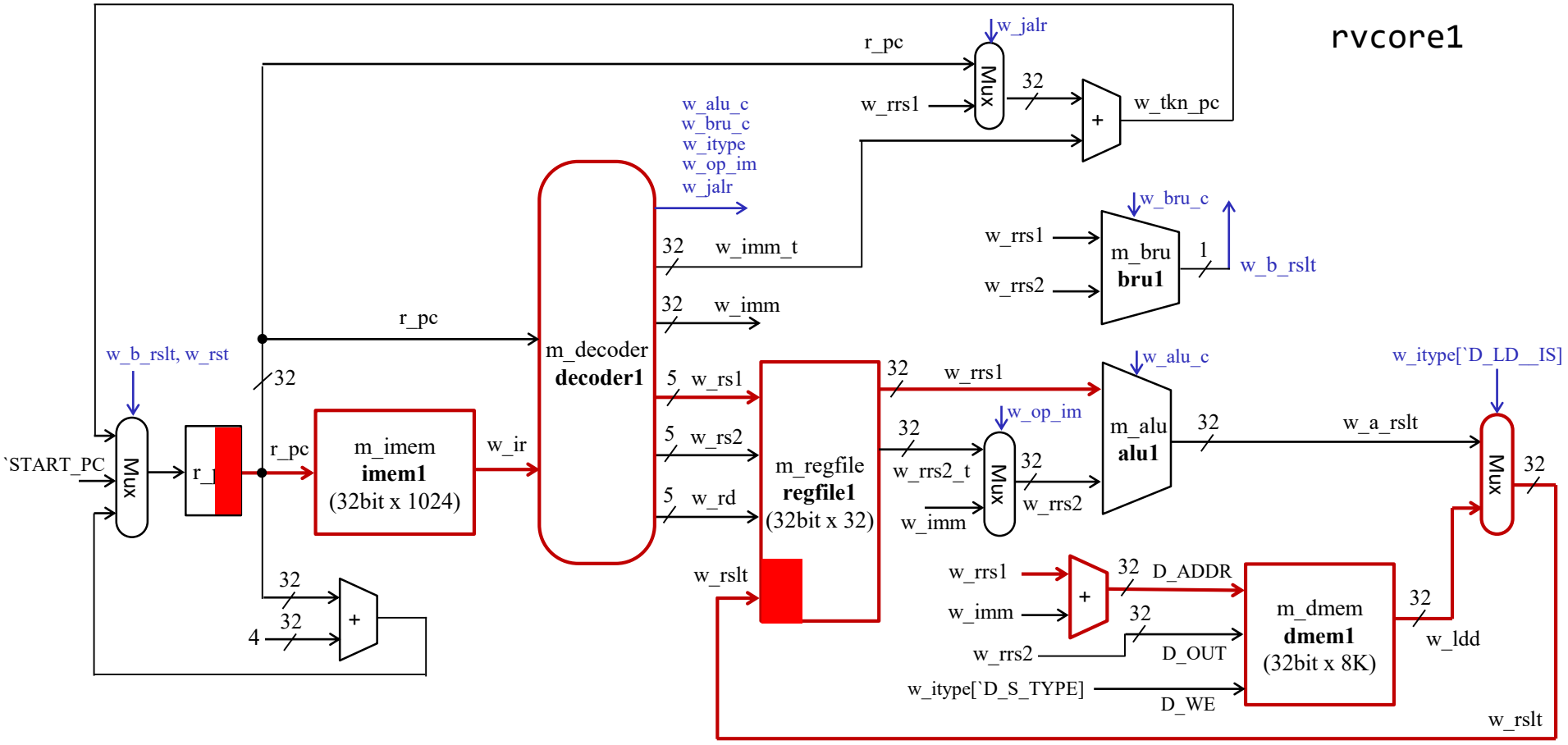
Single-cycle implementation of processors

- Single-cycle implementation also called single clock cycle implementation is the implementation in which an instruction is executed in one clock cycle.
  While easy to understand, it is too slow to be practical.
  It is useful as a baseline for lectures.



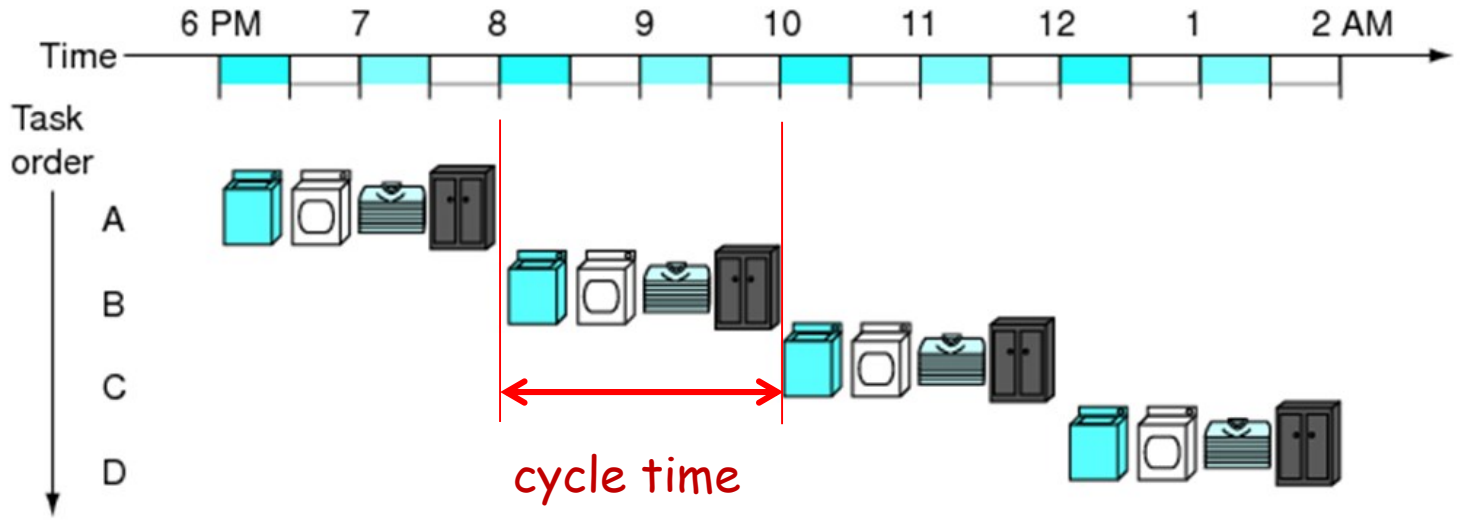
Critical path of rvcore1 (single-cycle version)

- It is too slow to be practical.



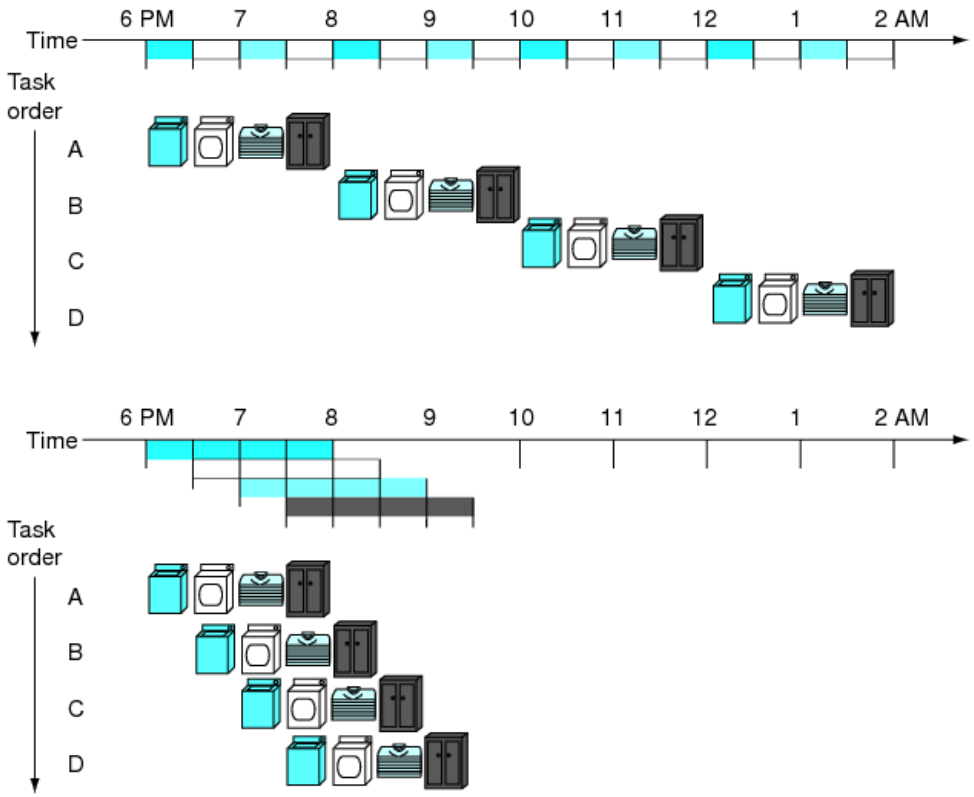
Single-cycle implementation of laundry

- (A) Ann, (B) Brian, (C) Cathy, and (D) Don each have dirty clothes to be *washed*, *dried*, *folded*, and *put away*, each taking 30 minutes.
- The cycle time (the time from the end of one load to the end of the next one) is 2 hours.
- For four loads, the sequential laundry takes 8 hours.

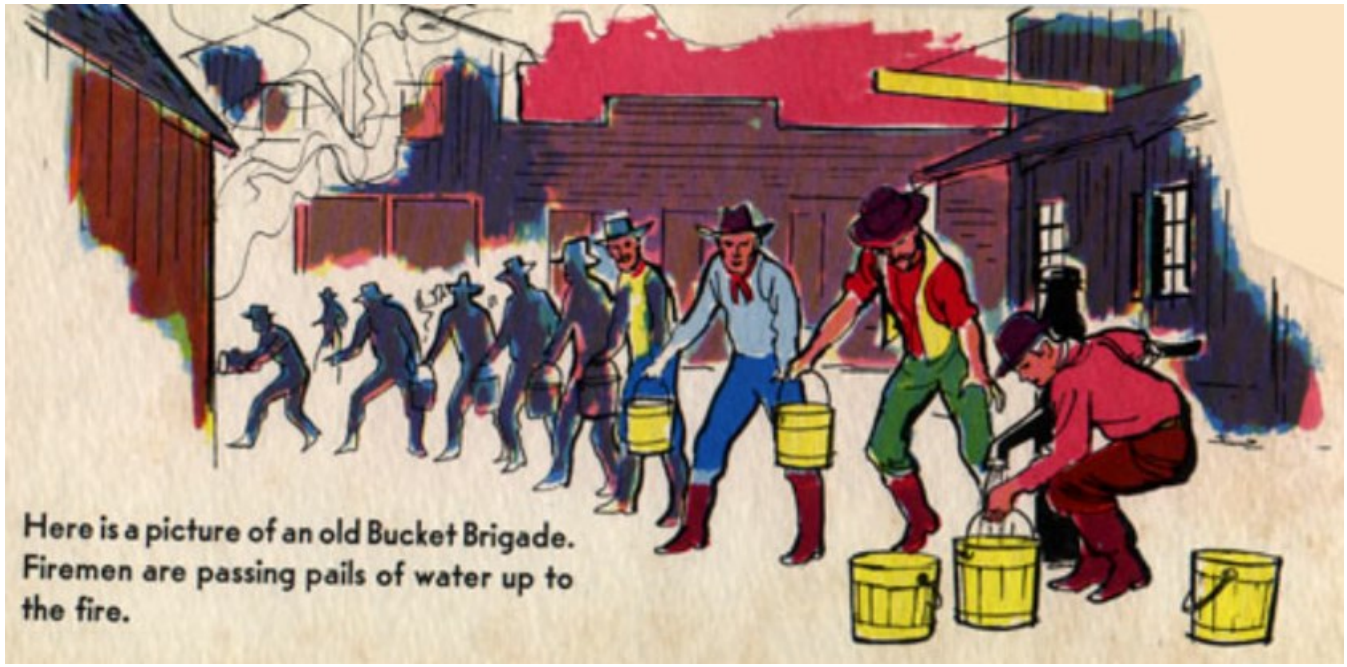


Single-cycle implementation and pipelining

- When the washing of load A is finished at 6:30 p.m., another washing of load B starts.
- Pipelined laundry takes 3.5 hours just using the same hardware resources. The cycle time is 30 minutes.
- What is the latency (execution time) of each load?

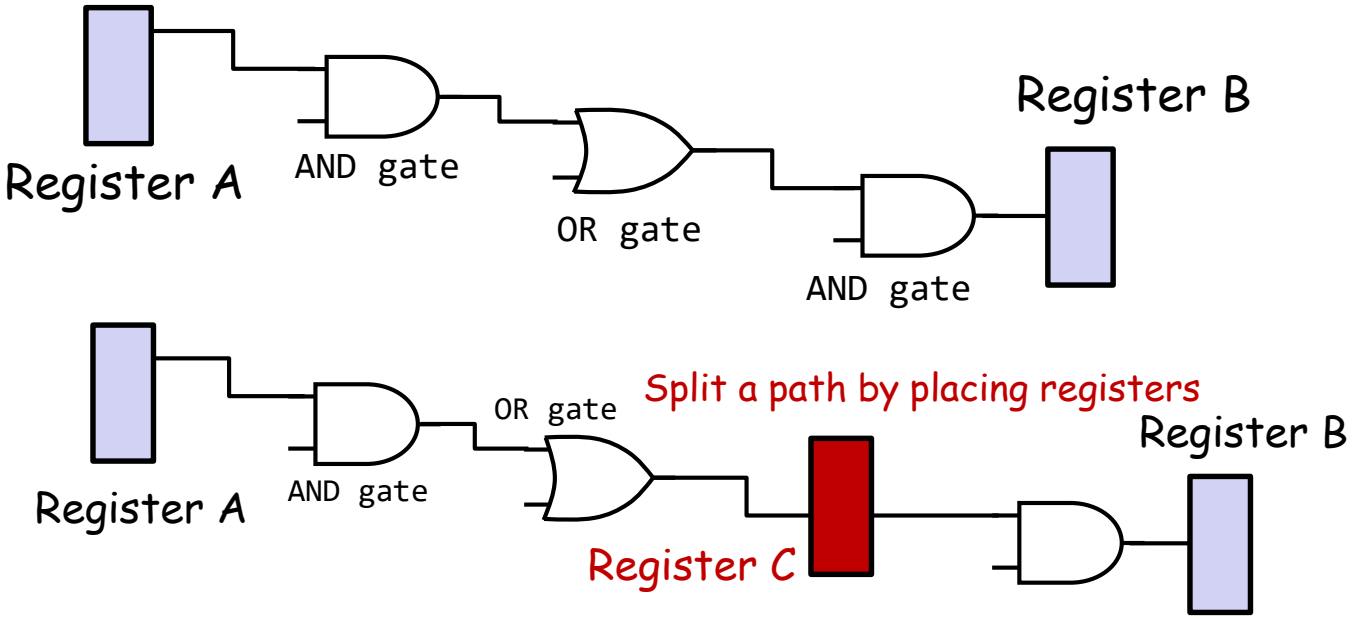


Bucket brigade



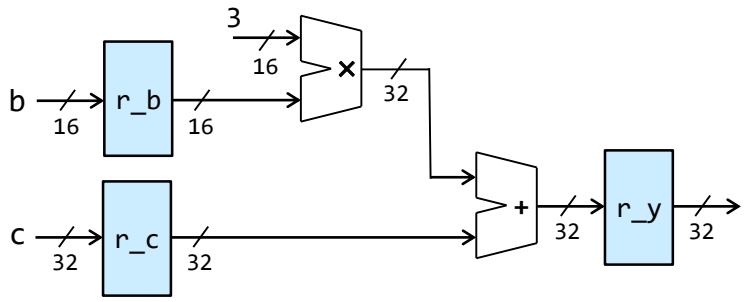
Clock rate is mainly determined by

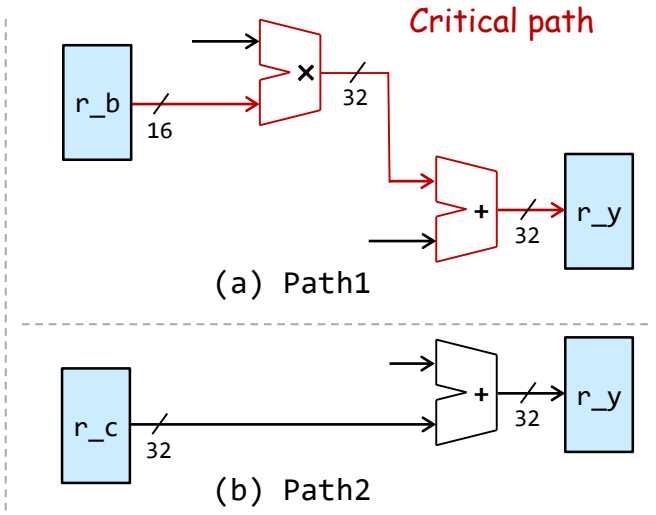
- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout



Pipelining example: multiply-add operation (1)

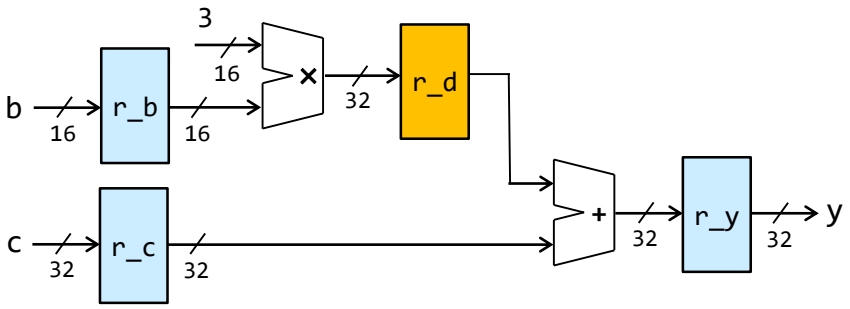
- As an example of pipelining, we will see a multiply-add circuit.
- r_b, r_c are input registers and r_y is output register of the circuit.
- This has two paths named path1 and path2, and path1 is the critical path to determine the maximum operating frequency.

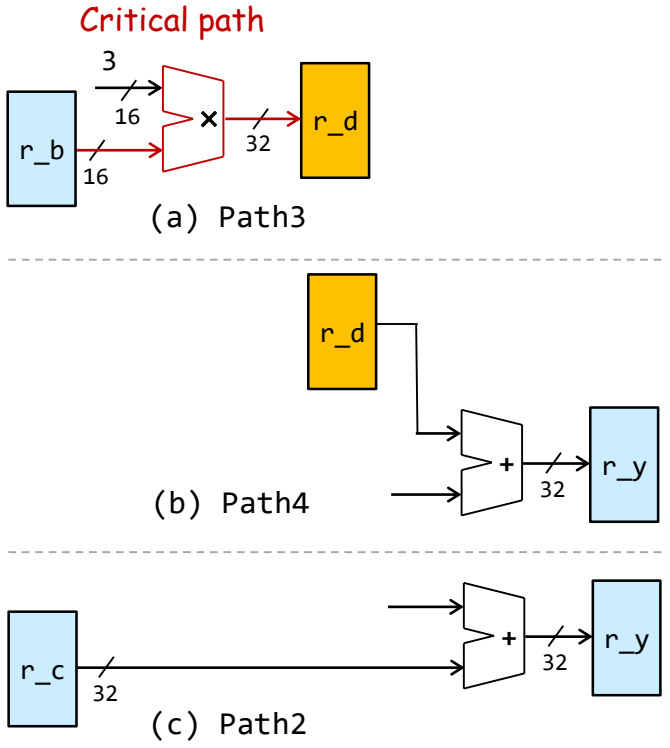




Pipelining example: multiply-add operation (2)

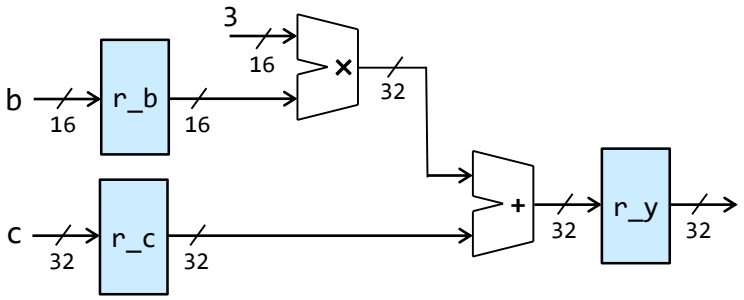
- By inserting register r_d, the critical path can be divided into Path3 and Path4.
- As a result, the new critical path becomes Path3.
- This has the disadvantage that input b and c in the same clock cycle cannot be processed.



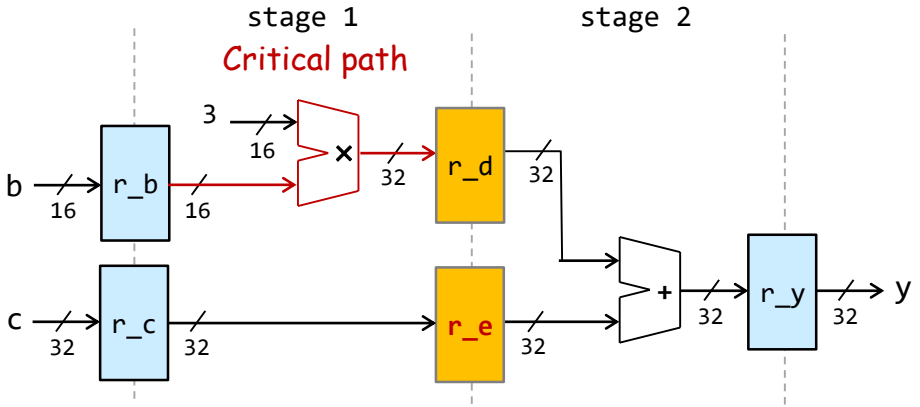


Pipelining example: multiply-add operation (3)

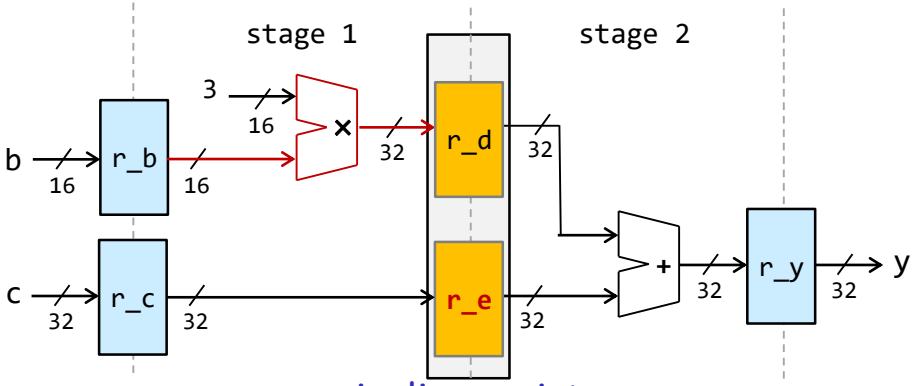
- To overcome this drawback, we insert register r_e.
- This realizes a pipeline with stages 1 and 2.
  A set of registers between two adjacent stages are called a pipeline register.



(a) original multiply-add circuit

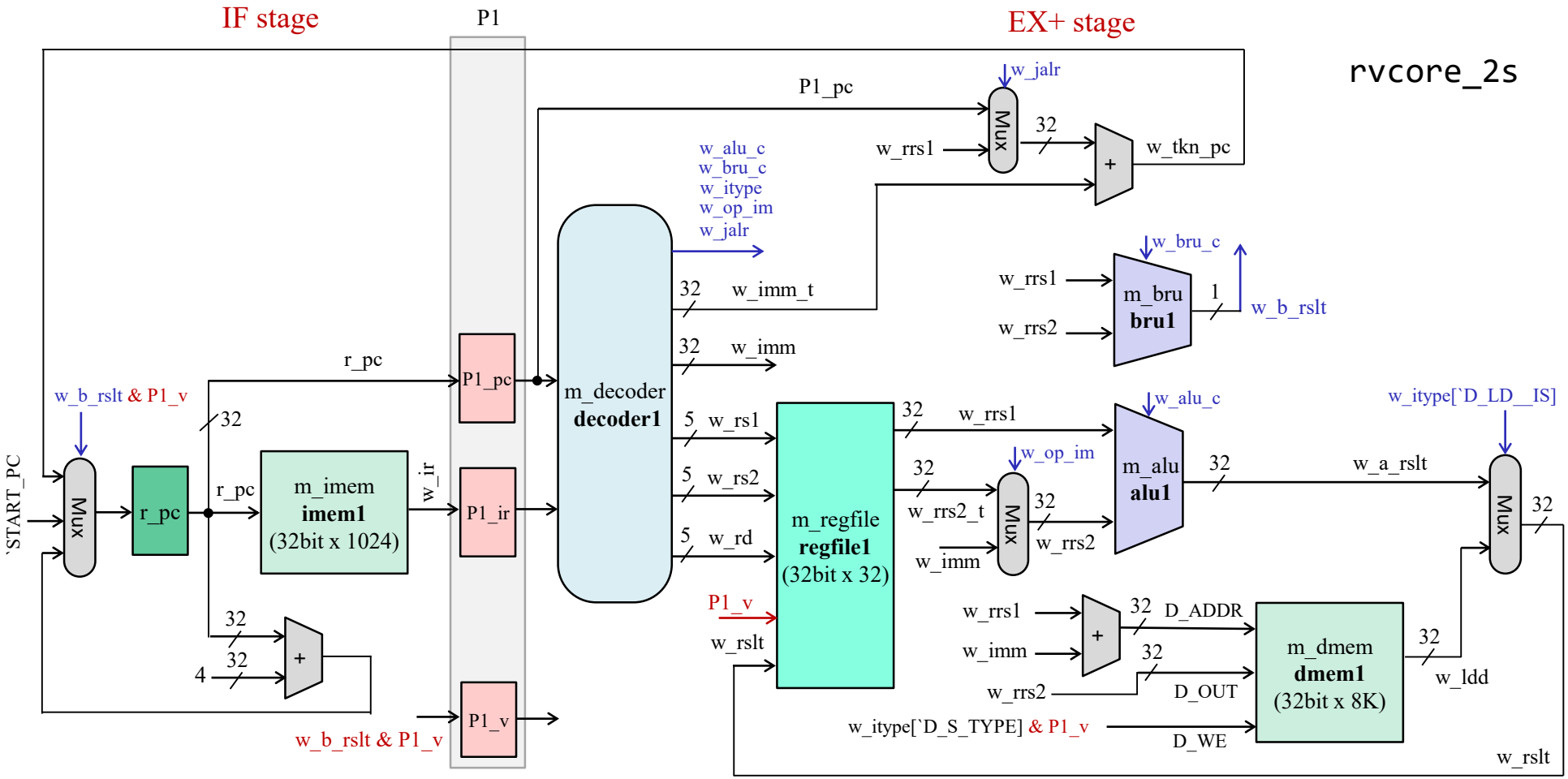


(b) two-stage pipelined circuit

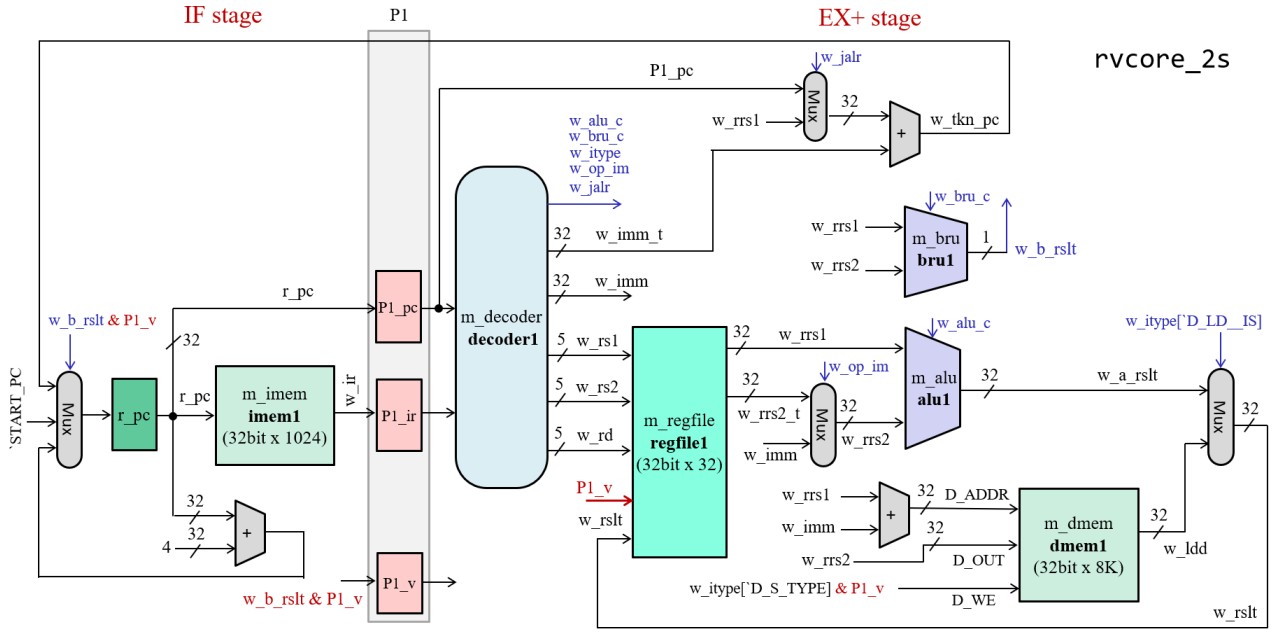


rvcore_2s : 2-stage pipelining processor

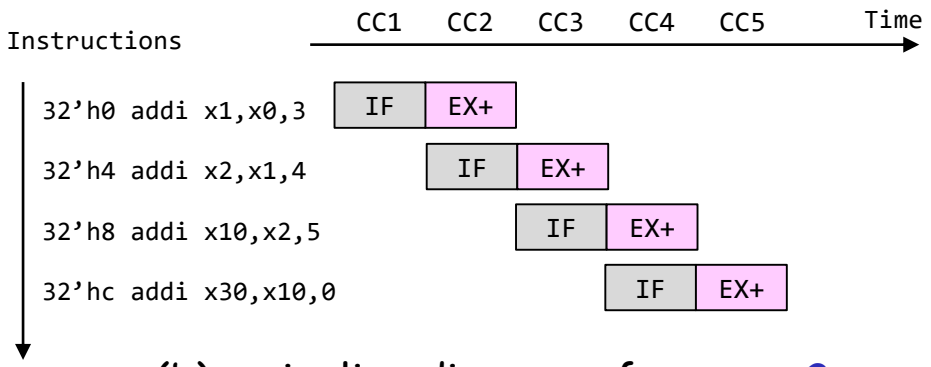
- The strategy is to separate the instruction fetch (IF) step and other (ID, EX, MA, WB) steps. The first stage is named IF. The other stage is named EX+.



rvcore_2s : 2-stage pipelining processor



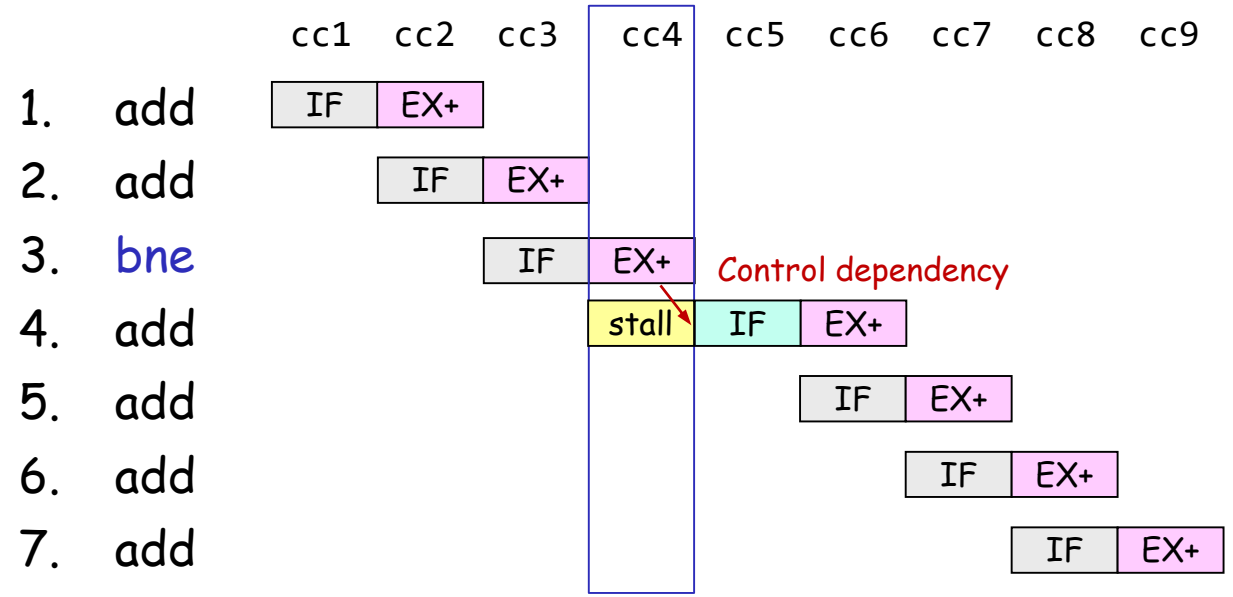
(a) rvcore_2s: 2-stage pipelining processor



(b) a pipeline diagram of rvcore_2s

Why do branch instructions degrade IPC?

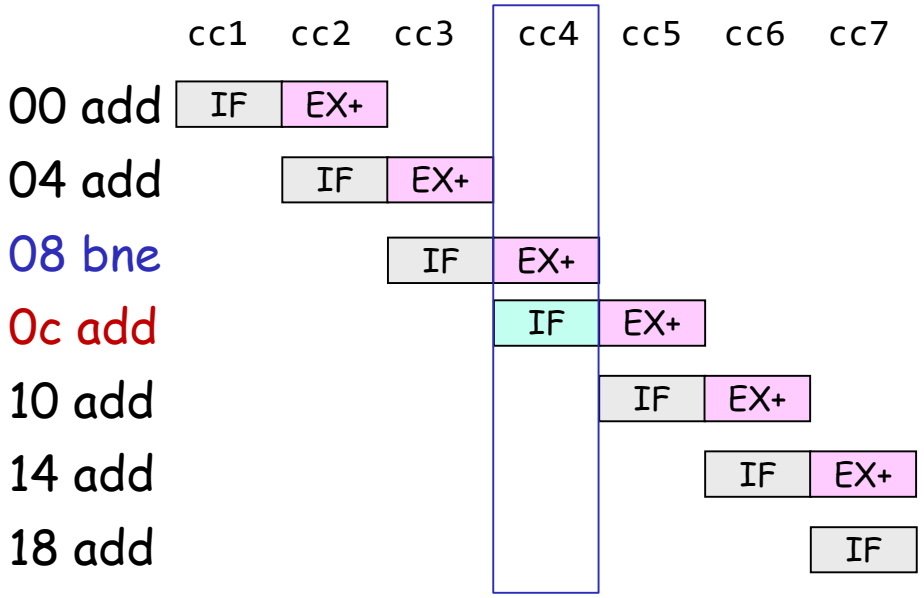
- The branch taken / untaken (branch result) is determined in the EX+ stage of the branch.
- The conservative approach is stalling instruction fetch until the branch direction is determined.



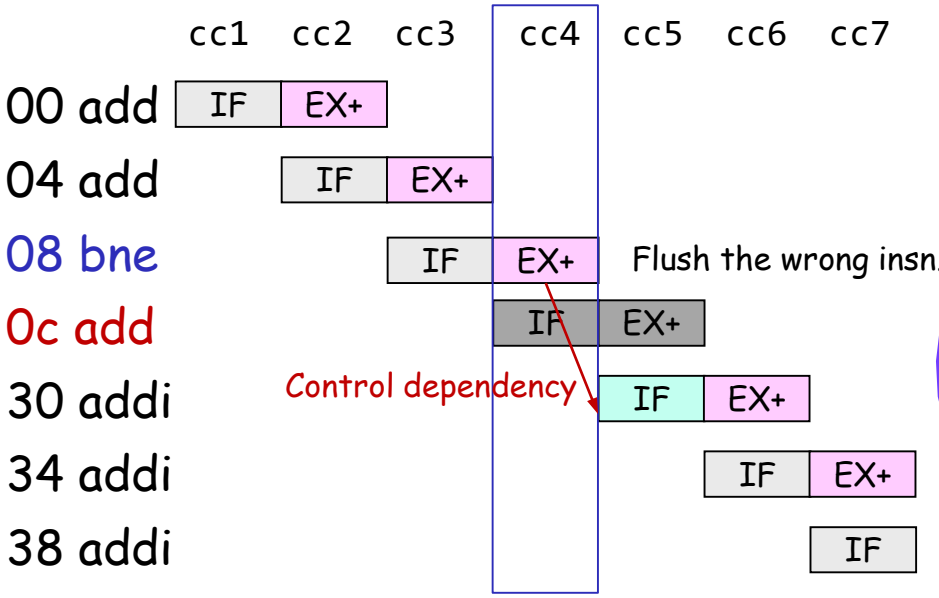
two-stage pipelining processor executing instruction sequence with a branch

Why do branch instructions degrade IPC?

- Another approach is fetching the following instruction (an instruction at the next address) when a branch (bne) is fetched.
- When a branch (08 bne) is taken, the wrong instruction fetched (0c add) is flushed.

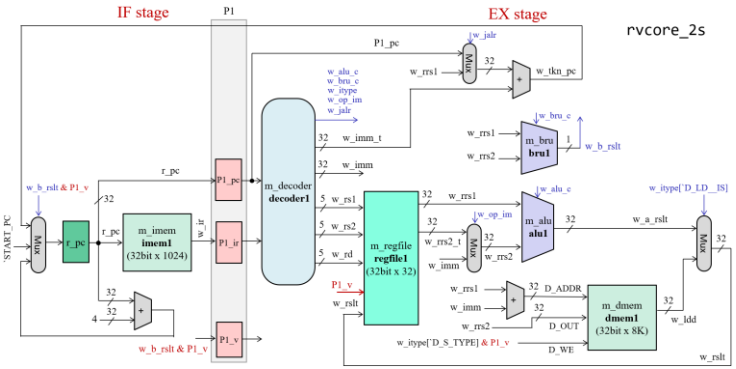


(a) branch untaken case



(b) branch taken case

Verilog HDL code for rvcore_2s



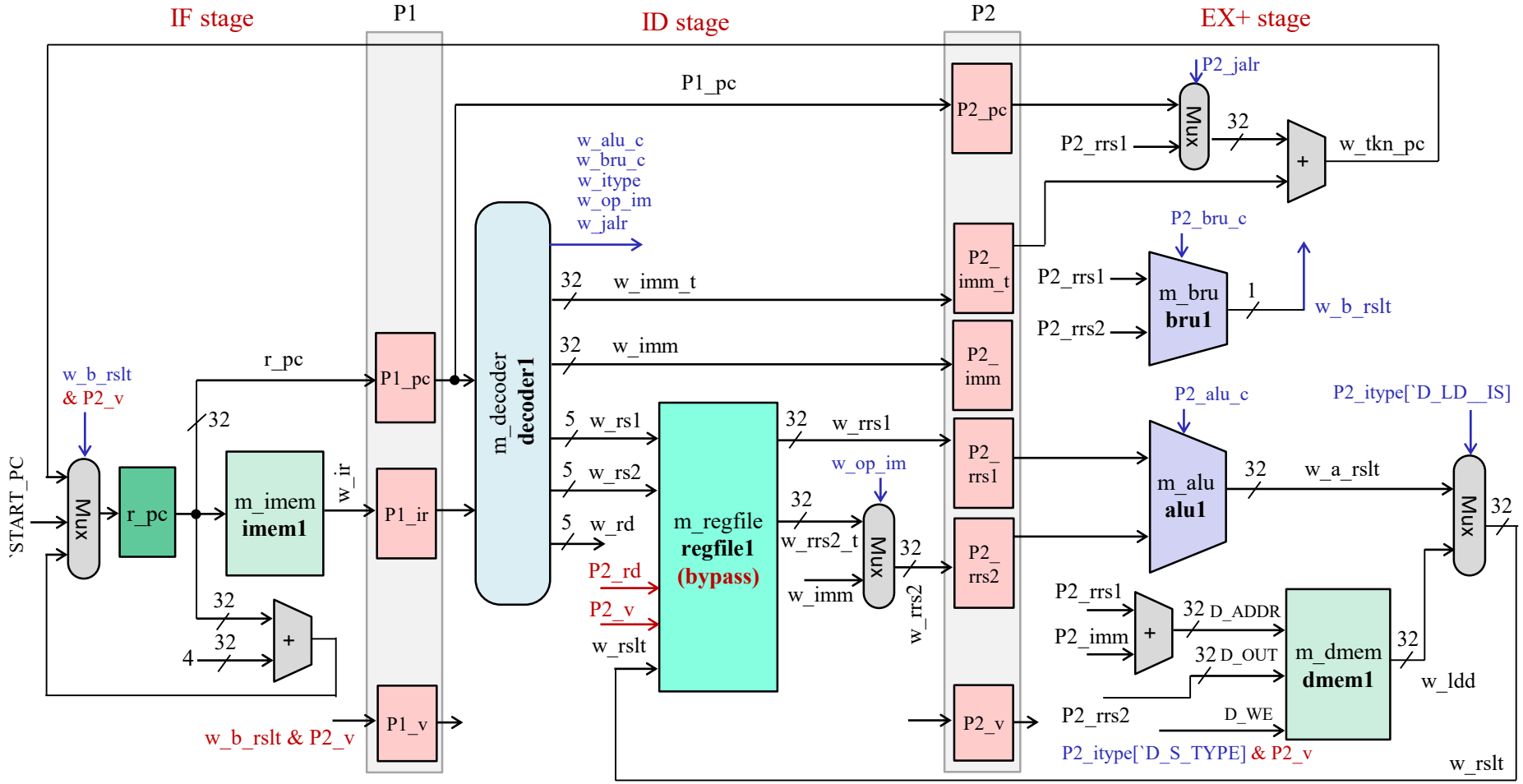

```
module m_rvcore_2s ( ///// RVCore two-stage pipelining version
    input  wire        w_clk,  // clock signal
    input  wire        w_rst,  // reset signal
    output wire [31:0] D_ADDR, // data memory, address
    output wire [31:0] D_OUT,  // data memory, output data
    output wire        D_WE    // data memory, write enable
);
    reg  [31:0] r_pc;
    reg  [31:0] P1_ir, P1_pc; /* pipeline register */
    reg         P1_v;         /* pipeline register */
    wire        w_jalr, w_op_im, w_b_rslt;
    wire [9:0]  w_itype;
    wire [10:0] w_alu_c;
    wire [6:0]  w_bru_c;
    wire [4:0]  w_rs1, w_rs2, w_rd;
    wire [31:0] w_ir, w_rrs1, w_rrs2_t, w_rrs2, w_imm_t, w_imm;
    wire [31:0] w_a_rslt, w_rslt, w_tkn_pc, w_ldd;
    wire w_miss = w_b_rslt & P1_v;

    m_imem imem1 (w_clk, r_pc, w_ir);
    m_decoder decoder1 (P1_pc, P1_ir, w_rd, w_rs1, w_rs2,
                        w_op_im, w_itype, w_jalr, w_alu_c, w_bru_c, w_imm_t, w_imm);
    m_regfile regfile1 (w_clk, w_rs1, w_rs2, w_rrs1, w_rrs2_t, w_rd, P1_v, w_rslt);
    assign w_rrs2 = (w_op_im) ? w_imm : w_rrs2_t;
    m_alu alu1 (w_rrs1, w_rrs2, w_alu_c, w_a_rslt);
    m_bru bru0 (w_rrs1, w_rrs2, w_bru_c, w_b_rslt);
    assign D_ADDR = w_rrs1 + w_imm;
    assign D_OUT  = w_rrs2;
    assign D_WE   = w_itype[`D_S_TYPE] & P1_v;
    m_dmem dmem1 (w_clk, D_WE, D_ADDR, D_OUT, w_ldd);

    assign w_rslt = (w_itype[`D_LD__IS]) ? w_ldd : w_a_rslt;
    assign w_tkn_pc = ((w_jalr) ? w_rrs1 : P1_pc) + w_imm_t;
    always @(posedge w_clk) begin
        r_pc  <= (w_rst) ? `START_PC :
                 (w_b_rslt & P1_v) ? w_tkn_pc : r_pc + 4;
        P1_ir <= w_ir;
        P1_pc <= r_pc;
        P1_v  <= (w_rst) ? 0 : !w_miss;
end
```

rvcore_3s : 3-stage pipelining processor

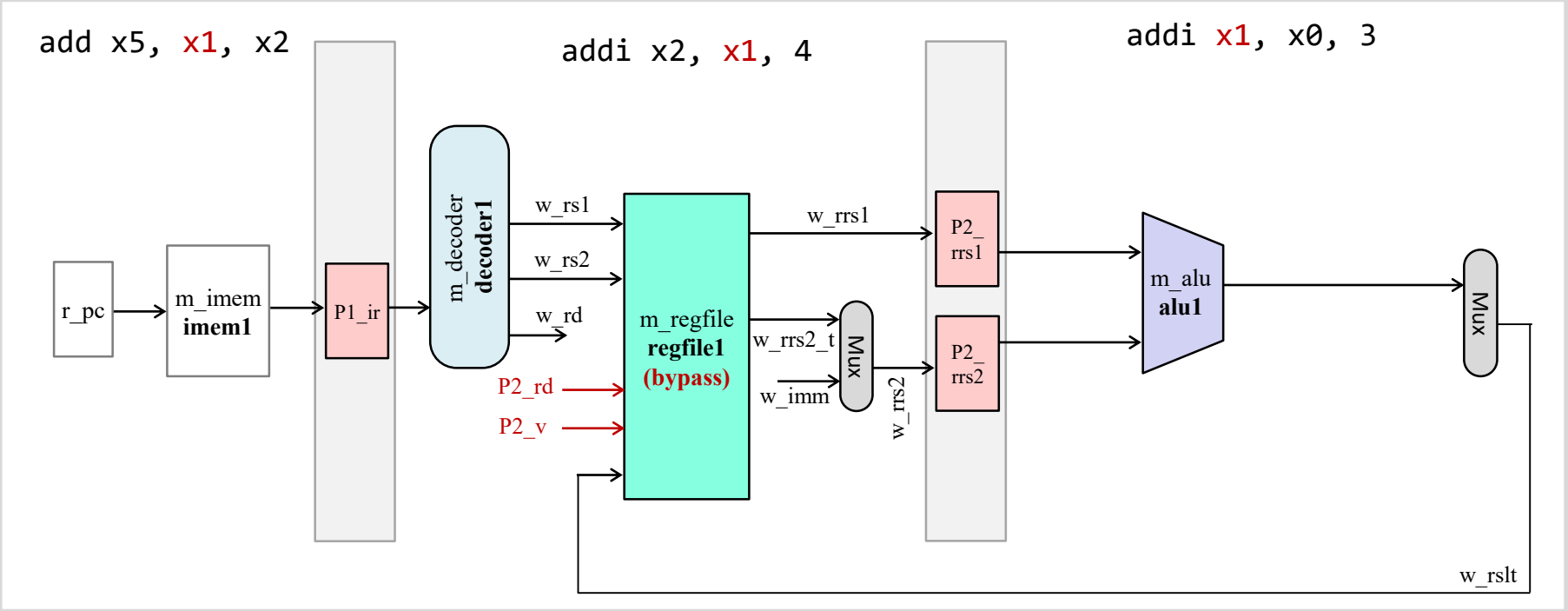
- The strategy is to separate the instruction fetch (IF) step, instruction decode (ID) step, and other (EX, MA, WB) steps. The first stage is named IF. The second stage is named ID. The last stage is named EX+.



Exercise 1

- Draw the main datapath of the processor rvcore_3s and write the valid values on wires when the processor is executing these three instructions

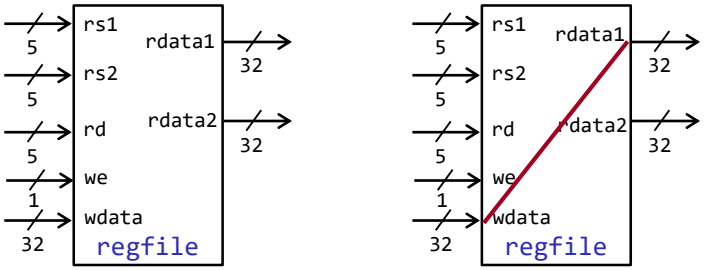
```
0x00   addi x1, x0, 3      # x1 = 3
0x04   addi x2, x1, 4      # x2 = 3 + 4 = 7
0x08   add  x5, x1, x2     # x5 = 3 + 7 = 10
```



m_regfile with bypassing

```
module m_regfile ( ///// register file
    input  wire        w_clk,
    input  wire [ 4:0] rs1, rs2,
    output wire [31:0] rdata1, rdata2,
    input  wire [ 4:0] rd,
    input  wire        we,
    input  wire [31:0] wdata
);
    reg [31:0] mem [0:31];
    integer i; initial begin for(i=0; i<32; i=i+1) mem[i]=0; end
    assign rdata1 = (rs1==0) ? 0 : mem[rs1];
    assign rdata2 = (rs2==0) ? 0 : mem[rs2];
    always @(posedge w_clk) if(rd!=0 && we) mem[rd] <= wdata;
endmodule
```

m_regfile for rvcore1 and rvcore_2s

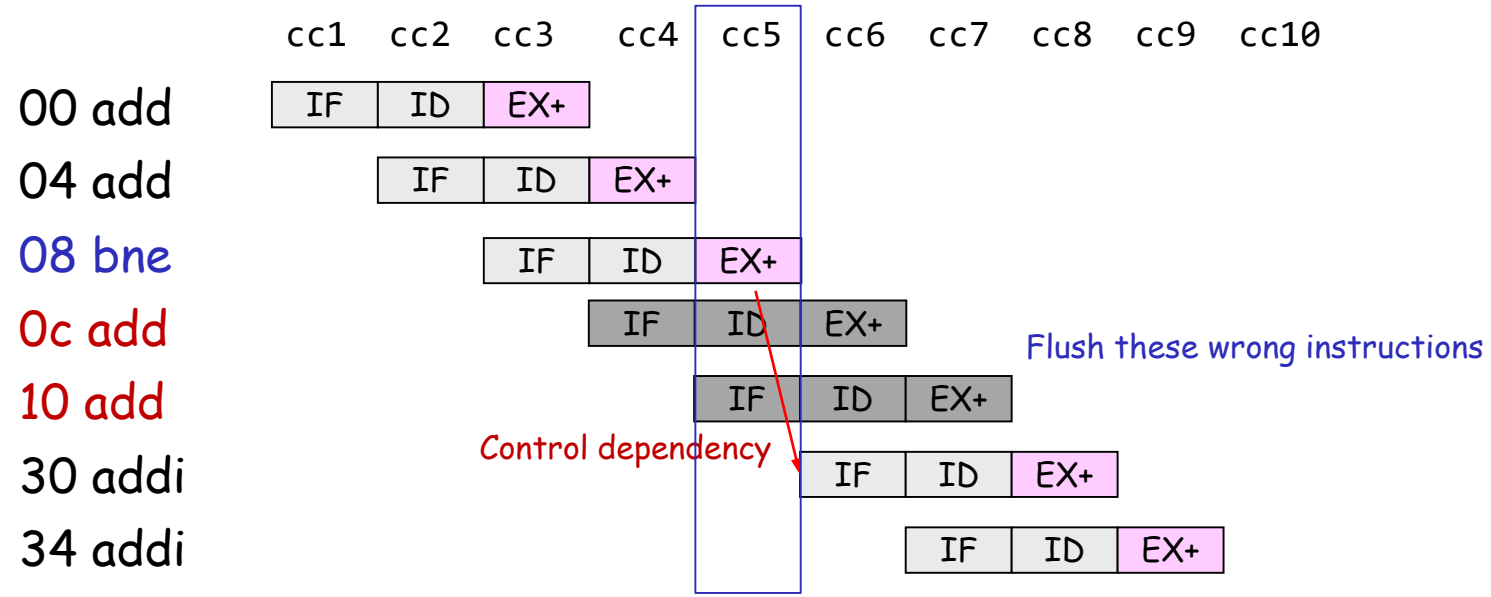


m_regfile for rvcore_3s

```
module m_regfile ( ///// register file
    input  wire        w_clk,
    input  wire [ 4:0] rs1, rs2,
    output wire [31:0] rdata1, rdata2,
    input  wire [ 4:0] rd,
    input  wire        we,
    input  wire [31:0] wdata
);
    reg [31:0] mem [0:31];
    integer i; initial begin for(i=0; i<32; i=i+1) mem[i]=0; end
    wire bp1 = (we & rs1==rd);  // bypassing for rdata1
    wire bp2 = (we & rs2==rd);  // bypassing for rdata2
    assign rdata1 = (rs1==0) ? 0 : (bp1) ? wdata : mem[rs1];
    assign rdata2 = (rs2==0) ? 0 : (bp2) ? wdata : mem[rs2];
    always @(posedge w_clk) if(rd!=0 && we) mem[rd] <= wdata;
endmodule
```

Why do branch instructions degrade IPC?

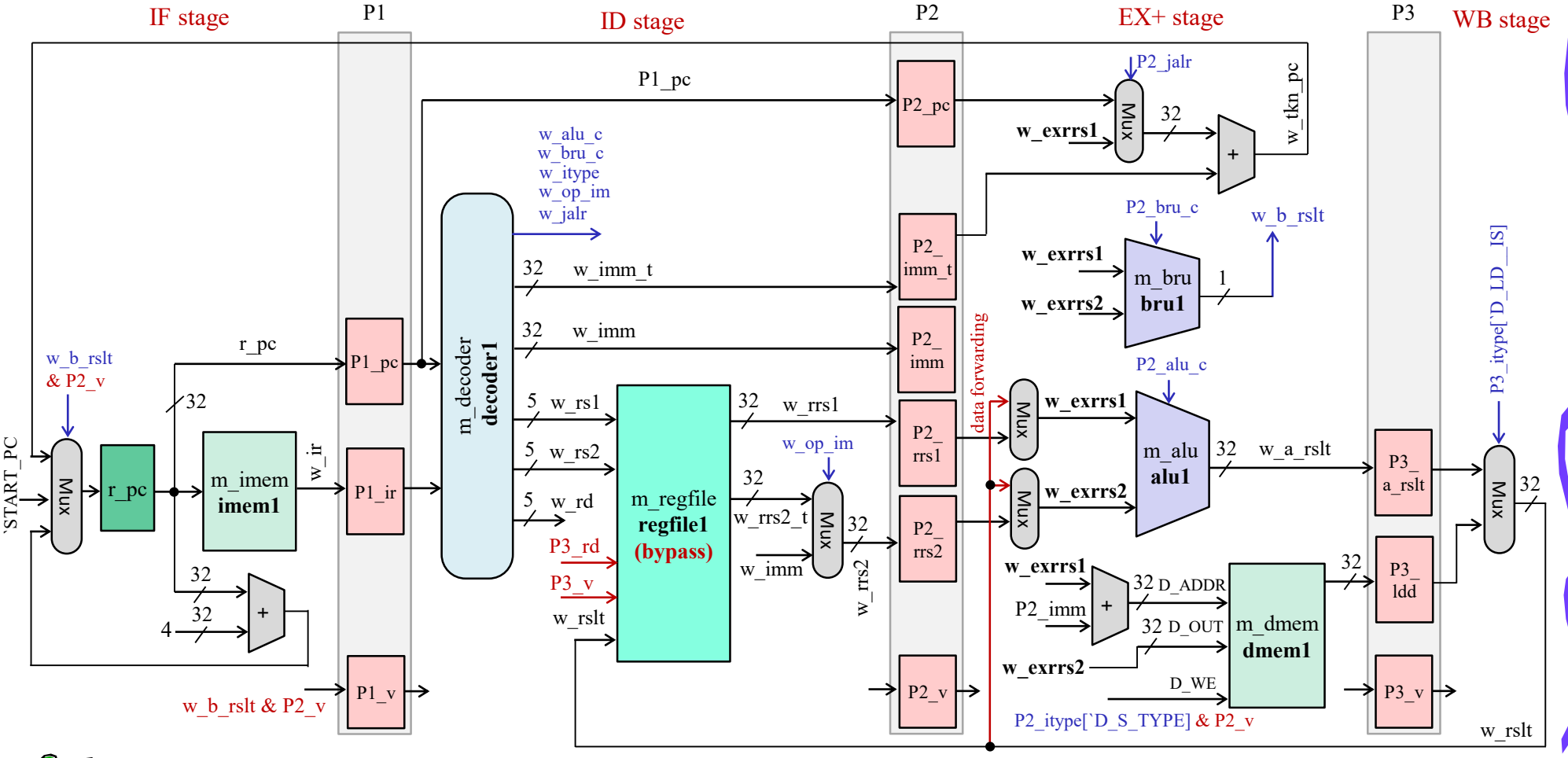
- Another approach is fetching the following instructions (0c add, 10 add) after a branch (bne) is fetched.
- When a branch (08 bne) is taken, the wrong instructions fetched (0c add, 10 add) are flushed.



three-stage pipelining processor executing instruction sequence with a taken branch

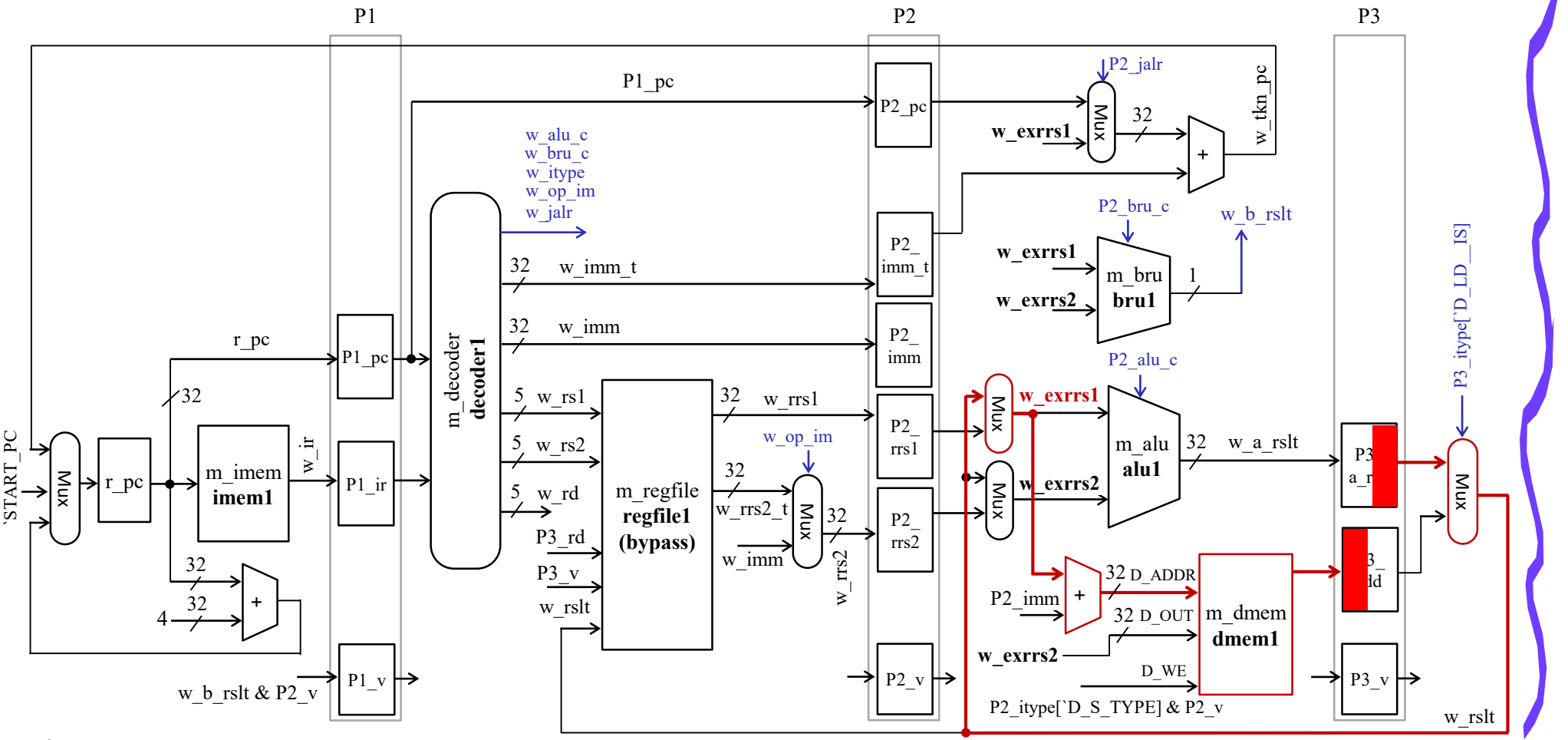
rvcore_4s : 4-stage pipelining processor

- The strategy is to separate the instruction fetch (IF) step, instruction decode (ID) step, and write back(WB) step, and other (EX, MA) steps. The first stage is named IF. The second stage is named ID. The third stage is named EX+. The last stage is named WB.



rvcore_4s : 4-stage pipelining processor

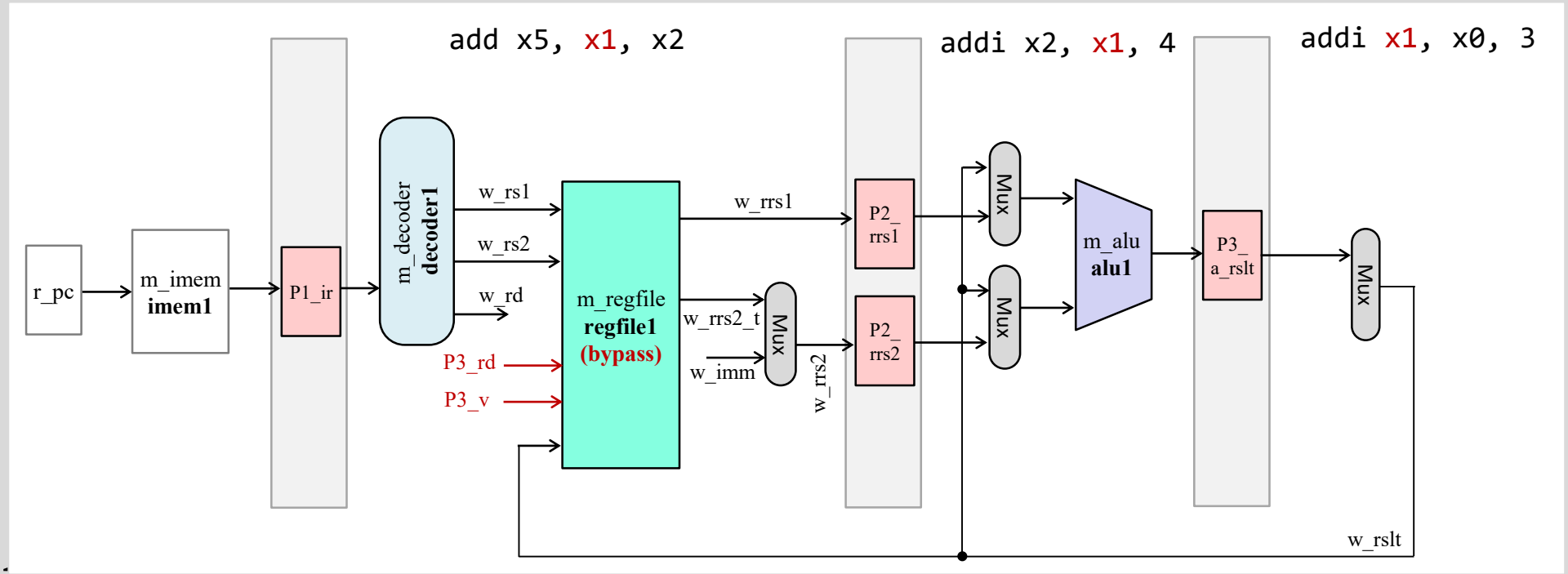
- Critical path
  - deley(register read) + delay(mux) + delay(mux) + delay(adder) + delay(dmem read)



Exercise 2

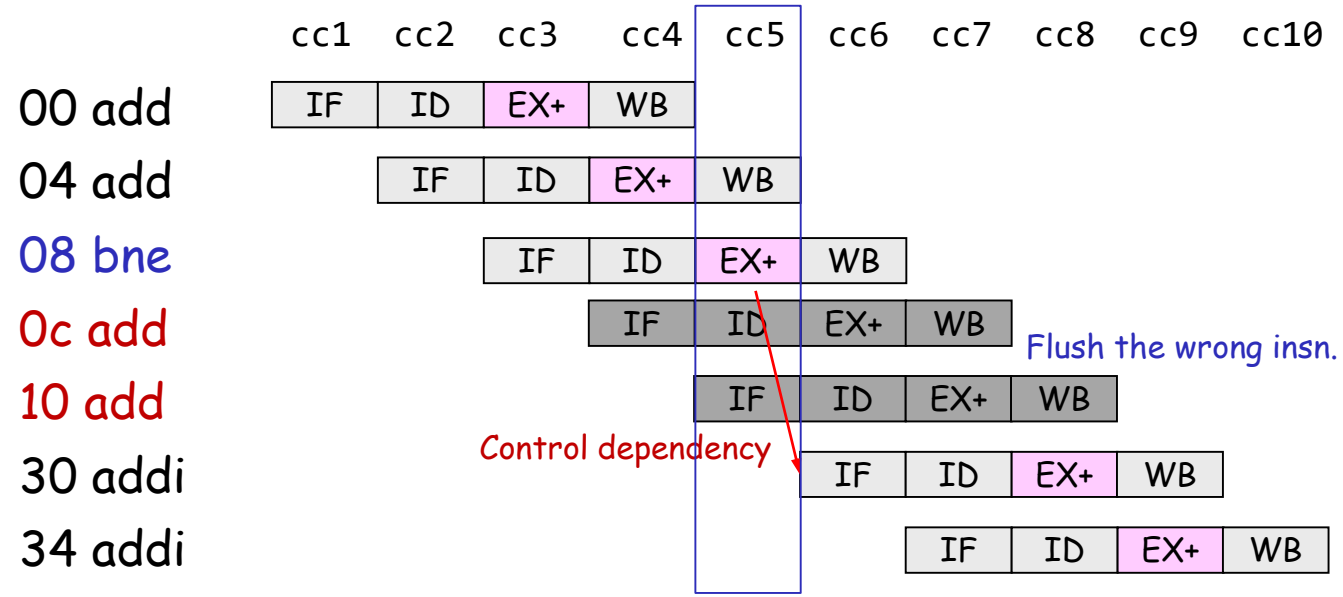
- Draw the main datapath of the processor rvcore_4s and write the valid values on wires when the processor is executing these three instructions in ID, EX+, and WB stages

```
0x00   addi x1, x0, 3      # x1 = 3
0x04   addi x2, x1, 4      # x2 = 3 + 4 = 7
0x08   add  x5, x1, x2     # x5 = 3 + 7 = 10
```



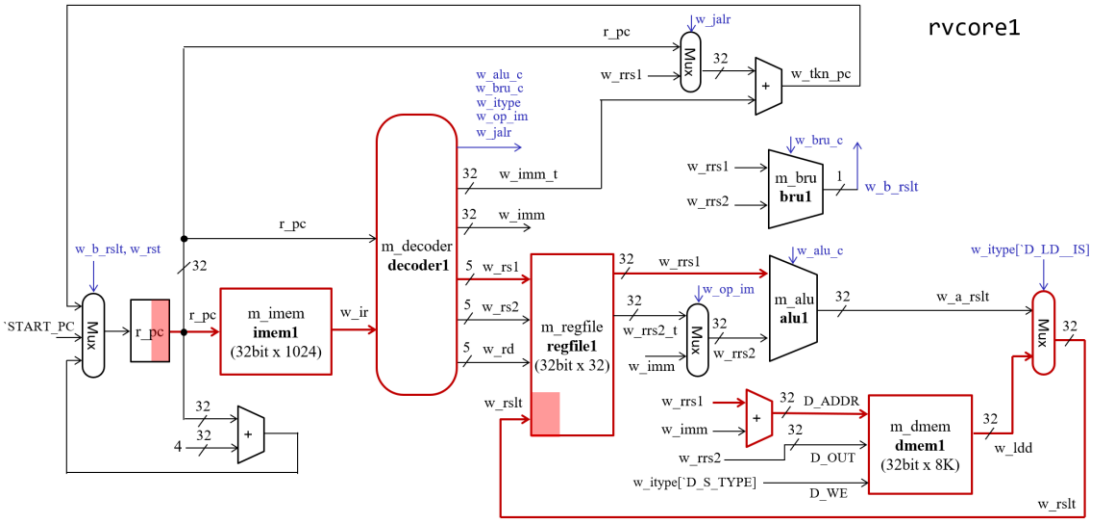
Why do branch instructions degrade IPC?

- Another approach is fetching the following instructions (0c add, 10 add) after a branch (bne) is fetched.
- When a branch (08 bne) is taken, the wrong instructions fetched (0c add, 10 add) are flushed.



four-stage pipelining processor executing instruction sequence with a taken branch

Comparison of critical path between rvcore1 and rvcore_4s

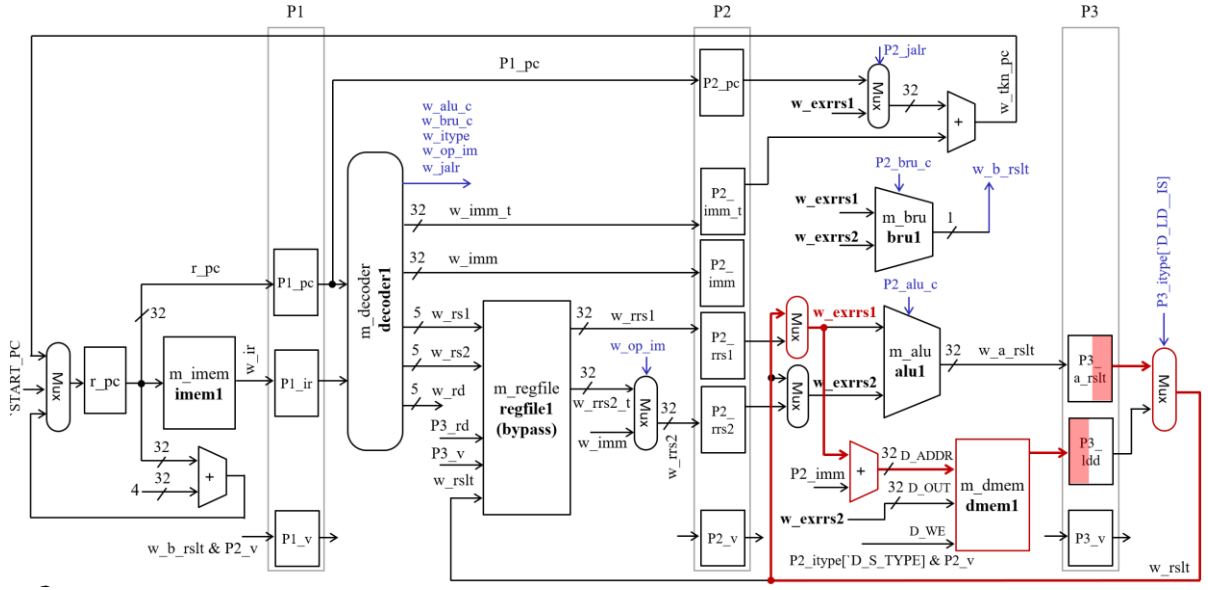


(a) the critical path of rvcore_1s

deley(register read) + delay(imem read) + delay(decode) + delay(regfile read) + delay(adder) + delay(dmem read) + delay(mux)

(b) the critical path of rvcore_4s

deley(register read) + delay(mux) + delay(mux) + delay(adder) + delay(dmem read)



Recommended Reading

- Increasing Processor Performance by Implementing Deeper Pipelines
  - Eric Sprangle , Doug Carmean (Intel Corporation)
  - ISCA-2002 pp. 25-34 (2002)

This paper will show that the branch misprediction latency is the single largest contributor to performance degradation as pipelines are stretched, and therefore branch prediction and fast branch recovery will continue to increase in importance. We will also show that higher performance cores, implemented with longer pipelines for example, will put more pressure on the memory system, and therefore require larger on-chip caches. Finally, we will show that in the same process technology, designing deeper pipelines can increase the processor frequency by 100%, which, when combined with larger on-chip caches can yield performance improvements of 35% to 90% over a Pentium® 4 like processor.

Basic Pentium® III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium® 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive