Fiscal Year 2023

Ver. 2023-12-14a

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

Advanced Computer Architecture

3. HDL, single-cycle processor

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

(1) Machine Language - Add instruction (add)

- Instructions are 32 bits long
- Arithmetic Instruction Format (R-type):

`add x7, x8, x9`

funct7	rs2	rs1	funct3	rd	opcode	R-type

opcode 7-bits *opcode* that specifies the operation

rs1 5-bits register file address of the first source operand

rs2 5-bits register file address of the second source operand

rd 5-bits register file address of the result's destination

funct3 and **funct7** 10-bits select the type of operation (function)

(2) RISC-V Add immediate instruction (addi)

- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like x0) for constants like 1
  - have special instructions that contain constants !

```
addi  x7, x8, -2      # x7 = x8 + (-2)
```

- Machine format (I format):

imm[11:0]	rs1	funct3	rd	opcode	I-type

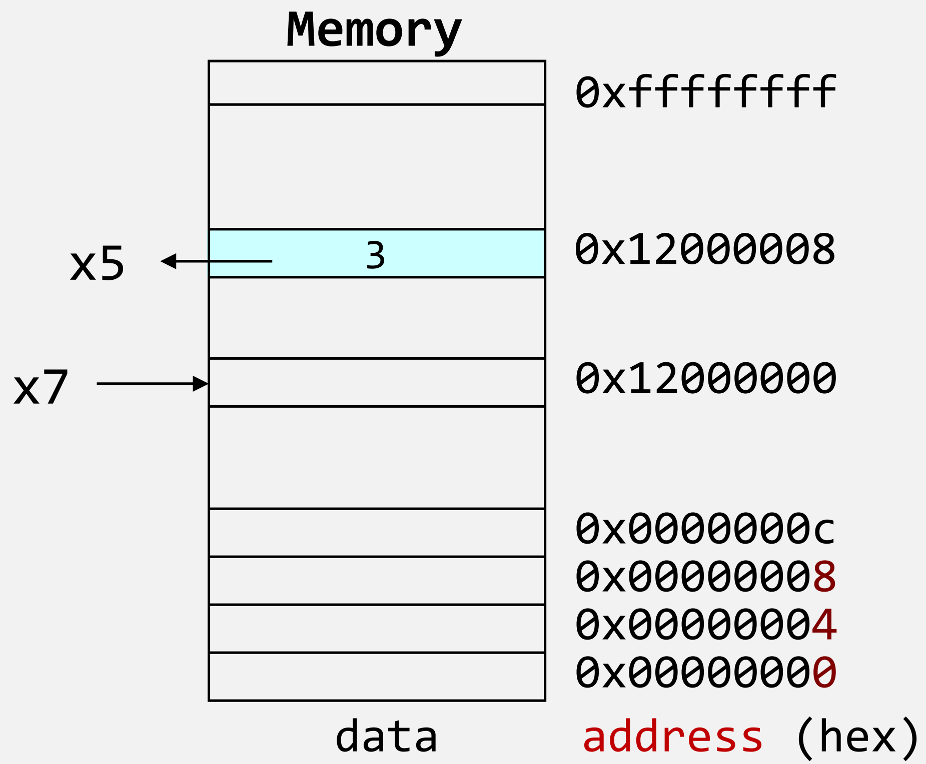
- The constant is kept inside the instruction itself
  - Immediate format limits values to the range $+2^{11}-1$ to -2^{11}

(3) Machine Language - Load word instruction (lw)

- Load Instruction Format (I-type):

```
lw x5, 8(x7)
```

imm[11:0]	rs1	funct3	rd	opcode	
					I-type

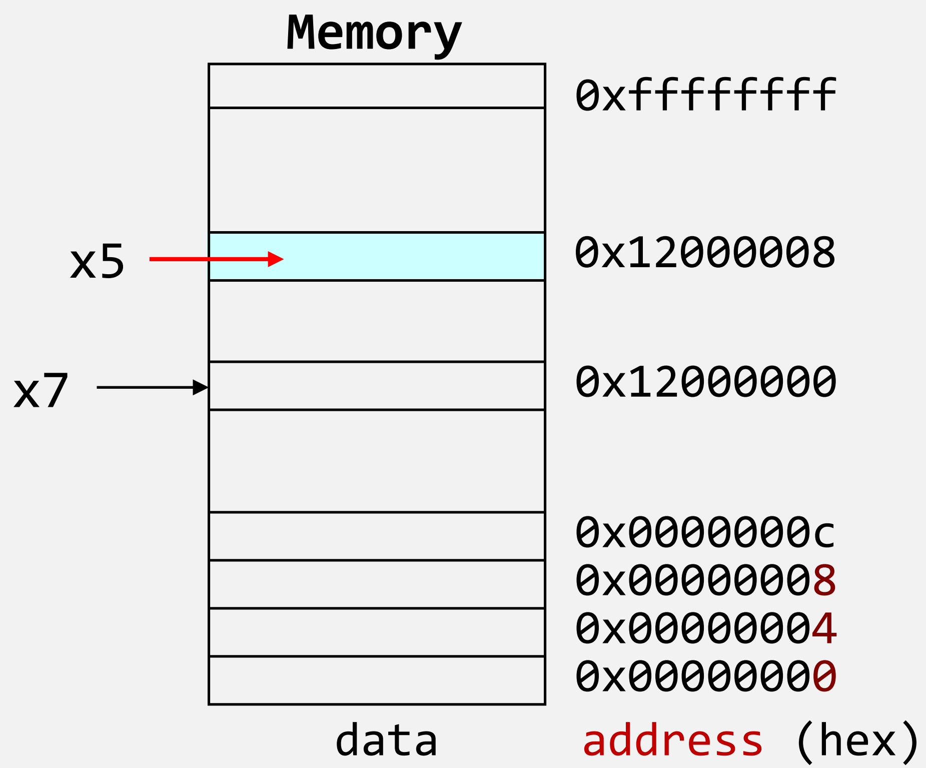


(4) Machine Language - Store word instruction (sw)

- Load Instruction Format (S-type):

`sw x5, 8(x7)`

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type



(5) RISC-V branch if not equal instructions (bne)

- RISC-V conditional branch instructions (bne, branch if not equal) :

```
bne x4, x5, Lbl   # go to Lbl if x4!=x5
```

```
Ex:    if (i==j) h = i + j;

       bne x4, x5, Lbl1    # if (i!=j) goto Lbl1
       add x6, x4, x5      # h = i + j;
Lbl1:  ...
```

- Instruction Format (B-type):

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

- How is the branch destination address specified?

Sample assembly code in RISC-V

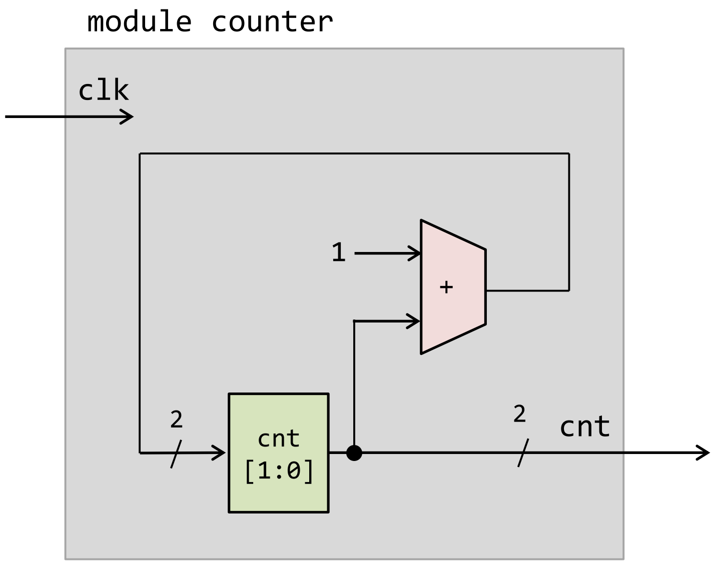
- sample assembly code in RISC-V with add, addi, lw, sw, bne instructions
- the leftmost number is the instruction memory address where the instruction is stored
- the first register x0 is zero register with hardwiring 0

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

```
0x00200293
0x00300313
0x006283B3
0x02702023
0x02002403
0x005404B3
0xFE6294E3
```

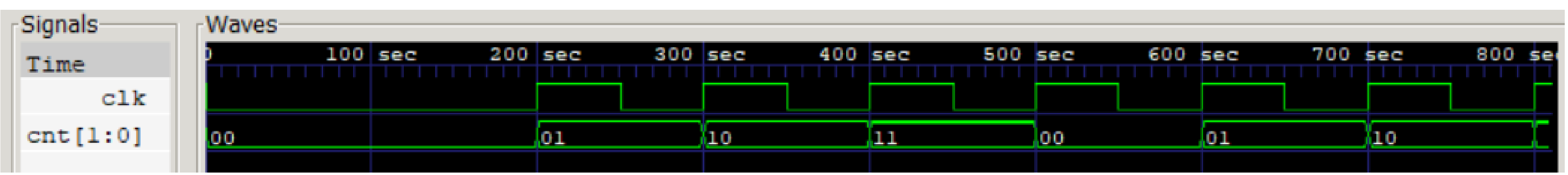
Sample circuit 3

- 2-bit counter as a simple *sequential circuit*



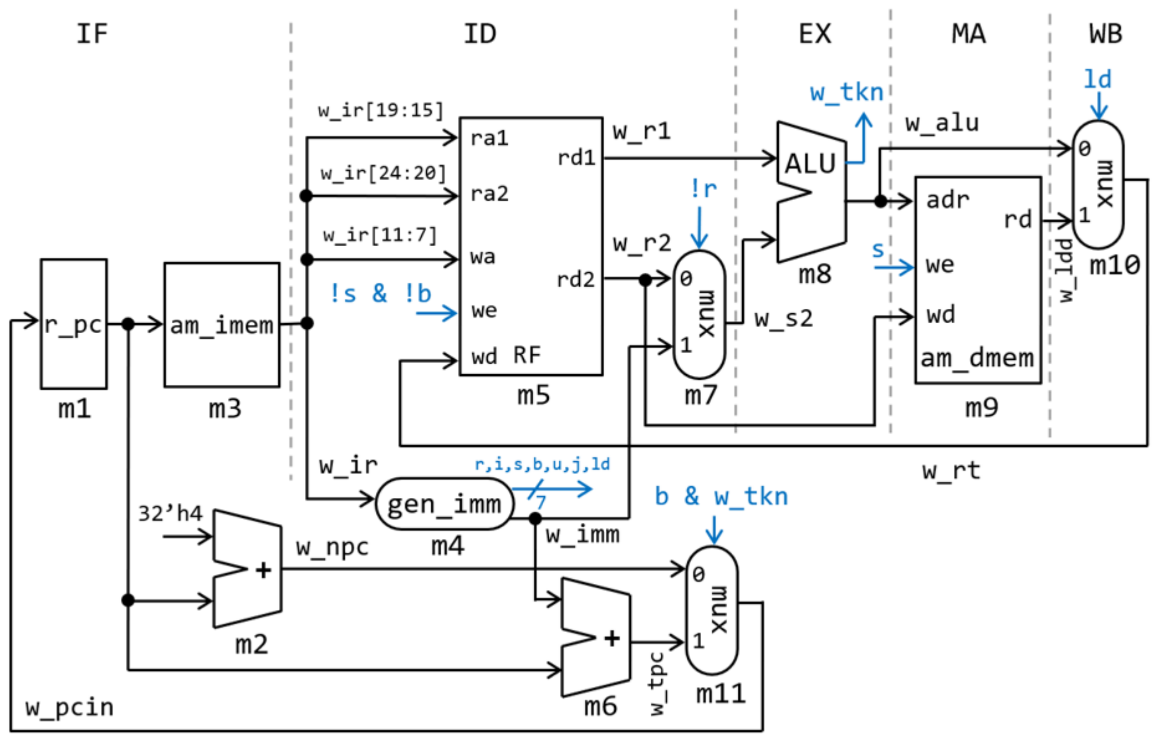

```
module top();
  reg r_clk=0;
  initial #150 forever #50 r_clk = ~r_clk;
  initial #810 $finish;
  wire [1:0] w_cnt;
  counter m1 (r_clk, w_cnt);
  initial $dumpvars(0, m1);
endmodule

module counter(clk, cnt);
  input wire clk;
  output reg [1:0] cnt;
  initial cnt = 0;
  always@(posedge clk) cnt <= cnt + 1;
endmodule
```



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.

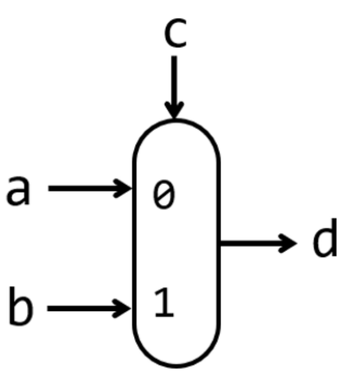


Steps in processing an instruction

- IF: Instruction Fetch
  fetch an instruction from instruction memory or instruction cache
- ID: Instruction Decode
  decode an instruction and read input operands from register file
- EX: Execution
  perform operation, calculate an address of lw/sw
- MEM: Memory Access
  access data memory or data cache for lw/sw
- WB: Write Back
  write operation result and loaded data to register file

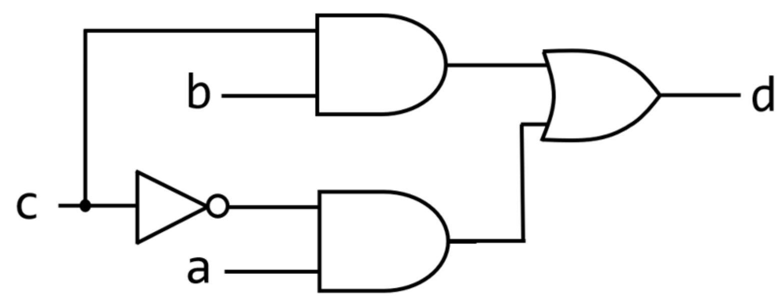
Multiplexer (MUX)

- The multiplexer, shortened to MUX, is a combinational logic circuit designed to switch one of several input lines through to a single common output line by a control signal.

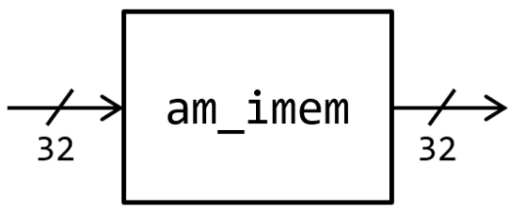


c	a	b	d
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

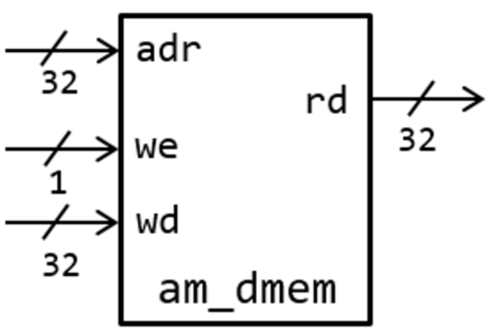
```
module m_mux(a, b, c, d);
  input  wire a, b, c;
  output wire d;;
  assign d = (c) ? b : a;   # ternary operator
endmodule
```



Instruction and data memory in Verilog HDL

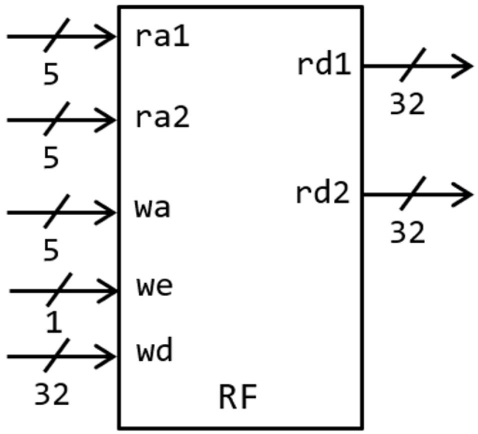



```
module m_am_imem(w_pc, w_insn);
  input  wire [31:0] w_pc;
  output wire [31:0] w_insn;
  reg [31:0] mem [0:63];
  assign w_insn = mem[w_pc[7:2]];
  integer i; initial for (i=0; i<64; i=i+1) mem[i] = 0;
endmodule
```

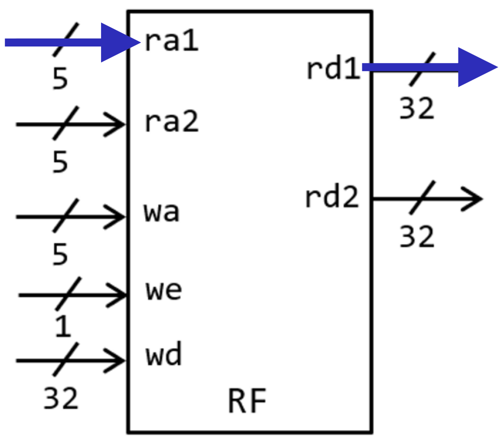



```
module m_am_dmem(w_clk, w_adr, w_we, w_wd, w_rd);
  input  wire w_clk, w_we;
  input  wire [31:0] w_adr, w_wd;
  output wire [31:0] w_rd;
  reg [31:0] mem [0:63];
  assign w_rd = mem[w_adr[7:2]];
  always @(posedge w_clk) if (w_we) mem[w_adr[7:2]] <= w_wd;
  integer i; initial for (i=0; i<64; i=i+1) mem[i] = 0;
endmodule
```

Register file (RF) in Verilog HDL (1)



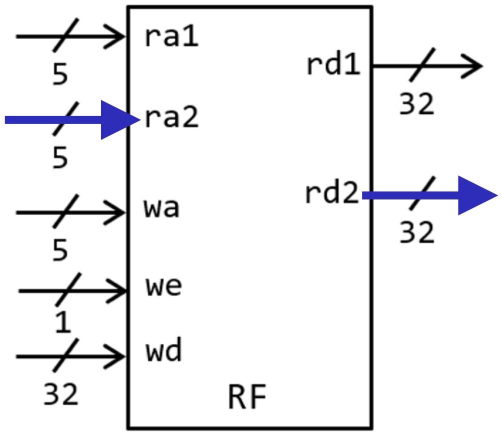

```
module m_RF(w_clk, w_ra1, w_ra2, w_rd1, w_rd2, w_wa, w_we, w_wd);
  input  wire w_clk, w_we;
  input  wire [4:0] w_ra1, w_ra2, w_wa;
  output wire [31:0] w_rd1, w_rd2;
  input  wire [31:0] w_wd;
  reg [31:0] mem [0:31];
  assign w_rd1 = (w_ra1==0) ? 0 : mem[w_ra1];
  assign w_rd2 = (w_ra2==0) ? 0 : mem[w_ra2];
  always @(posedge w_clk) if (w_we) mem[w_wa] <= w_wd;
  integer i; initial for (i=0; i<32; i=i+1) mem[i] = 0;
endmodule
```




```
module m_RF(w_clk, w_ra1, w_ra2, w_rd1, w_rd2, w_wa, w_we, w_wd);
  input  wire w_clk, w_we;
  input  wire [4:0] w_ra1, w_ra2, w_wa;
  output wire [31:0] w_rd1, w_rd2;
  input  wire [31:0] w_wd;
  reg [31:0] mem [0:31];
  assign w_rd1 = (w_ra1==0) ? 0 : mem[w_ra1];
  assign w_rd2 = (w_ra2==0) ? 0 : mem[w_ra2];
  always @(posedge w_clk) if (w_we) mem[w_wa] <= w_wd;
  integer i; initial for (i=0; i<32; i=i+1) mem[i] = 0;
endmodule
```

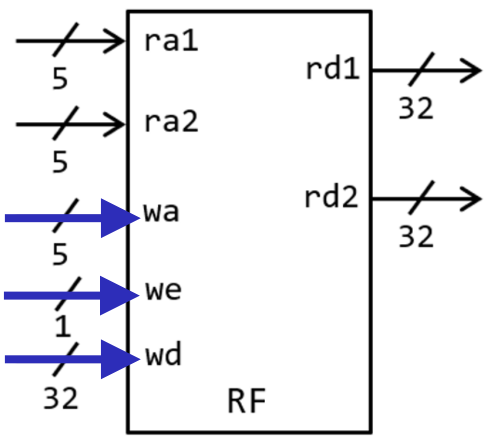

first read port

Register file (RF) in Verilog HDL (2)




```
module m_RF(w_clk, w_ra1, w_ra2, w_rd1, w_rd2, w_wa, w_we, w_wd);
  input  wire w_clk, w_we;
  input  wire [4:0] w_ra1, w_ra2, w_wa;
  output wire [31:0] w_rd1, w_rd2;
  input  wire [31:0] w_wd;
  reg [31:0] mem [0:31];
  assign w_rd1 = (w_ra1==0) ? 0 : mem[w_ra1];
  assign w_rd2 = (w_ra2==0) ? 0 : mem[w_ra2];
  always @(posedge w_clk) if (w_we) mem[w_wa] <= w_wd;
  integer i; initial for (i=0; i<32; i=i+1) mem[i] = 0;
endmodule
```

second read port

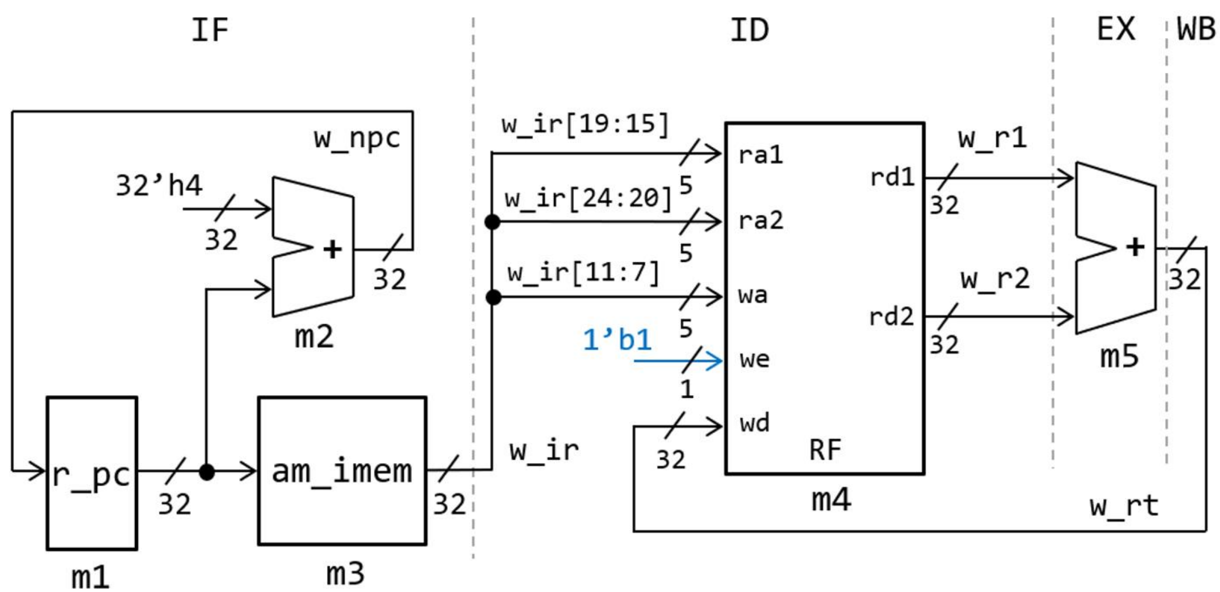



```
module m_RF(w_clk, w_ra1, w_ra2, w_rd1, w_rd2, w_wa, w_we, w_wd);
  input  wire w_clk, w_we;
  input  wire [4:0] w_ra1, w_ra2, w_wa;
  output wire [31:0] w_rd1, w_rd2;
  input  wire [31:0] w_wd;
  reg [31:0] mem [0:31];
  assign w_rd1 = (w_ra1==0) ? 0 : mem[w_ra1];
  assign w_rd2 = (w_ra2==0) ? 0 : mem[w_ra2];
  always @(posedge w_clk) if (w_we) mem[w_wa] <= w_wd;
  integer i; initial for (i=0; i<32; i=i+1) mem[i] = 0;
endmodule
```

write port

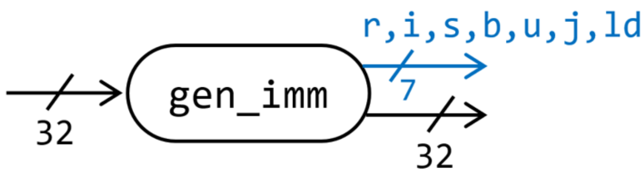
proc02: single cycle proc. supporting add

```
module m_proc2(w_clk);
  input wire w_clk;
  wire [31:0] w_npc, w_ir, w_r1, w_r2, w_rt;
  reg [31:0] r_pc = 0;                          // m1
  assign w_npc = r_pc + 4;                      // m2
  m_am_imem m3 (r_pc, w_ir);
  m_RF m4 (w_clk, w_ir[19:15], w_ir[24:20], w_r1, w_r2, w_ir[11:7], 1, w_rt);
  assign w_rt = w_r1 + w_r2;                    // m5
  always @(posedge w_clk) r_pc <= w_npc;        // m1
endmodule
```



funct7	rs2	rs1	funct3	rd	opcode	
31 30 25	24 21 20	19 15	14 12	11 8 7	6 0	R-type

immediate generation module (gen_imm)



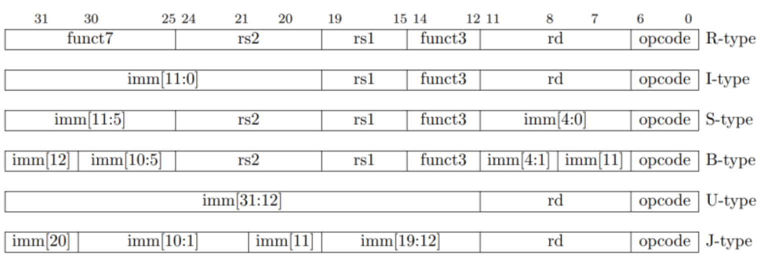


Figure 2.3: RISC-V base instruction formats showing immediate variants.

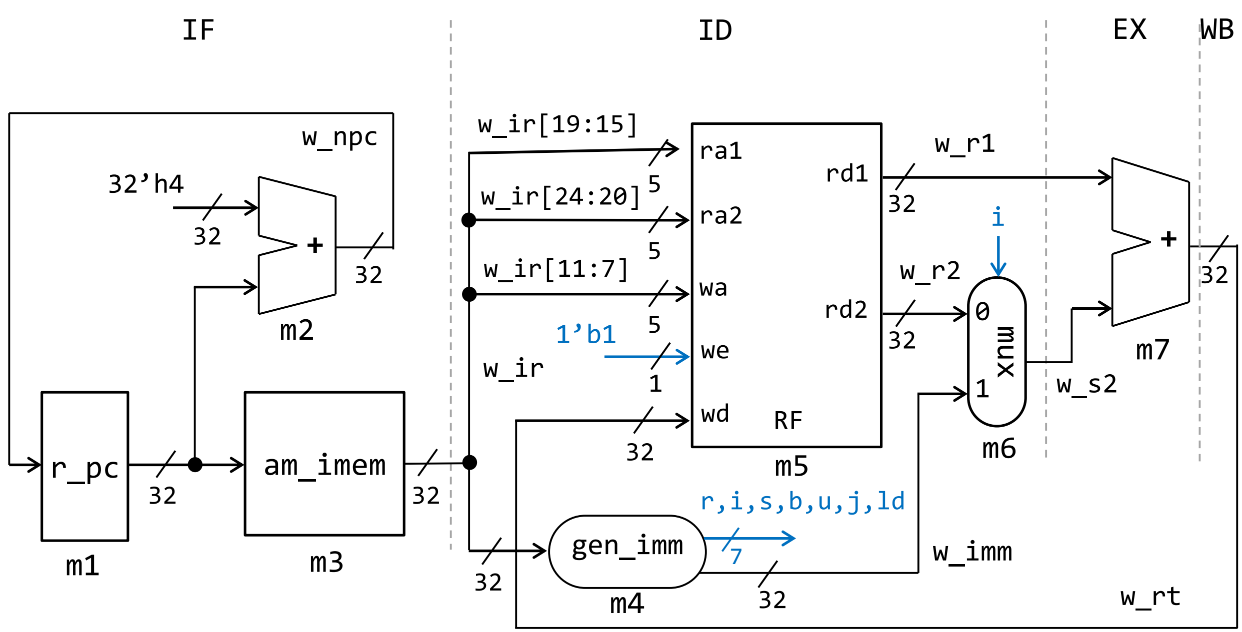
```
module m_get_type(opcode5, r, i, s, b, u, j);
  input  wire [4:0] opcode5;
  output wire r, i, s, b, u, j;
  assign j = (opcode5==5'b11011);
  assign b = (opcode5==5'b11000);
  assign s = (opcode5==5'b01000);
  assign r = (opcode5==5'b01100);
  assign u = (opcode5==5'b01101 || opcode5==5'b00101);
  assign i = ~(j | b | s | r | u);
endmodule
```

```
module m_get_imm(ir, i, s, b, u, j, imm);
  input wire [31:0] ir;
  input wire i, s, b, u, j;
  output wire [31:0] imm;
  assign imm= (i) ? {{20{ir[31]}},ir[31:20]} :
              (s) ? {{20{ir[31]}},ir[31:25],ir[11:7]} :
              (b) ? {{20{ir[31]}},ir[7],ir[30:25],ir[11:8],1'b0} :
              (u) ? {ir[31:12],12'b0} :
              (j) ? {{12{ir[31]}},ir[19:12],ir[20],ir[30:21],1'b0} : 0;
endmodule
```

```
module m_gen_imm(w_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
  input  wire [31:0] w_ir;
  output wire [31:0] w_imm;
  output wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld;
  m_get_type m1 (w_ir[6:2], w_r, w_i, w_s, w_b, w_u, w_j);
  m_get_imm m2 (w_ir, w_i, w_s, w_b, w_u, w_j, w_imm);
  assing w_ld = (w_ir[6:2]==0);
endmodule
```

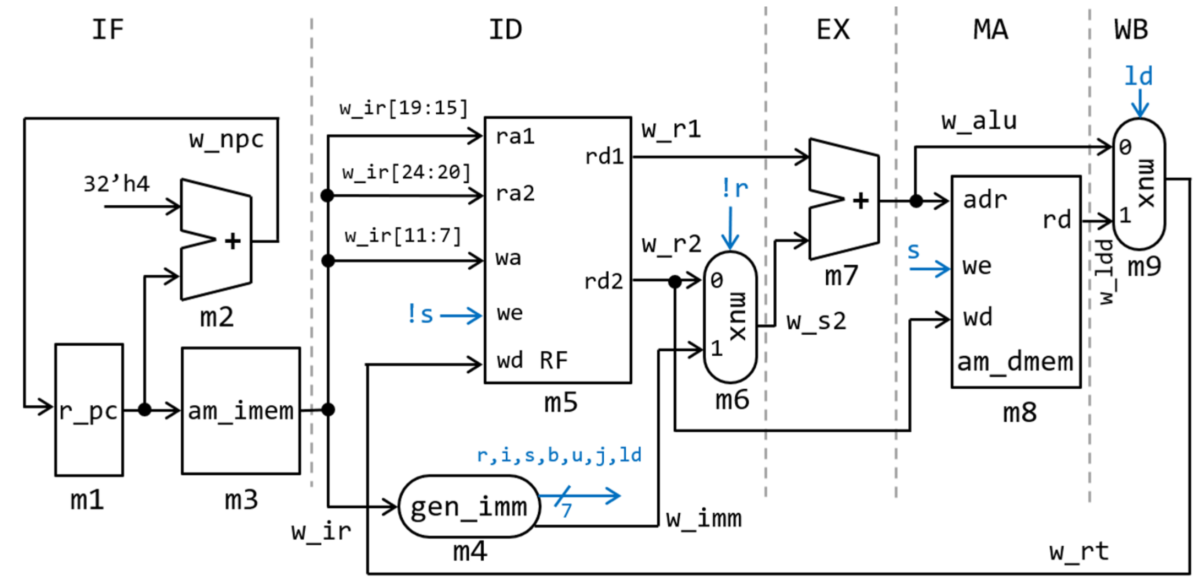
proc03: single cycle proc. supporting add, addi

```
module m_proc3(w_clk);
  input wire w_clk;
  wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
  wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld;
  reg [31:0] r_pc = 0;                          // m1
  assign w_npc = r_pc + 4;                      // m2
  m_am_imem m3 (r_pc, w_ir);
  m_gen_imm m4 (w_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
  m_RF m5 (w_clk, w_ir[19:15], w_ir[24:20], w_r1, w_r2, w_ir[11:7], 1, w_rt);
  assign w_s2 = (w_i) ? w_imm : w_r2;           // m6
  assing w_rt = w_r1 + w_s2;                    // m7
  always @(posedge w_clk) r_pc <= w_npc;        // m1
endmodule
```



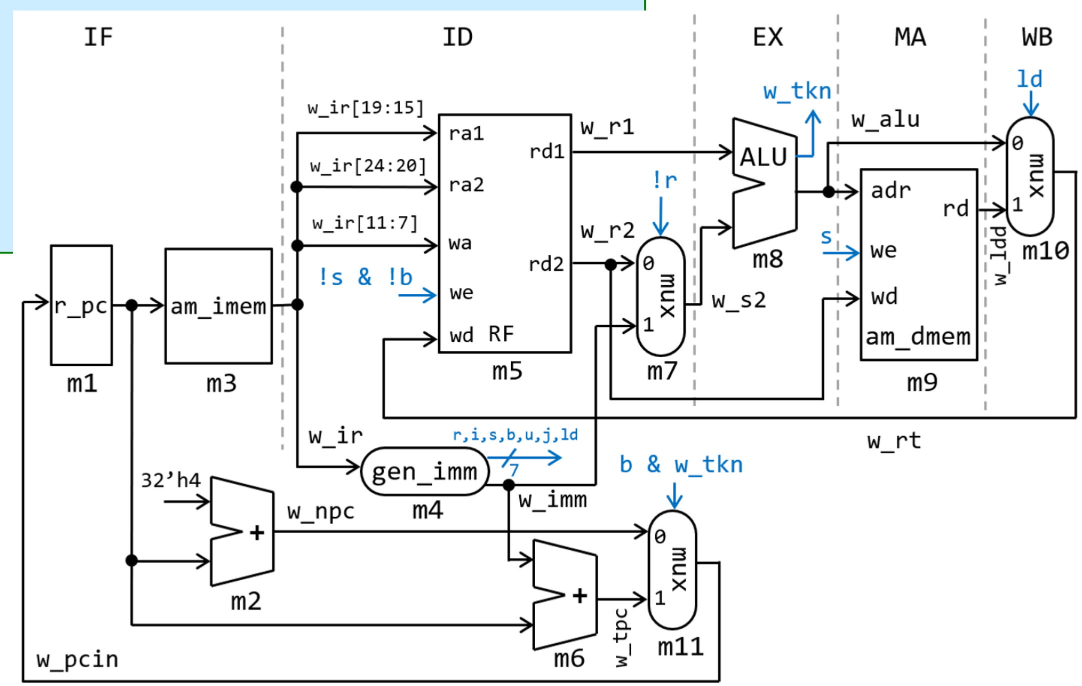
proc04: single cycle proc. supporting add, addi, lw, sw

```
module m_proc4(w_clk);
  input wire w_clk;
  wire [31:0] w_npc, w_ir, w_imm, w_r1, w_r2, w_s2, w_rt;
  wire [31:0] w_alu, w_ldd;
  reg [31:0] r_pc = 0;
  wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld;
  assign w_npc = r_pc + 4;
  m_am_imem m3 (r_pc, w_ir);
  m_gen_imm m4 (w_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
  m_RF m5 (w_clk, w_ir[19:15], w_ir[24:20], w_r1, w_r2, w_ir[11:7], !w_s, w_rt);
  assign w_s2 = (!w_r) ? w_imm : w_r2;
  assign w_alu = w_r1 + w_s2;
  m_am_dmem m8 (w_clk, w_alu, w_s, w_r2, w_ldd);
  assign w_rt = (w_ld) ? w_ldd : w_alu;
  always @(posedge w_clk) r_pc <= w_npc;
endmodule
```



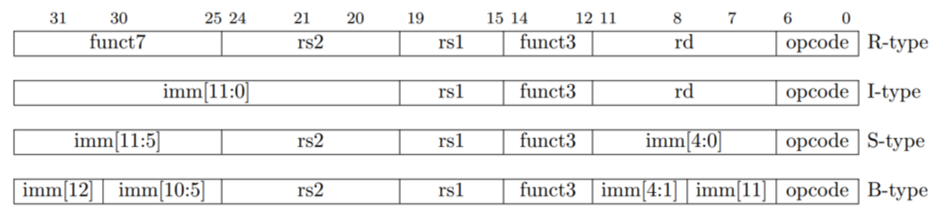
proc05: single cycle proc. supporting add, addi, lw, sw, bne

```
module m_proc5(w_clk);
  input wire w_clk;
  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;
  wire w_tkn;
  reg [31:0] r_pc=0;
  assign w_pcin = (w_b & w_tkn) ? w_tpc : w_npc;     # m11
  assign w_npc = r_pc + 4;
  m_am_imem m3 (r_pc, w_ir);
  wire w_r, w_i, w_s, w_b, w_u, w_j, w_ld;
  m_gen_imm m4 (w_ir, w_imm, w_r, w_i, w_s, w_b, w_u, w_j, w_ld);
  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);
  assing w_tpc = r_pc + w_imm;    # m6
  assign w_s2 = (!w_r & !w_b) ? w_imm : w_r2;
  assign w_alu = w_r1 + w_s2;     # m8
  assign w_tkn = w_r1 != w_s2;    # m8
  m_am_dmem m9 (w_clk, w_alu, w_s, w_r2, w_ldd);
  assign w_rt = (w_ld) ? w_ldd : w_alu;
  always @(posedge w_clk) r_pc <= w_pcin;
endmodule
```

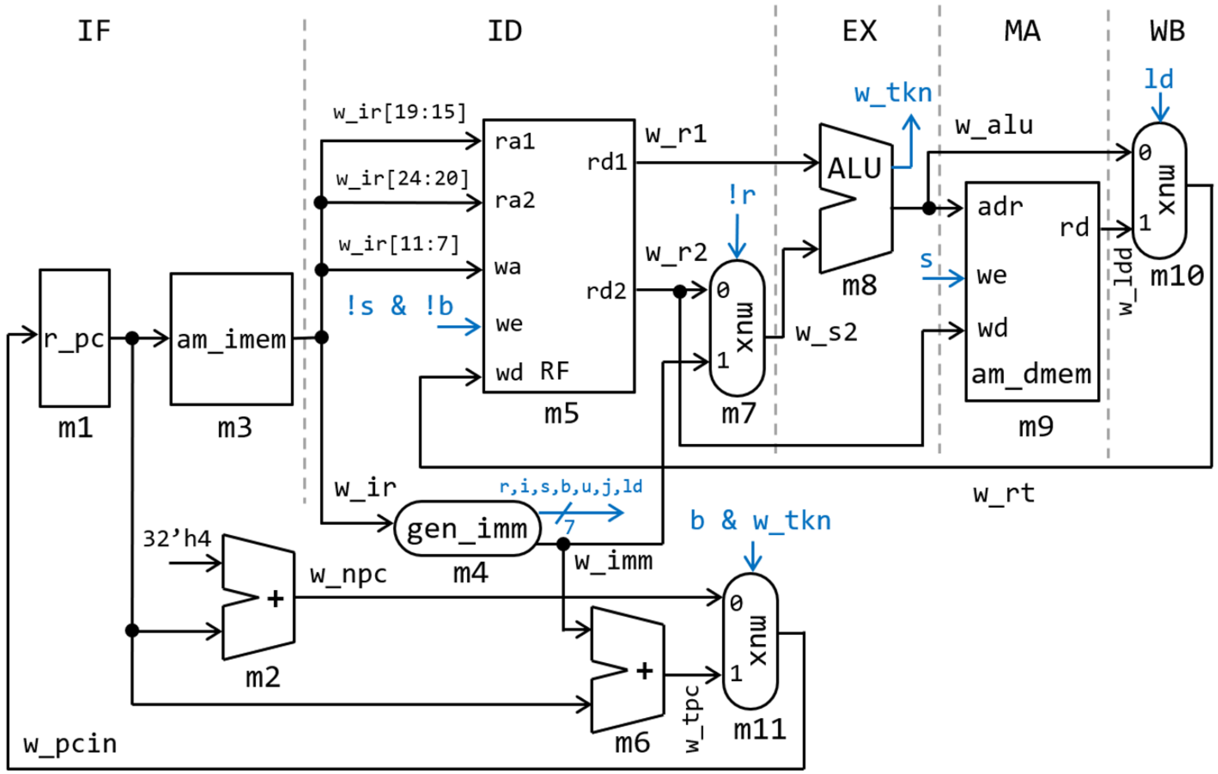


proc05: single cycle proc. supporting add, addi, lw, sw, bne

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

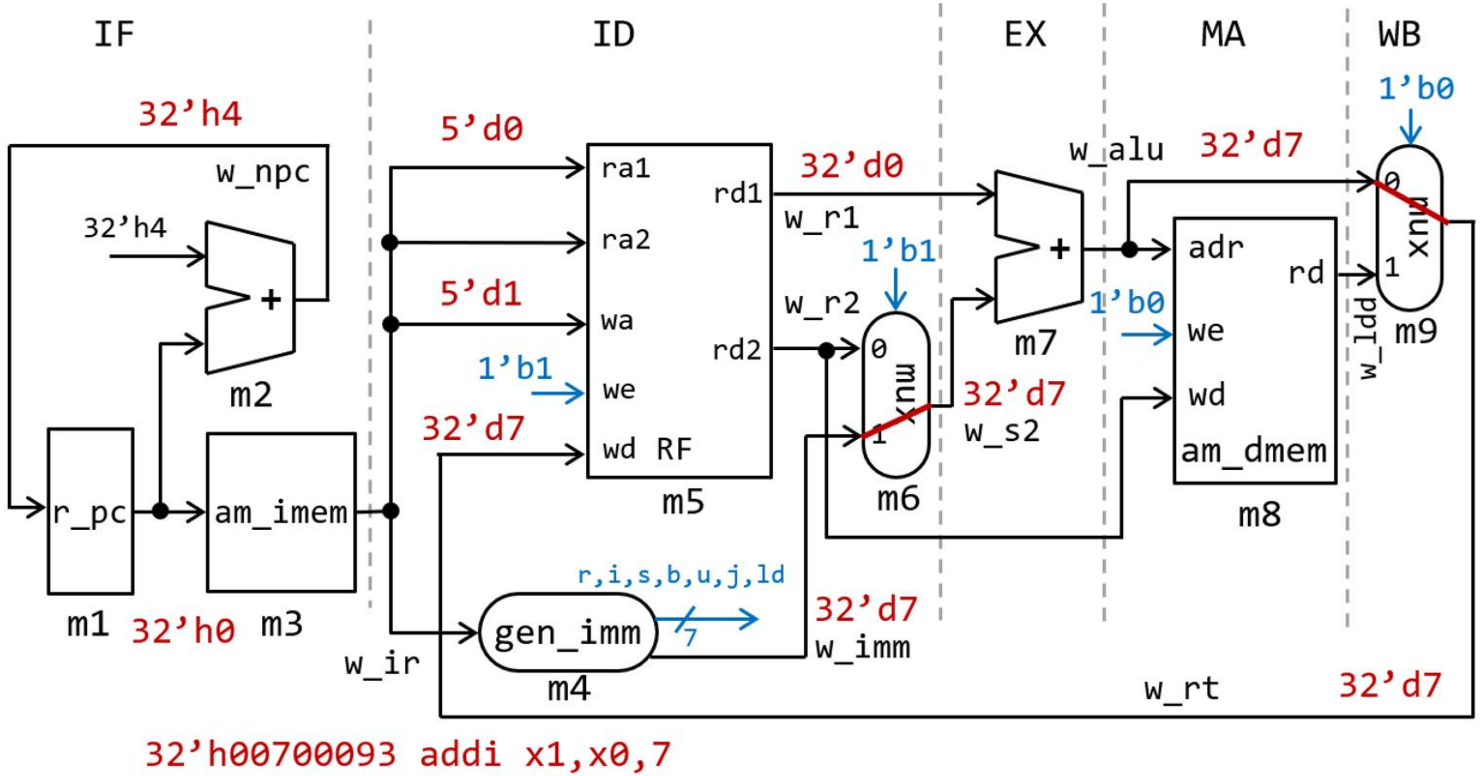


R-type for add, I-type for addi and lw, S-type for sw, and B-type for bne



Processing behabior of proc4

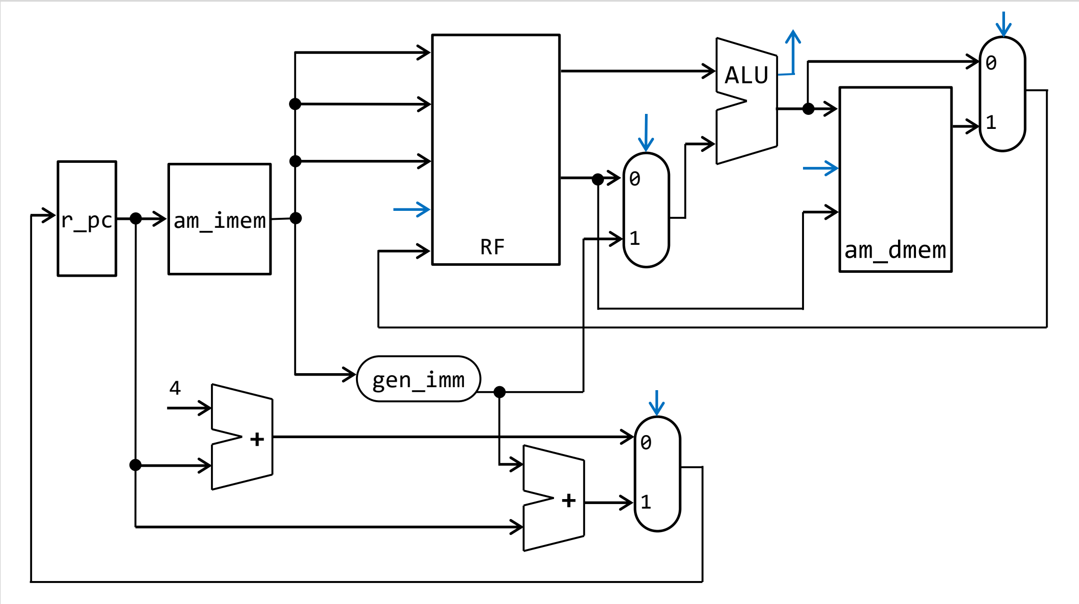
- executing `addi x1, x0, 7` of address 0x00



Exercise 1

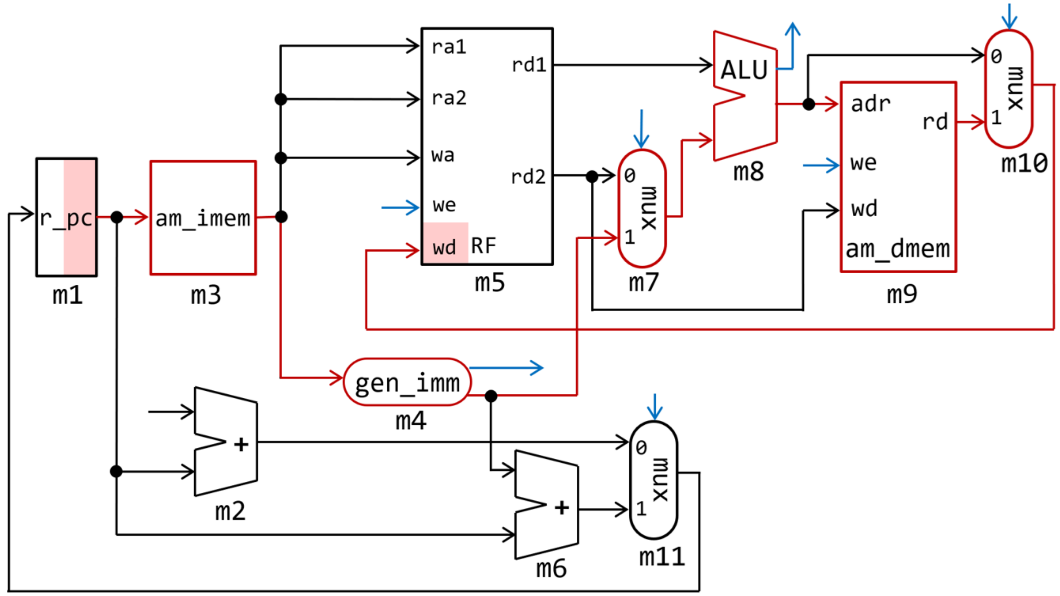
- Draw a block diagram of the processor proc05 and write the valid values on wires when the processor is executing the lw instruction

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

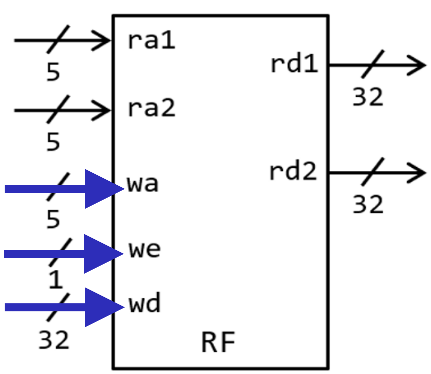


Critical path of proc5

- It is too slow to be practical.



write port




```
module m_RF(w_clk, w_ra1, w_ra2, w_rd1, w_rd2, w_wa, w_we, w_wd);
  input  wire w_clk, w_we;
  input  wire [4:0] w_ra1, w_ra2, w_wa;
  output wire [31:0] w_rd1, w_rd2;
  input  wire [31:0] w_wd;
  reg [31:0] mem [0:31];
  assign w_rd1 = (w_ra1==0) ? 0 : mem[w_ra1];
  assign w_rd2 = (w_ra2==0) ? 0 : mem[w_ra2];
  always @(posedge w_clk) if (w_we) mem[w_wa] <= w_wd;
  integer i; initial for (i=0; i<32; i=i+1) mem[i] = 0;
endmodule
```

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