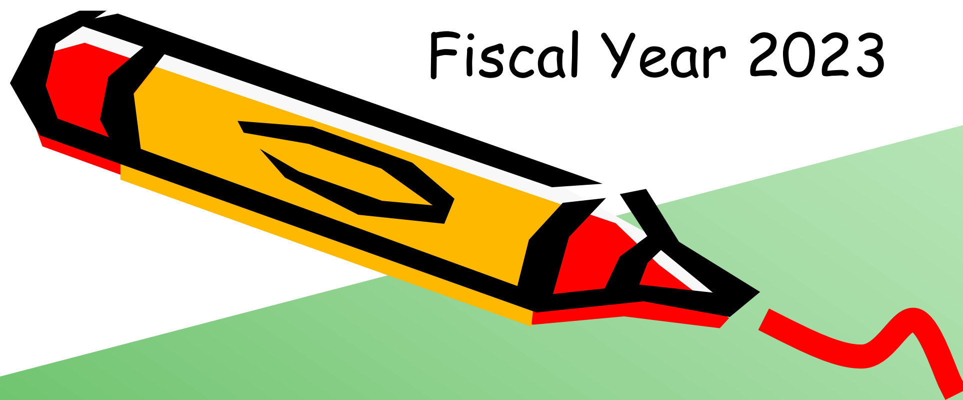


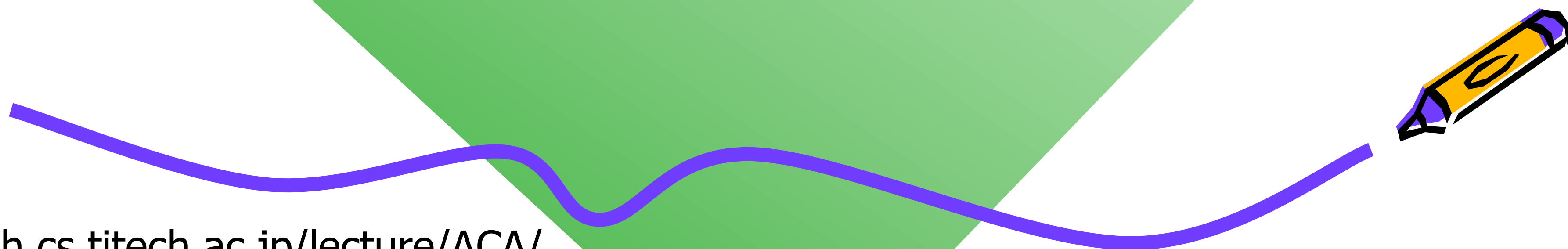
Fiscal Year 2023

Ver. 2023-12-21a

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.W834, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

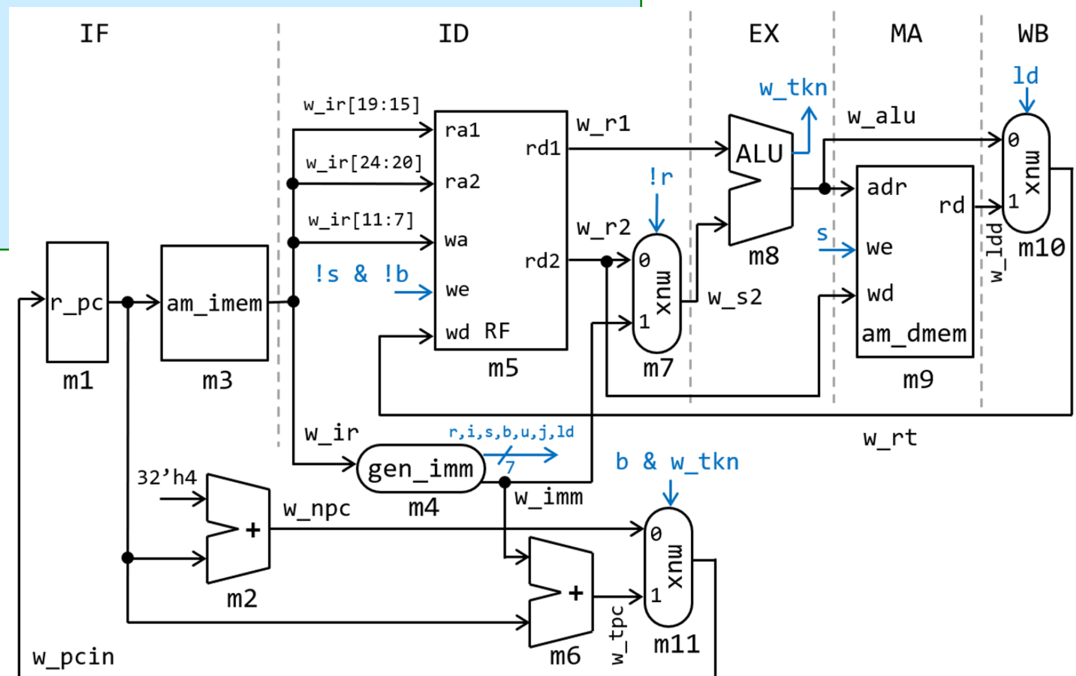
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

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.

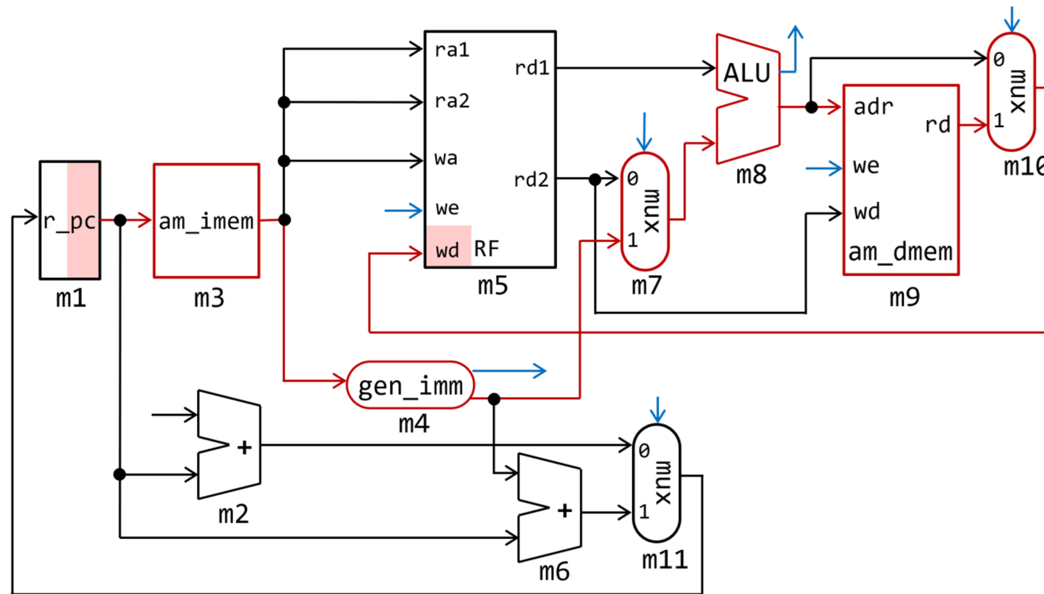
proc05: single cycle proc. supporting add, addi, lw, sw, bne

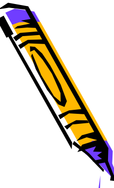
```
1  module m_proc5(w_clk);
2    input wire w_clk;
3    wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
4    wire [31:0] w_alu, w_ldd, w_tpc, w_pcin;
5    wire w_tkn;
6    reg [31:0] r_pc=0;
7    assign w_pcin = (w_b & w_tkn) ? w_tpc : w_npc;    # m11
8    assign w_npc = r_pc + 4;
9    m_am_imem m3 (r_pc, w_ir);
10   wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld;
11   m_gen_imm m4 (w_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
12   m_RF m5 (w_clk, w_ir[19:15], w_ir[24:20], w_r1, w_r2, w_ir[11:7], !w_s & !w_b, w_rt);
13   assing w_tpc = r_pc + w_imm;   # m6
14   assign w_s2 = (!w_r & !w_b) ? w_imm : w_r2;
15   assign w_alu = w_r1 + w_s2;    # m8
16   assign w_tkn = w_r1 != w_s2;   # m8
17   m_am_dmem m9 (w_clk, w_alu, w_s, w_r2, w_ldd);
18   assign w_rt = (w_ld) ? w_ldd : w_alu;
19   always @(posedge w_clk) r_pc <= w_pcin;
20 endmodule
```



Critical path of proc5

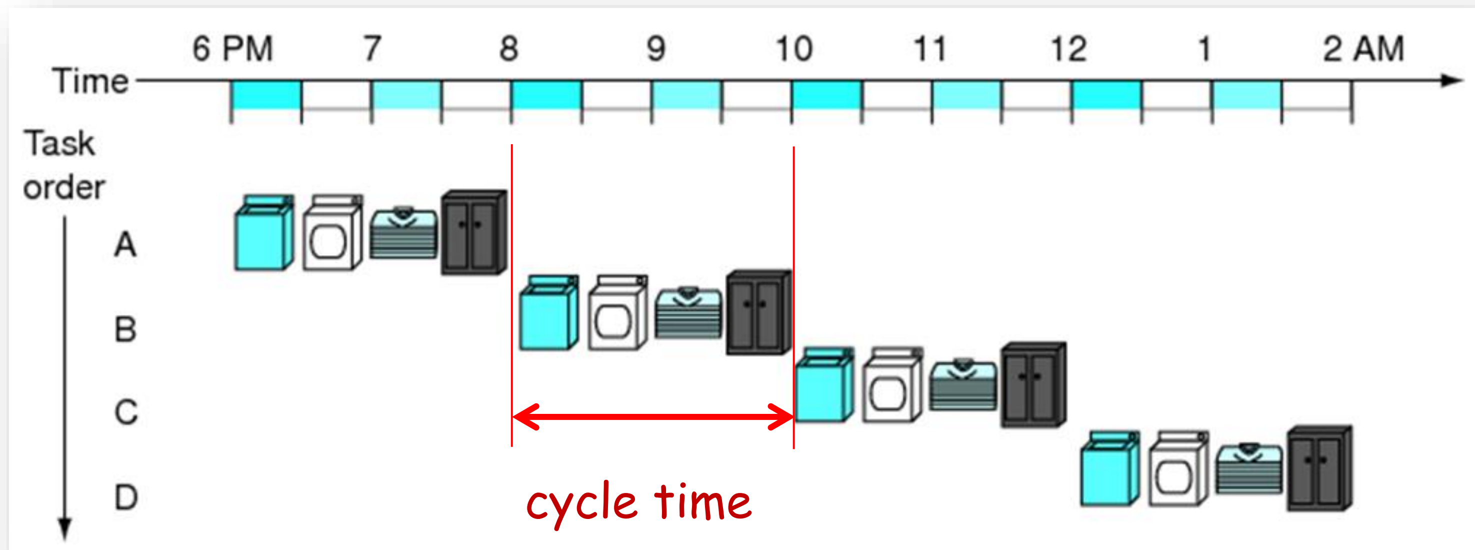
- It is too slow to be practical.





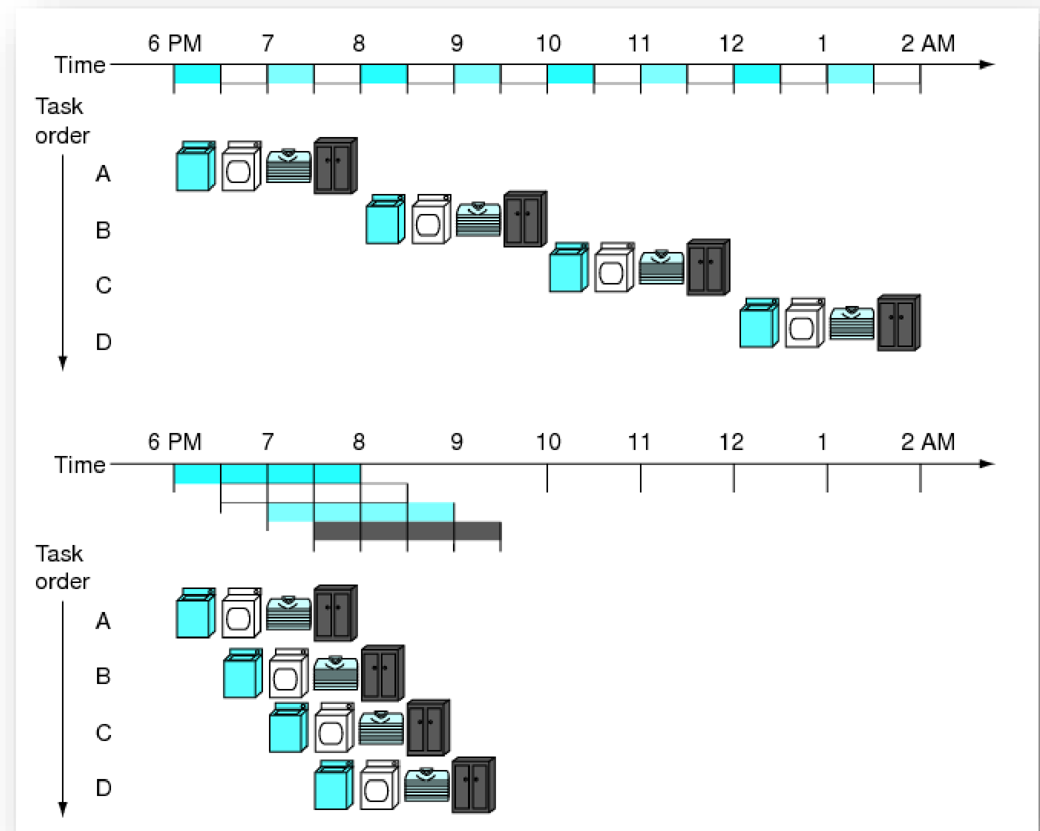
Single-cycle implementation of laundry

- (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.



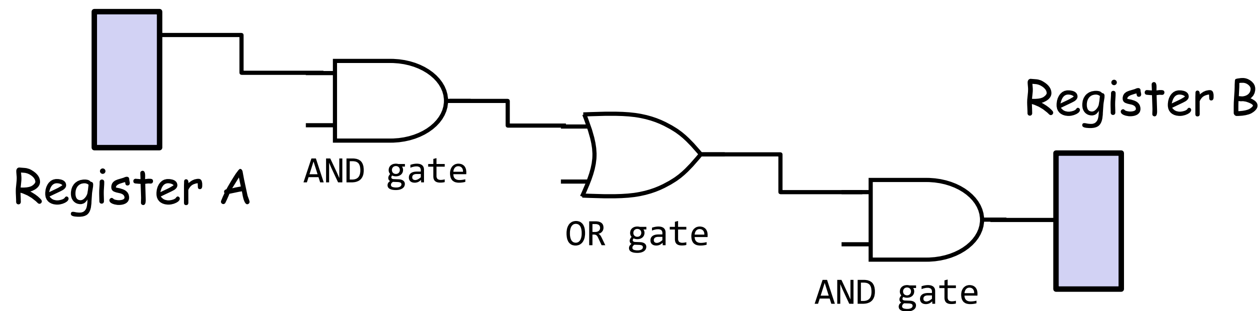
Single-cycle implementation and pipelining

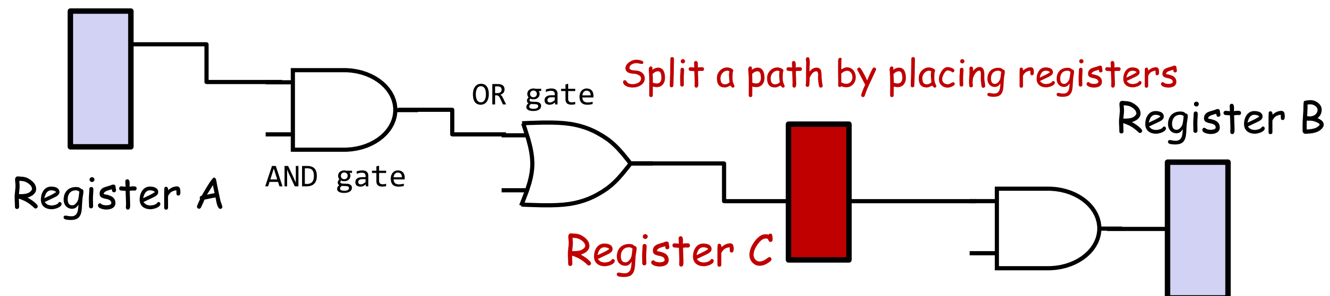
- When the washing of load A is finished at 6:30 p.m., another washing of load B starts.
- Pipelined laundry takes 3.5 hours just using the same hardware resources. The cycle time is 30 minutes.
- What is the latency (execution time) of each load?



Clock rate is mainly determined by

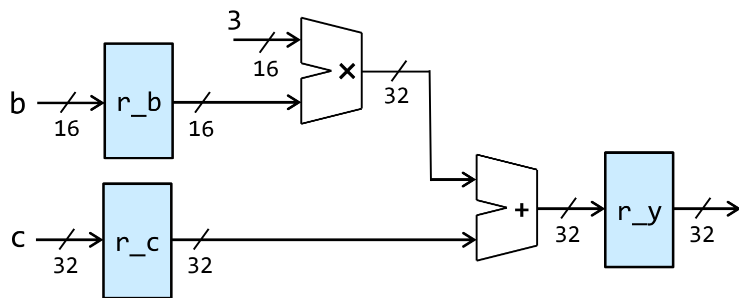
- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout

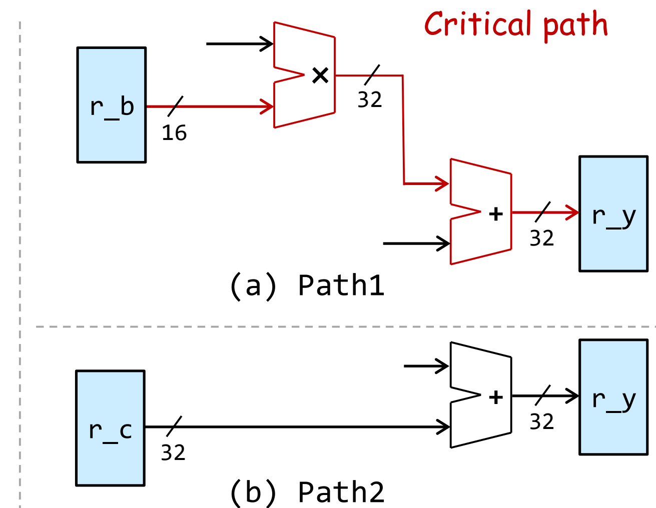




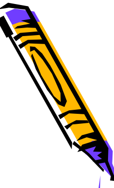
Pipelining example: multiply-add operation (1)

- 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.

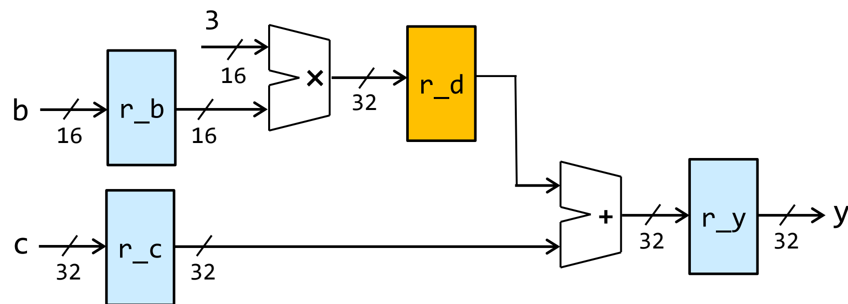


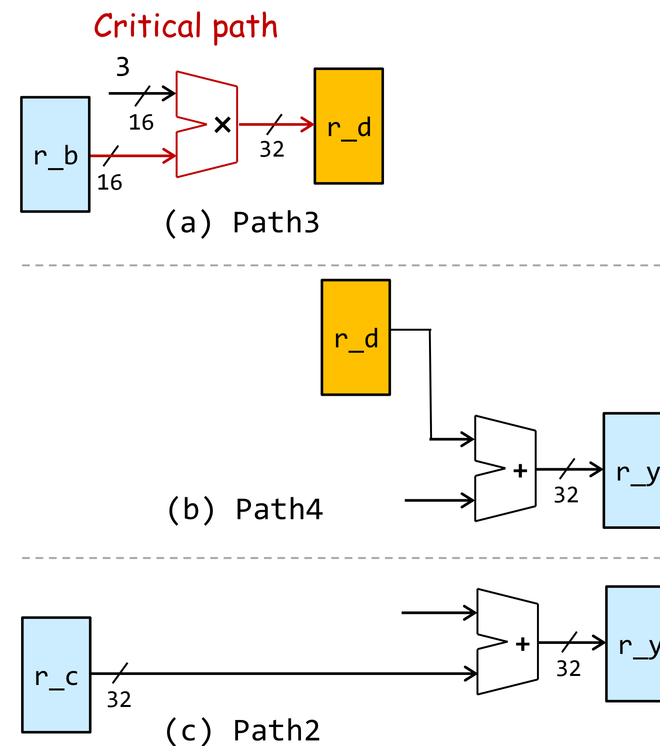


Pipelining example: multiply-add operation (2)



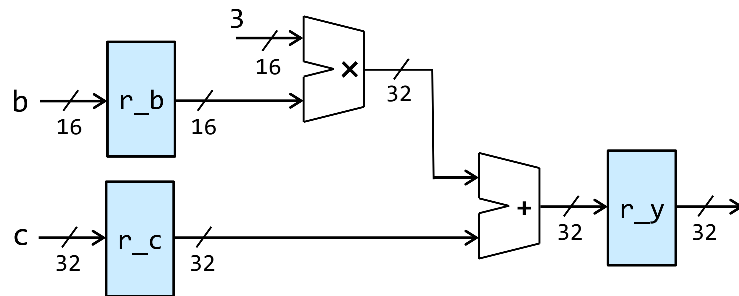
- By inserting register r_d, the critical path can be divided into Path3 and Path4.
- As a result, the new critical path becomes Path3.
- This has the disadvantage that input b and c in the same clock cycle cannot be processed.



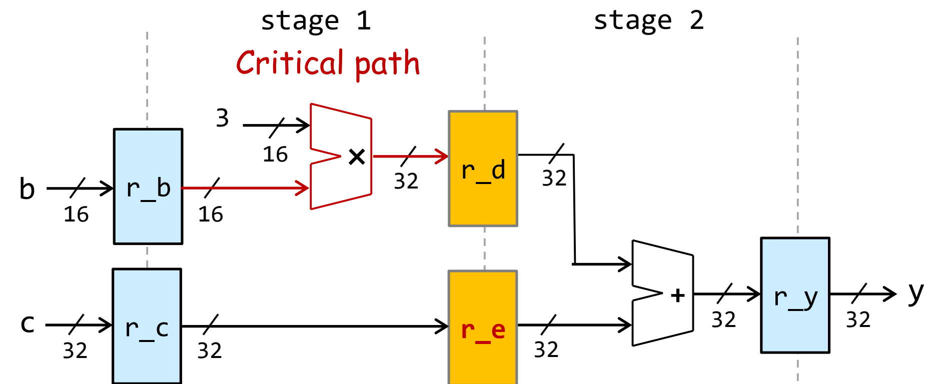


Pipelining example: multiply-add operation (3)

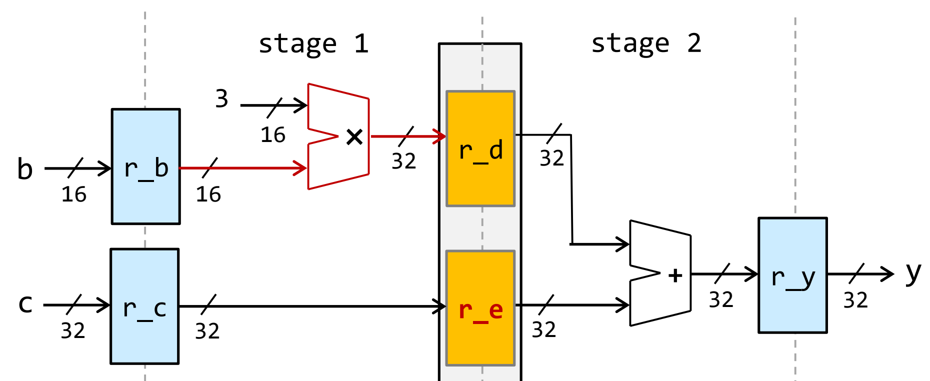
- To overcome this drawback, we insert register r_e.
- This realizes a pipeline with stages 1 and 2. A set of registers between two adjacent stages are called a pipeline register.

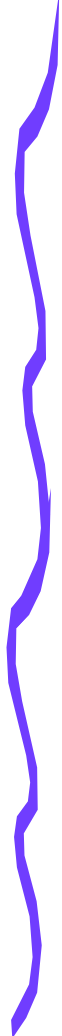


(a) original multiply-add circuit



(b) two-stage pipelined circuit



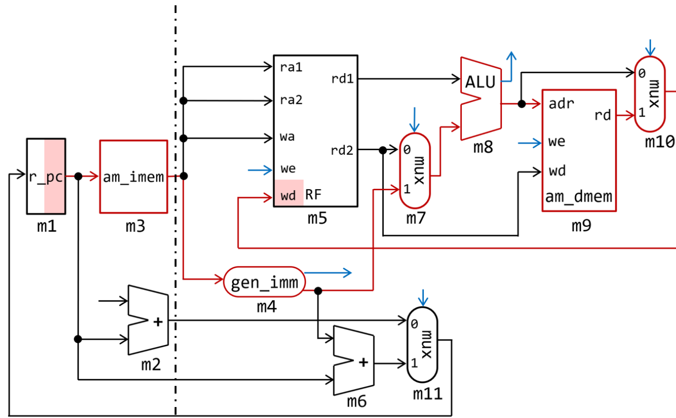


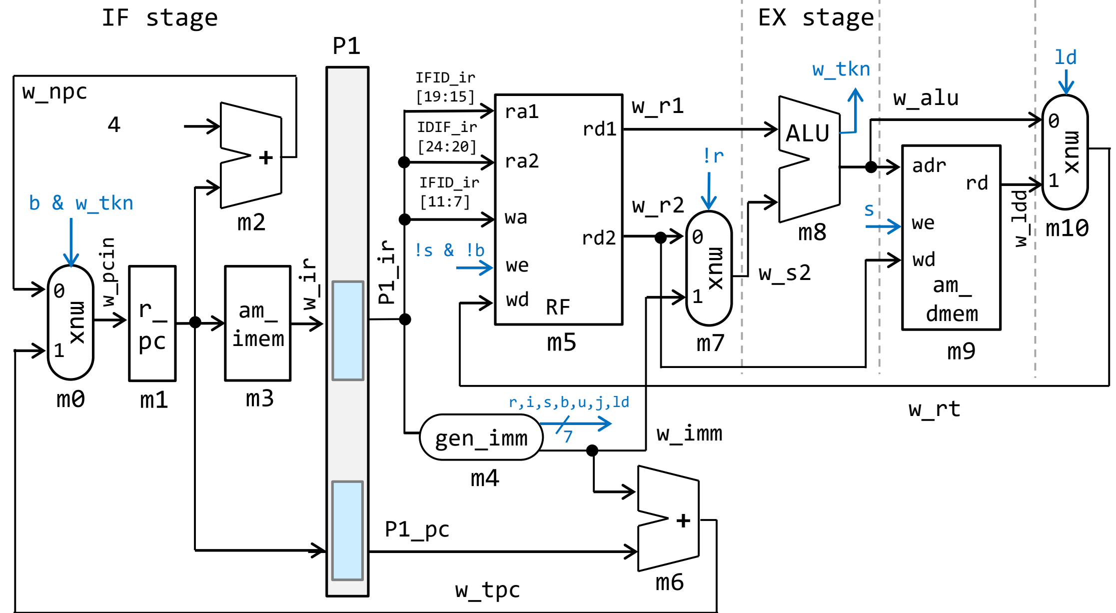
Single cycle processor to 2-stage pipelining proc.

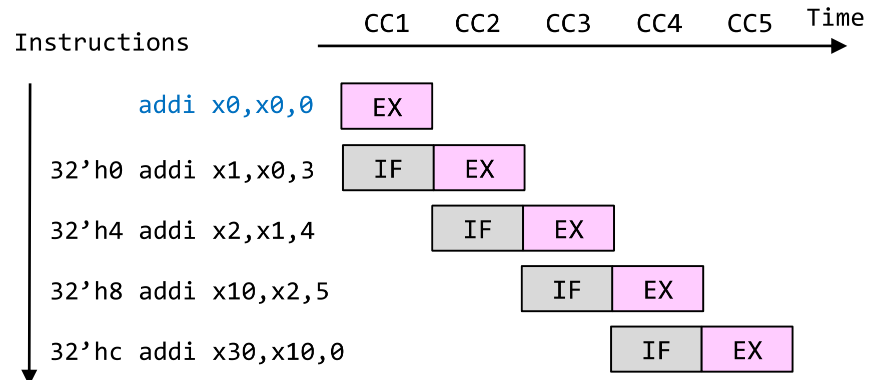
- This single cycle processor is too slow to be practical.
- The strategy is to separate the instruction fetch step (IF) and other steps (ID, EX, MA, WB).

(a) proc5: single-cycle processor

proc6: 2-stage pipelining processor

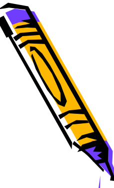

(a) proc5: single-cycle processor


(b) proc6: 2-stage pipelining processor

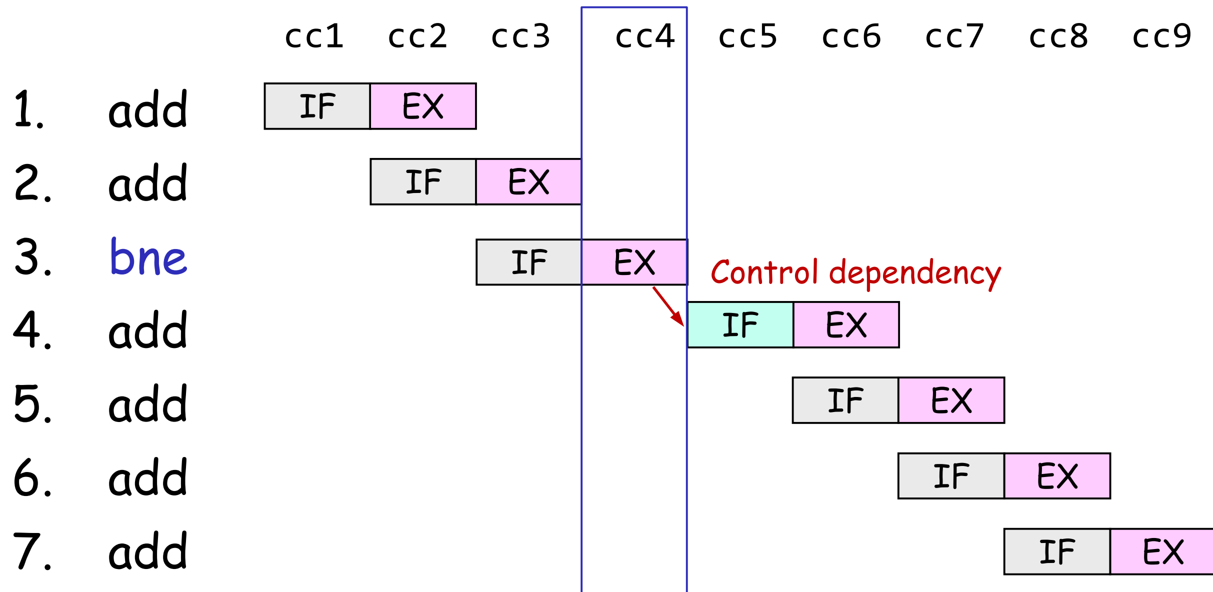


(c) a pipeline diagram of proc6

Why do branch instructions degrade IPC?



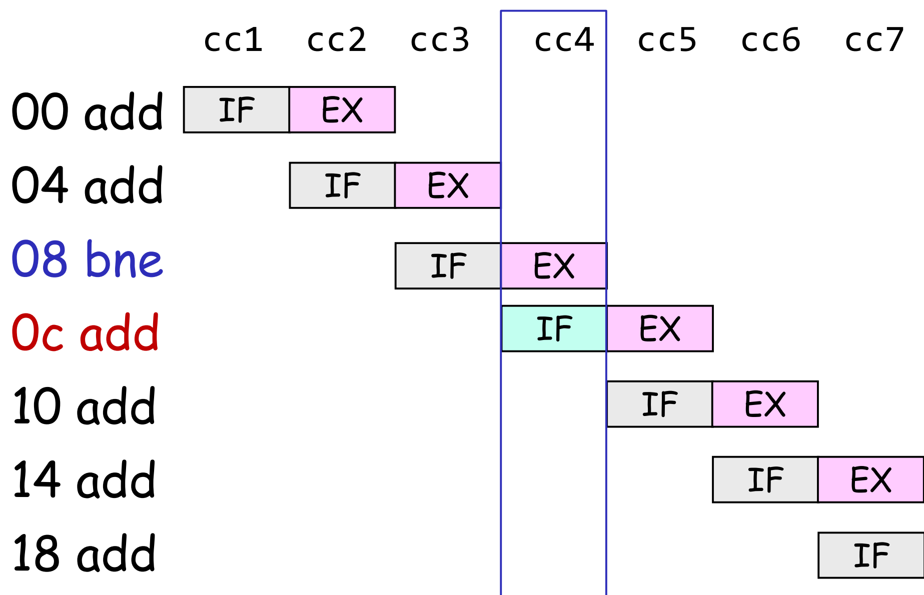
- The branch taken / untaken is determined in the execution stage (EX) of the branch.
- The conservative approach is stalling instruction fetch until the branch direction is determined.



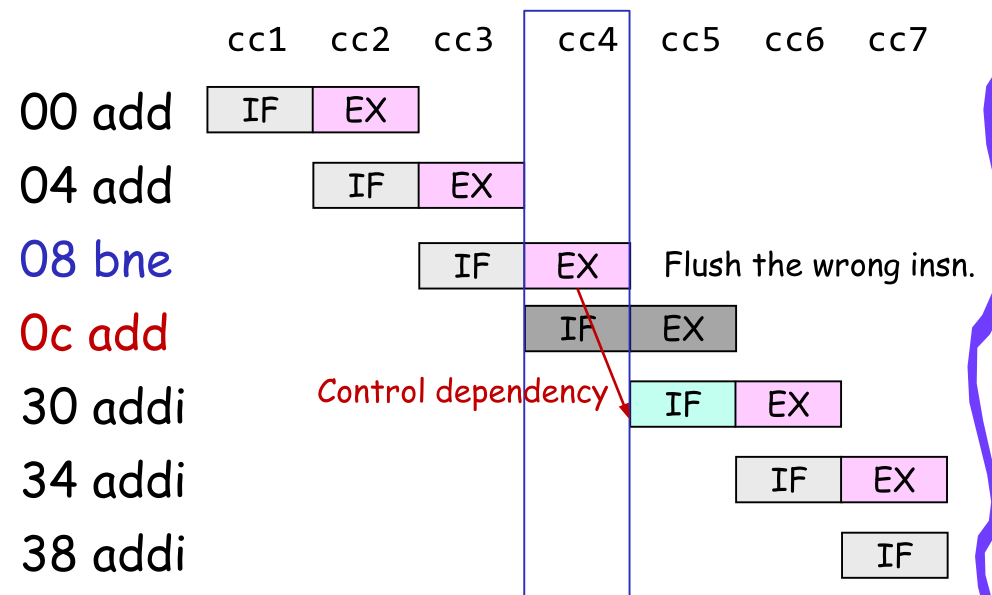
five stages pipelined processor executing instruction sequence with a branch

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.



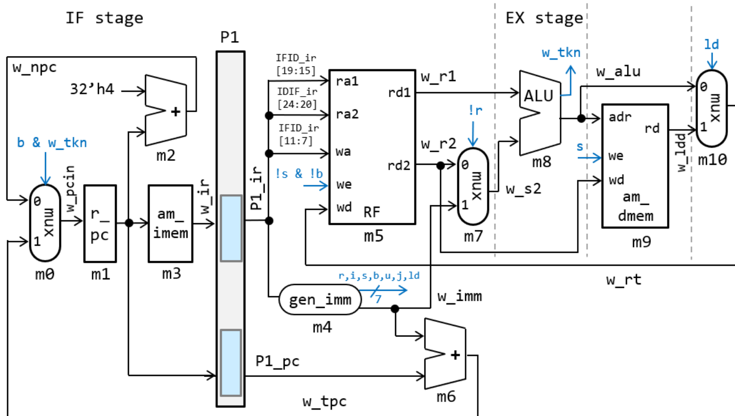
(a) branch untaken case



(b) branch taken case

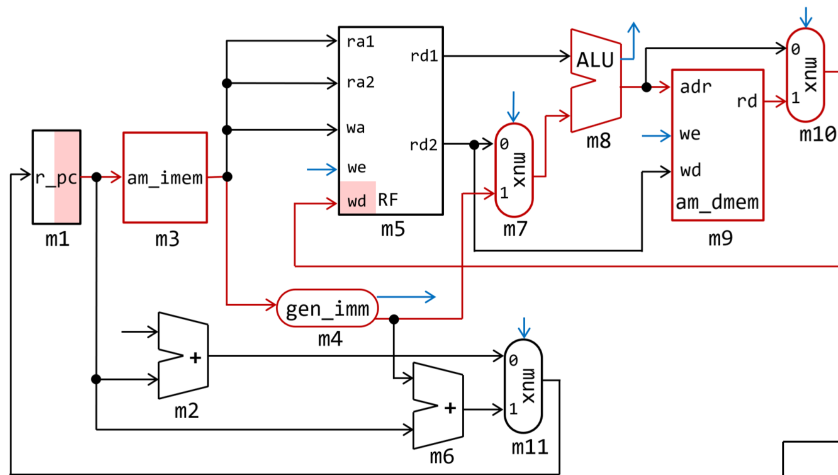
Verilog HDL code for proc5 and proc6

```
1   module m_proc5(w_clk);
2     input wire w_clk;
3     wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
4     wire [31:0] w_alu, w_ldd, w_tpc, w_pcin;
5     wire w_tkn;
6     reg [31:0] r_pc=0;
7     assign w_pcin = (w_b & w_tkn) ? w_tpc : w_npc;    # m11
8     assign w_npc = r_pc + 4;
9     m_am_imem m3 (r_pc, w_ir);
10    wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld;
11    m_gen_imm m4 (w_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
12    m_RF m5 (w_clk, w_ir[19:15], w_ir[24:20],
                w_r1, w_r2, w_ir[11:7], !w_s & !w_b, w_rt);
13    assing w_tpc = r_pc + w_imm;   # m6
14    assign w_s2 = (!w_r & !w_b) ? w_imm : w_r2;
15    assign w_alu = w_r1 + w_s2;    # m8
16    assign w_tkn = w_r1 != w_s2;   # m8
17    m_am_dmem m9 (w_clk, w_alu, w_s, w_r2, w_ldd);
18    assign w_rt = (w_ld) ? w_ldd : w_alu;
19    always @(posedge w_clk) r_pc <= w_pcin;
20  endmodule
```

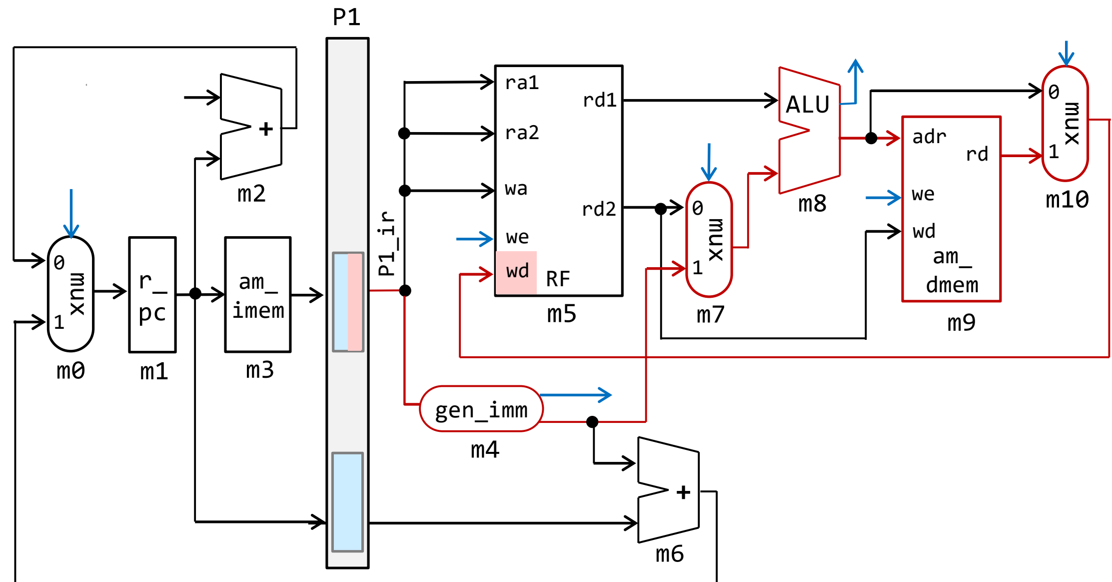



```
1   module m_proc6(w_clk);
2     input wire w_clk;
3     reg [31:0] P1_ir=32'h13, P1_pc=0; reg P1_v=0;
4     wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
5     wire [31:0] w_alu, w_ldd, w_tpc, w_pcin;
6     wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld, w_tkn;
7     reg [31:0] r_pc=0;
8     wire w_miss = w_b & w_tkn & P1_v;
9     assign w_pcin = (w_miss) ? w_tpc : w_npc;
10    assign w_npc = r_pc + 4;
11    m_am_imem m3 (r_pc, w_ir);
12    always @(posedge w_clk)
13      {r_pc, P1_ir, P1_pc, P1_v} <= {w_pcin, w_ir, r_pc, !w_miss};
14    m_gen_imm m4 (P1_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
15    m_RF m5 (w_clk, P1_ir[19:15], P1_ir[24:20], w_r1, w_r2,
                P1_ir[11:7], !w_s & !w_b & P1_v, w_rt);
16    assign w_tpc = P1_pc + w_imm;
17    assign w_s2 = (!w_r & !w_b) ? w_imm : w_r2;
18    assign w_alu = w_r1 + w_s2;
19    assign w_tkn = w_r1 != w_s2;
20    m_am_dmem m9 (w_clk, w_alu, w_s & P1_v, w_r2, w_ldd);
21    assign w_rt = (w_ld) ? w_ldd : w_alu;
22  endmodule
```

Comparison of critical path between proc5 and proc6



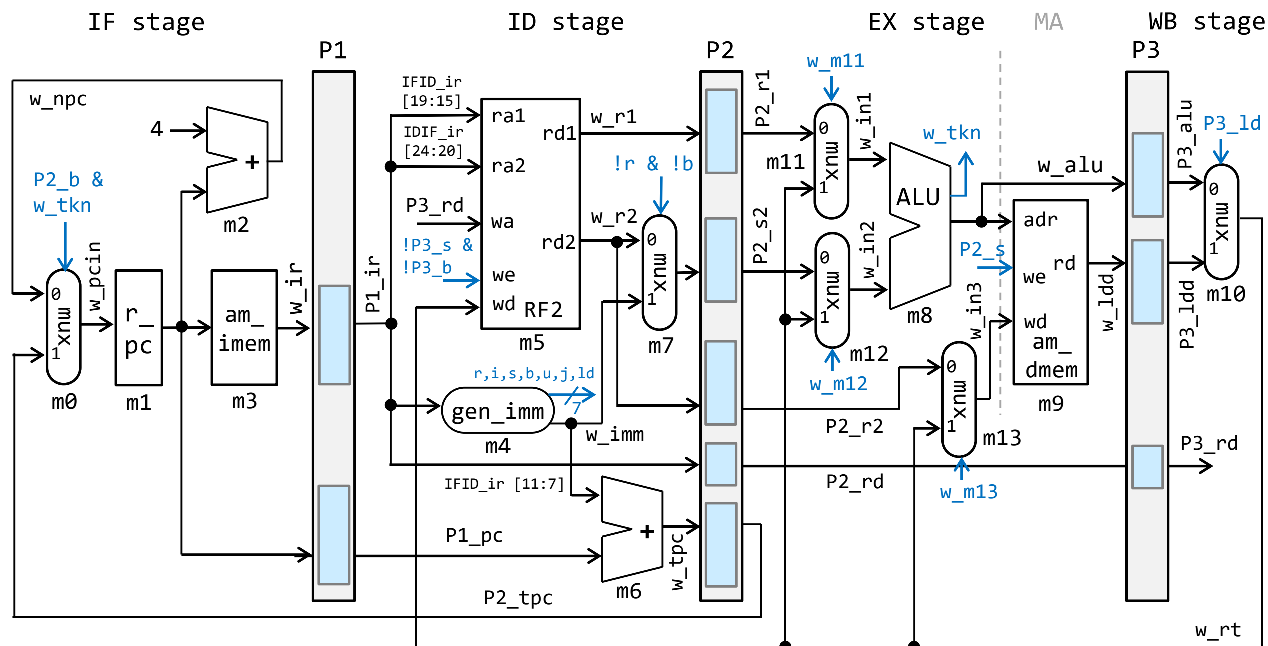
(a) the critical path of proc5



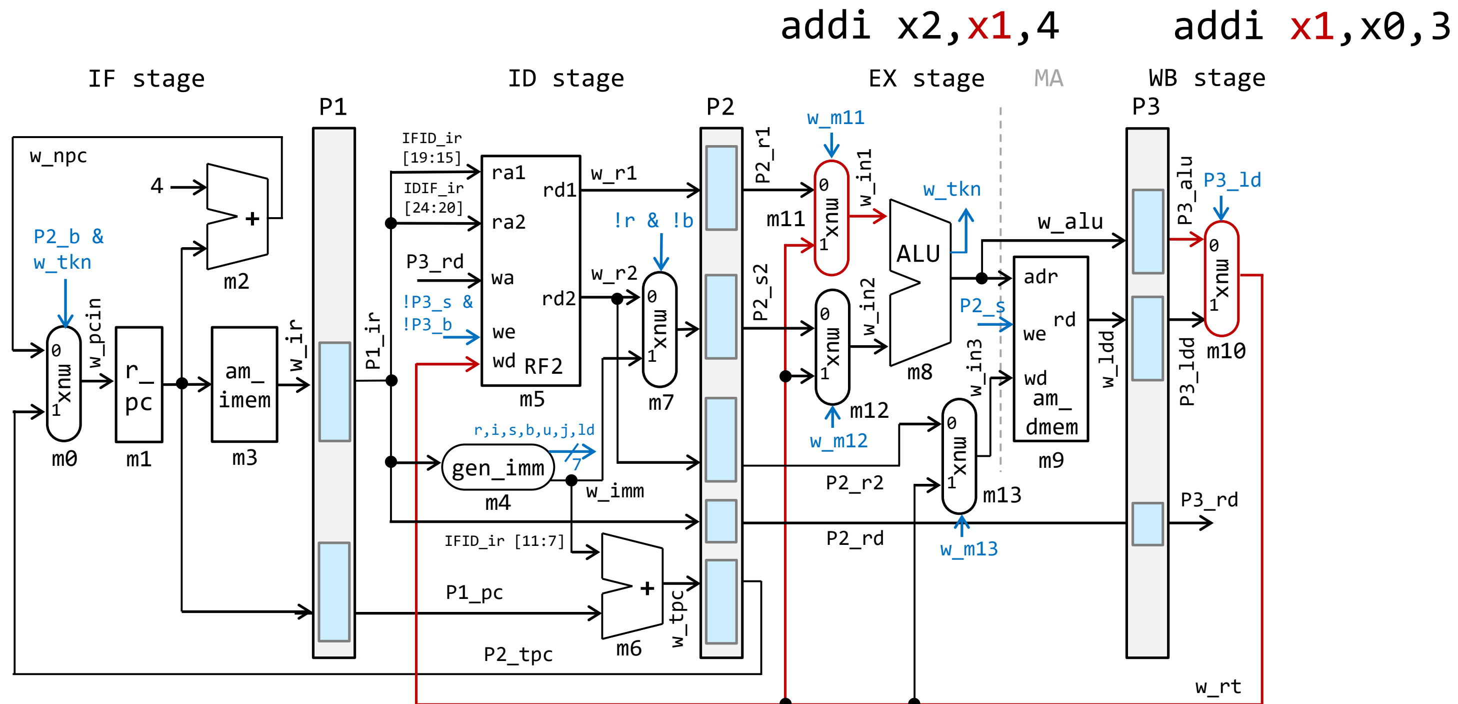
(b) the critical path of proc6

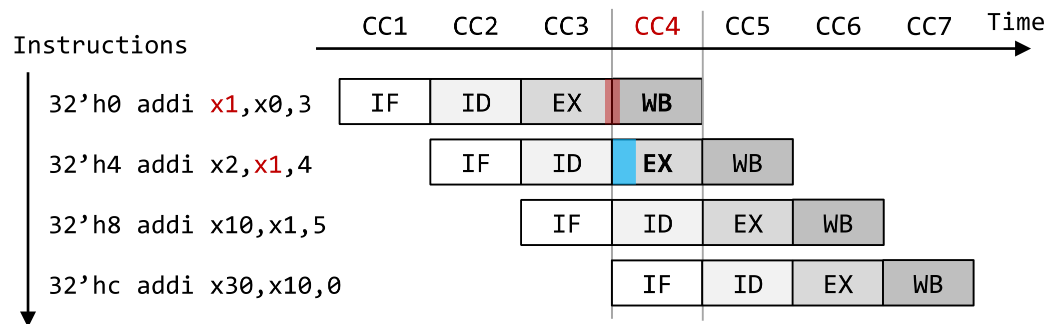
proc8: 4-stage pipelining processor

- The strategy is to separate instruction fetch step (IF), instruction decode step (ID), execution and memory access steps (EX), and write back step (WB).
- Use the pipeline register P2 between ID and EX, and pipeline register P3 between EX and WB.



proc8: 4-stage pipelining processor

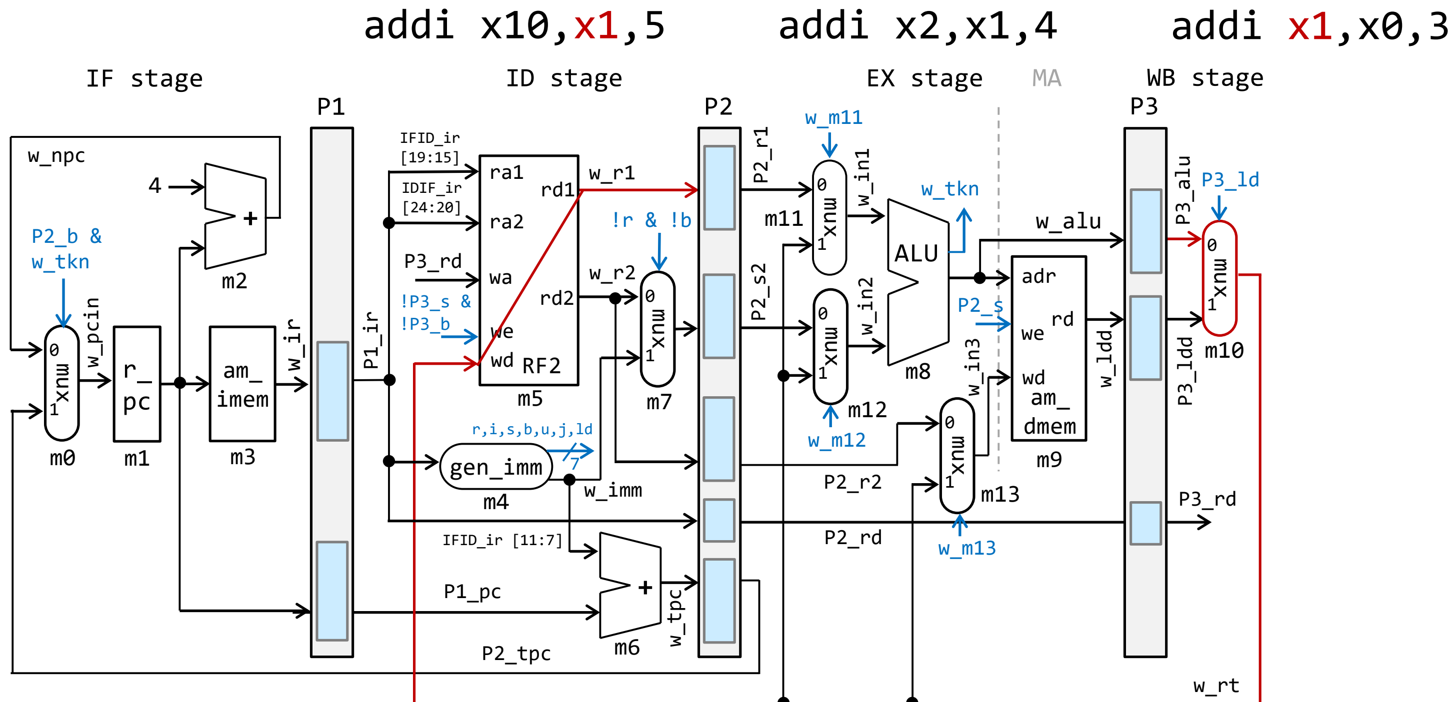


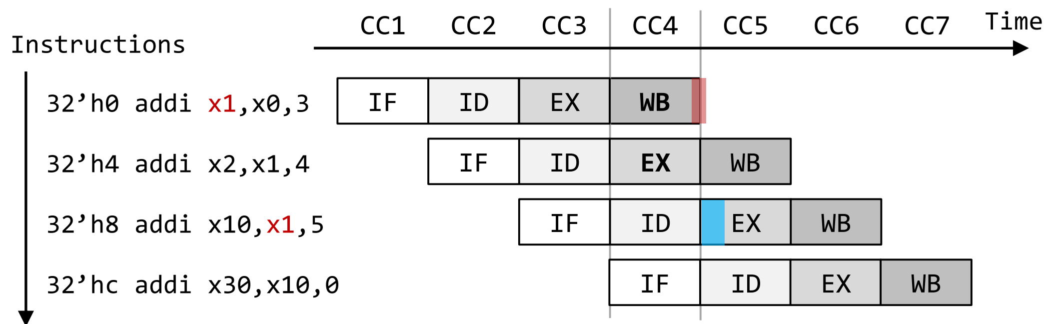


Data forwarding

A mechanism for supplying data to the ALU from pipeline registers of the subsequent stages.

proc8: 4-stage pipelining processor





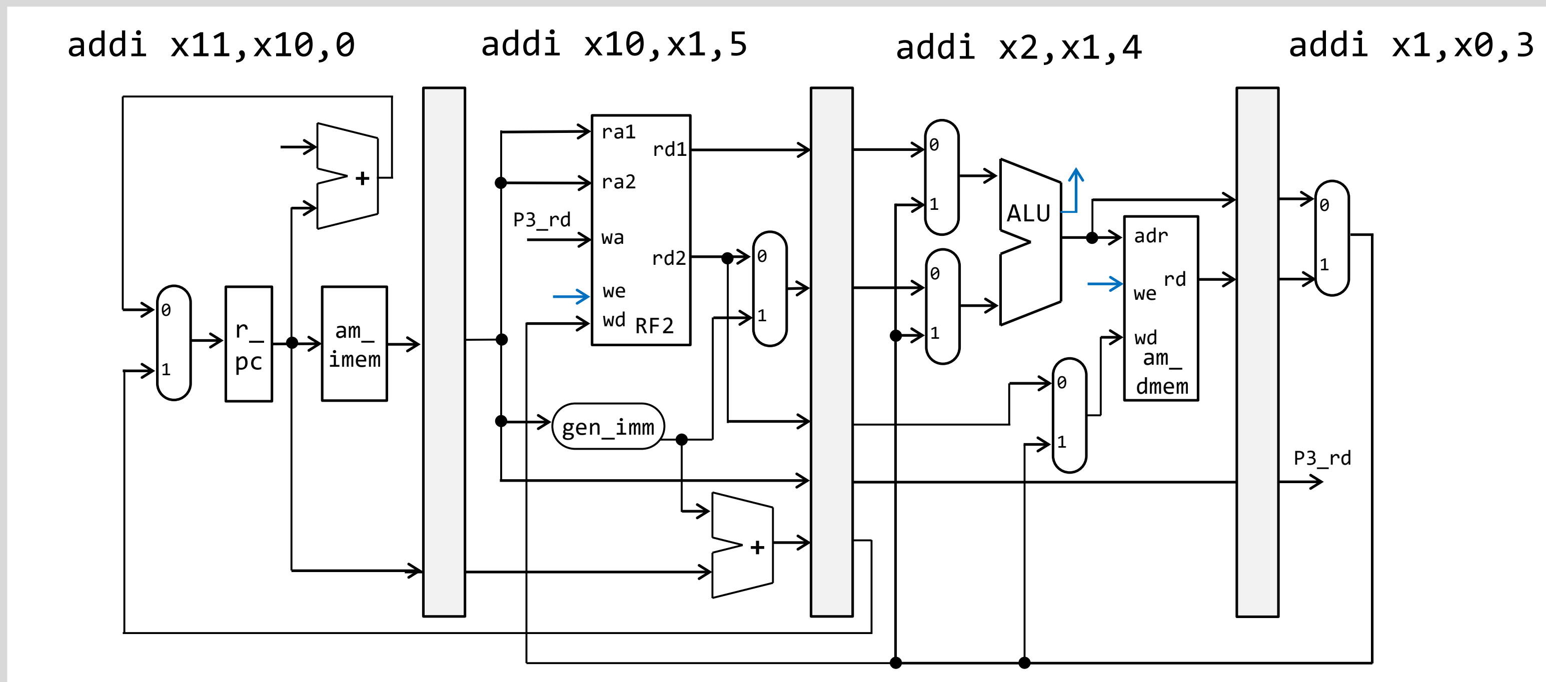
Instructions	CC1	CC2	CC3	CC4	CC5	CC6	CC7
32'h0 addi x1,x0,3	IF	ID	EX	WB			
32'h4 addi x2,x1,4		IF	ID	EX	WB		
32'h8 addi x10,x1,5			IF	ID	EX	WB	
32'hc addi x30,x10,0				IF	ID	EX	WB

Time

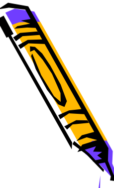
Exercise 1

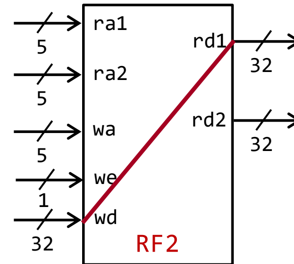
- Draw a block diagram of the processor proc08 and write the valid values on wires when the processor is executing the four instructions

```
0x00  addi x1,x0,3
0x04  addi x2,x1,4
0x08  addi x10,x1,5
0x0c  addi x11,x10,0
```

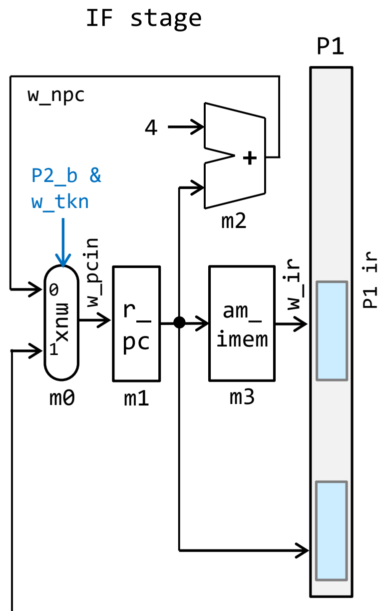


RF2 (Register File) with bypassing

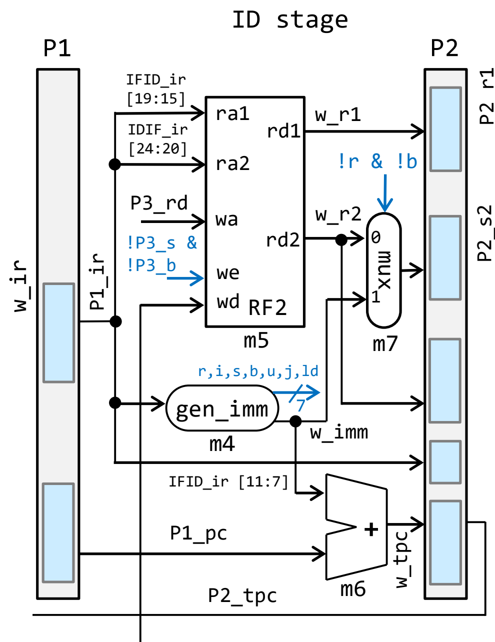




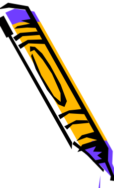

```
module m_RF2(w_clk, w_radr1, w_radr2, w_rd1, w_rd2, w_wadr, w_we, w_wd);
  input  wire w_clk, w_we;
  input  wire [4:0] w_radr1, w_radr2, w_wadr;
  output wire [31:0] w_rd1, w_rd2;
  input  wire [31:0] w_wd;
  reg [31:0] mem [0:31];
  wire w_bp1 = (w_we & w_radr1==w_wadr);
  wire w_bp2 = (w_we & w_radr2==w_wadr);
  assign w_rd1 = (w_radr1==5'd0) ? 32'd0 : (w_bp1) ? w_wd : mem[w_radr1];
  assign w_rd2 = (w_radr2==5'd0) ? 32'd0 : (w_bp2) ? w_wd : mem[w_radr2];
  always @(posedge w_clk) if (w_we) mem[w_wadr] <= w_wd;
  integer i; initial for (i=0; i<32; i=i+1) mem[i] = 0;
endmodule
```

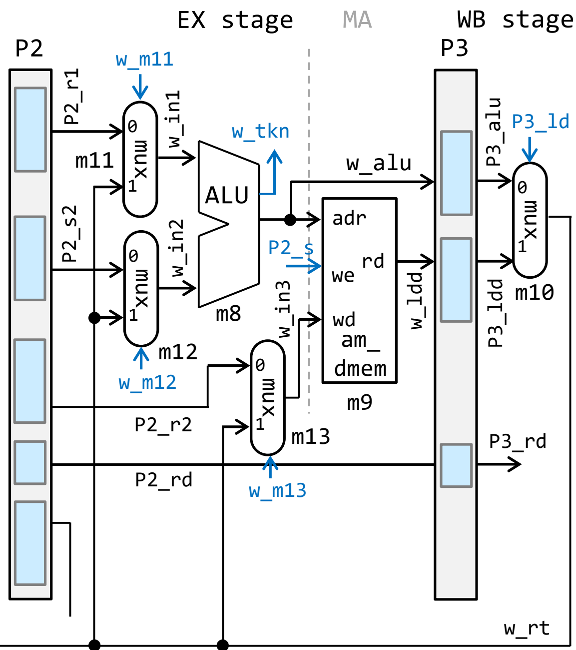



```
module m_proc8(w_clk);
  input wire w_clk;
  reg [31:0] P1_ir=32'h13, P1_pc=0, P2_pc=0, P3_pc=0;
  reg [31:0] P2_r1=0, P2_s2=0, P2_r2=0, P2_tpc=0;
  reg [31:0] P3_alu=0, P3_ldd=0;
  reg P2_r=0, P2_s=0, P2_b=0, P2_ld=0, P3_s=0, P3_b=0, P3_ld=0;
  reg [4:0] P2_rd=0, P2_rs1=0, P2_rs2=0, P3_rd=0;
  reg P1_v=0, P2_v=0, P3_v=0;
  wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
  wire [31:0] w_alu, w_ldd, w_tpc, w_pcin, w_in1, w_in2, w_in3;
  wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld, w_tkn;
  reg [31:0] r_pc = 0; // m1
  wire w_miss = P2_b & w_tkn & P2_v;
  assign w_pcin = (w_miss) ? P2_tpc : w_npc; // m0
  assign w_npc = r_pc + 4; // m2
  m_am_imem m3 (r_pc, w_ir);
  m_gen_imm m4 (P1_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
  m_RF2 m5 (w_clk, P1_ir[19:15], P1_ir[24:20], w_r1, w_r2,
            P3_rd, !P3_s & !P3_b & P3_v, w_rt);
  assign w_tpc = P1_pc + w_imm; // m6
  assign w_s2 = (!w_r & !w_b) ? w_imm : w_r2; // m7
  always @(posedge w_clk) begin
    {P1_v, P2_v, P3_v} <= {!w_miss, !w_miss & P1_v, P2_v};
    {r_pc, P1_ir, P1_pc, P2_pc} <= {w_pcin, w_ir, r_pc, P1_pc};
    {P2_r1, P2_r2, P2_s2, P2_tpc} <= {w_r1, w_r2, w_s2, w_tpc};
    {P2_r, P2_s, P2_b, P2_ld} <= {w_r, w_s, w_b, w_ld};
    {P2_rs2, P2_rs1, P2_rd} <= {P1_ir[24:15], P1_ir[11:7]};
    {P3_pc, P3_ld} <= {P2_pc, P2_ld};
    {P3_alu, P3_ldd, P3_rd} <= {w_alu, w_ldd, P2_rd};
  end
  assign w_alu = w_in1 + w_in2;   // m8
  assign w_tkn = w_in1 != w_in2;  // m8
  m_am_dmem m9 (w_clk, w_alu, P2_s & P2_v, w_in3, w_ldd);
  assign w_rt = (P3_ld) ? P3_ldd : P3_alu; // m10
  assign w_in1 = (|P3_rd & P2_rs1==P3_rd) ? w_rt : P2_r1; // m11
  assign w_in3 = (!P3_rd & P2_rs2==P3_rd) ? w_rt : P2_r2; // m13
  assign w_in2 = (|P3_rd & (P2_r|P2_b) & P2_rs2==P3_rd) ? w_rt : P2_s2;// m12
endmodule
```




```
module m_proc8(w_clk);
  input wire w_clk;
  reg [31:0] P1_ir=32'h13, P1_pc=0, P2_pc=0, P3_pc=0;
  reg [31:0] P2_r1=0, P2_s2=0, P2_r2=0, P2_tpc=0;
  reg [31:0] P3_alu=0, P3_ldd=0;
  reg P2_r=0, P2_s=0, P2_b=0, P2_ld=0, P3_s=0, P3_b=0, P3_ld=0;
  reg [4:0] P2_rd=0, P2_rs1=0, P2_rs2=0, P3_rd=0;
  reg P1_v=0, P2_v=0, P3_v=0;
  wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
  wire [31:0] w_alu, w_ldd, w_tpc, w_pcin, w_in1, w_in2, w_in3;
  wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld, w_tkn;
  reg [31:0] r_pc = 0; // m1
  wire w_miss = P2_b & w_tkn & P2_v;
  assign w_pcin = (w_miss) ? P2_tpc : w_npc; // m0
  assign w_npc = r_pc + 4; // m2
  m_am_imem m3 (r_pc, w_ir);
  m_gen_imm m4 (P1_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
  m_RF2 m5 (w_clk, P1_ir[19:15], P1_ir[24:20], w_r1, w_r2,
            P3_rd, !P3_s & !P3_b & P3_v, w_rt);
  assign w_tpc = P1_pc + w_imm; // m6
  assign w_s2 = (!w_r & !w_b) ? w_imm : w_r2; // m7
  always @(posedge w_clk) begin
    {P1_v, P2_v, P3_v} <= {!w_miss, !w_miss & P1_v, P2_v};
    {r_pc, P1_ir, P1_pc, P2_pc} <= {w_pcin, w_ir, r_pc, P1_pc};
    {P2_r1, P2_r2, P2_s2, P2_tpc} <= {w_r1, w_r2, w_s2, w_tpc};
    {P2_r, P2_s, P2_b, P2_ld} <= {w_r, w_s, w_b, w_ld};
    {P2_rs2, P2_rs1, P2_rd} <= {P1_ir[24:15], P1_ir[11:7]};
    {P3_pc, P3_ld} <= {P2_pc, P2_ld};
    {P3_alu, P3_ldd, P3_rd} <= {w_alu, w_ldd, P2_rd};
  end
  assign w_alu = w_in1 + w_in2;   // m8
  assign w_tkn = w_in1 != w_in2;  // m8
  m_am_dmem m9 (w_clk, w_alu, P2_s & P2_v, w_in3, w_ldd);
  assign w_rt = (P3_ld) ? P3_ldd : P3_alu; // m10
  assign w_in1 = (|P3_rd & P2_rs1==P3_rd) ? w_rt : P2_r1; // m11
  assign w_in3 = (!P3_rd & P2_rs2==P3_rd) ? w_rt : P2_r2; // m13
  assign w_in2 = (|P3_rd & (P2_r|P2_b) & P2_rs2==P3_rd) ? w_rt : P2_s2;// m12
endmodule
```

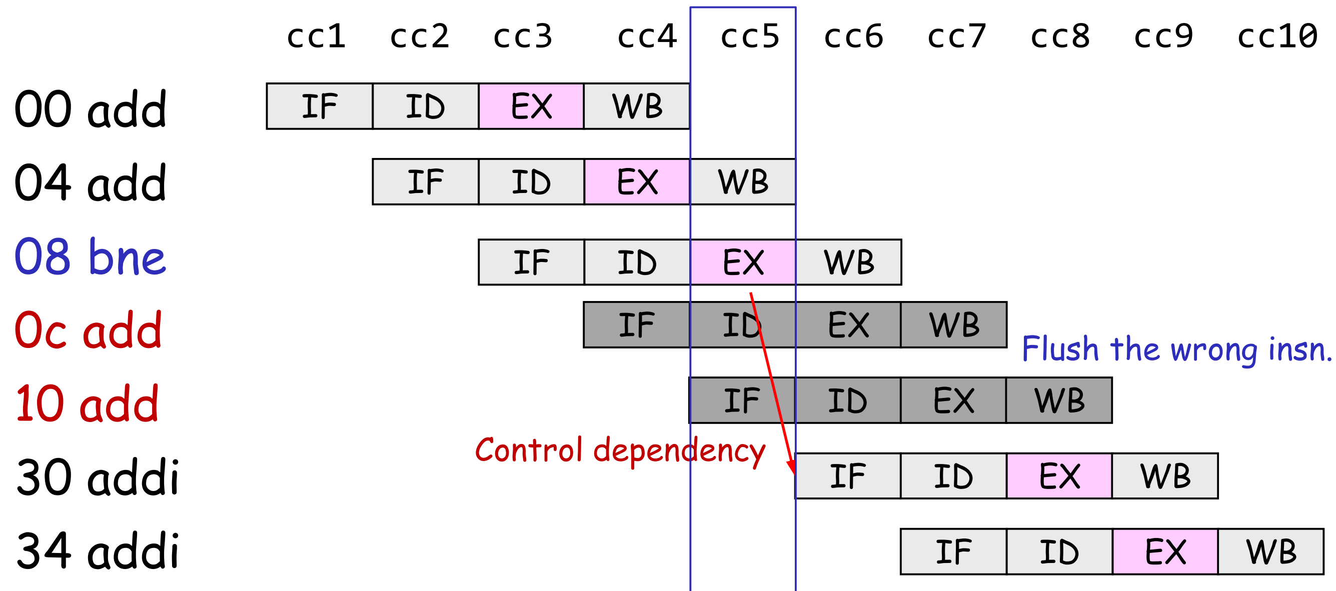




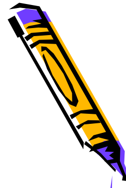

```
module m_proc8(w_clk);
  input wire w_clk;
  reg [31:0] P1_ir=32'h13, P1_pc=0, P2_pc=0, P3_pc=0;
  reg [31:0] P2_r1=0, P2_s2=0, P2_r2=0, P2_tpc=0;
  reg [31:0] P3_alu=0, P3_ldd=0;
  reg P2_r=0, P2_s=0, P2_b=0, P2_ld=0, P3_s=0, P3_b=0, P3_ld=0;
  reg [4:0] P2_rd=0, P2_rs1=0, P2_rs2=0, P3_rd=0;
  reg P1_v=0, P2_v=0, P3_v=0;
  wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
  wire [31:0] w_alu, w_ldd, w_tpc, w_pcin, w_in1, w_in2, w_in3;
  wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld, w_tkn;
  reg [31:0] r_pc = 0; // m1
  wire w_miss = P2_b & w_tkn & P2_v;
  assign w_pcin = (w_miss) ? P2_tpc : w_npc; // m0
  assign w_npc = r_pc + 4; // m2
  m_am_imem m3 (r_pc, w_ir);
  m_gen_imm m4 (P1_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
  m_RF2 m5 (w_clk, P1_ir[19:15], P1_ir[24:20], w_r1, w_r2,
            P3_rd, !P3_s & !P3_b & P3_v, w_rt);
  assign w_tpc = P1_pc + w_imm; // m6
  assign w_s2 = (!w_r & !w_b) ? w_imm : w_r2; // m7
  always @(posedge w_clk) begin
    {P1_v, P2_v, P3_v} <= {!w_miss, !w_miss & P1_v, P2_v};
    {r_pc, P1_ir, P1_pc, P2_pc} <= {w_pcin, w_ir, r_pc, P1_pc};
    {P2_r1, P2_r2, P2_s2, P2_tpc} <= {w_r1, w_r2, w_s2, w_tpc};
    {P2_r, P2_s, P2_b, P2_ld} <= {w_r, w_s, w_b, w_ld};
    {P2_rs2, P2_rs1, P2_rd} <= {P1_ir[24:15], P1_ir[11:7]};
    {P3_pc, P3_ld} <= {P2_pc, P2_ld};
    {P3_alu, P3_ldd, P3_rd} <= {w_alu, w_ldd, P2_rd};
  end
  assign w_alu = w_in1 + w_in2;   // m8
  assign w_tkn = w_in1 != w_in2;  // m8
  m_am_dmem m9 (w_clk, w_alu, P2_s & P2_v, w_in3, w_ldd);
  assign w_rt = (P3_ld) ? P3_ldd : P3_alu; // m10
  assign w_in1 = (|P3_rd & P2_rs1==P3_rd) ? w_rt : P2_r1; // m11
  assign w_in3 = (!P3_rd & P2_rs2==P3_rd) ? w_rt : P2_r2; // m13
  assign w_in2 = (|P3_rd & (P2_r|P2_b) & P2_rs2==P3_rd) ? w_rt : P2_s2;// m12
endmodule
```

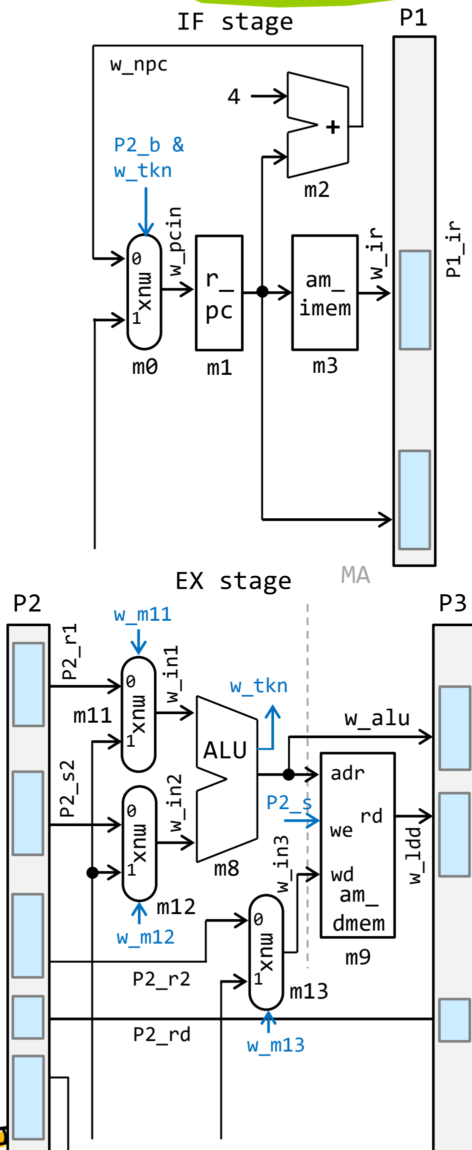
Why do branch instructions degrade IPC?

- 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.



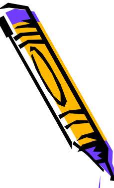
five stages pipelined processor executing instruction sequence with a branch

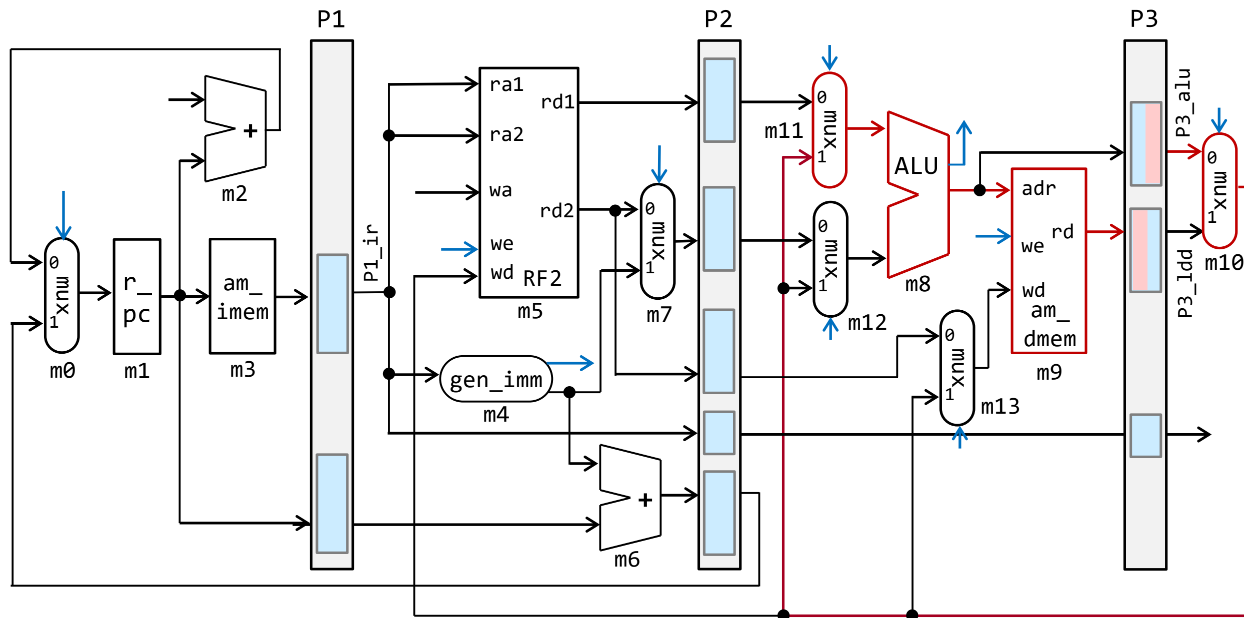





```
module m_proc8(w_clk);
  input wire w_clk;
  reg [31:0] P1_ir=32'h13, P1_pc=0, P2_pc=0, P3_pc=0;
  reg [31:0] P2_r1=0, P2_s2=0, P2_r2=0, P2_tpc=0;
  reg [31:0] P3_alu=0, P3_ldd=0;
  reg P2_r=0, P2_s=0, P2_b=0, P2_ld=0, P3_s=0, P3_b=0, P3_ld=0;
  reg [4:0] P2_rd=0, P2_rs1=0, P2_rs2=0, P3_rd=0;
  reg P1_v=0, P2_v=0, P3_v=0;
  wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
  wire [31:0] w_alu, w_ldd, w_tpc, w_pcin, w_in1, w_in2, w_in3;
  wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld, w_tkn;
  reg [31:0] r_pc = 0; // m1
  wire w_miss = P2_b & w_tkn & P2_v;
  assign w_pcin = (w_miss) ? P2_tpc : w_npc; // m0
  assign w_npc = r_pc + 4; // m2
  m_am_imem m3 (r_pc, w_ir);
  m_gen_imm m4 (P1_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
  m_RF2 m5 (w_clk, P1_ir[19:15], P1_ir[24:20], w_r1, w_r2,
            P3_rd, !P3_s & !P3_b & P3_v, w_rt);
  assign w_tpc = P1_pc + w_imm; // m6
  assign w_s2 = (!w_r & !w_b) ? w_imm : w_r2; // m7
  always @(posedge w_clk) begin
    {P1_v, P2_v, P3_v} <= {!w_miss, !w_miss & P1_v, P2_v};
    {r_pc, P1_ir, P1_pc, P2_pc} <= {w_pcin, w_ir, r_pc, P1_pc};
    {P2_r1, P2_r2, P2_s2, P2_tpc} <= {w_r1, w_r2, w_s2, w_tpc};
    {P2_r, P2_s, P2_b, P2_ld} <= {w_r, w_s, w_b, w_ld};
    {P2_rs2, P2_rs1, P2_rd} <= {P1_ir[24:15], P1_ir[11:7]};
    {P3_pc, P3_ld} <= {P2_pc, P2_ld};
    {P3_alu, P3_ldd, P3_rd} <= {w_alu, w_ldd, P2_rd};
  end
  assign w_alu = w_in1 + w_in2;   // m8
  assign w_tkn = w_in1 != w_in2;  // m8
  m_am_dmem m9 (w_clk, w_alu, P2_s & P2_v, w_in3, w_ldd);
  assign w_rt = (P3_ld) ? P3_ldd : P3_alu; // m10
  assign w_in1 = (|P3_rd & P2_rs1==P3_rd) ? w_rt : P2_r1; // m11
  assign w_in3 = (!P3_rd & P2_rs2==P3_rd) ? w_rt : P2_r2; // m13
  assign w_in2 = (|P3_rd & (P2_r|P2_b) & P2_rs2==P3_rd) ? w_rt : P2_s2;// m12
endmodule
```

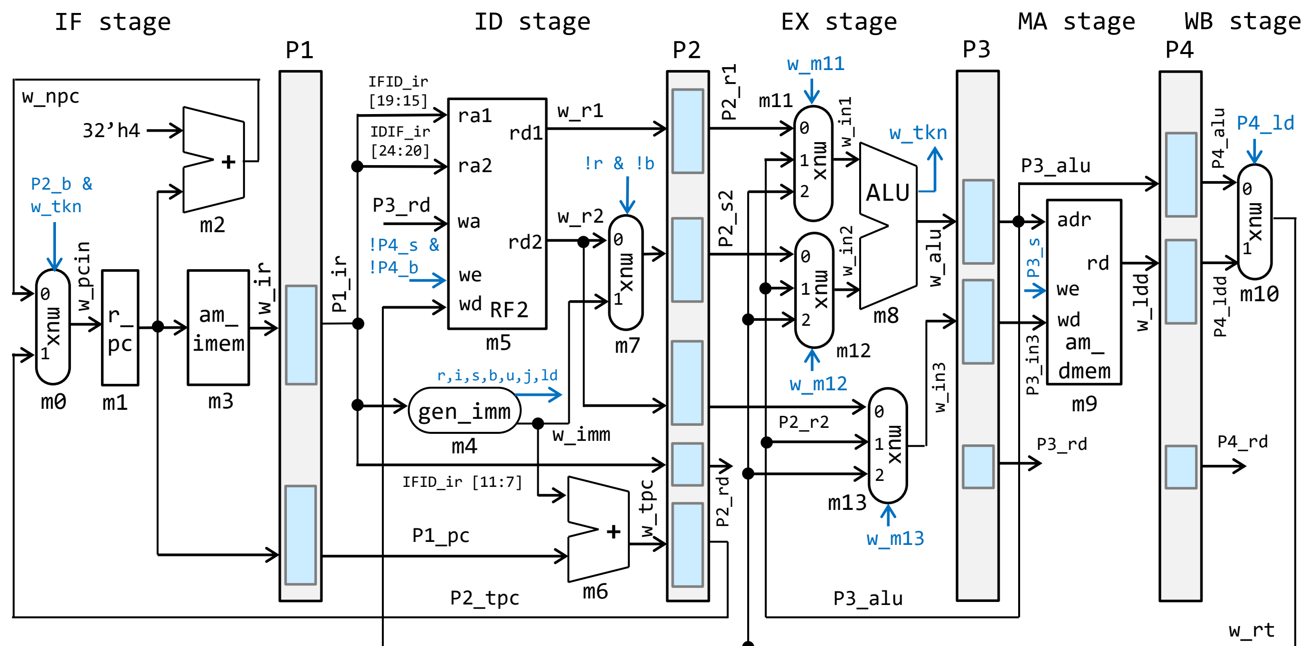
proc8: critical path of 4-stage pipelining processor



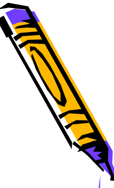


proc9: 5-stage pipelining processor

- The strategy is to separate instruction fetch step (IF), instruction decode step (ID), execution step (EX), memory access step (EX), and write back step (WB).
- Use the pipeline register P3 between EX and MA, and pipeline register P4 between EX and WB.



proc9: critical path of 5-stage pipelining processor



- The path from P4 pipeline register to PC is the critical path.
- This 5-stage organization is commonly explained in typical computer architecture textbooks.