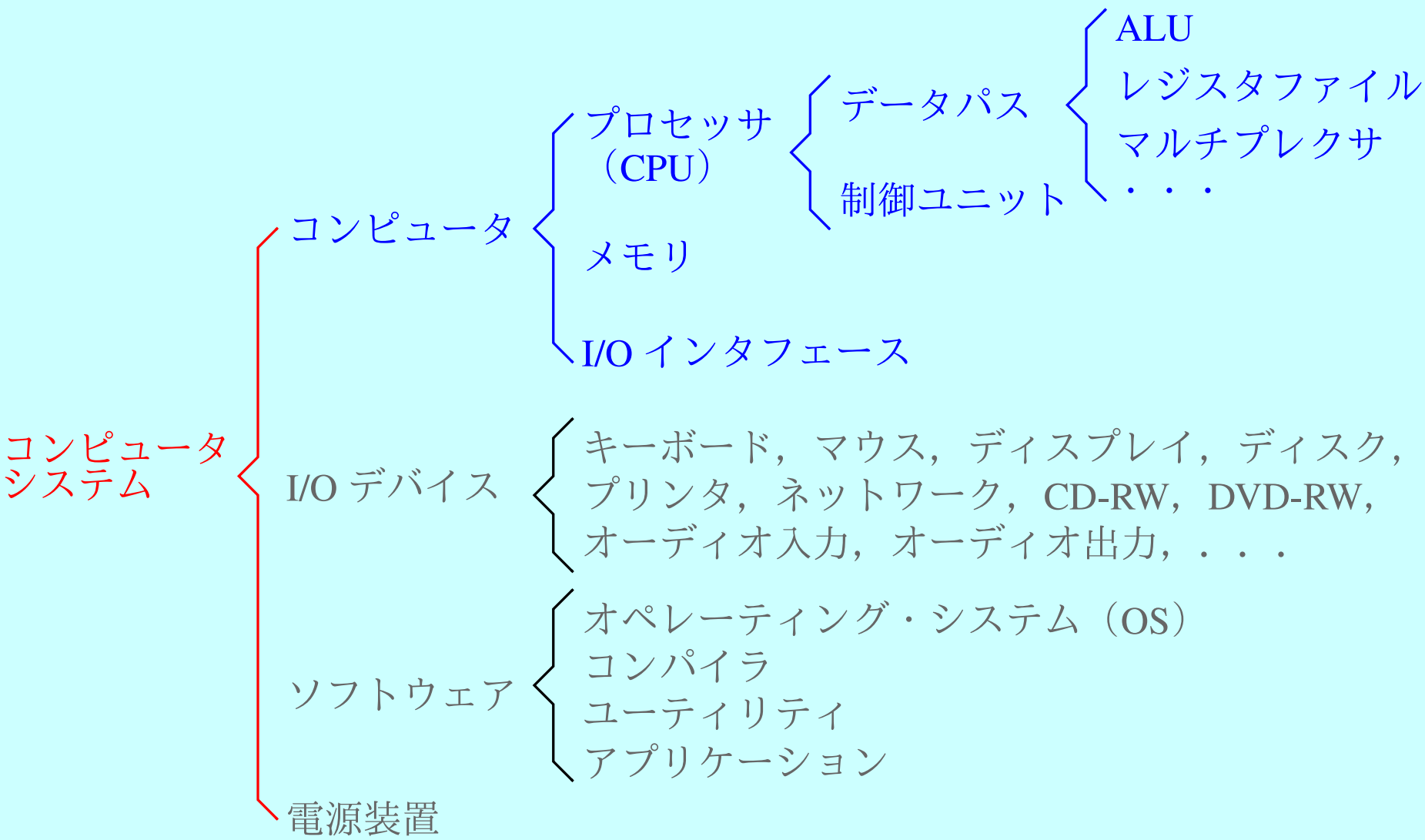
コンピュータ構成と設計（９）

パイプライン CPU 設計

李　亜民

2024 年 11 月 28 日 (木)

コンピュータとコンピュータシステム



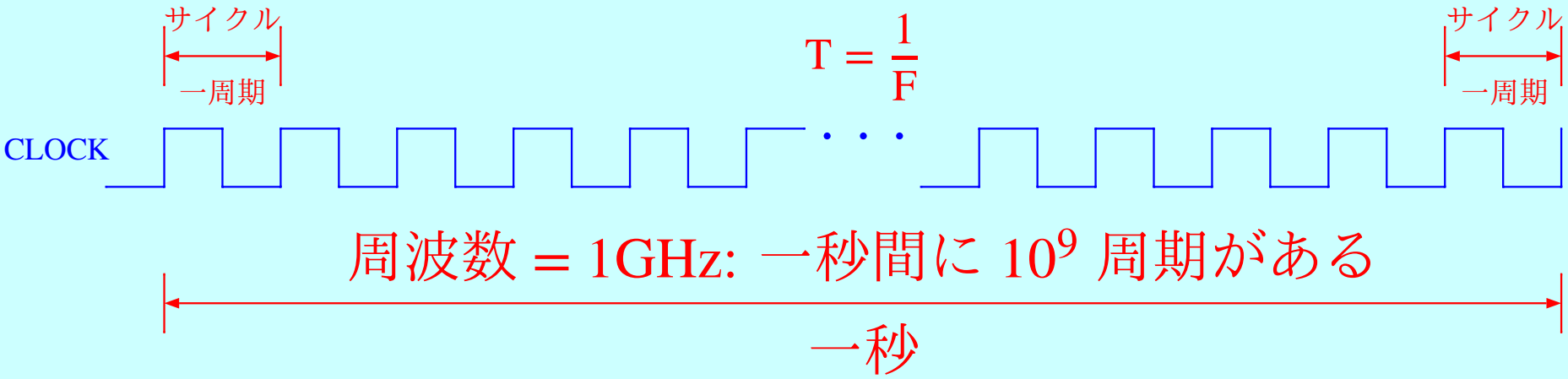
20 RISC-V 命令のまとめ

```
 1. add  rd, rs1, rs2    # rd <- rs1 + rs2
 2. sub  rd, rs1, rs2    # rd <- rs1 - rs2
 3. slt  rd, rs1, rs2    # rd <- rs1 < rs2 (signed)
 4. xor  rd, rs1, rs2    # rd <- rs1 ^ rs2
 5. or   rd, rs1, rs2    # rd <- rs1 | rs2
 6. and  rd, rs1, rs2    # rd <- rs1 & rs2
 7. slli rd, rs1, shamt  # rd <- rs1 << shamt
 8. srli rd, rs1, shamt  # rd <- rs1 >> shamt
 9. srai rd, rs1, shamt  # rd <- rs1 >>>shamt
10. jalr rd, rs1, imm    # rd <- pc+4; pc <- rs1+imm
11. addi rd, rs1, imm    # rd <- rs1 + imm
12. xori rd, rs1, imm    # rd <- rs1 ^ imm
13. ori  rd, rs1, imm    # rd <- rs1 | imm
14. andi rd, rs1, imm    # rd <- rs1 & imm
15. lw   rd,  imm(rs1)   # rd <- memory[rs1+imm]
16. sw   rs2, imm(rs1)   # memory[rs1+imm] <- rs2
17. beq  rs1, rs2, label # if (rs1==rs2) pc <- label
18. bne  rs1, rs2, label # if (rs1!=rs2) pc <- label
19. jal  rd,  label      # rd <- pc+4;   pc <- label
20. lui  rd,  imm        # rd <- imm,000000000000
```

RV32I Base Instruction Set Encoding

31 25	24 20	19 15	14 12	11 7	6 0		
0000000	rs2	rs1	000	rd	0110011	1.	add
0100000	rs2	rs1	000	rd	0110011	2.	sub
0000000	rs2	rs1	010	rd	0110011	3.	slt
0000000	rs2	rs1	100	rd	0110011	4.	xor
0000000	rs2	rs1	110	rd	0110011	5.	or
0000000	rs2	rs1	111	rd	0110011	6.	and
0000000	shamt	rs1	001	rd	0010011	7.	slli
0000000	shamt	rs1	101	rd	0010011	8.	srli
0100000	shamt	rs1	101	rd	0010011	9.	srai
imm[11:0]		rs1	000	rd	1100111	10.	jalr
imm[11:0]		rs1	000	rd	0010011	11.	addi
imm[11:0]		rs1	100	rd	0010011	12.	xori
imm[11:0]		rs1	110	rd	0010011	13.	ori
imm[11:0]		rs1	111	rd	0010011	14.	andi
imm[11:0]		rs1	010	rd	0000011	15.	lw
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	16.	sw
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	17.	beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	18.	bne
imm[20|10:1|11|19:12]				rd	1101111	19.	jal
imm[31:12]				rd	0110111	20.	lui

周波数と一サイクルの時間の関係



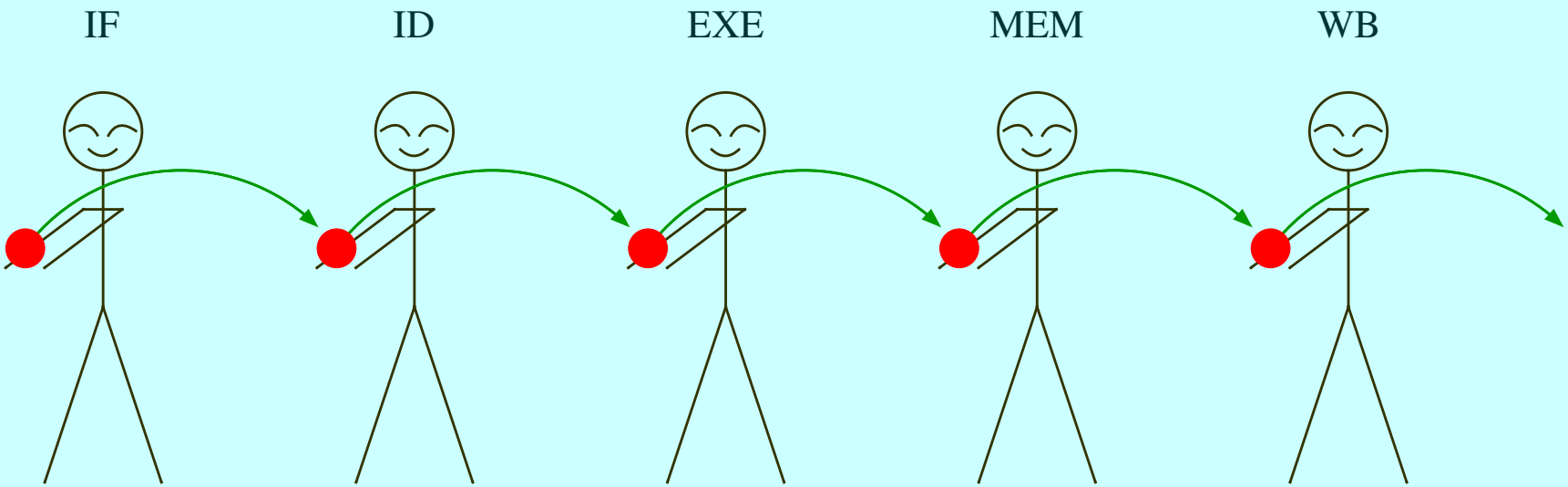
周波数 F = 1GHz: 一周期の時間 $T = \frac{1}{1 \times 10^9} = 10^{-9}\mathrm{s} = 1\mathrm{ns}$

周波数 F = 1MHz: 一周期の時間 $T = \frac{1}{1 \times 10^6} = 10^{-6}\mathrm{s} = 1\mu\mathrm{s}$

周波数 F = 1kHz: 一周期の時間 $T = \frac{1}{1 \times 10^3} = 10^{-3}\mathrm{s} = 1\mathrm{ms}$

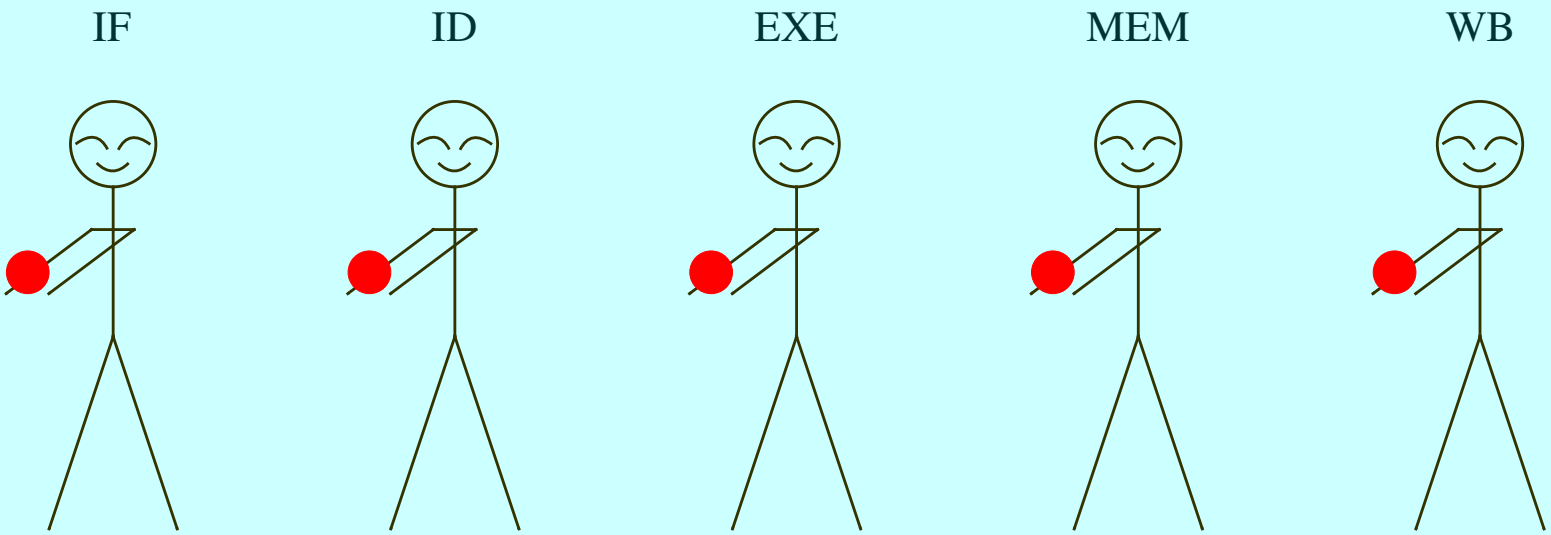
周波数 F = 1Hz: 一周期の時間 $T = \frac{1}{1 \times 10^0} = 10^{-0}\mathrm{s} = 1\mathrm{s}$ (秒)

パイプライン方式



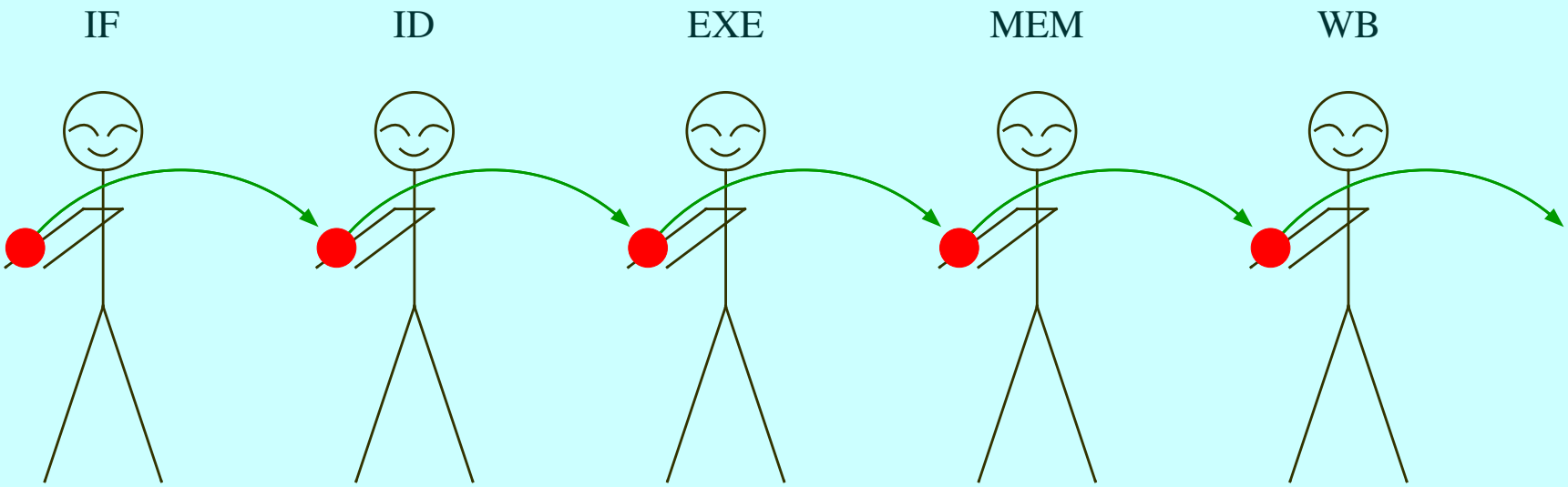
せーのっ !!!

パイプライン方式



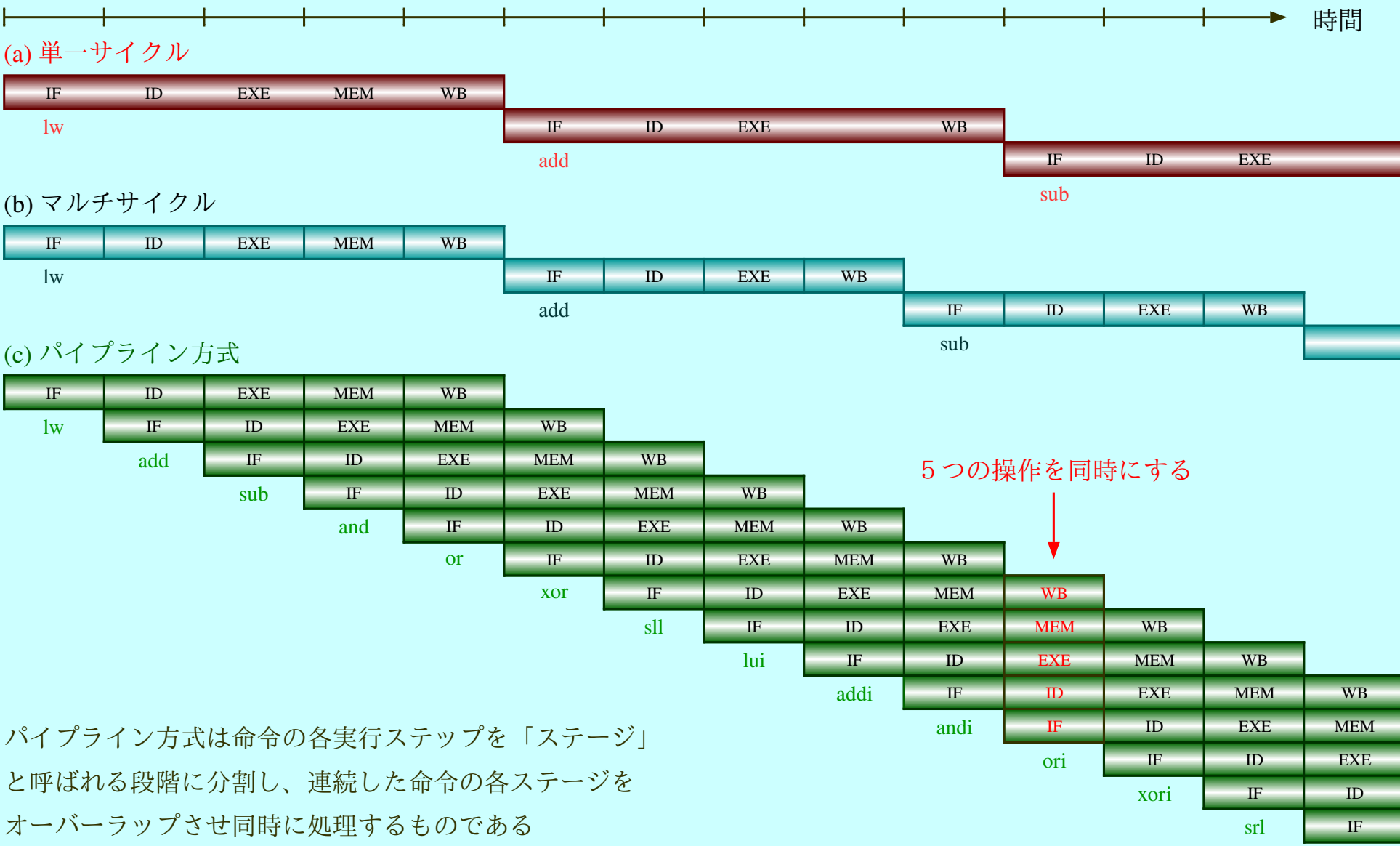
Display this PDF with Adobe Acrobat Reader DC

パイプライン方式



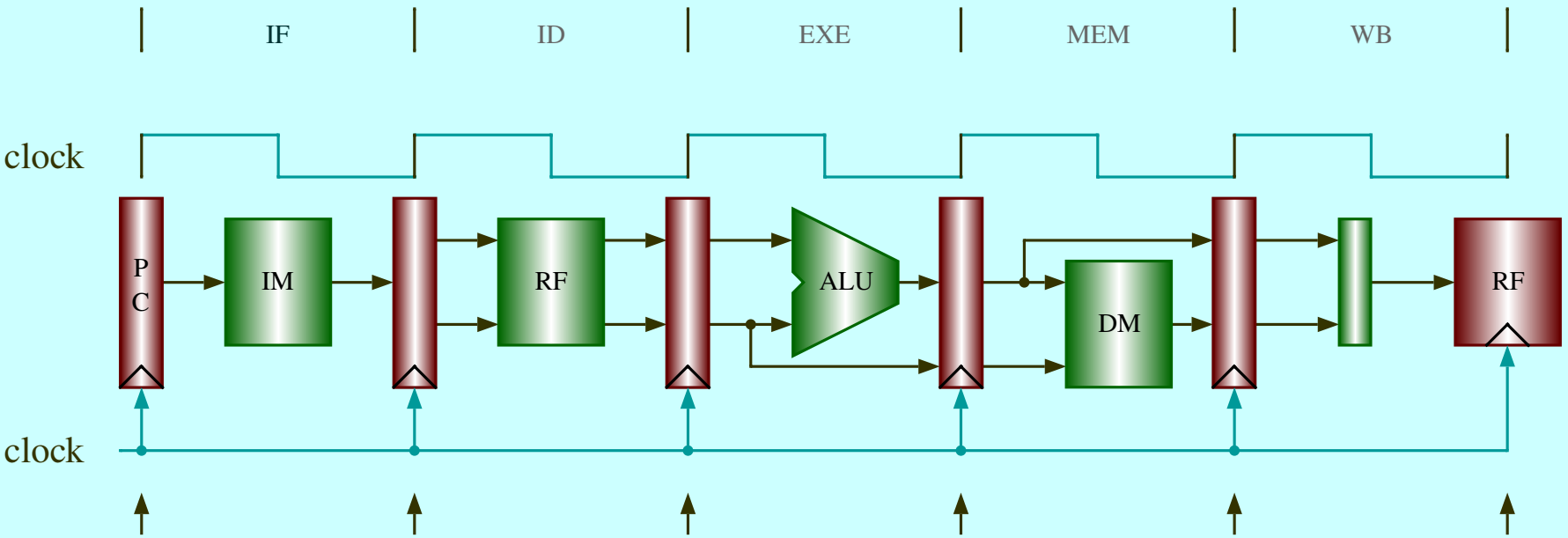
Display this PDF with Adobe Acrobat Reader DC

パイプラインの概念



パイプライン方式は命令の各実行ステップを「ステージ」と呼ばれる段階に分割し、連続した命令の各ステージをオーバーラップさせ同時に処理するものである

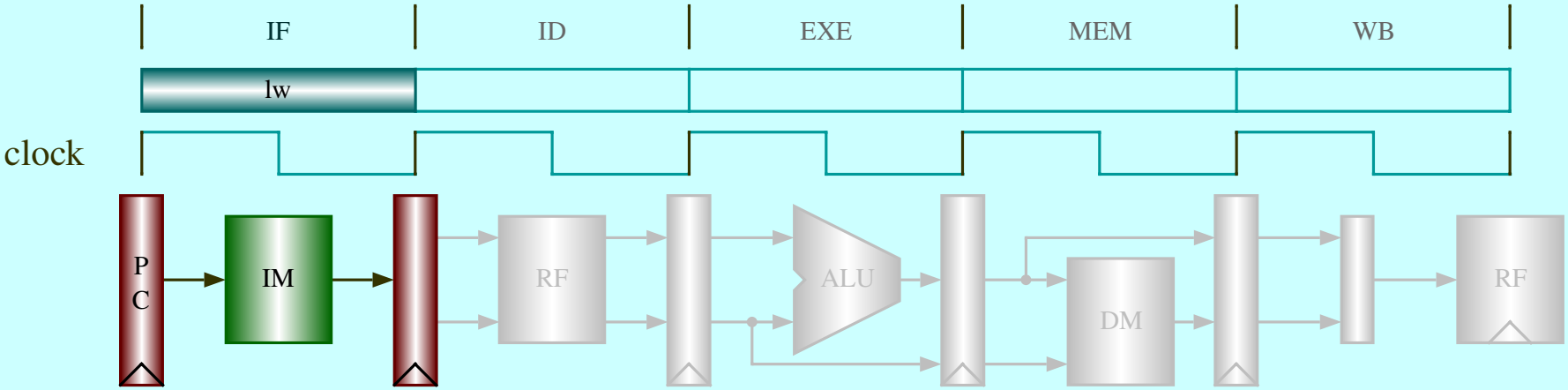
簡略化したパイプライン CPU 回路



クロックの立ち上がりで
データを
パイプラインレジスタに
保存する

注意：図の中の２つのRFは同一のレジスタファイルである

パイプライン方式で命令を実行

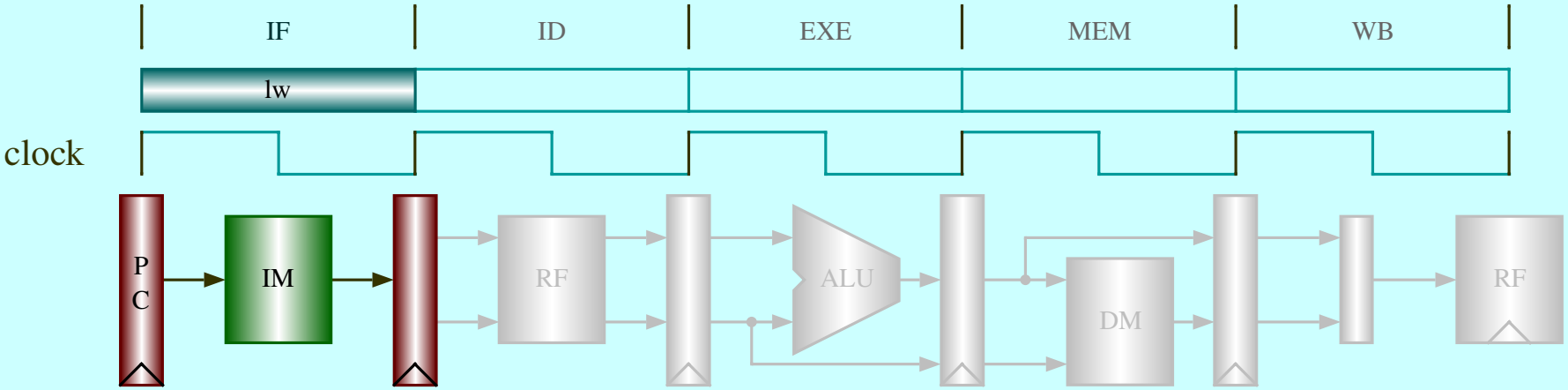


2012年 PC → lw → IF　一年生　2012年新しい学部ができた

2013年 add
2014年 sub
2015年 and
2016年 or
2017年 xor
2018年 slli
2019年 lui
2020年 addi
2021年 andi
2022年 ori
2023年 xori
2024年 srli

Display this PDF with
Adobe Acrobat Reader DC

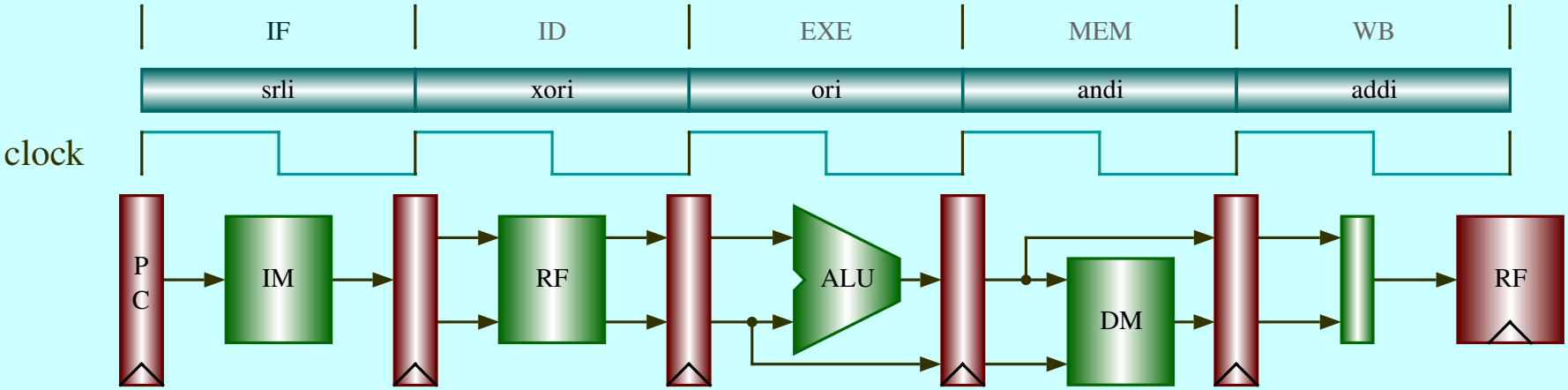
パイプライン方式で命令を実行



2012年 PC → lw → IF 一年生 2012年新しい学部ができた

2013年 add
2014年 sub
2015年 and
2016年 or
2017年 xor
2018年 slli
2019年 lui
2020年 addi
2021年 andi
2022年 ori
2023年 xori
2024年 srli

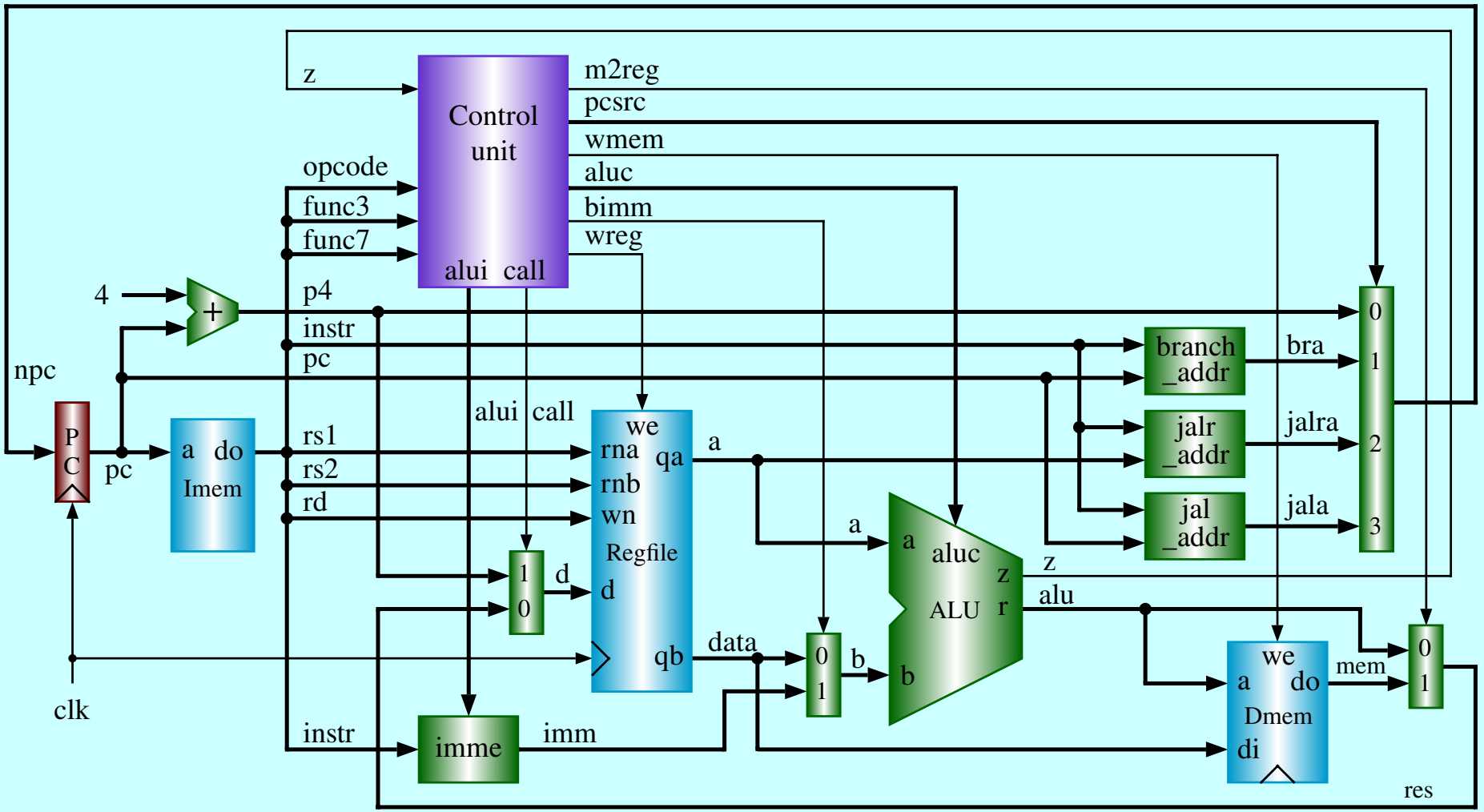
パイプライン方式で命令を実行



2012年		lw		卒業した
2013年		add		卒業した
2014年		sub		卒業した
2015年		and		卒業した
2016年		or		卒業した
2017年		xor		卒業した
2018年		slli		卒業した
2019年		lui		卒業した
2020年		addi	→ WB	五年生
2021年		andi	→ MEM	四年生
2022年		ori	→ EXE	三年生
2023年		xori	→ ID	二年生
2024年	PC →	srli	→ IF	一年生

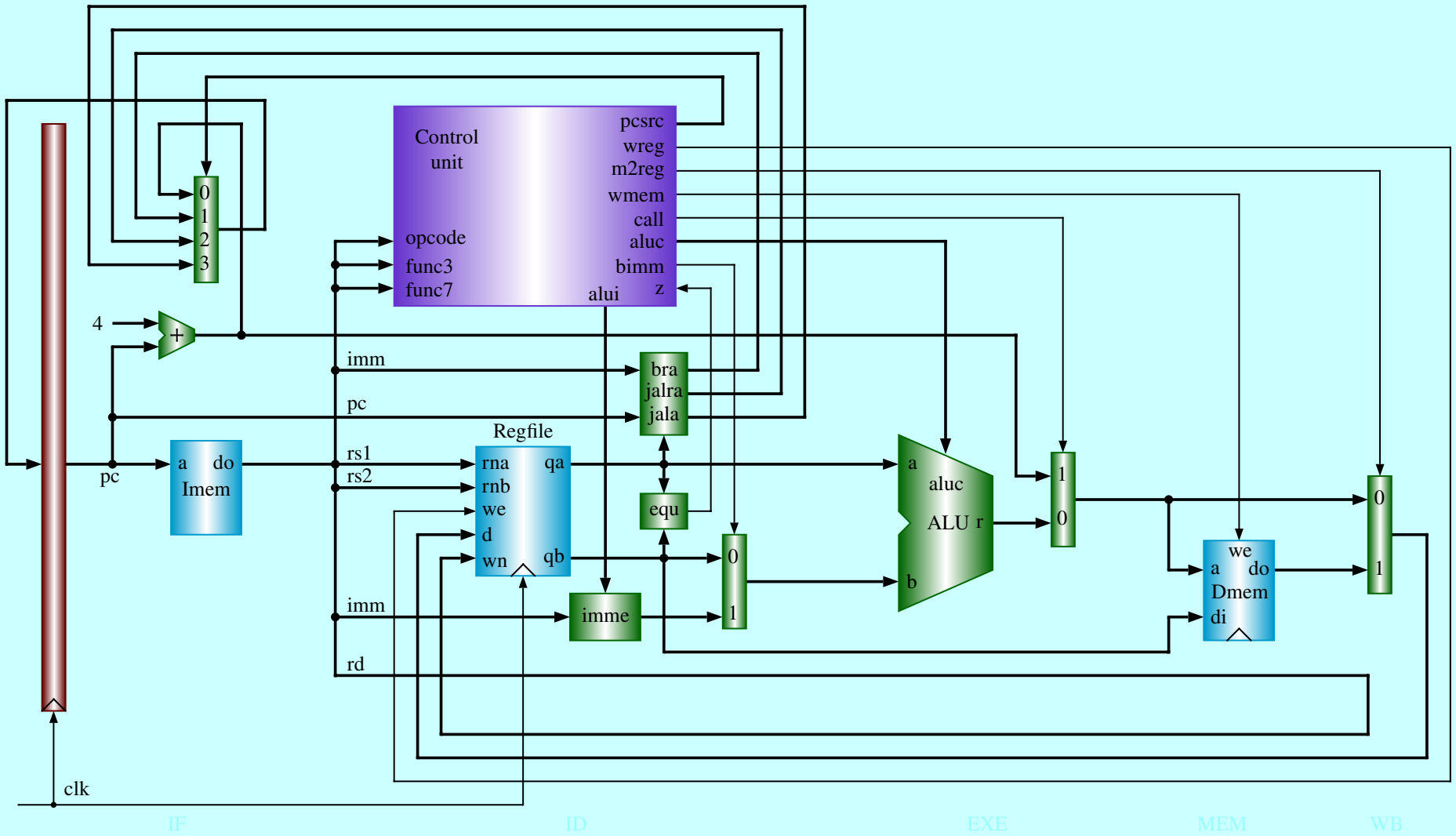
2012年新しい学部ができた

単一サイクル CPU 回路



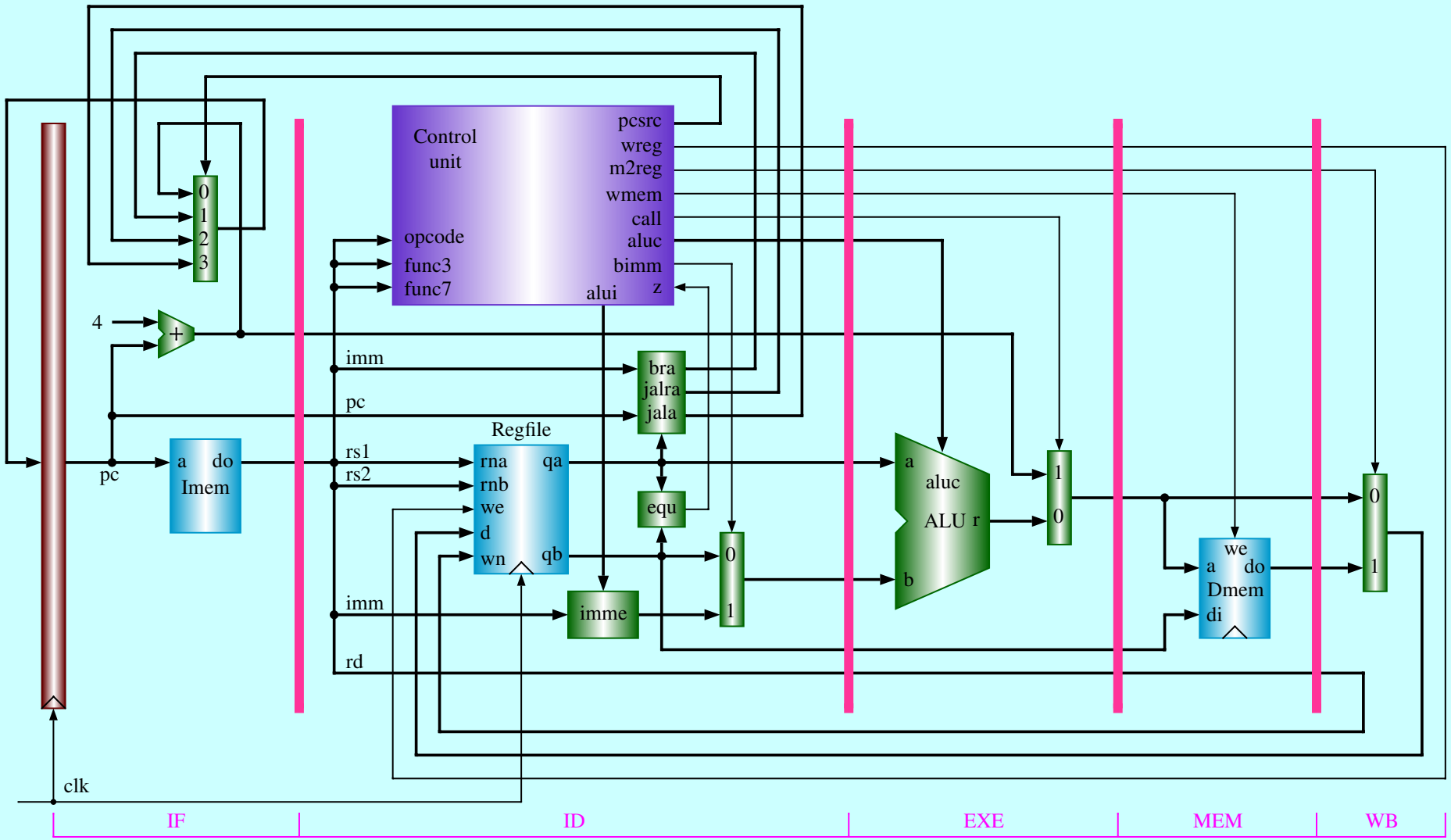
CPU + 命令メモリ Imem + データメモリ Dmem

単一サイクル CPU 回路



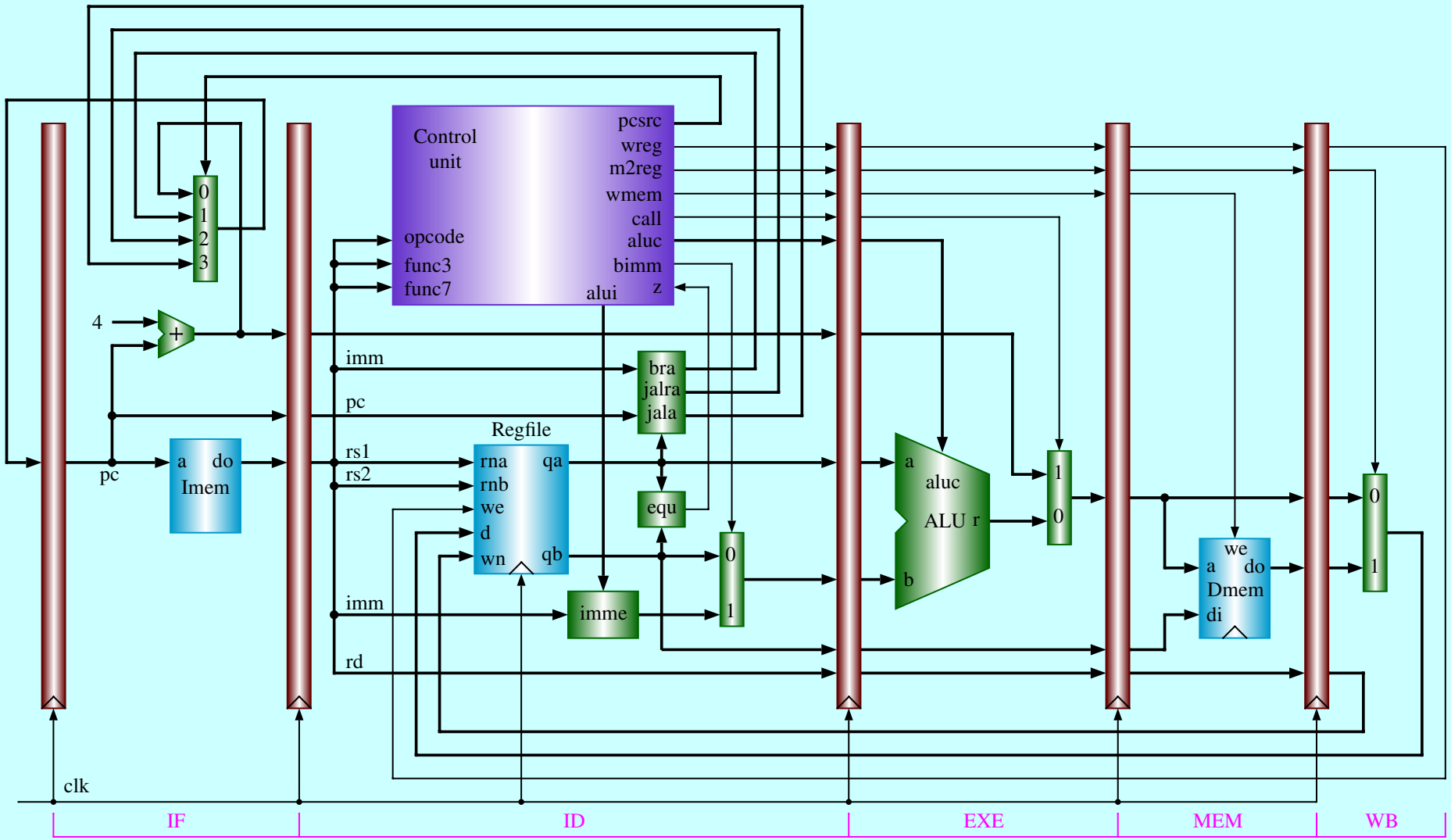
同じものである

単一サイクル CPU 回路



ステージに分割する

パイプライン CPU 回路



ステージとステージの間にはレジスタを挿入する

パイプラインハザードの種類

ハザード ― 命令ストリームにある次の命令を想定されたクロック・サイクル中に実行することを妨げる状況が存在する

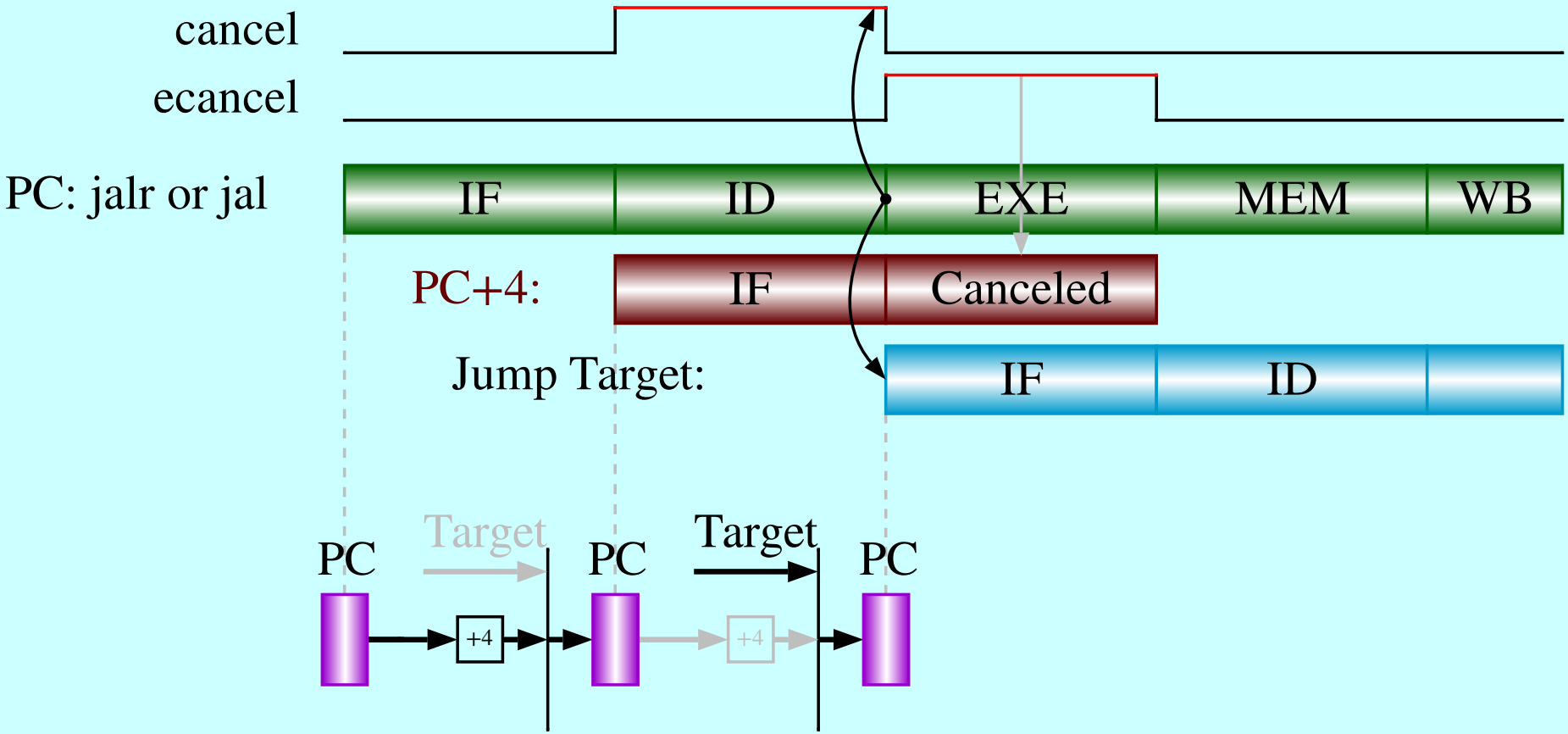
1. 構造ハザード (Structural Hazard):
   - ハードウェア資源の競合によって起こる。資源の競合とは同時多重実行されうる複数の命令が同じハードウェア資源を要求したため，システムがその要求を満たせない状態である
2. 制御ハザード (Control Hazard):
   - 分岐命令や割り込みなどパイプラインの流れを変えてしまうことによって発生する
3. データハザード (Data Hazard):
   - 命令が直前の命令の実行結果を使用している時に発生する

パイプラインハザードの解消方法

ハザード — 命令ストリームにある次の命令を想定されたクロック・サイクル中に実行することを妨げる状況が存在する

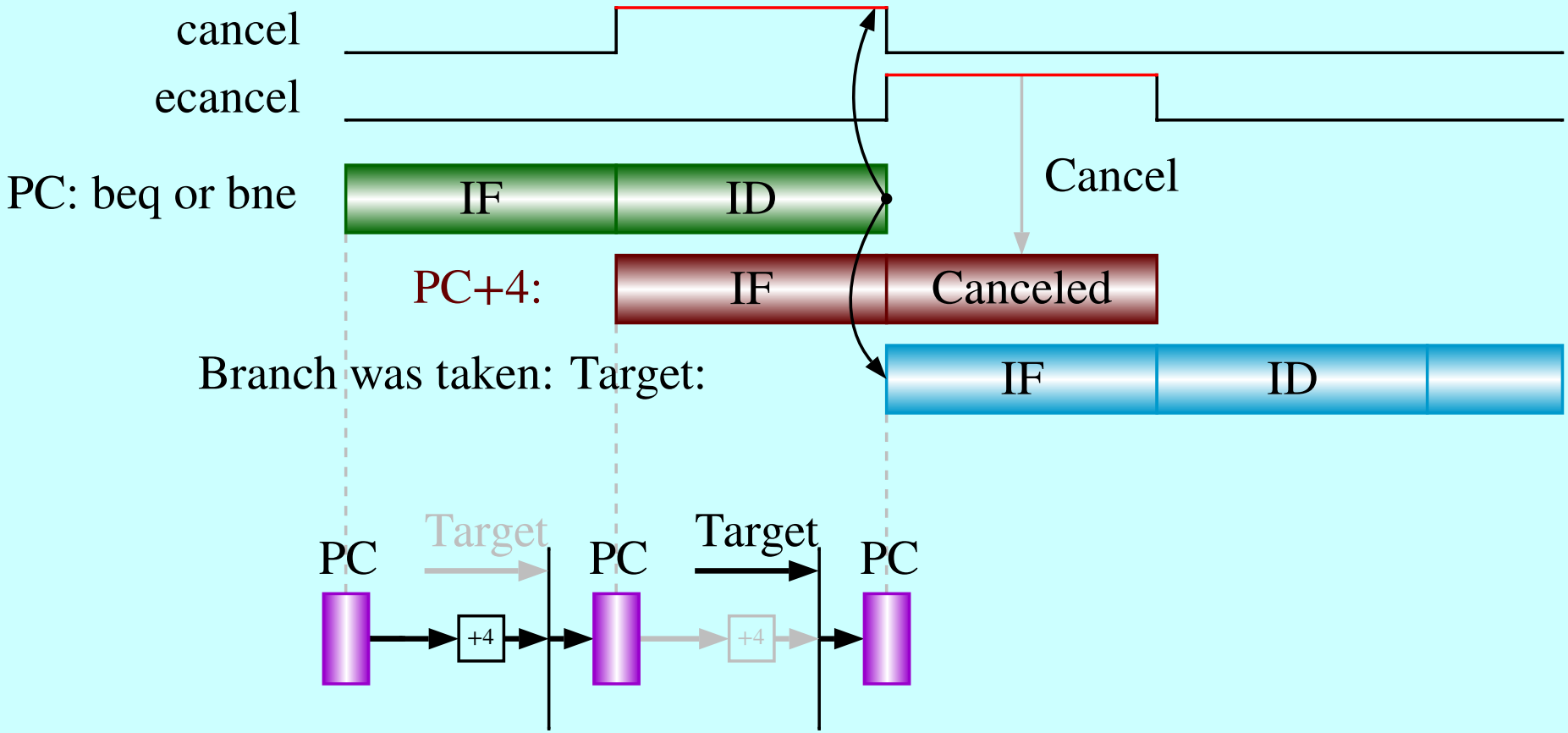
1. 構造ハザード (Structural Hazard):
   - 複数のアクセスに対応したハードウェア資源を採用する
   - コンパイラによるスケジューリング
2. 制御ハザード (Control Hazard):
   - 遅延分岐
   - 分岐予測機構による投機実行
3. データハザード (Data Hazard):
   - コンパイラによるスケジューリング
   - フォアワーディング (バイパスを設ける)

Jump: Cancel the Next Instruction



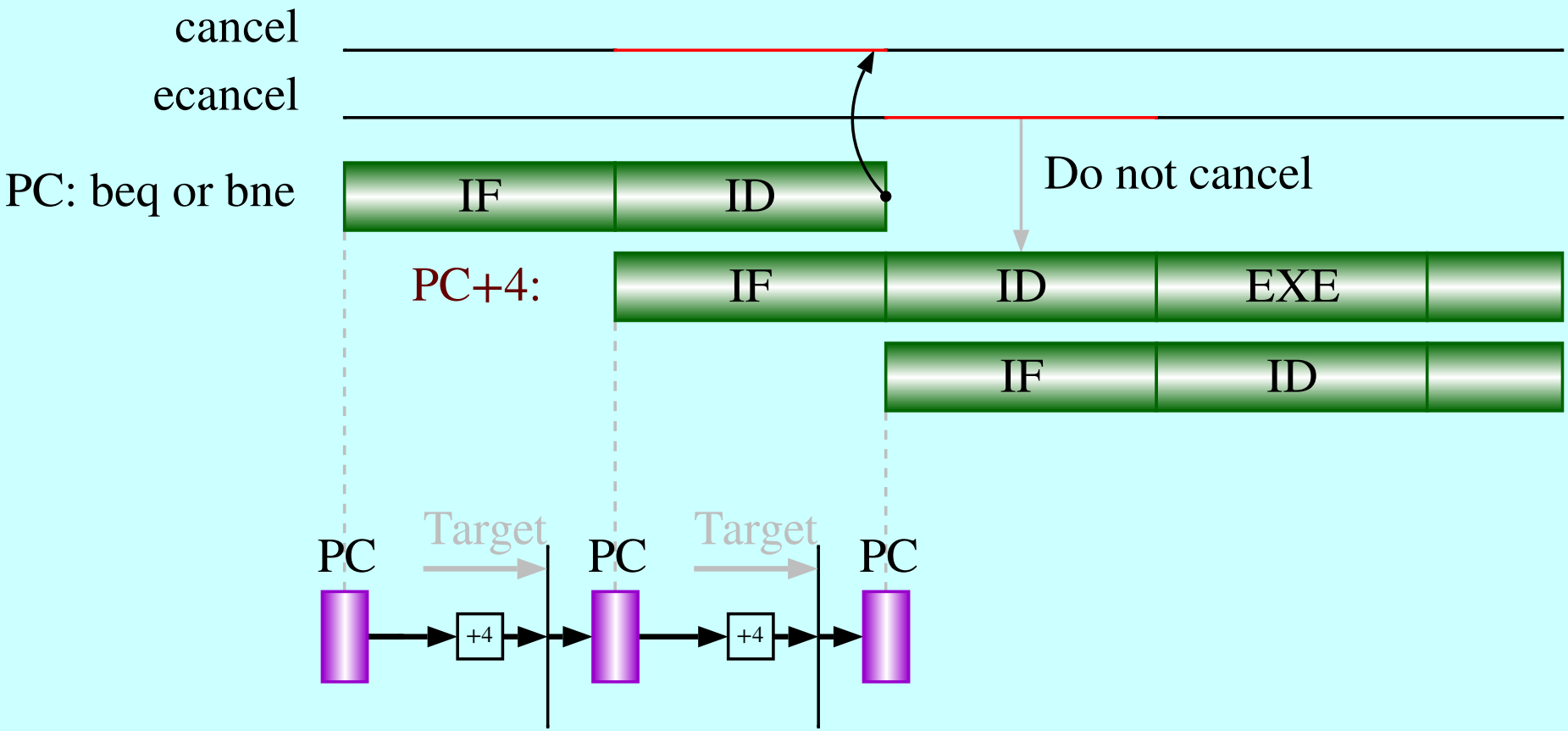
Unconditional jump

Branch: Cancel the Next Instruction



Conditional branch: Branch is taken

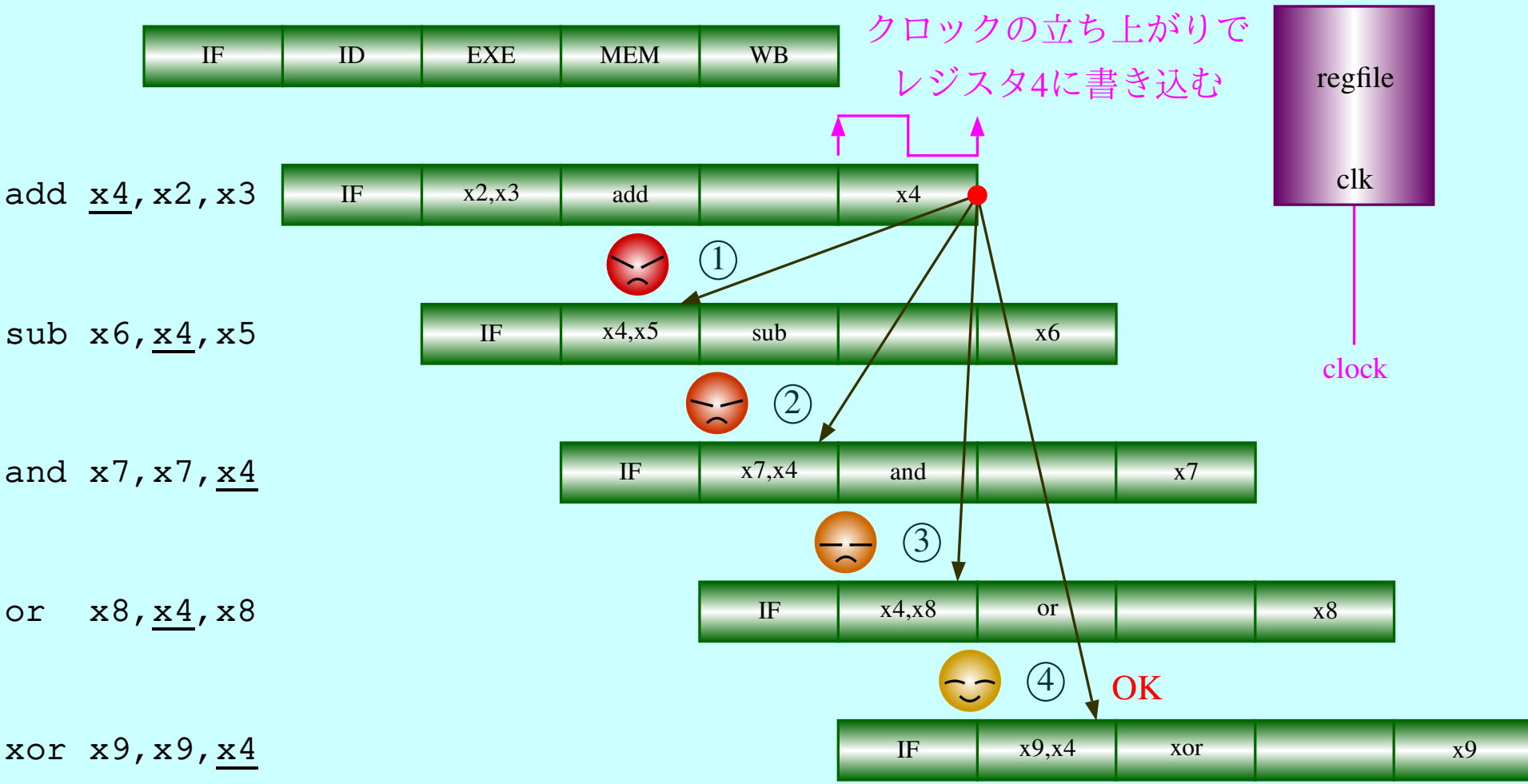
Branch is not Taken: Do not Cancel



Conditional branch: Branch is not taken

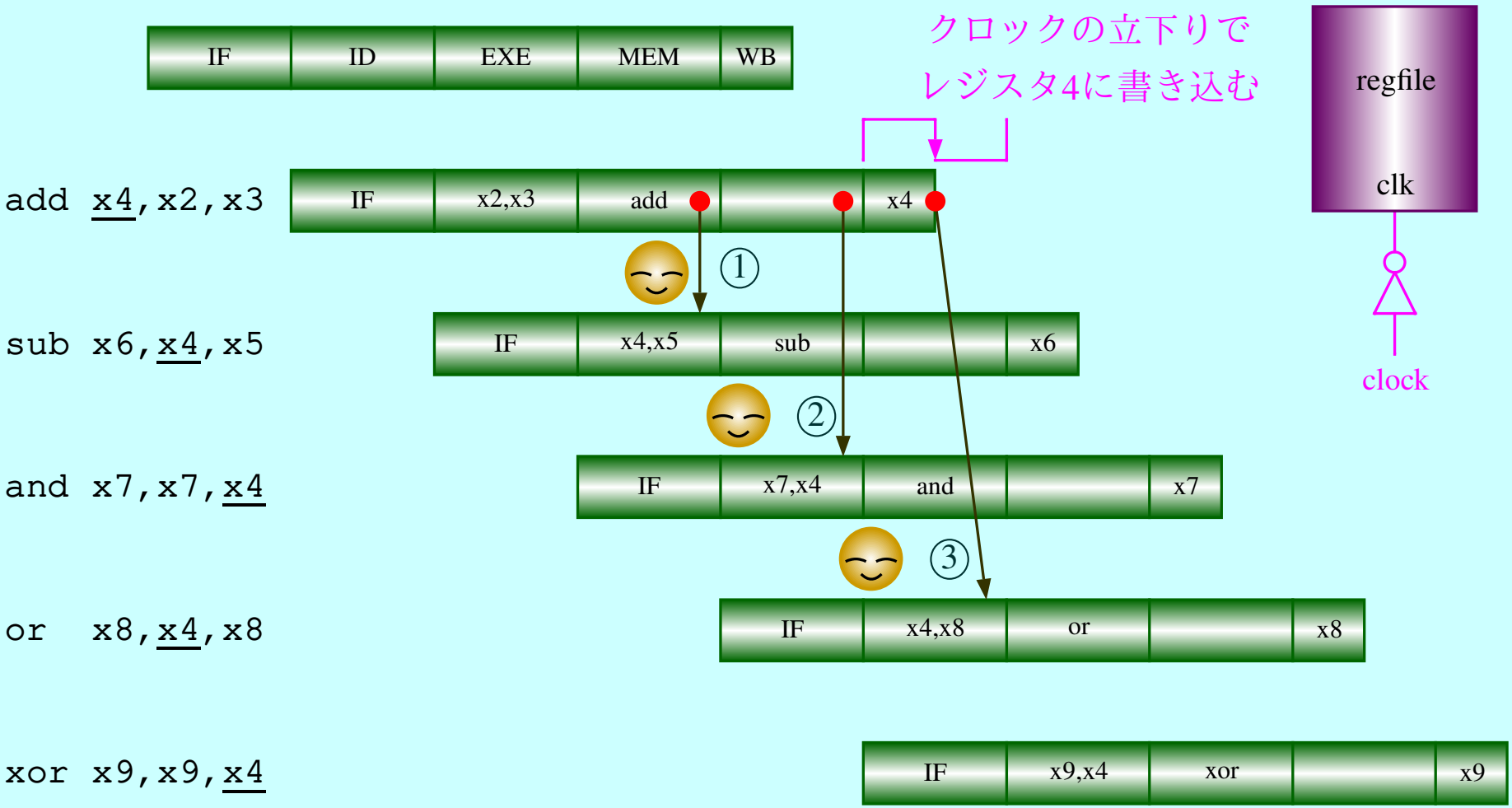
データハザード (Data Hazard)

データハザード: 命令が直前の命令の実行結果を使用する場合に発生するハザード

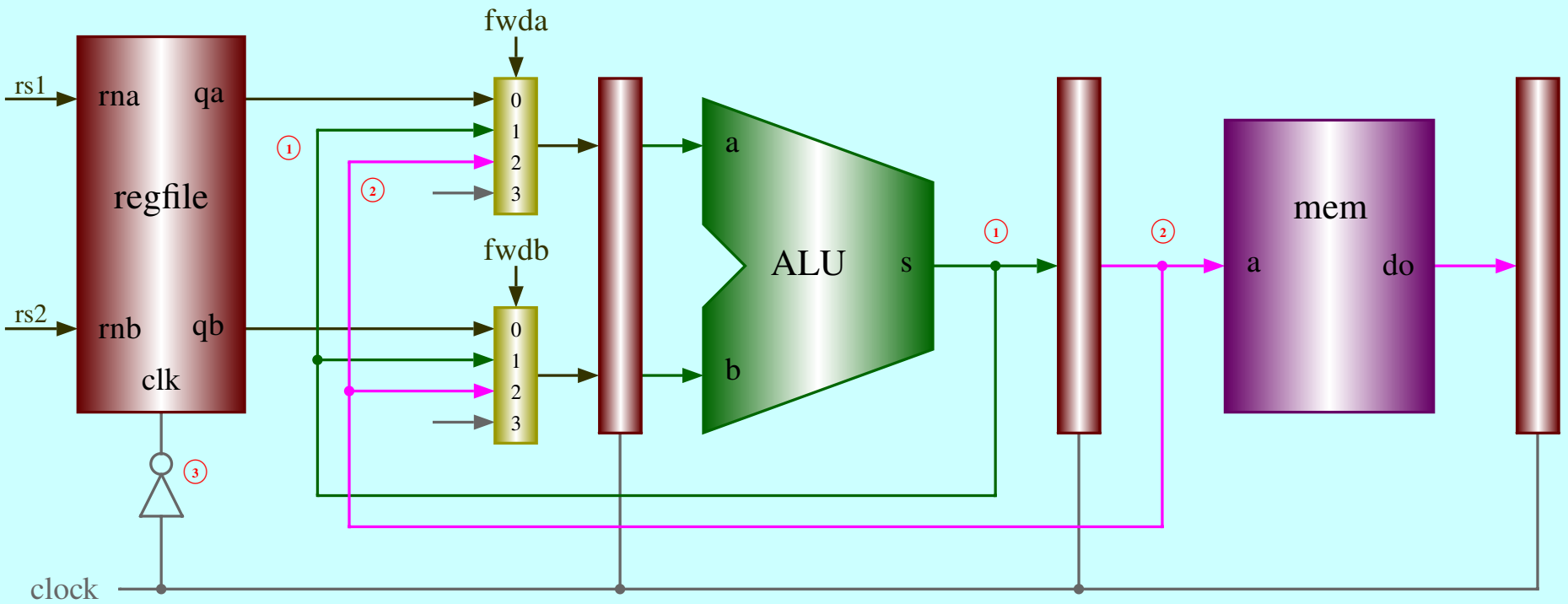


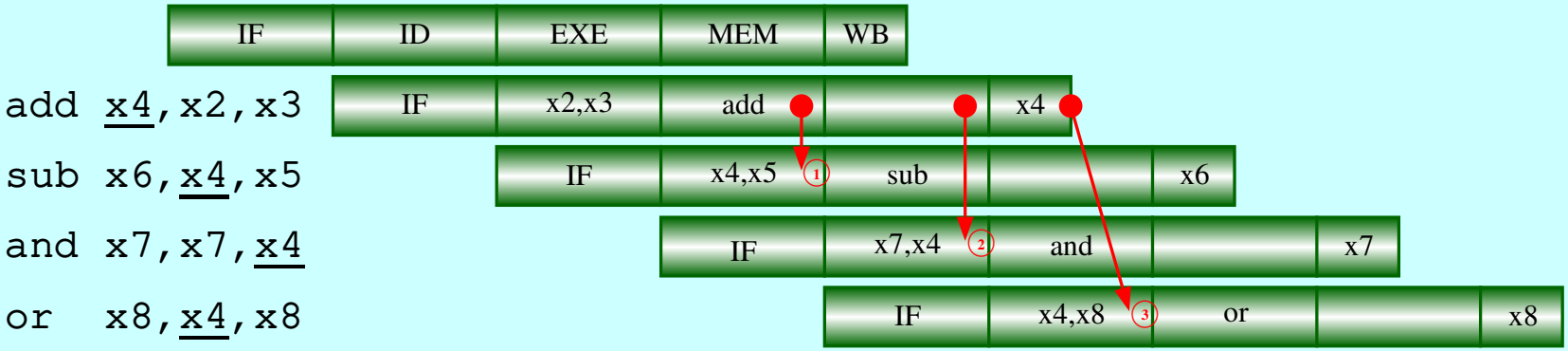
フォアワーディング (Forwarding)

バイパスを設け、立下り時に書き込む (１サイクルの半分のところ)

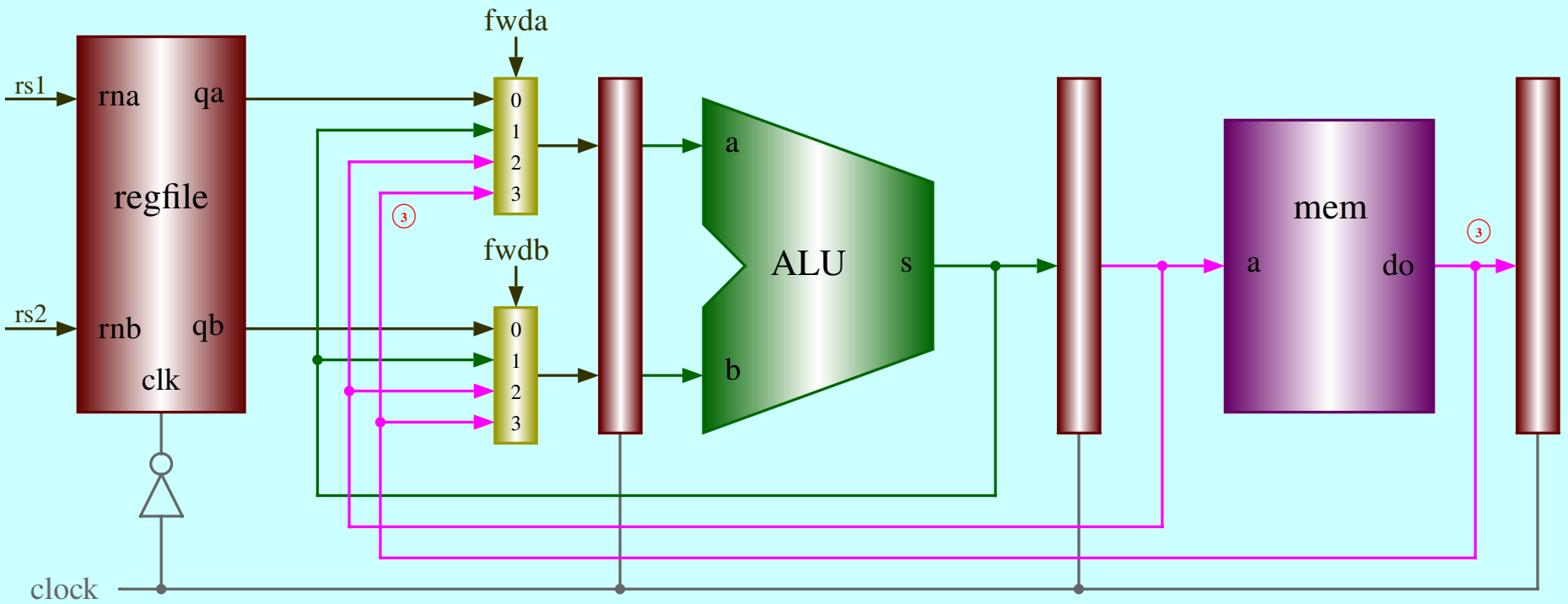


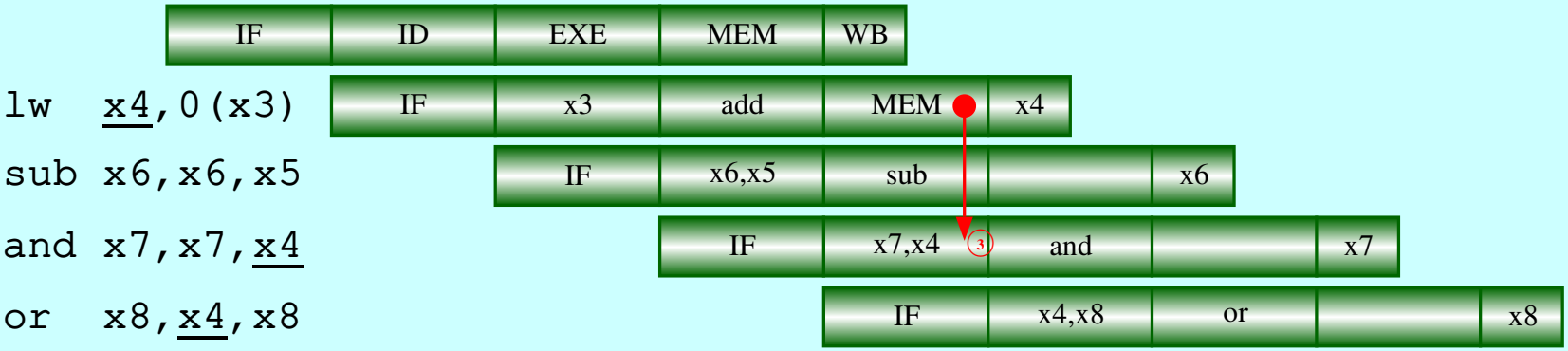
フォアワーディング — from add





フォアワーディング — from lw





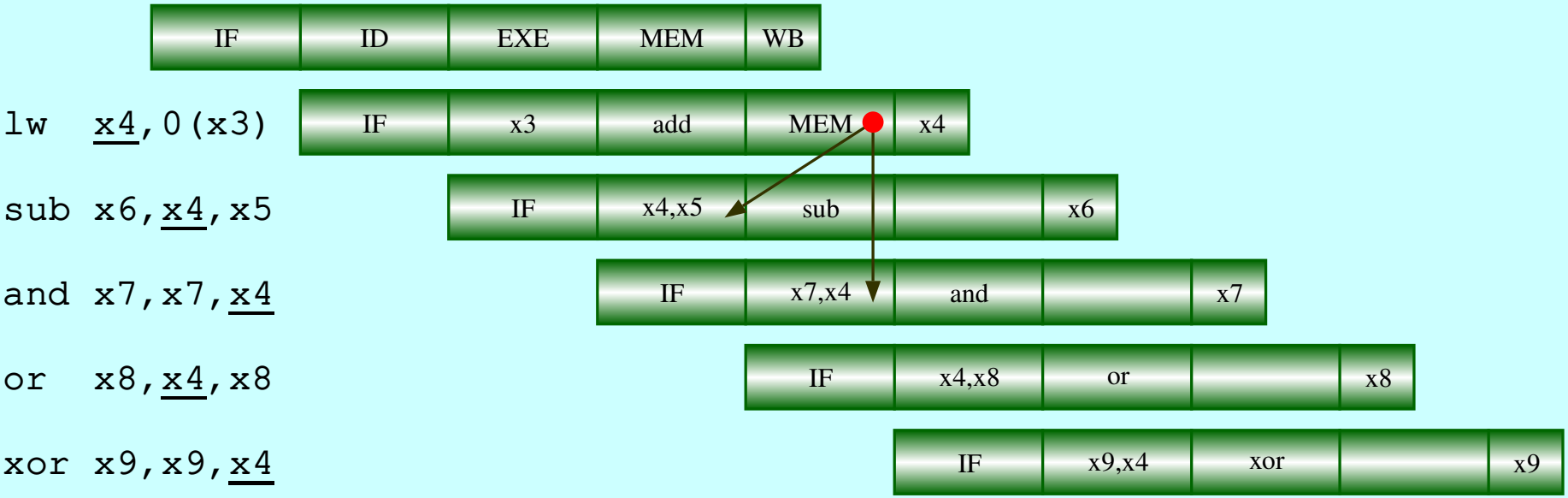
フォアワーディング — fwda

```
fwda = 2'b00; // default: no hazards
if (ewreg & (erd != 0) & (erd == rs1) & ~em2reg) begin
    fwda = 2'b01; // select exe_alu
end else begin
    if (mwreg & (mrd != 0) & (mrd == rs1) & ~mm2reg) begin
        fwda = 2'b10; // select mem_alu
    end else begin
        if (mwreg & (mrd != 0) & (mrd == rs1) & mm2reg) begin
            fwda = 2'b11; // select mem_lw
        end
    end
end
```

フォアワーディング — fwdb

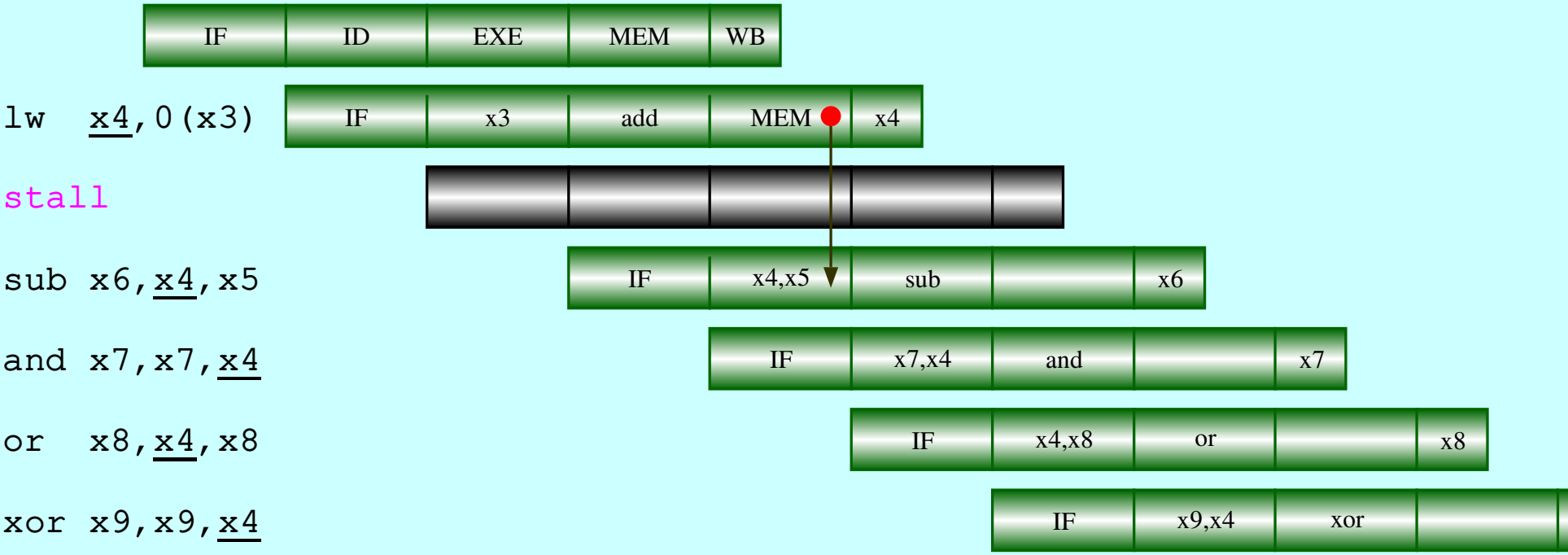
```
fwdb = 2'b00; // default: no hazards
if (ewreg & (erd != 0) & (erd == rs2) & ~em2reg) begin
    fwdb = 2'b01; // select exe_alu
end else begin
    if (mwreg & (mrd != 0) & (mrd == rs2) & ~mm2reg) begin
        fwdb = 2'b10; // select mem_alu
    end else begin
        if (mwreg & (mrd != 0) & (mrd == rs2) & mm2reg) begin
            fwdb = 2'b11; // select mem_lw
        end
    end
end
```

lwは次の命令にデータを転送できない



ロード (lw) 命令は and 命令には結果を転送できるが，sub 命令にはできない。なぜなら，sub 命令がデータを必要とするときがロード命令の結果が得られたときよりも “時間的に前” だからである

パイプライン・ストールを起こす



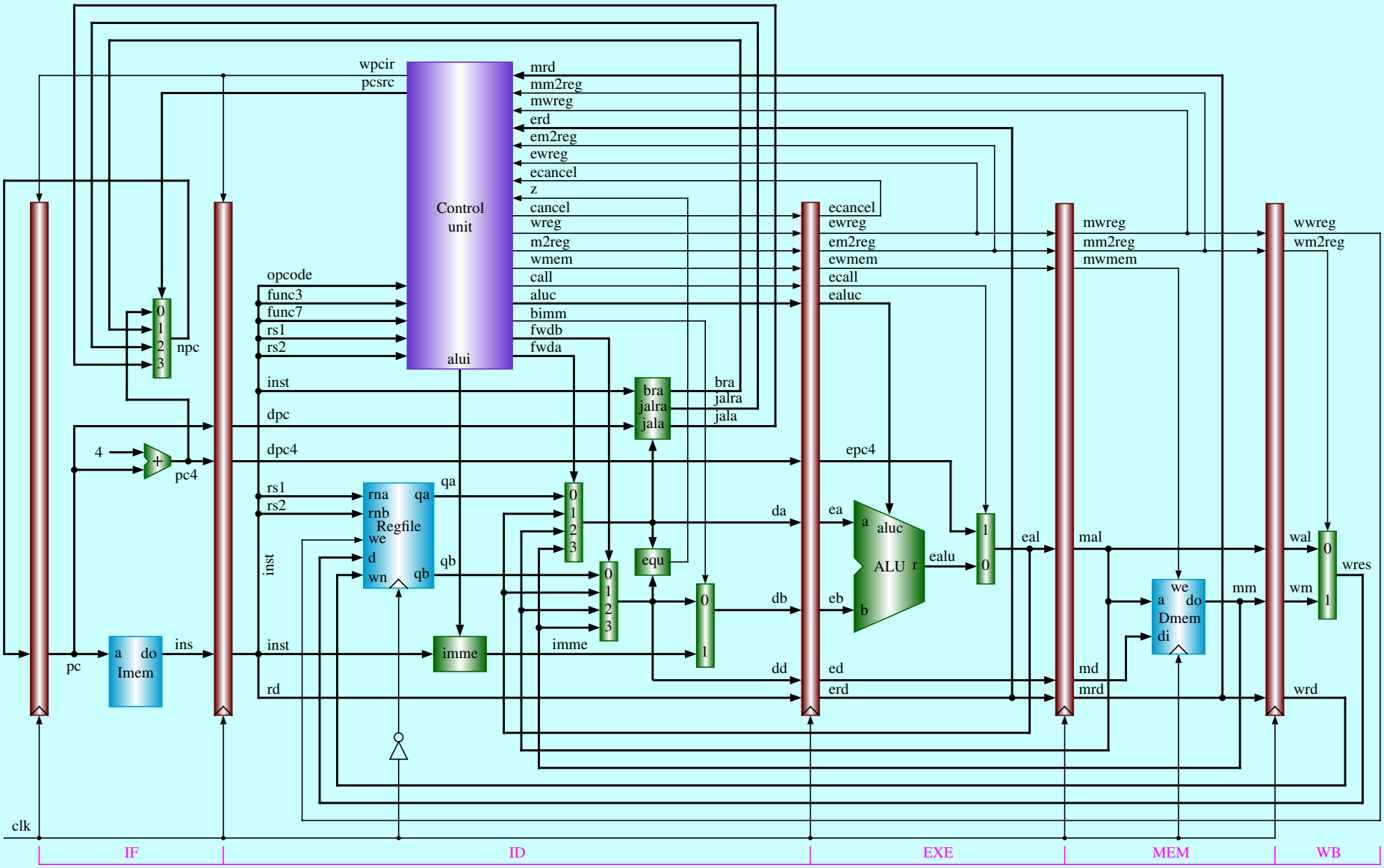
パイプラインをストール(Stall)させるために，PC と ID パイプライン・レジスタへの書き込みを防ぐ信号 wpcir を生成する

パイプライン・ストール — wpcir

```
i_rs1 = i_jalr | i_beq  | i_bne  | i_lw   | i_sw   | i_addi |
        i_xori | i_ori  | i_andi | i_slli | i_srli | i_srai |
        i_add  | i_sub  | i_slt  | i_xor  | i_or   | i_and;

i_rs2 = i_beq  | i_bne  | i_sw   |
        i_add  | i_sub  | i_slt  | i_xor  | i_or   | i_and;

wpcir = ~(ewreg & em2reg & (erd != 0) &
         (i_rs1 & (erd == rs1) |
          i_rs2 & (erd == rs2)));

wreg = (i_lui  | i_jal  | i_jalr | i_lw   | i_addi | i_xori |
        i_ori  | i_andi | i_slli | i_srli | i_srai | i_add  |
        i_sub  | i_slt  | i_xor  | i_or   | i_and) & wpcir & ~ecancel;

wmem  = i_sw & wpcir & ~ecancel;
```

wpcir は 0 の時、PC と IR に書き込まない

パイプライン CPU 回路



pl_computer.v

```
‘timescale 1ns/1ns
module pl_computer (clk,clrn,pc,inst,eal,mal,wres);                    // pipelined cpu
    input         clk;             // clock                           // plus inst mem
    input         clrn;            // reset                           // and  data mem
    output [31:0] pc;              // program counter
    output [31:0] inst;            // instruction in ID stage
    output [31:0] eal;             // alu or epc4 in EXE stage
    output [31:0] mal;             // eal in MEM stage
    output [31:0] wres;            // data to be written into register file
    // signals in IF stage
    wire   [31:0] pc4;             // pc+4 in IF stage
    wire   [31:0] ins;             // instruction in IF stage
    wire   [31:0] npc;             // next pc in IF stage
    // signals in ID stage
    wire   [31:0] dpc;             // pc in ID stage
    wire   [31:0] dpc4;            // pc+4 in ID stage
    wire   [31:0] bra;             // branch target of beq and bne instructions
    wire   [31:0] jalra;           // jump target of jalr instruction
    wire   [31:0] jala;            // jump target of jal instruction
    wire   [31:0] da;              // operand a in ID stage
    wire   [31:0] db;              // operand b in ID stage
    wire   [31:0] dd;              // reg data to mem in ID stage
    wire    [4:0] rd = inst[11:7]; // destination register number in ID stage
    wire    [3:0] aluc;            // alu control in ID stage
    wire    [1:0] pcsrc;           // next pc (npc) select in ID stage
    wire          wpcir;           // pipepc and pipeir write enable
    wire          m2reg;           // memory to register in ID stage
    wire          wreg;            // register file write enable in ID stage
    wire          wmem;            // memory write in ID stage
    wire          call;            // jalr, jal in ID stage
    wire          cancel;          // cancel in ID stage
    // signals in EXE stage
    wire   [31:0] epc4;            // pc+4 in EXE stage
    wire   [31:0] ea;              // operand a in EXE stage
    wire   [31:0] eb;              // operand b in EXE stage
    wire   [31:0] ed;              // reg data to mem in EXE stage
    wire    [4:0] erd;             // destination register number in EXE stage
    wire    [3:0] ealuc;           // alu control in EXE stage
```

pl_computer.v

```
    wire          em2reg;          // memory to register in EXE stage
    wire          ewreg;           // register file write enable in EXE stage
    wire          ewmem;           // memory write in EXE stage
    wire          ecall;           // jalr, jal in EXE stage
    wire          ecancel;         // cancel in EXE stage
    // signals in MEM stage
    wire   [31:0] mm;              // memory data out in MEM stage
    wire   [31:0] md;              // reg data to mem in MEM stage
    wire    [4:0] mrd;             // destination register number in MEM stage
    wire          mm2reg;          // memory to register in MEM stage
    wire          mwreg;           // register file write enable in MEM stage
    wire          mwmem;           // memory write in MEM stage
    // signals in WB stage
    wire   [31:0] wal;             // mal in WB stage
    wire   [31:0] wm;              // memory data out in WB stage
    wire    [4:0] wrd;             // destination register number in WB stage
    wire          wm2reg;          // memory to register in WB stage
    wire          wwreg;           // register file write enable in WB stage
    // program counter
    pl_reg_pc prog_cnt (npc,wpcir,clk,clrn, pc);
    pl_stage_if if_stage (pcsrc,pc,bra,jalra,jala,npc,pc4,ins);             // IF stage
    // IF/ID pipeline register
    pl_reg_ir fd_reg ( pc, pc4,ins, wpcir,clk,clrn, dpc,dpc4,inst);
    pl_stage_id id_stage (mrd,mm2reg,mwreg,erd,em2reg,ewreg,ecancel,dpc,inst,eal,mal,mm,
                          wrd,wres,wwreg,clk,clrn,bra,jalra,jala,pcsrc,wpcir,cancel,wreg,
                          m2reg,wmem,call,aluc,da,db,dd);                   // ID stage
    // ID/EXE pipeline register
    pl_reg_de de_reg ( cancel, wreg, m2reg, wmem, call, aluc, rd,dpc4,da,db,dd,clk,clrn,
                       ecancel,ewreg,em2reg,ewmem,ecall,ealuc,erd,epc4,ea,eb,ed);
    pl_stage_exe exe_stage (ea,eb,epc4,ealuc,ecall, eal);                   // EXE stage
    // EXE/MEM pipeline register
    pl_reg_em em_reg (ewreg,em2reg,ewmem,eal,ed,erd,clk,clrn,
                      mwreg,mm2reg,mwmem,mal,md,mrd);
    pl_stage_mem mem_stage (mwmem,mal,md,clk, mm);                          // MEM stage
    // MEM/WB pipeline register
    pl_reg_mw mw_reg (mwreg,mm2reg,mm,mal,mrd,clk,clrn,wwreg,wm2reg,wm,wal,wrd);
    pl_stage_wb wb_stage (wal,wm,wm2reg, wres);                             // WB stage
endmodule
```

命令メモリ (テスト・プログラム)

```
‘timescale 1ns/1ns
module pl_instmem (a,inst);
    input  [31:0] a;
    output [31:0] inst;
    wire   [31:0] rom [0:31];                                  // (pc)
    assign rom[5’h00] = 32’b00000000000000000000000010110111; // (00) main:   lui  x1, 0         # x1 <- 0
    assign rom[5’h01] = 32’b00000101000000001110001000010011; // (04)         ori  x4, x1, 80    # x1 <- 80
    assign rom[5’h02] = 32’b00000000010000000000001010010011; // (08)         addi x5, x0,  4    # x5 <- 4
    assign rom[5’h03] = 32’b00000101100000000000000011101111; // (0c) call:   jal  x1, sum      # x1 <- 0x10 (return address), call sum
    assign rom[5’h04] = 32’b00000000011000100010000000100011; // (10)         sw   x6, 0(x4)     # memory[x4+0] <- x6
    assign rom[5’h05] = 32’b00000000000000010001001010000011; // (14)         lw   x9, 0(x4)     # x6 <- memory[x4+0]
    assign rom[5’h06] = 32’b01000000010001001000010000110011; // (18)         sub  x8, x9, x4    # x8 <- x9 - x4, pipeline stall
    assign rom[5’h07] = 32’b00000000000110000000000010010011; // (1c)         addi x5, x0,  3    # x5 <- 3
    assign rom[5’h08] = 32’b11111111111100101000001010010011; // (20) loop2:  addi x5, x5, -1    # x5 <- x5 - 1
    assign rom[5’h09] = 32’b11111111111100101110010000010011; // (24)         ori  x8, x5, -1    # x8 <- x5 | 0xffffffff = 0xffffffff
    assign rom[5’h0a] = 32’b01010101010101000100010000010011; // (28)         xori x8, x8, 0x555 # x8 <- x8 ^ 0x00000555 = 0xfffffaaa
    assign rom[5’h0b] = 32’b11111111111100000000010010010011; // (2c)         addi x9, x0, -1    # x9 <- 0xffffffff
    assign rom[5’h0c] = 32’b11111111111101001111010100010011; // (30)         andi x10,x9, -1    # x10<- x9 & 0xffffffff = 0xffffffff
    assign rom[5’h0d] = 32’b00000000100101010110001000110011; // (34)         or   x4, x10, x9   # x4 <- x10 | x9 = 0xffffffff
    assign rom[5’h0e] = 32’b00000000100101010100010000110011; // (38)         xor  x8, x10, x9   # x8 <- x10 ^ x9 = 0x00000000
    assign rom[5’h0f] = 32’b00000000010001010111001110110011; // (3c)         and  x7, x10, x4   # x7 <- x10 & x4 = 0xffffffff
    assign rom[5’h10] = 32’b00000000000000010100001000110011; // (40)         beq  x5, x0, shift # if x5 = 0, goto shift
    assign rom[5’h11] = 32’b11111101110111111111000001101111; // (44)         jal  x0, loop2     # jump loop2
    assign rom[5’h12] = 32’b11111111111100000000001010010011; // (48) shift:  addi x5, x0, -1    # x5 <- 0xffffffff
    assign rom[5’h13] = 32’b00000000111100101001010000010011; // (4c)         slli x8, x5, 15    # x8 <- 0xffffffff <<  15 = 0xffff8000
    assign rom[5’h14] = 32’b00000001000001000001010000010011; // (50)         slli x8, x8, 16    # x8 <- 0xffff8000 <<  16 = 0x80000000
    assign rom[5’h15] = 32’b01000001000001000101010000010011; // (54)         srai x8, x8, 16    # x8 <- 0x80000000 >>> 16 = 0xffff8000
    assign rom[5’h16] = 32’b00000000111101000101010000010011; // (58)         srli x8, x8, 15    # x8 <- 0xffff8000 >>  15 = 0x0001ffff
    assign rom[5’h17] = 32’b00000000011000100010000110110011; // (5c)         slt  x3, x4, x6    # x3 <- 0xffffffff < 0x000002ff = 1
    assign rom[5’h18] = 32’b00000000000000000000000001101111; // (60) finish: jal  x0, finish    # dead loop
    assign rom[5’h19] = 32’b00000000000000000000000001100110011; // (64) sum:    add  x6, x0, x0    # x6 <- 0 (subroutine entry)
    assign rom[5’h1a] = 32’b00000000000000010001001010000011; // (68) loop:   lw   x9, 0(x4)     # x9 <- memory[x4+0]
    assign rom[5’h1b] = 32’b00000000010000100000001000010011; // (6c)         addi x4, x4,  4    # x4 <- x4 + 4 (address+4)
    assign rom[5’h1c] = 32’b00000000100100110000001100110011; // (70)         add  x6, x6, x9    # x6 <- x6 + x9 (sum)
    assign rom[5’h1d] = 32’b11111111111100101000001010010011; // (74)         addi x5, x5, -1    # x5 <- x5 - 1 (counter--)
    assign rom[5’h1e] = 32’b11111110000000101001100011100011; // (78)         bne  x5, x0, loop  # if x5 != 0, goto loop
    assign rom[5’h1f] = 32’b00000000000000000100000001100111; // (7c)         ret  x1            # return from subroutine
    assign inst = rom[a[6:2]];
endmodule
```

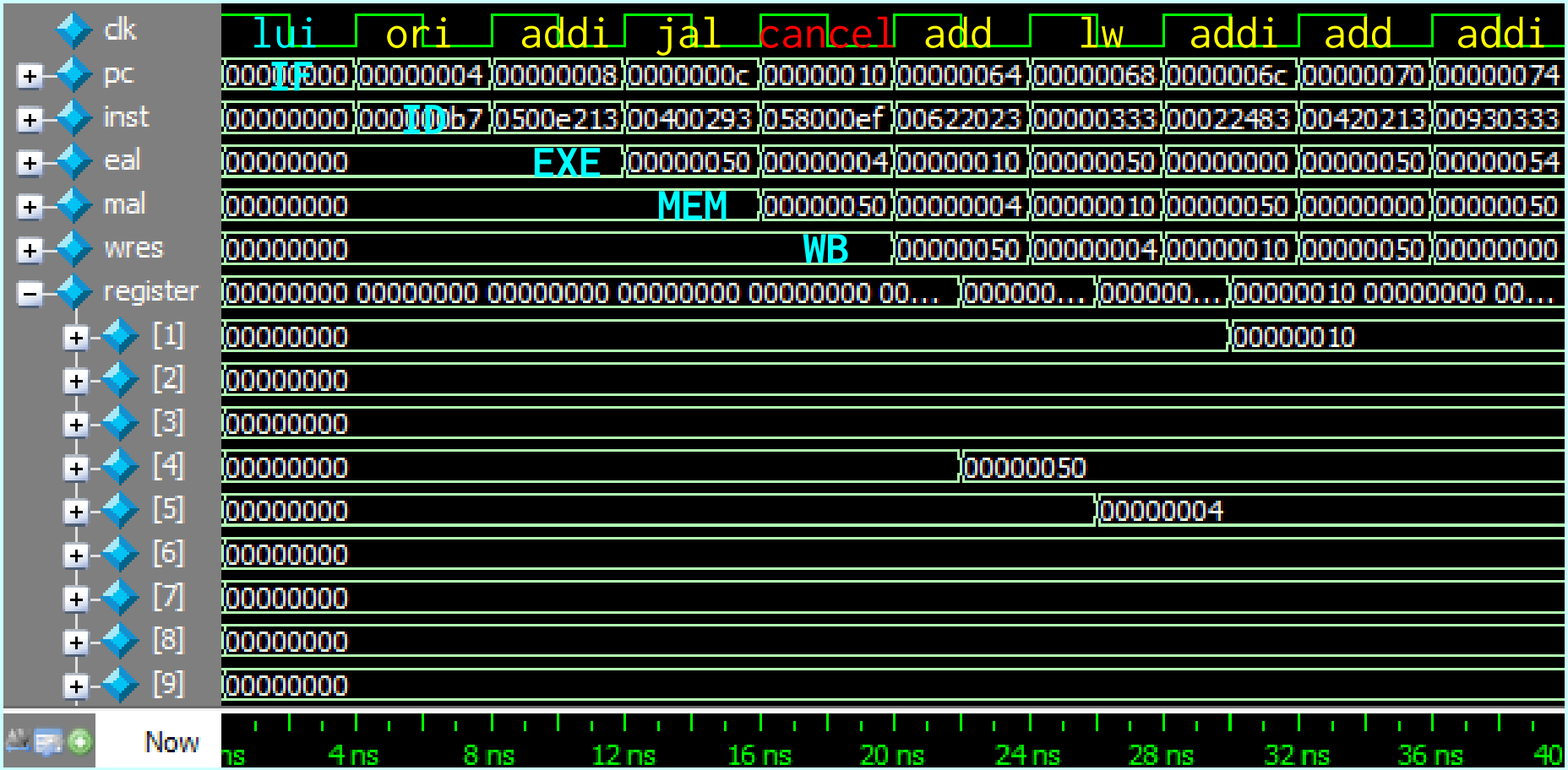
pl_instmem.s.txt

データメモリ (テスト・データ)

```
‘timescale 1ns/1ns
module pl_datamem (addr,datain,we,clk,dataout); // data memory, ram
    input         clk;                          // clock
    input         we;                           // write enable
    input  [31:0] datain;                       // data in (to memory)
    input  [31:0] addr;                         // ram address
    output [31:0] dataout;                      // data out (from memory)
    reg    [31:0] ram [0:31];                   // ram cells: 32 words * 32 bits
    assign dataout = ram[addr[6:2]];            // use word address to read ram
    always @ (posedge clk)
        if (we) ram[addr[6:2]] = datain;        // use word address to write ram
    integer i;
    initial begin                               // initialize memory
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data                // (byte_addr) item in data array
        ram[5’h14] = 32’h000000f2;              // (50)  data[0]
        ram[5’h15] = 32’h0000000e;              // (54)  data[1]
        ram[5’h16] = 32’h00000200;              // (58)  data[2]
        ram[5’h17] = 32’hffffffff;              // (5c)  data[3]
        // ram[5’h18] the sum stored by sw instruction
    end
endmodule
```

pl_instmem.s.txt

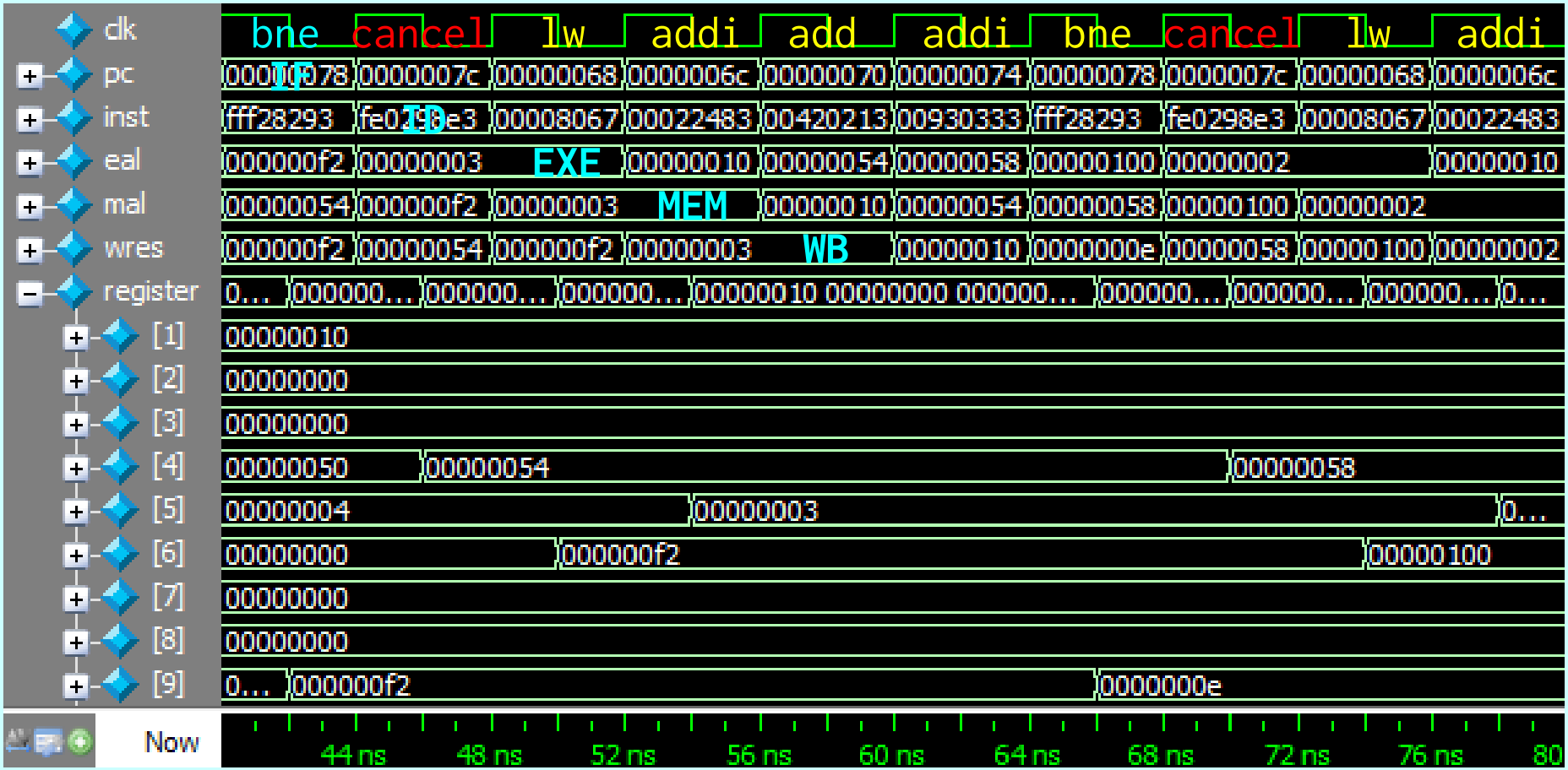
シミュレーション — Registers




```
(00) main:   lui  x1, 0
(04)         ori  x4, x1, 80
(08)         addi x5, x0,  4
(0c) call:   jal  x1, sum
(10) canceled
```

```
(64) sum:    add  x6, x0, x0
(68) loop:   lw   x9, 0(x4)
(6c)         addi x4, x4,  4
(70)         add  x6, x6, x9
(74)         addi x5, x5, -1
```

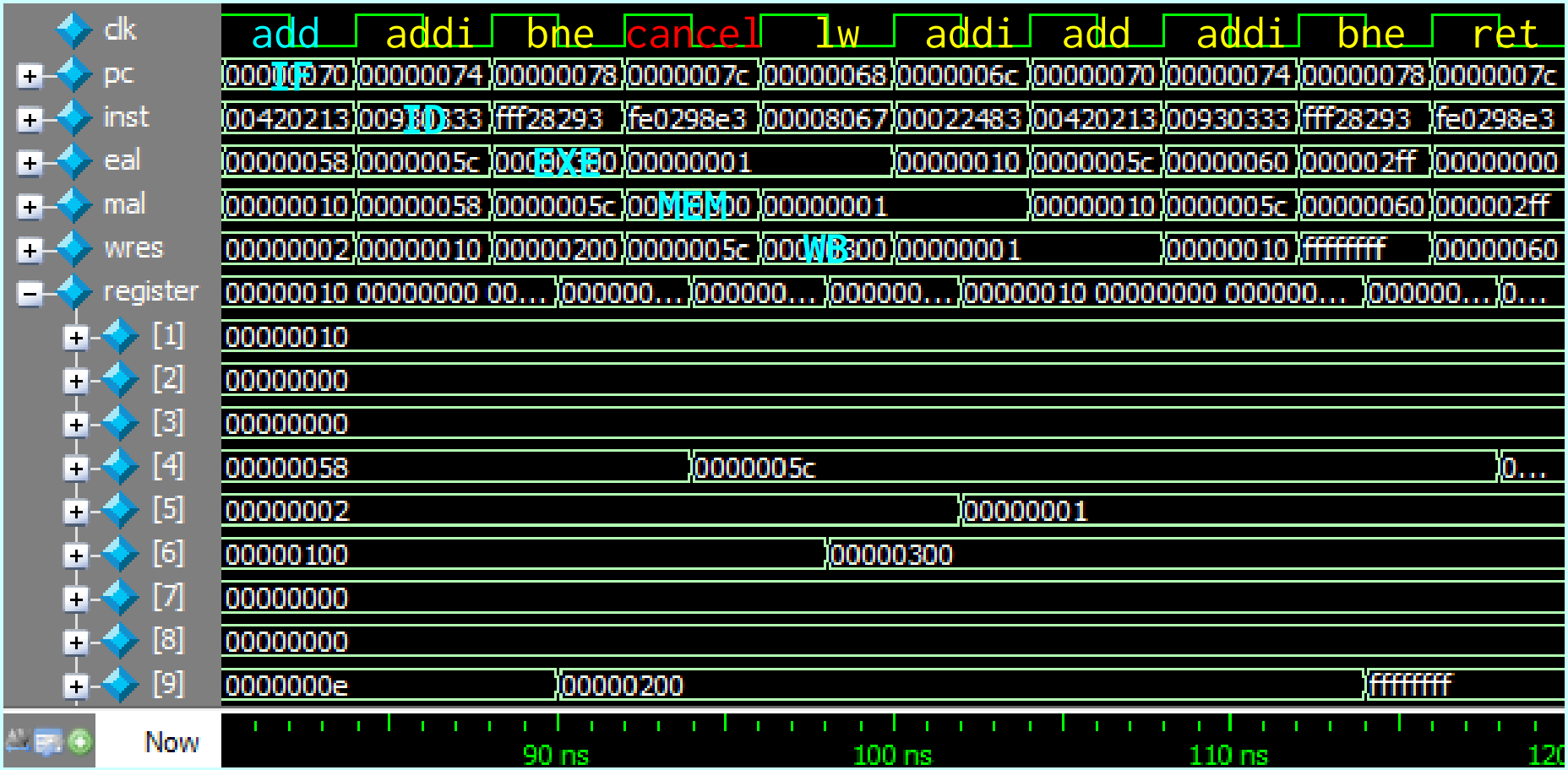
シミュレーション — Registers




```
(78)          bne  x5, x0, loop
(7c) canceled
(68) loop:    lw   x9, 0(x4)
(6c)          addi x4, x4,  4
(70)          add  x6, x6, x9
```

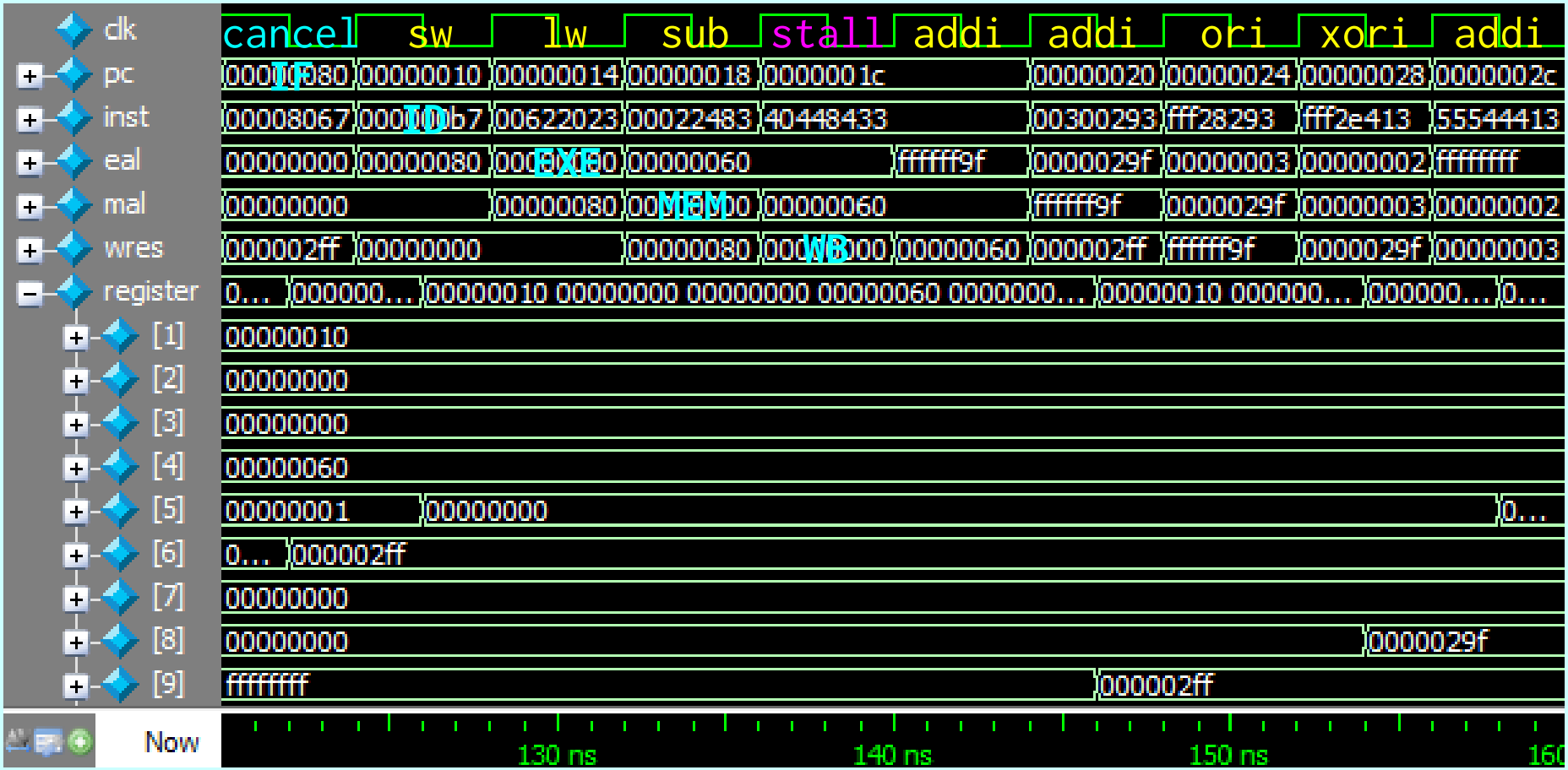
```
(74)          addi x5, x5, -1
(78)          bne  x5, x0, loop
(7c) canceled
(68) loop:    lw   x9, 0(x4)
(6c)          addi x4, x4,  4
```

シミュレーション — Registers




```
(70)          add  x6, x6, x9          (6c)          addi x4, x4,  4
(74)          addi x5, x5, -1          (70)          add  x6, x6, x9
(78)          bne  x5, x0, loop        (74)          addi x5, x5, -1
(7c) canceled                          (78)          bne  x5, x0, loop
(68) loop:    lw   x9, 0(x4)           (7c)          ret  x1
```

シミュレーション — Registers

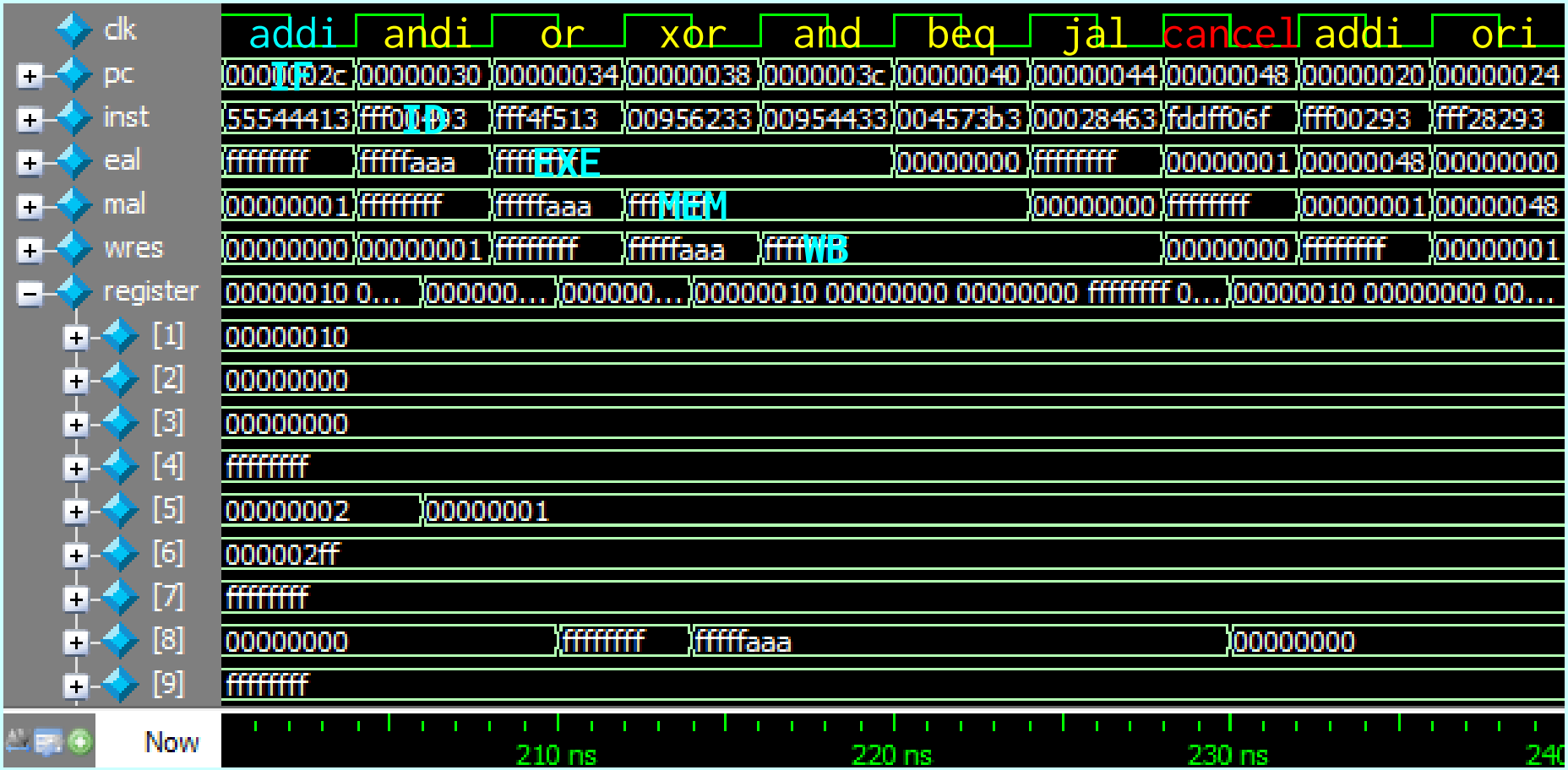



```
(80) canceled                          (1c)          addi x5, x0,  3
(10)          sw   x6, 0(x4)           (20) loop2:   addi x5, x5, -1
(14)          lw   x9, 0(x4)           (24)          ori  x8, x5, -1
(18)          sub  x8, x9, x4          (28)          xori x8, x8, 0x555
(1c) stall due to lw-sub               (2c)          addi x9, x0, -1
```

シミュレーション — Registers

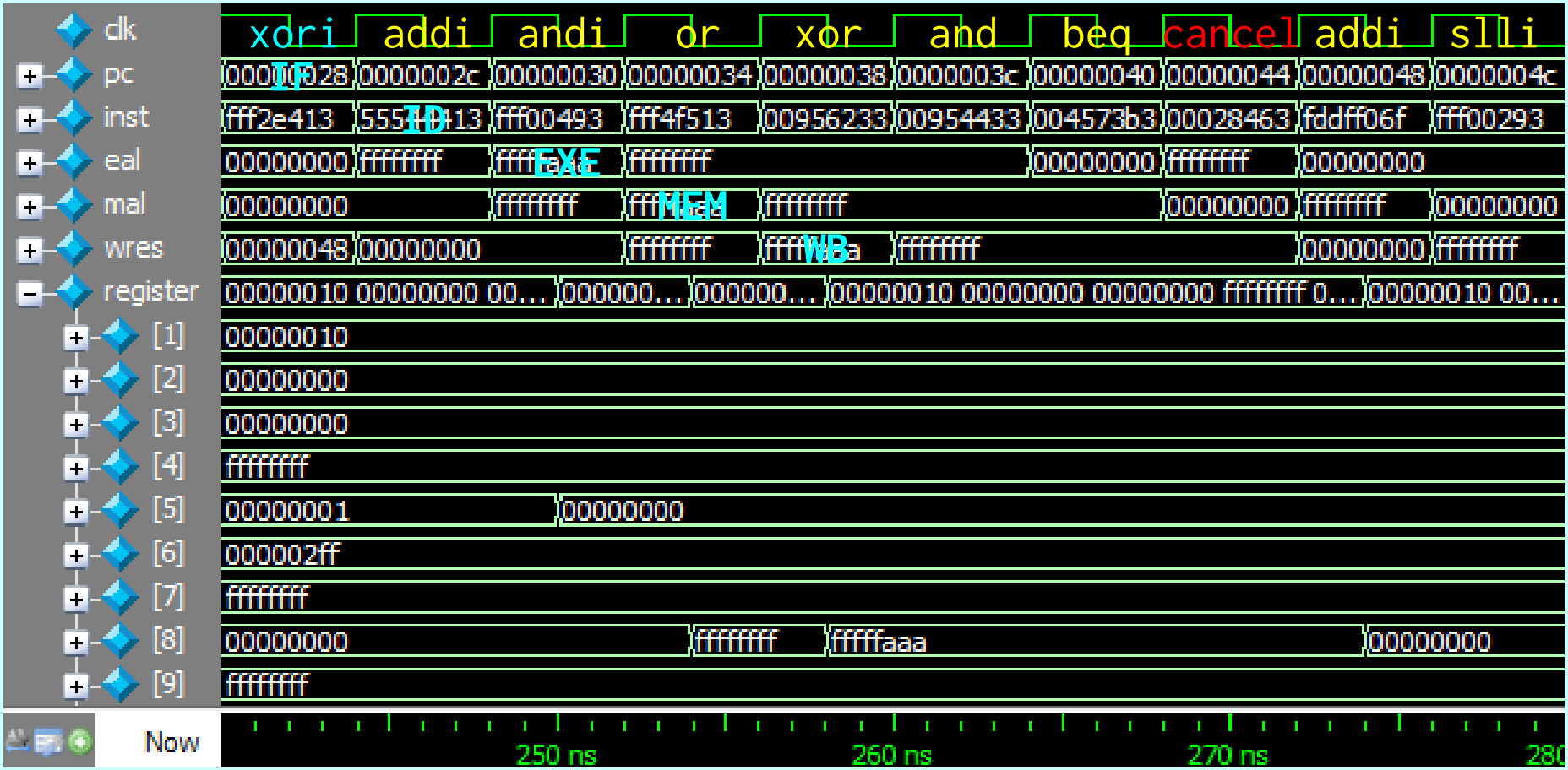

```
(30)          andi x10,x9, -1          (44)          jal  x0, loop2
(34)          or   x4, x10, x9         (48) cancel
(38)          xor  x8, x10, x9         (20) loop2:   addi x5, x5, -1
(3c)          and  x7, x10, x4         (24)          ori  x8, x5, -1
(40)          beq  x5, x0, shift       (28)          xori x8, x8, 0x555
```

シミュレーション — Registers



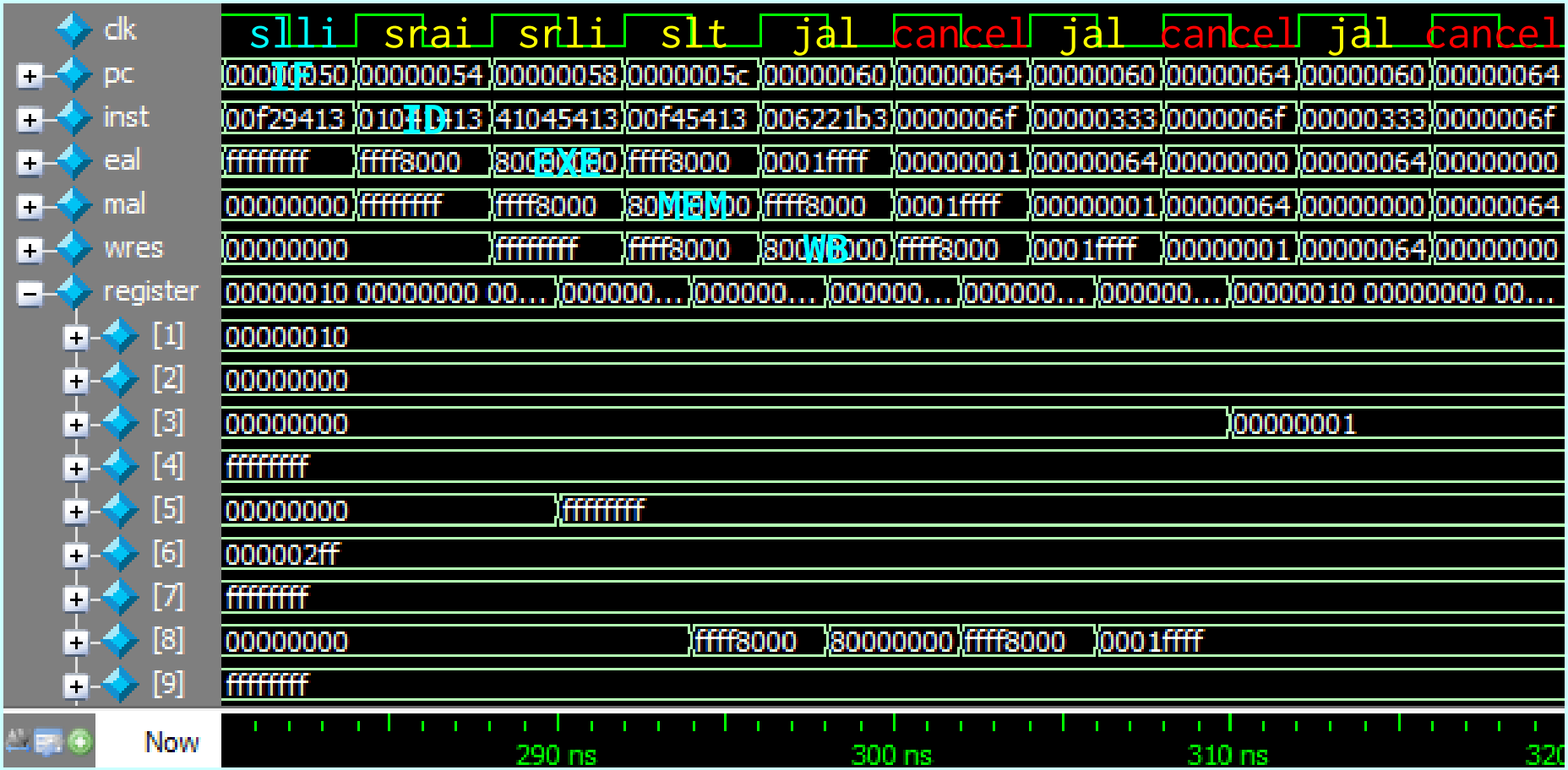

```
(2c)          addi x9, x0, -1          (40)          beq  x5, x0, shift
(30)          andi x10,x9, -1          (44)          jal  x0, loop2
(34)          or   x4, x10, x9         (48) canceled
(38)          xor  x8, x10, x9         (20) loop2:   addi x5, x5, -1
(3c)          and  x7, x10, x4         (24)          ori  x8, x5, -1
```

シミュレーション — Registers



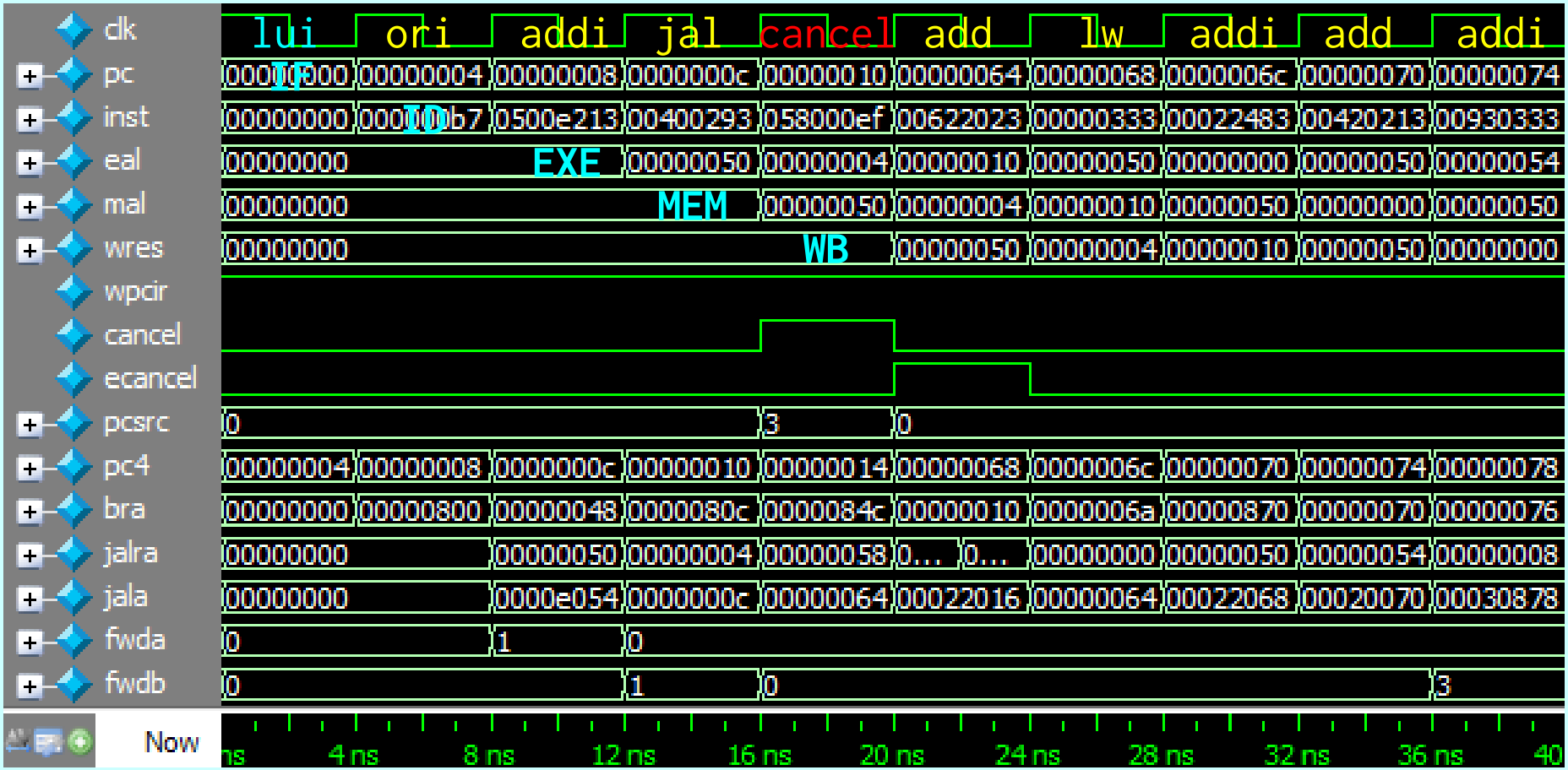

```
(28)          xori x8, x8, 0x555          (3c)          and  x7, x10, x4
(2c)          addi x9, x0, -1             (40)          beq  x5, x0, shift
(30)          andi x10,x9, -1             (44) canceled
(34)          or   x4, x10, x9            (48) shift:   addi x5, x0, -1
(38)          xor  x8, x10, x9            (4c)          slli x8, x5, 15
```

シミュレーション — Registers



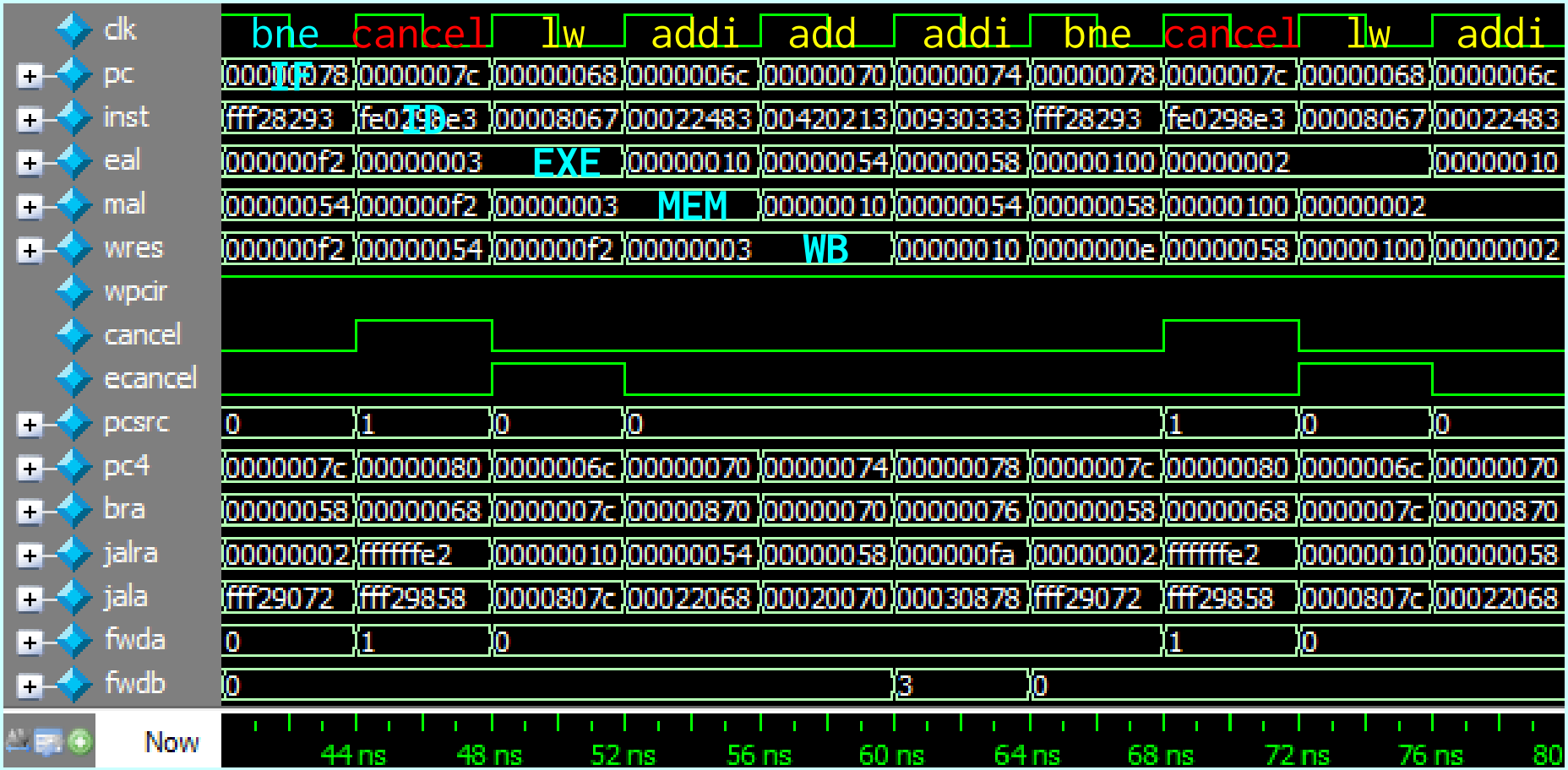

```
(50)          slli x8, x8, 16
(54)          srai x8, x8, 16
(58)          srli x8, x8, 15
(5c)          slt  x3, x4, x6
(60) finish:  jal  x0, finish
(64) canceled
(60) finish:  jal  x0, finish
(64) canceled
(60) finish:  jal  x0, finish
(64) canceled
```

シミュレーション — Branch & Jump




```
(00) main:   lui  x1, 0                (64) sum:    add  x6, x0, x0
(04)         ori  x4, x1, 80           (68) loop:   lw   x9, 0(x4)
(08)         addi x5, x0,  4           (6c)         addi x4, x4,  4
(0c) call:   jal  x1, sum              (70)         add  x6, x6, x9
(10) canceled                          (74)         addi x5, x5, -1
```

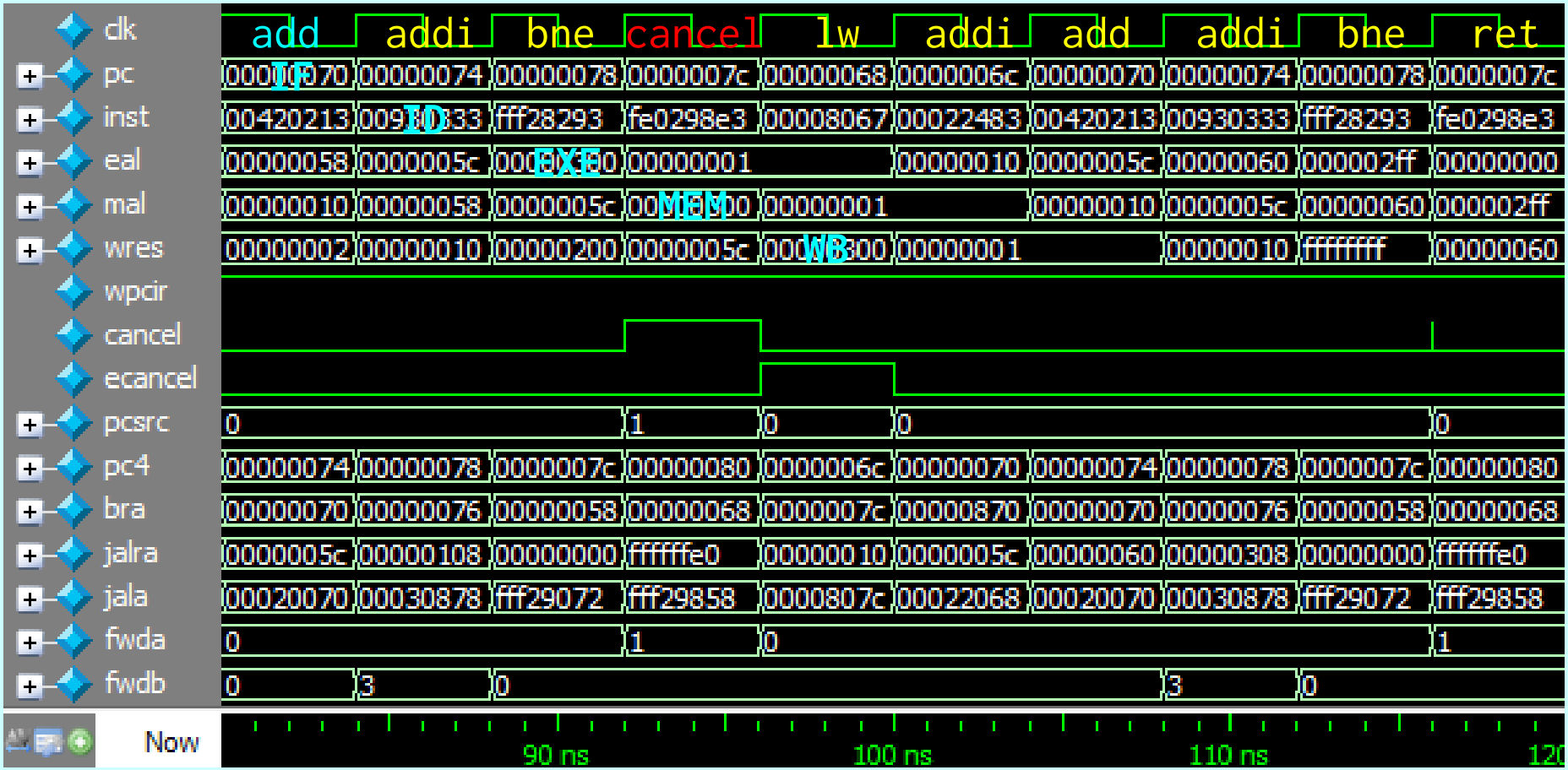
シミュレーション — Branch & Jump




```
(78)          bne  x5, x0, loop
(7c) canceled
(68) loop:    lw   x9, 0(x4)
(6c)          addi x4, x4,  4
(70)          add  x6, x6, x9
```

```
(74)          addi x5, x5, -1
(78)          bne  x5, x0, loop
(7c) canceled
(68) loop:    lw   x9, 0(x4)
(6c)          addi x4, x4,  4
```

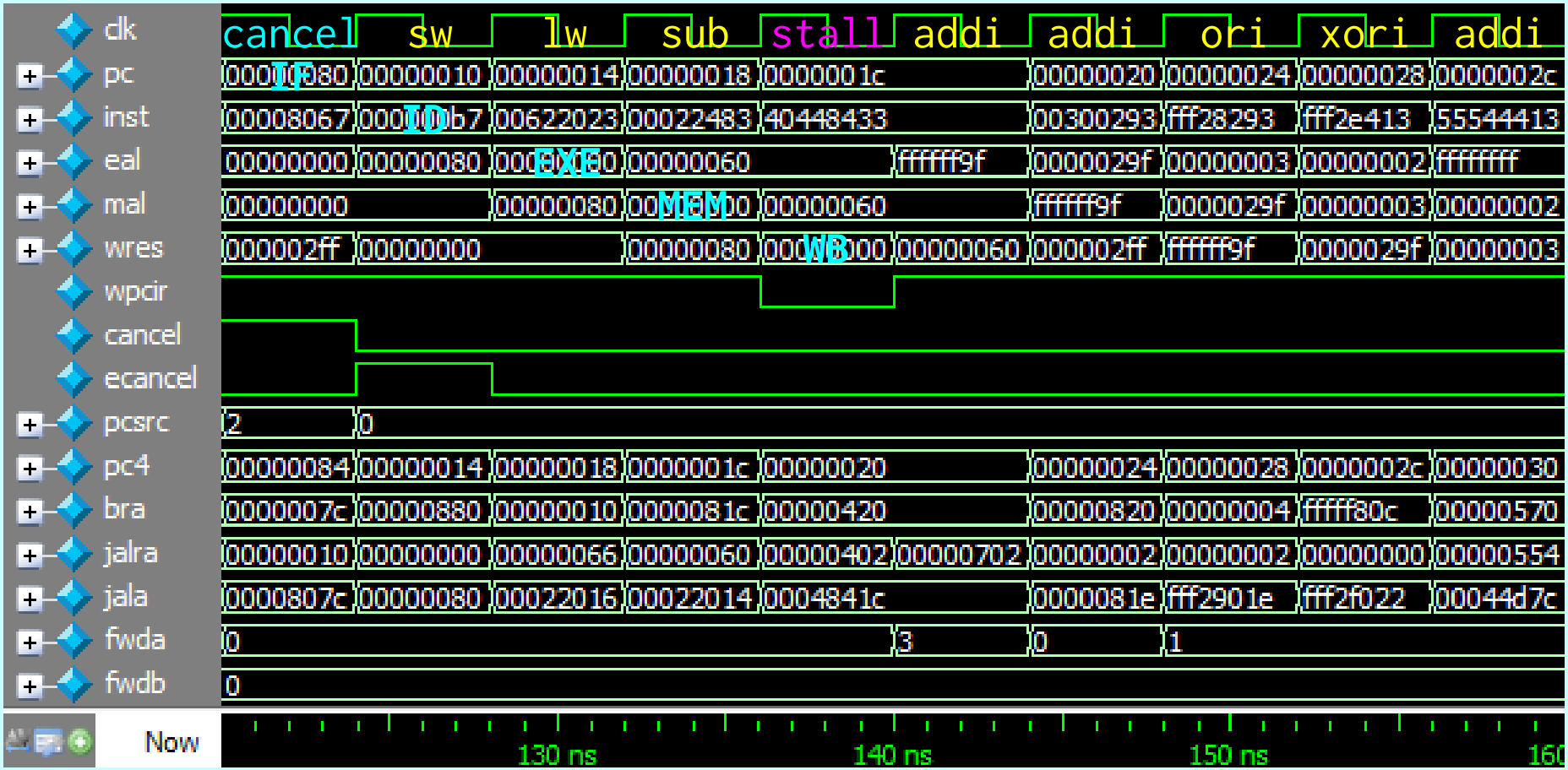
シミュレーション — Branch & Jump




```
(70)          add  x6, x6, x9
(74)          addi x5, x5, -1
(78)          bne  x5, x0, loop
(7c) canceled
(68) loop:    lw   x9, 0(x4)
```

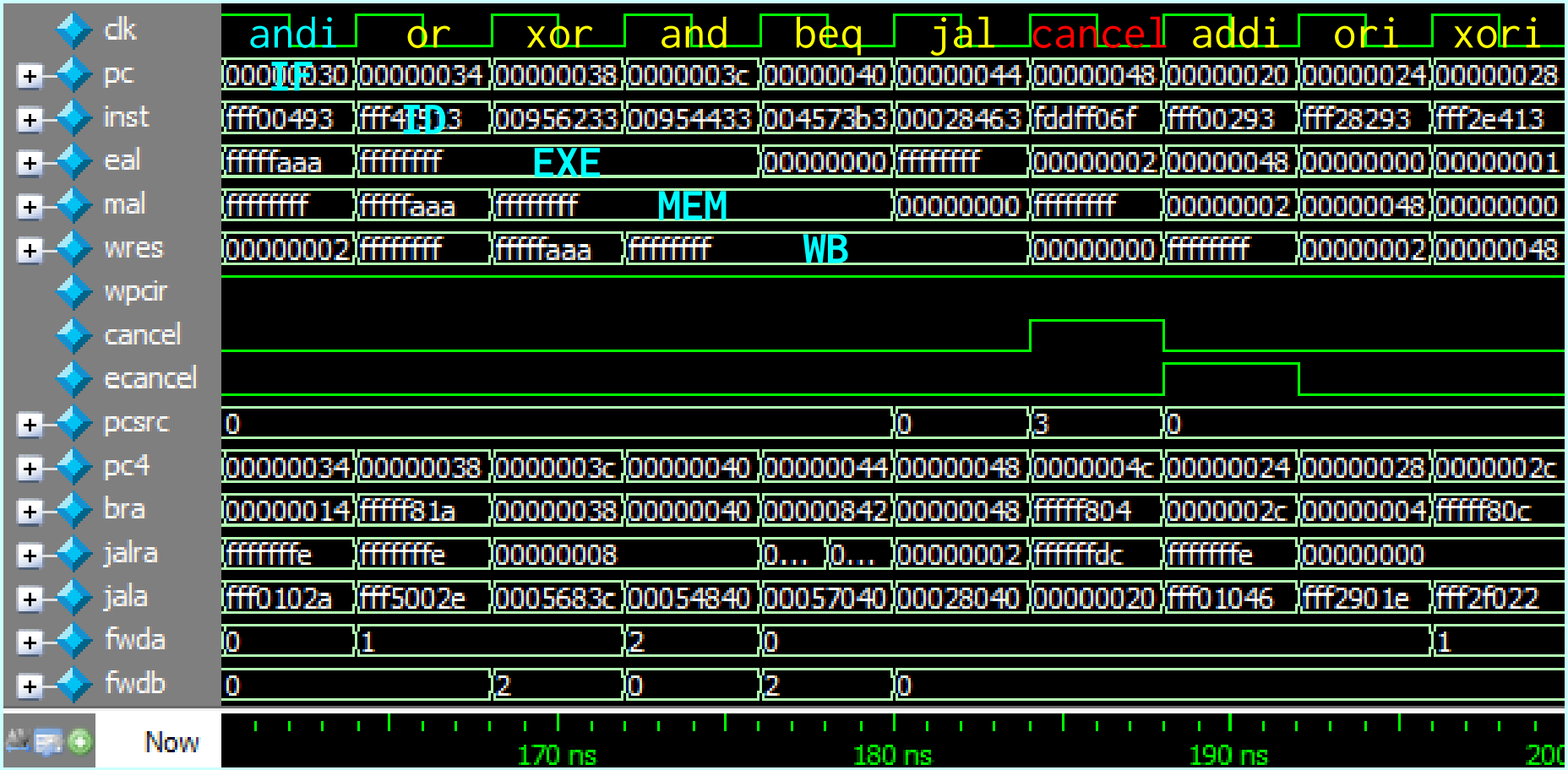
```
(6c)          addi x4, x4,  4
(70)          add  x6, x6, x9
(74)          addi x5, x5, -1
(78)          bne  x5, x0, loop
(7c)          ret  x1
```

シミュレーション — Branch & Jump



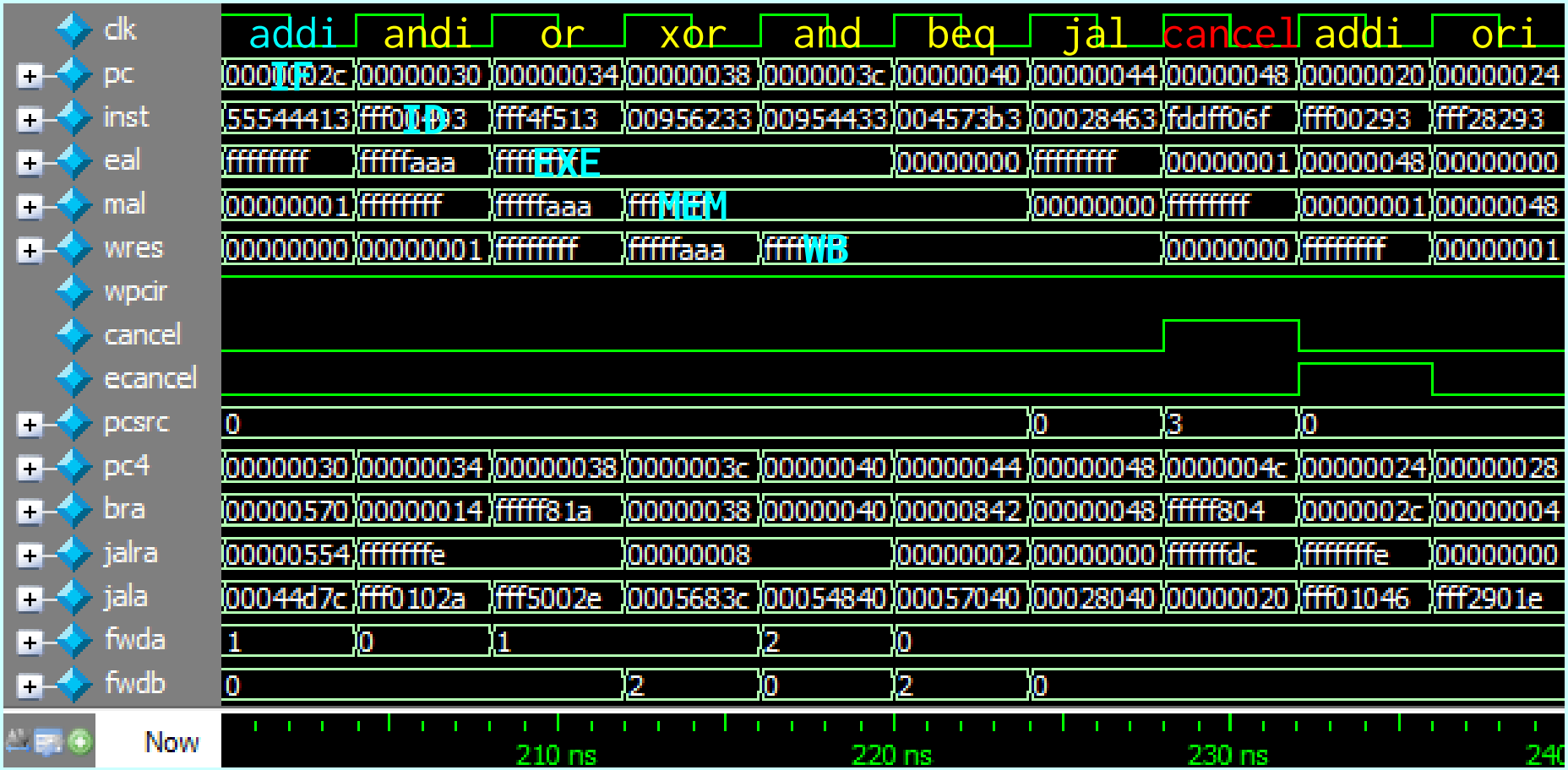

```
(80) canceled                                  (1c)          addi x5, x0,  3
(10)          sw   x6, 0(x4)                   (20) loop2:   addi x5, x5, -1
(14)          lw   x9, 0(x4)                   (24)          ori  x8, x5, -1
(18)          sub  x8, x9, x4                  (28)          xori x8, x8, 0x555
(1c) stall due to lw-sub                       (2c)          addi x9, x0, -1
```

シミュレーション — Branch & Jump




```
(30)          andi x10,x9, -1
(34)          or   x4, x10, x9
(38)          xor  x8, x10, x9
(3c)          and  x7, x10, x4
(40)          beq  x5, x0, shift
(44)               jal  x0, loop2
(48)          cancel
(20)          loop2:  addi x5, x5, -1
(24)                  ori  x8, x5, -1
(28)                  xori x8, x8, 0x555
```

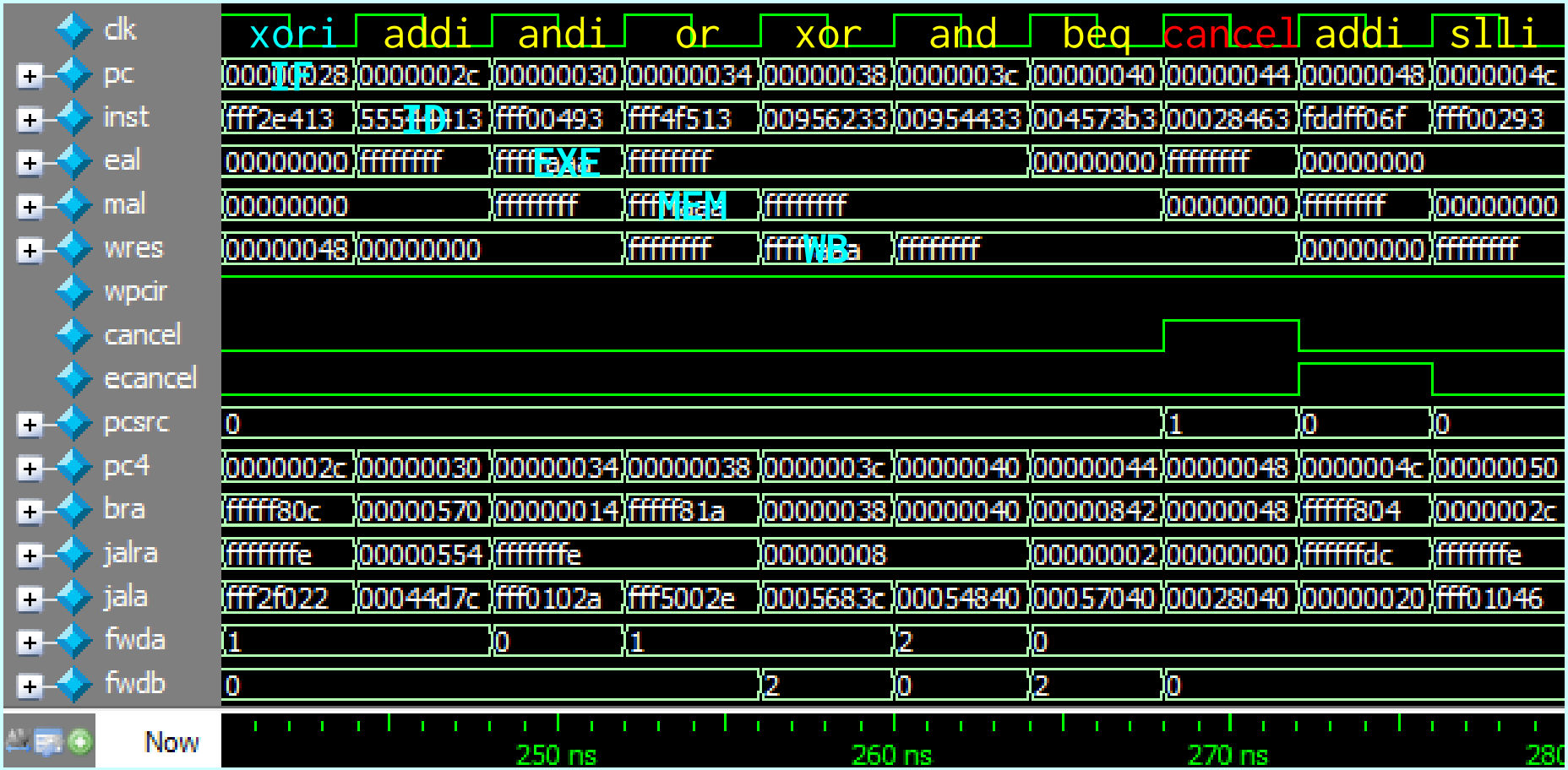
シミュレーション — Branch & Jump




```
(2c)          addi x9, x0, -1
(30)          andi x10,x9, -1
(34)          or   x4, x10, x9
(38)          xor  x8, x10, x9
(3c)          and  x7, x10, x4
```

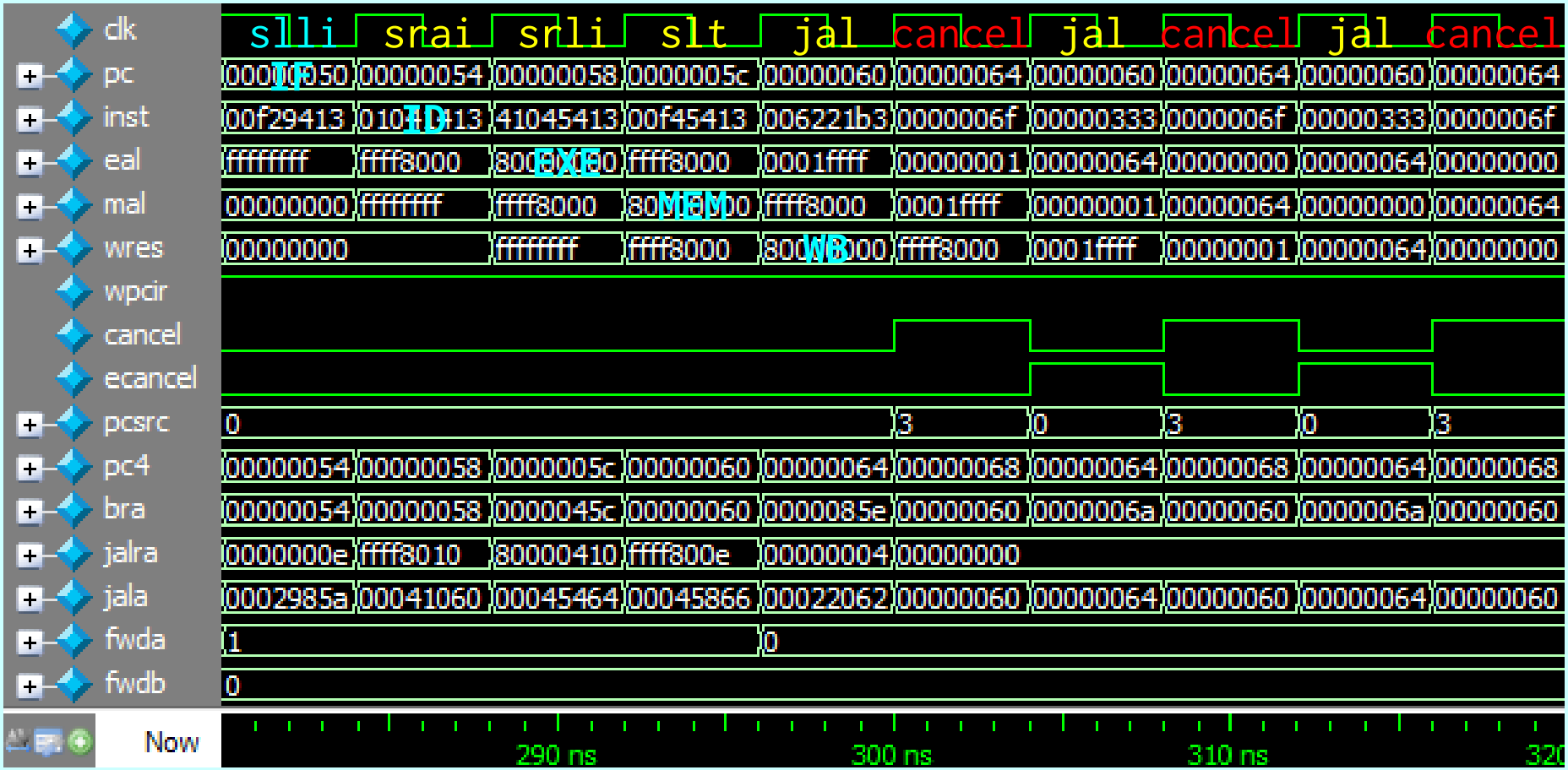
```
(40)          beq  x5, x0, shift
(44)          jal  x0, loop2
(48) canceled
(20) loop2:   addi x5, x5, -1
(24)          ori  x8, x5, -1
```

シミュレーション — Branch & Jump




```
(28)          xori x8, x8, 0x555
(2c)          addi x9, x0, -1
(30)          andi x10,x9, -1
(34)          or   x4, x10, x9
(38)          xor  x8, x10, x9
(3c)          and  x7, x10, x4
(40)          beq  x5, x0, shift
(44) canceled
(48) shift:   addi x5, x0, -1
(4c)          slli x8, x5, 15
```

シミュレーション — Branch & Jump




```
(50)          slli x8, x8, 16
(54)          srai x8, x8, 16
(58)          srli x8, x8, 15
(5c)          slt  x3, x4, x6
(60) finish:  jal  x0, finish
(64) canceled
(60) finish:  jal  x0, finish
(64) canceled
(60) finish:  jal  x0, finish
(64) canceled
```

RV32M (Mul/Div/Rem) Instructions

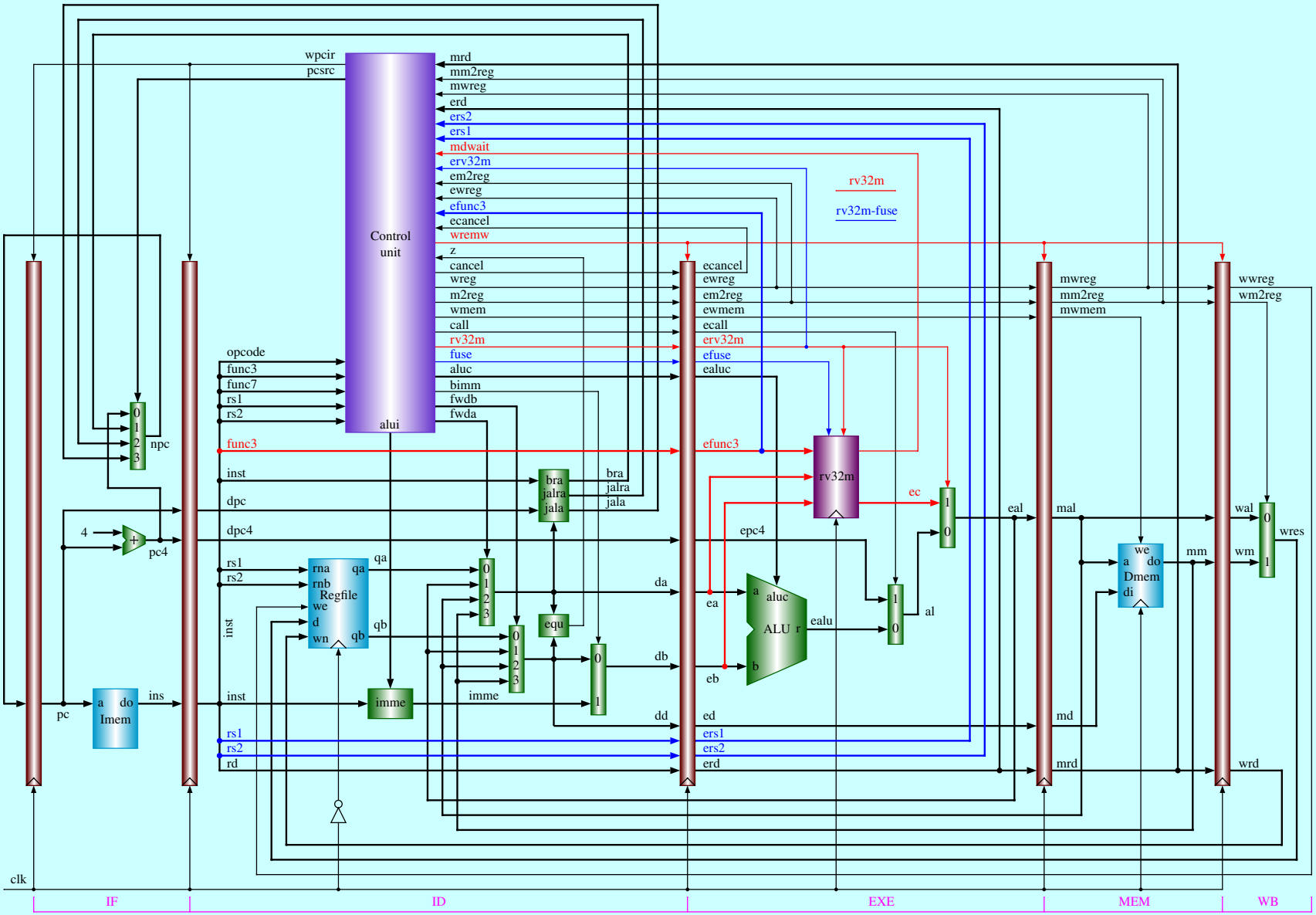
0000001	rs2	rs1	000	rd	0110011	mul
0000001	rs2	rs1	001	rd	0110011	mulh
0000001	rs2	rs1	010	rd	0110011	mulhsu
0000001	rs2	rs1	011	rd	0110011	mulhu
0000001	rs2	rs1	100	rd	0110011	div
0000001	rs2	rs1	101	rd	0110011	divu
0000001	rs2	rs1	110	rd	0110011	rem
0000001	rs2	rs1	111	rd	0110011	remu

```
mul    rd, rs1, rs2   # rd <- rs1 * rs2 (product low, same for signed and unsigned)
mulh   rd, rs1, rs2   # rd <- rs1 * rs2 (product high of   signed *   signed)
mulhsu rd, rs1, rs2   # rd <- rs1 * rs2 (product high of   signed * unsigned)
mulhu  rd, rs1, rs2   # rd <- rs1 * rs2 (product high of unsigned * unsigned)
div    rd, rs1, rs2   # rd <- rs1 / rs2 (  signed)
divu   rd, rs1, rs2   # rd <- rs1 / rs2 (unsigned)
rem    rd, rs1, rs2   # rd <- rs1 % rs2 (  signed)
remu   rd, rs1, rs2   # rd <- rs1 % rs2 (unsigned)
```

RISC-V RV32M Muse

- “If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1, rs2 (source register specifiers must be in same order and rdh cannot be the same as rs1 or rs2). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.”
- “If both the quotient and remainder are required from the same division, the recommended code sequence is: DIV[U] rdq, rs1, rs2; REM[U] rdr, rs1, rs2 (rdq cannot be the same as rs1 or rs2). Microarchitectures can then fuse these into a single divide operation instead of performing two separate divides.”

RISC-V Pipelined RV32IM



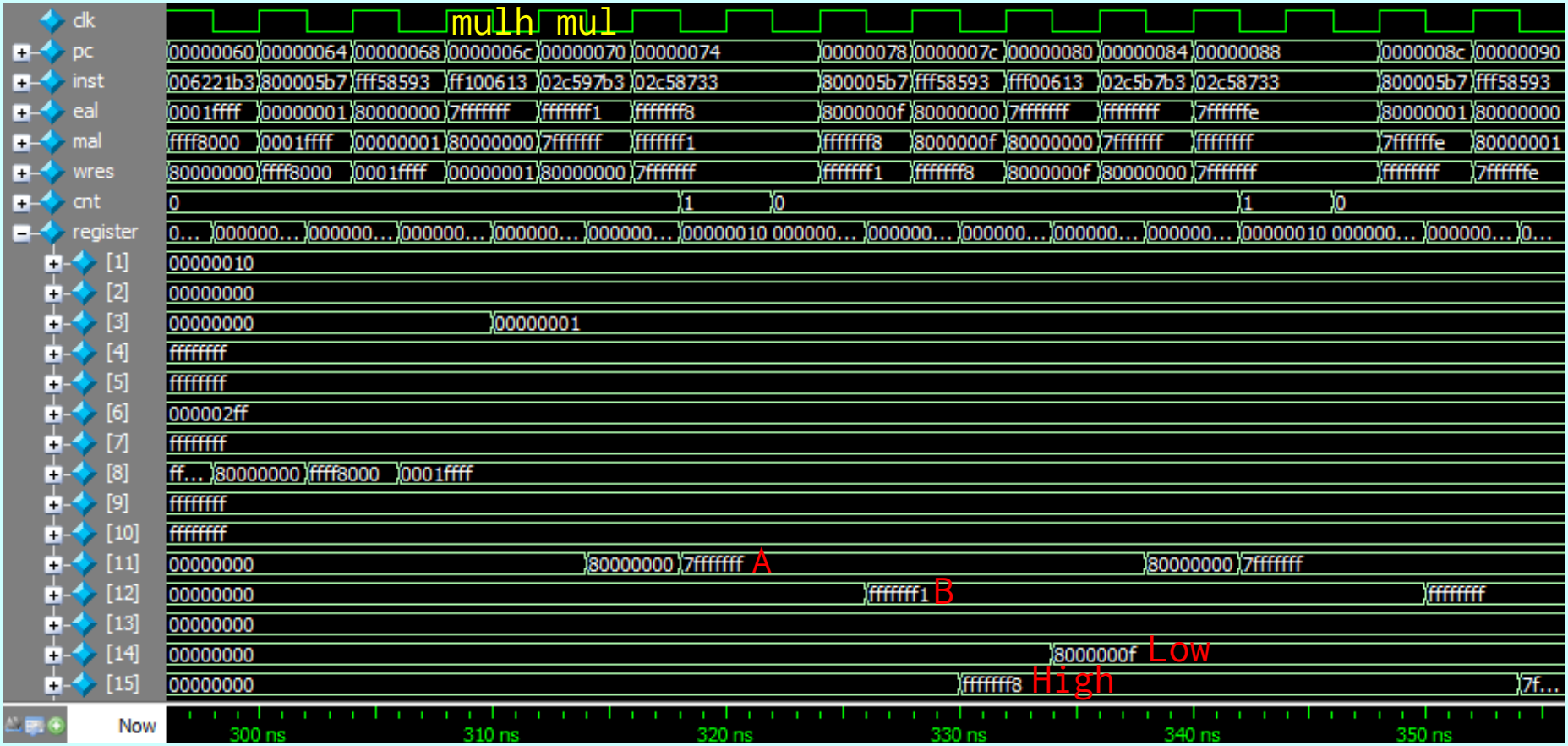
RISC-V RV32M Test Program

```
.text
# ...
s_x_s:  li     a1, 0x7fffffff # (60,64) a            H = 0xfffffff8
        li     a2, -15        # (68) b                L = 0x8000000f
        mulh   a5,  a1, a2    # (6c) product high
        mul    a4,  a1, a2    # (70) product low, fused with mulh
u_x_u:  li     a1, 0x7fffffff # (74,78) a            H = 0x7ffffffe
        li     a2, -1         # (7c) b                L = 0x80000001
        mulhu  a5,  a1, a2    # (80) product high
        mul    a4,  a1, a2    # (84) product low, fused with mulhu
s_x_u:  li     a1, 0x7fffffff # (88,8c) a            H = 0x7ffffffe
        li     a2, -1         # (90) b                L = 0x80000001
        mulhsu a5,  a1, a2    # (94) product high
        mul    a4,  a1, a2    # (98) product low, fused with mulhsu
```

RISC-V RV32M Test Program

```
sign0:  li     a1, 0x7fffffff # (9c,a0) a              Q = 0x2aaaaaaa
        li     a2,   3        # (a4) b                 R = 0x00000001
        div    a5,  a1, a2    # (a8) div signed
        rem    a4,  a1, a2    # (ac) rem signed, fused with div
sign1:  li     a1, 0xfffffff2 # (b0) a                 Q = 0xfffffffc
        li     a2,   3        # (b4) b                 R = 0xfffffffe
        div    a5,  a1, a2    # (b8) div signed
        rem    a4,  a1, a2    # (bc) rem signed, fused with div
unsign: li     a1, 0xfffffff2 # (c0) a                 Q = 0x55555550
        li     a2,   3        # (c4) b                 R = 0x00000002
        divu   a5,  a1, a2    # (c8) div unsigned
        remu   a4,  a1, a2    # (cc) rem unsigned, fused with divu
finish: jal    x0, finish     # (d0) dead loop
# ...
```

Simulation Waveform

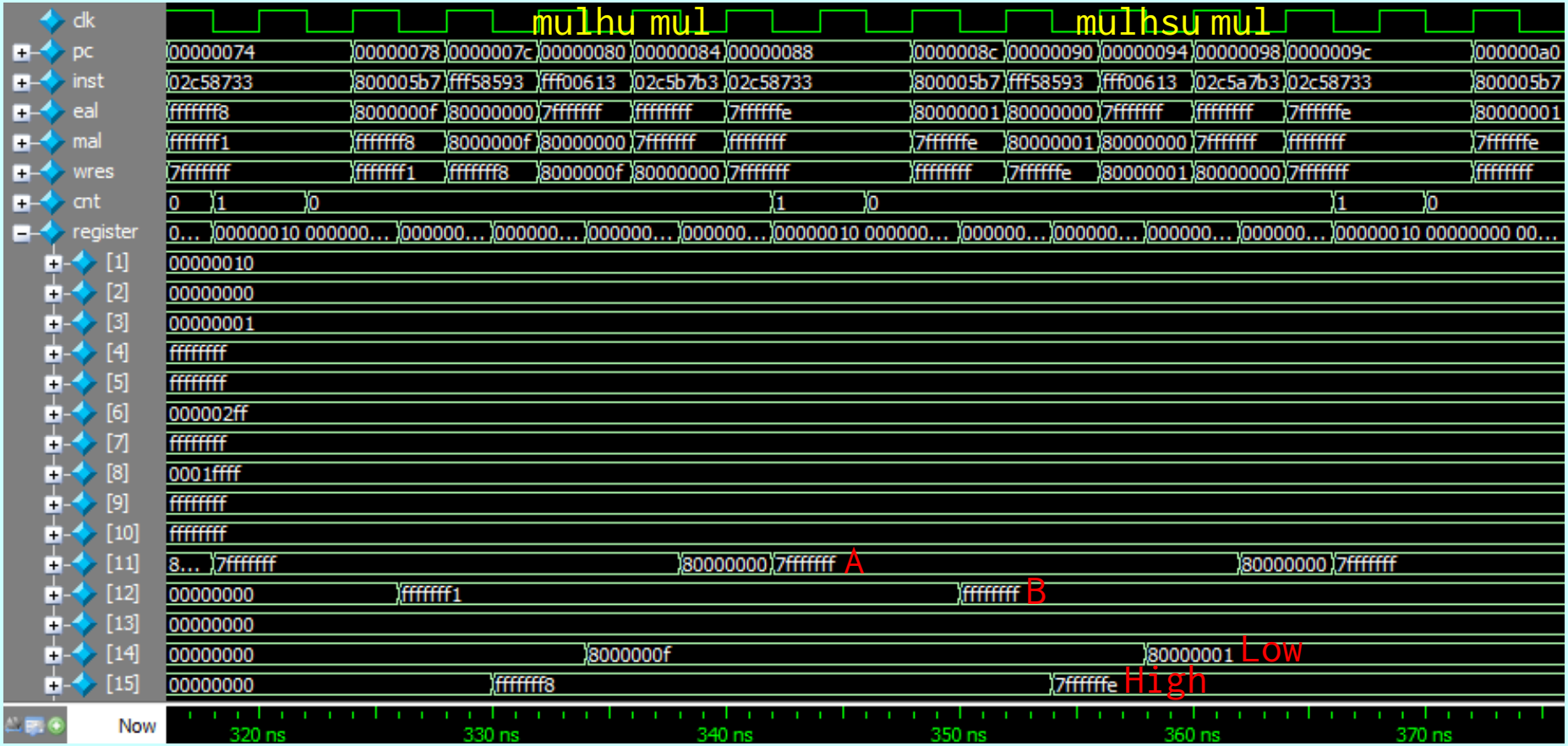


$$\texttt{0x7fffffff} \times \texttt{0xfffffff1} = \texttt{0xfffffff8\_8000000f}$$

A B High Low

signed × signed

Simulation Waveform

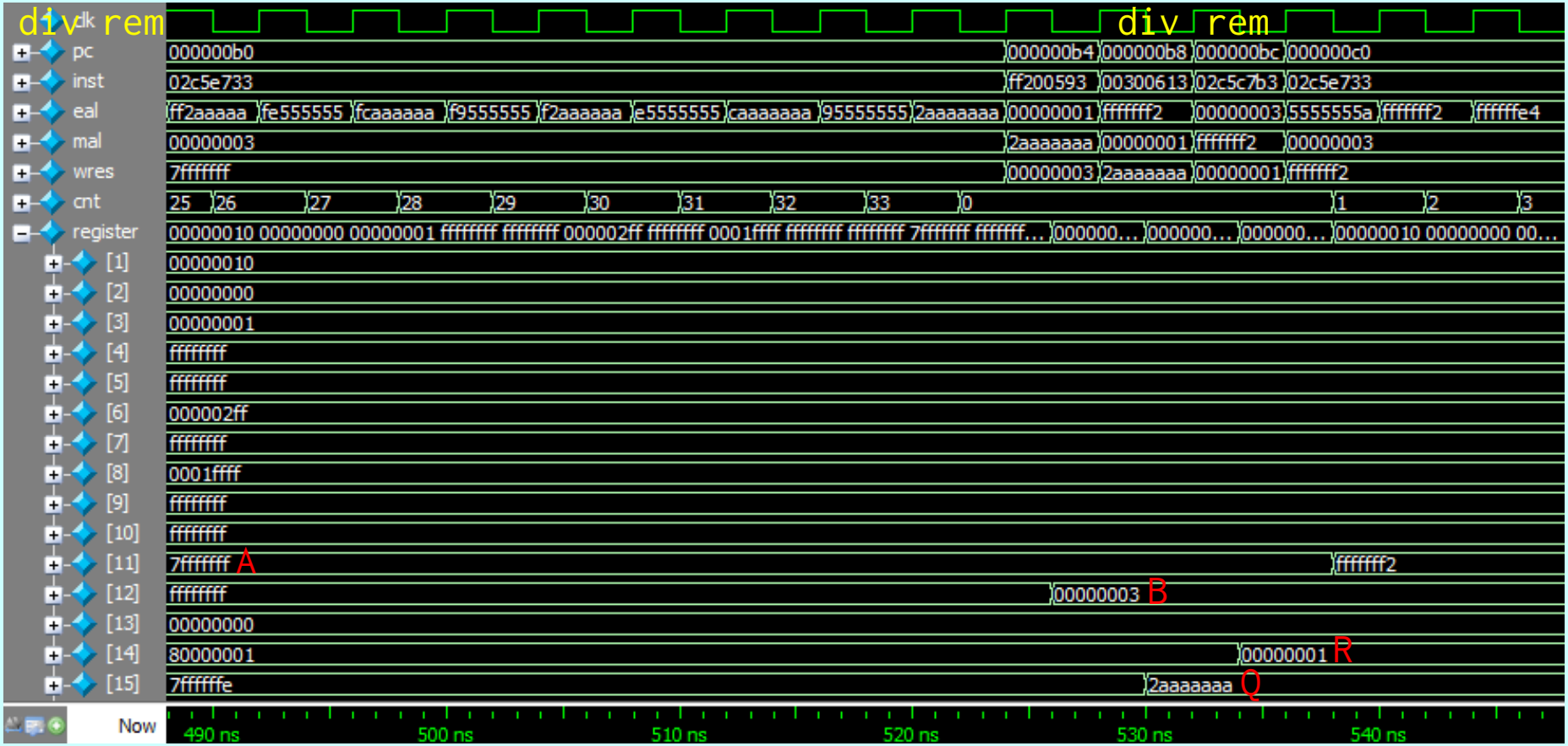


$$\texttt{0x7fffffff} \times \texttt{0xffffffff} = \texttt{0x7ffffffe\_80000001}$$

A B High Low

unsigned × unsigned

Simulation Waveform

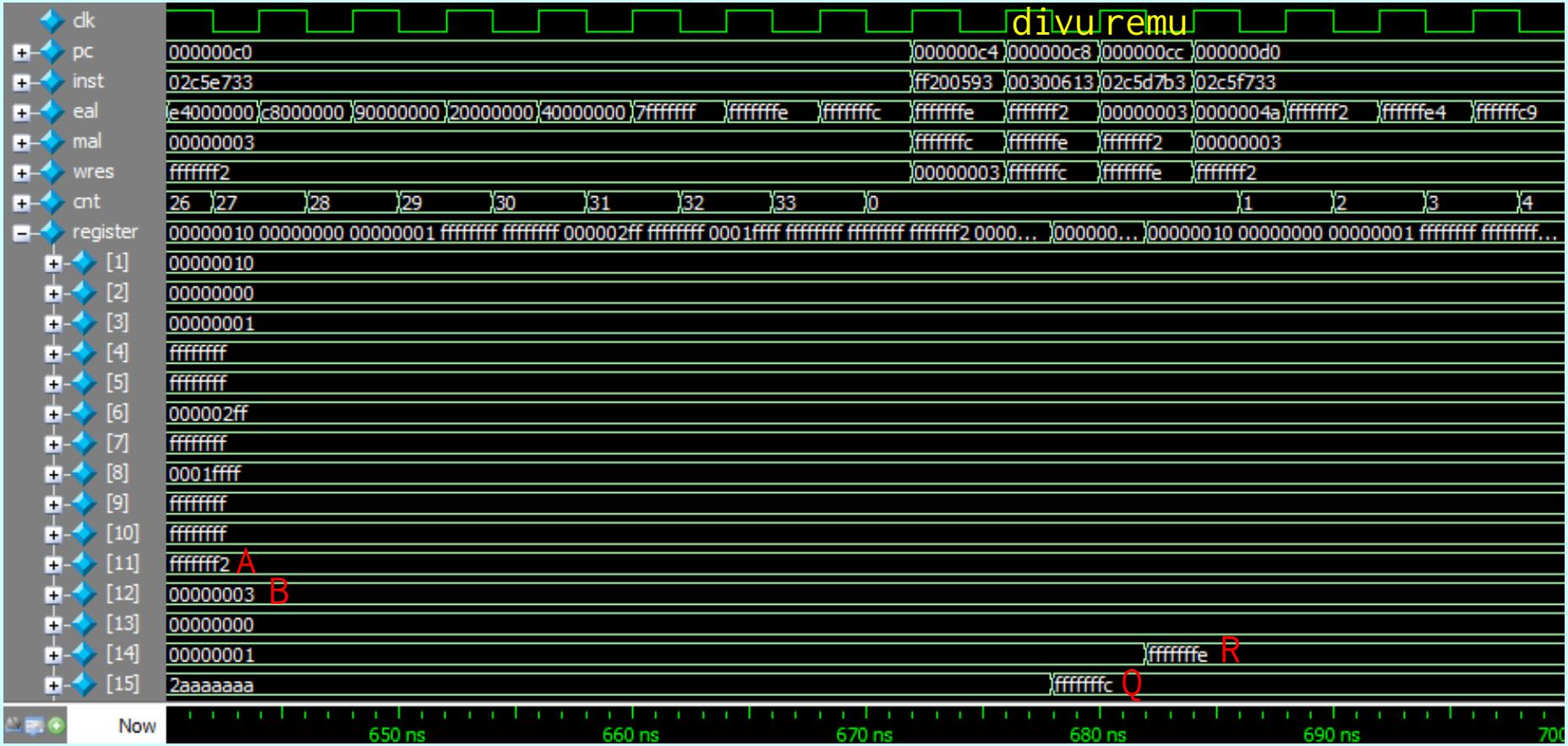


`0x7fffffff / 3 = 0x2aaaaaaa … 0x00000001`

A B Q R

signed / signed

Simulation Waveform

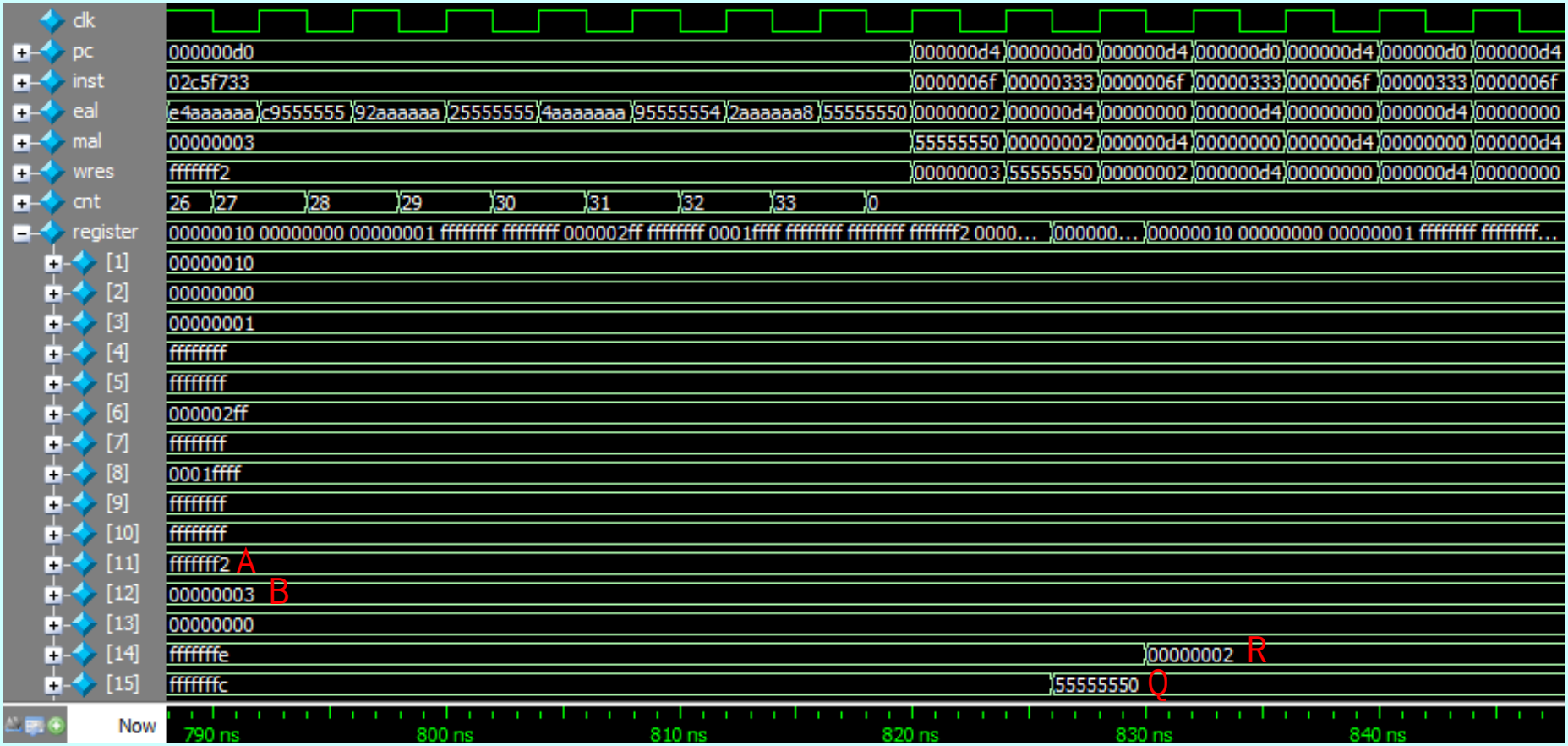


`0xfffffff2 / 3 = 0xfffffffc … 0xfffffffe`

A B Q R

signed / signed

Simulation Waveform



`0xfffffff2 / 3 = 0x55555550 … 0x00000002`

A B Q R

unsigned / unsigned

課題 IX (100 点 + 100 点)

1. RISC-V 単一サイクル CPU、マルチサイクル CPU、パイプライン CPU の 3 つの CPU を設計したとする。単一サイクル CPU のクロック周波数は 200MHz である。他の 2 つの周波数は両方とも 1GHz である。マルチサイクル CPU では、「lw」命令は 5 クロックサイクル、「bne」は 3 サイクル、残りは 4 サイクルかかる。パイプライン CPU は、内部転送、およびパイプラインストールの手法を使用する。3 つの CPU が次のプログラムを実行するときの実行時間（ns）を計算しなさい。

```
.text
main:
    lui  x4, %hi(array)     # address of array[0]
    addi x4, x4, %lo(array) # address of array[0]
    add  x2, x0, x0         # sum = 0
    addi x5, x0, 5          # counter = 5
loop:
    lw   x8, 0(x4)          # load array[i] from memory
    add  x2, x2, x8         # sum = sum + array[i]
    addi x4, x4, 4          # address + 4
    addi x5, x5, -1         # counter--
    bne  x5, x0, loop       # if counter != 0, go to loop
    sw   x2, 0(x4)
.data
array: .word 6, 5, -3, 9, 7, 0
.end
```

さらに、命令の順序を変更して、ロード命令によって引き起こされるパイプラインストールを排除する。そして、実行時間を再計算しなさい。

課題 IX (100 点 + 100 点)

2. オプション（+50 点）: パイプライン CPU RV32I を設計とシミュレーションしなさい。

   RV32I: Integer

3. オプション（+50 点）: パイプライン CPU RV32IM を設計とシミュレーションしなさい。

   RV32I: Integer

   RV32M: Multiplication/Division (see P53)