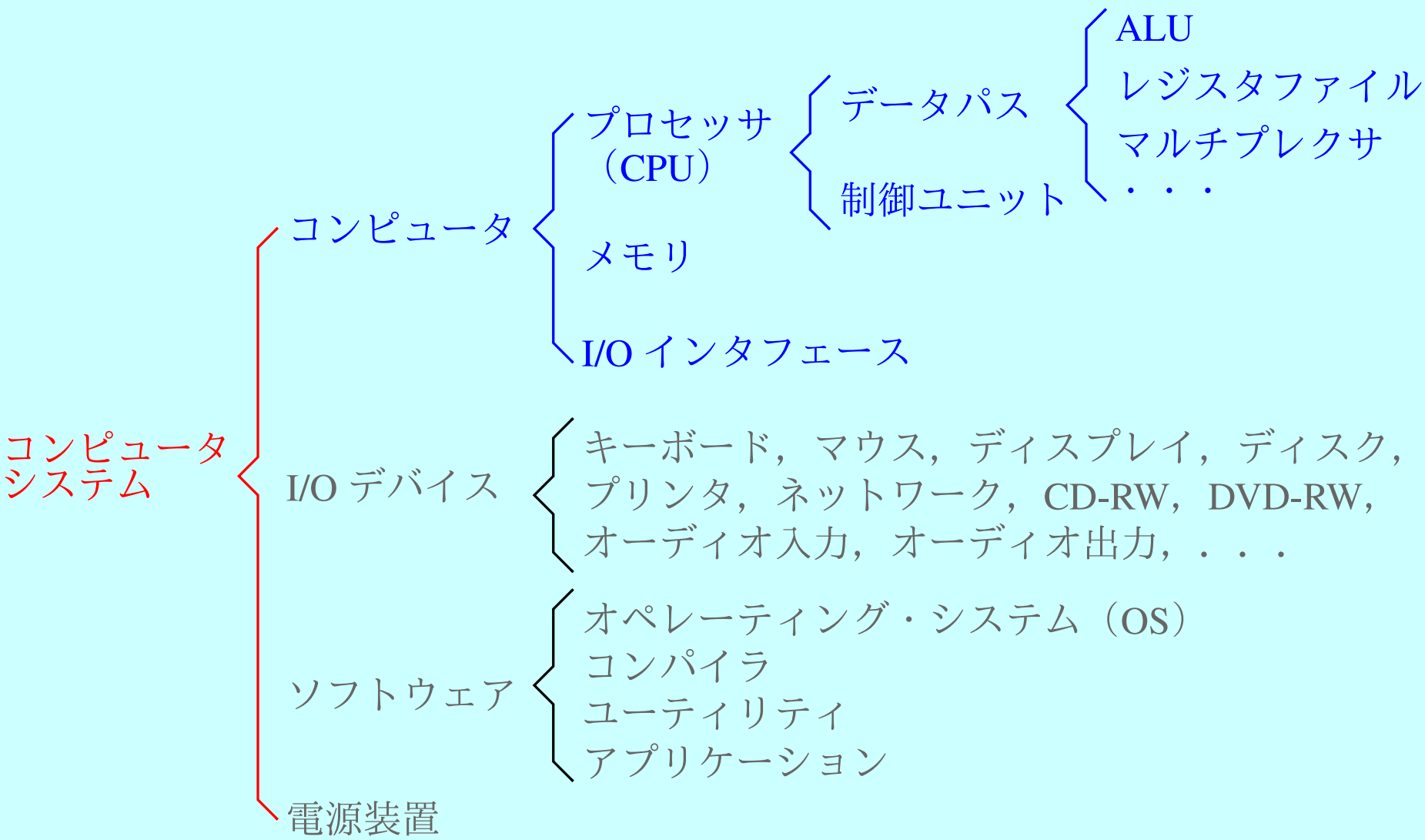
コンピュータ構成と設計（６）

単一サイクル CPU 設計

李　亜民

2024 年 11 月 7 日 (木)

コンピュータとコンピュータシステム



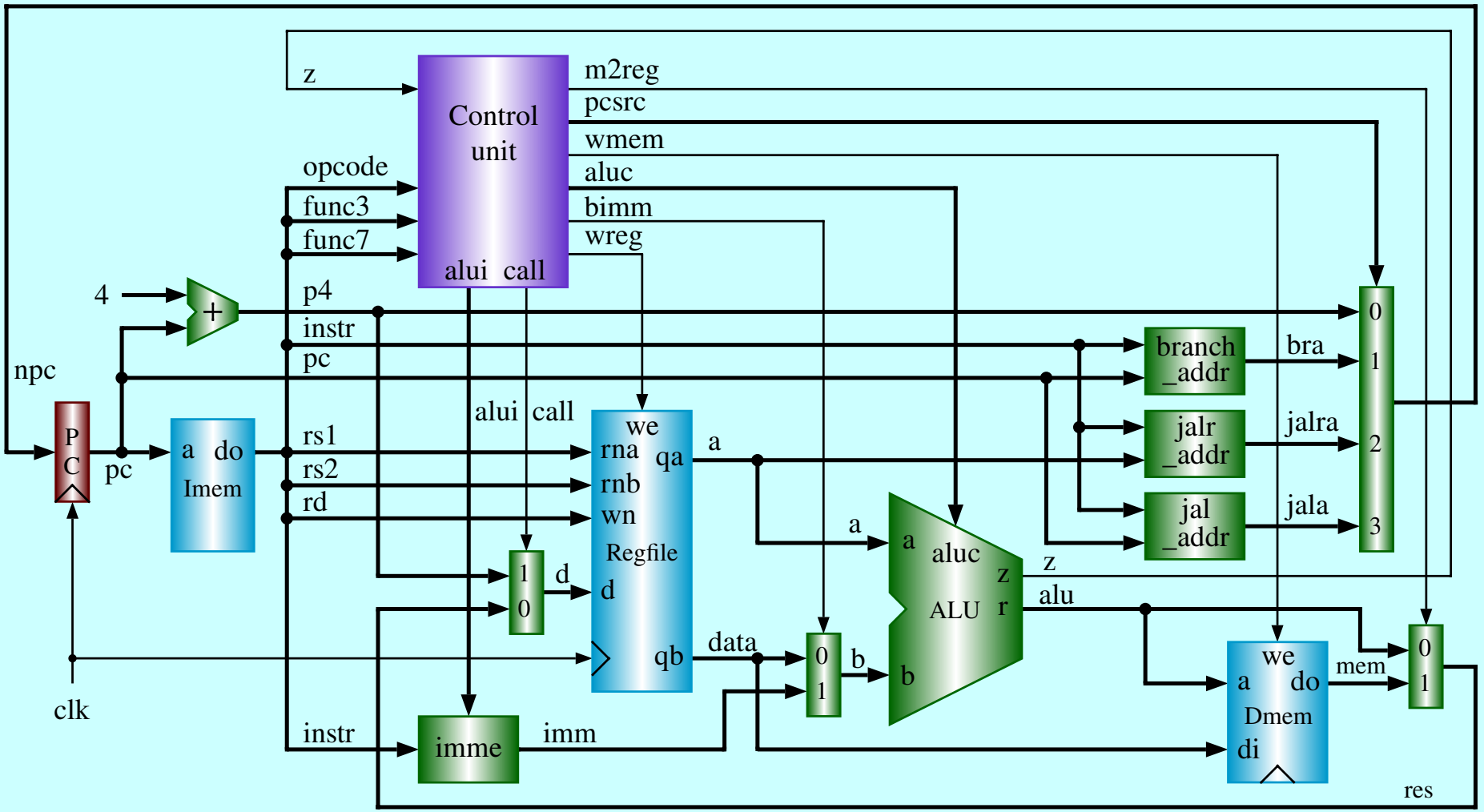
20 RISC-V 命令のまとめ

```
 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

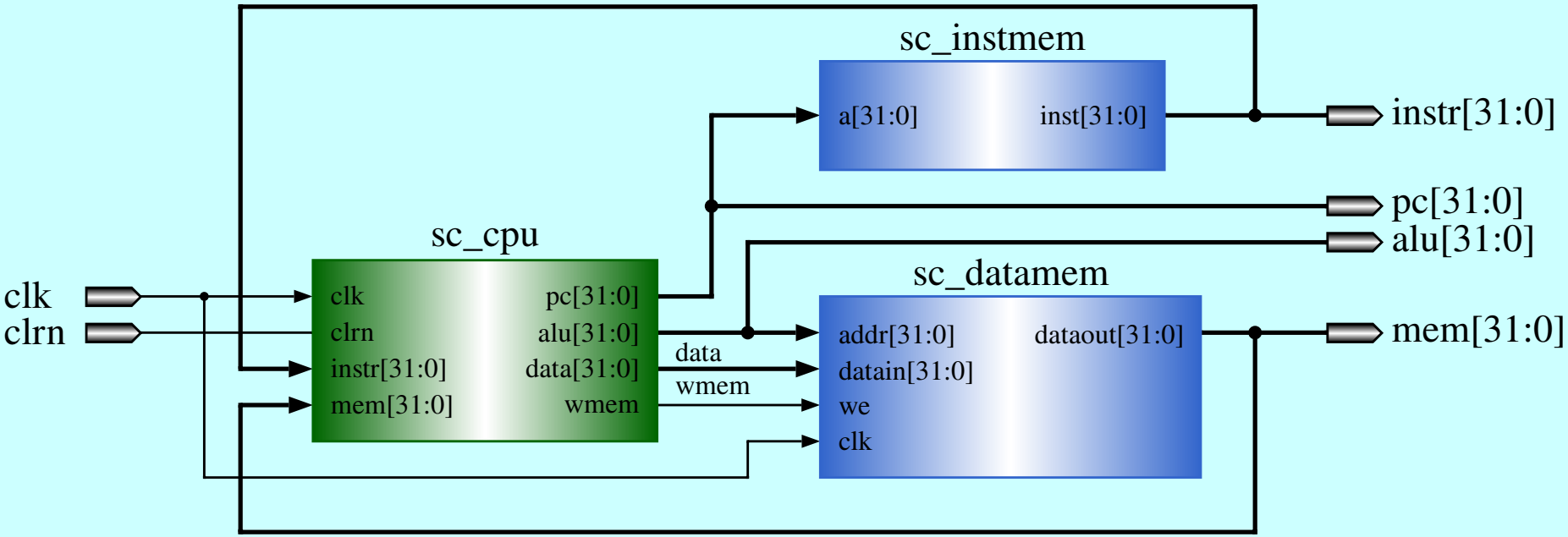
RISC-V コンピュータ



CPU + 命令メモリ Imem + データメモリ Dmem

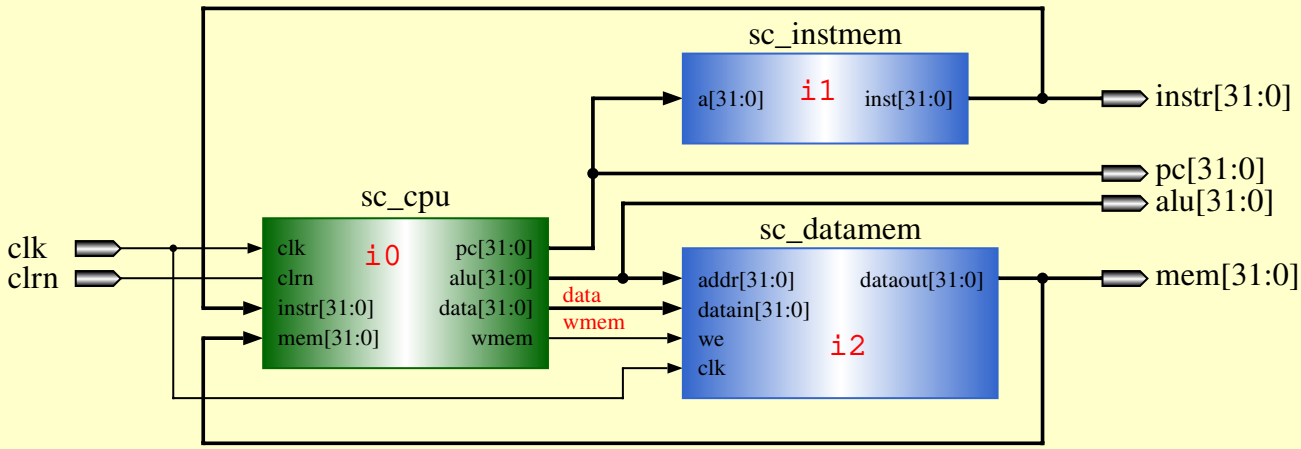
RISC-V CPU とメモリの回路

単一サイクル RISC-V CPU + 命令メモリ + データメモリ



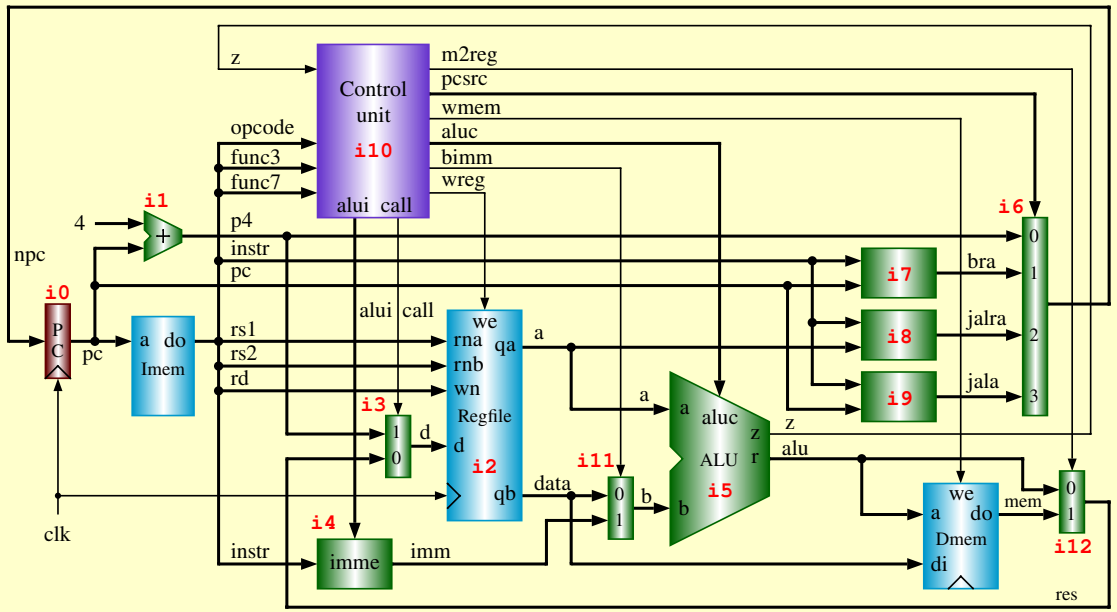
RISC-V sc_computer の回路

```
‘timescale 1ns/1ns
module sc_computer(clk,
                   clrn,
                   alu,
                   instr,
                   mem,
                   pc
                   );

    input         clk;
    input         clrn;
    output [31:0] alu;
    output [31:0] instr;
    output [31:0] mem;
    output [31:0] pc;

    wire          wmem;
    wire   [31:0] data;

    sc_cpu     i0 (.clk(clk),
                   .clrn(clrn),
                   .instr(instr),
                   .mem(mem),
                   .wmem(wmem),
                   .alu(alu),
                   .data(data),
                   .pc(pc));

    sc_instmem i1 (.a(pc),
                   .inst(instr));

    sc_datamem i2 (.we(wmem),  // .we: signal of sc_datamem; wmem: wire of sc_computer
                   .clk(clk),
                   .addr(alu),
                   .datain(data),
                   .dataout(mem));
endmodule
```



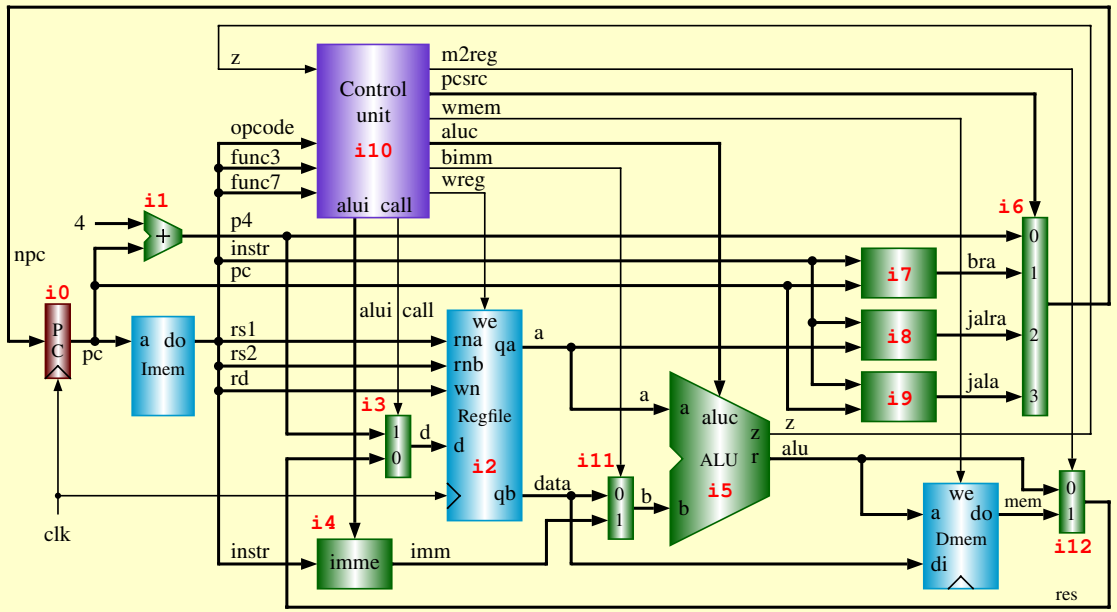
RISC-V sc_cpu の回路

```
‘timescale 1ns/1ns
module sc_cpu (
               clk,
               clrn,
               instr,
               mem,
               wmem,
               alu,
               data,
               pc
               );

    input          clk;
    input          clrn;
    input  [31:0] instr;
    input  [31:0] mem;
    output         wmem;
    output [31:0] alu;
    output [31:0] data;
    output [31:0] pc;

    wire   [31:0] a;
    wire   [31:0] b;
    wire    [3:0] aluc;
    wire    [1:0] alui;
    wire          bimm;
    wire   [31:0] bra;
    wire          call;
    wire   [31:0] d;
    wire   [31:0] imm;
    wire   [31:0] jala;
    wire   [31:0] jalra;
```



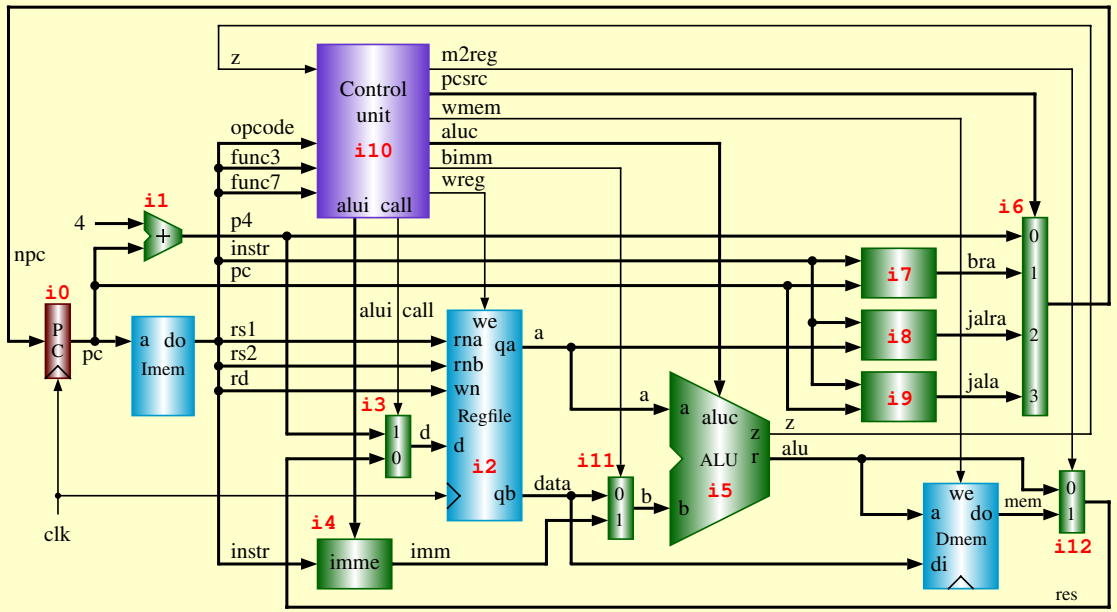
RISC-V sc_cpu の回路

```
wire          m2reg;
wire   [31:0] npc;
wire   [31:0] p4;
wire    [1:0] pcsrc;
wire   [31:0] res;
wire          wreg;
wire          z;

dff32 i0 (
        .clk(clk),
        .clrn(clrn),
        .d(npc),
        .q(pc));

pc4   i1 (
        .pc(pc),
        .p4(p4));

sc_cu i10 (
        .z(z),
        .func3(instr[14:12]),
        .func7(instr[31:25]),
        .opcode(instr[6:0]),
        .m2reg(m2reg),
        .bimm(bimm),
        .call(call),
        .wreg(wreg),
        .wmem(wmem),
        .aluc(aluc),
        .alui(alui),
        .pcsrc(pcsrc));
```



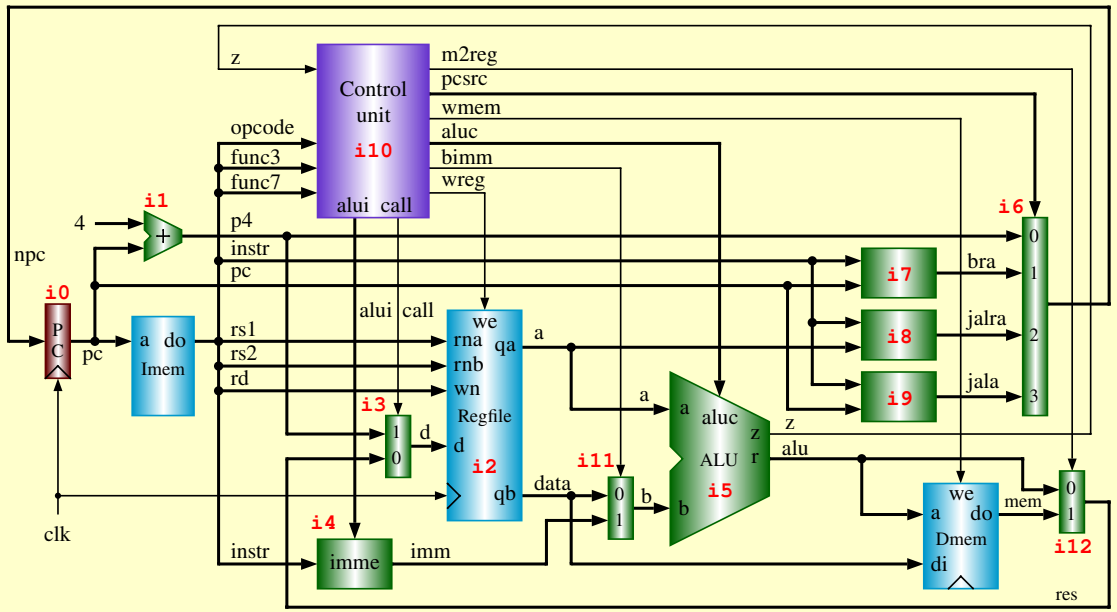
RISC-V sc_cpu の回路

```
mux2x32 i11 (
        .s(bimm),
        .a0(data),
        .a1(imm),
        .y(b));

mux2x32 i12 (
        .s(m2reg),
        .a0(alu),
        .a1(mem),
        .y(res));

alu i5 (
        .a(a),
        .aluc(aluc),
        .b(b),
        .z(z),
        .r(alu));

regfile i2 (
        .we(wreg),
        .clk(clk),
        .clrn(clrn),
        .d(d),
        .rna(instr[19:15]),
        .rnb(instr[24:20]),
        .wn(instr[11:7]),
        .qa(a),
        .qb(data));
```



RISC-V sc_cpu の回路

```
    mux2x32 i3 (
            .s(call),
            .a0(res),
            .a1(p4),
            .y(d));

    imme i4 (
            .alui(alui),
            .inst(instr),
            .imm(imm));

    mux4x32 i6 (
            .a0(p4),
            .a1(bra),
            .a2(jalra),
            .a3(jala),
            .s(pcsrc),
            .y(npc));

    branch_addr i7 (
            .inst(instr),
            .pc(pc),
            .addr(bra));

    jalr_addr i8 (
            .inst(instr),
            .rs1(a),
            .addr(jalra));

    jal_addr i9 (
            .inst(instr),
            .pc(pc),
            .addr(jala));
endmodule
```

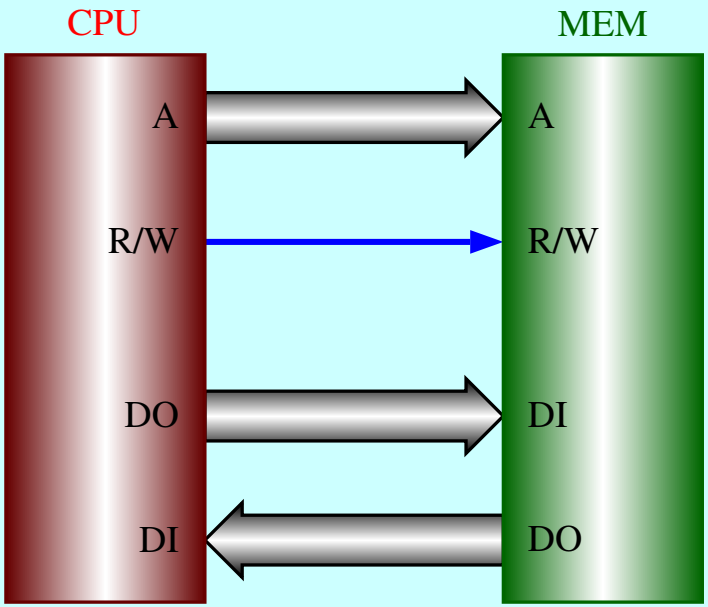


メモリ容量

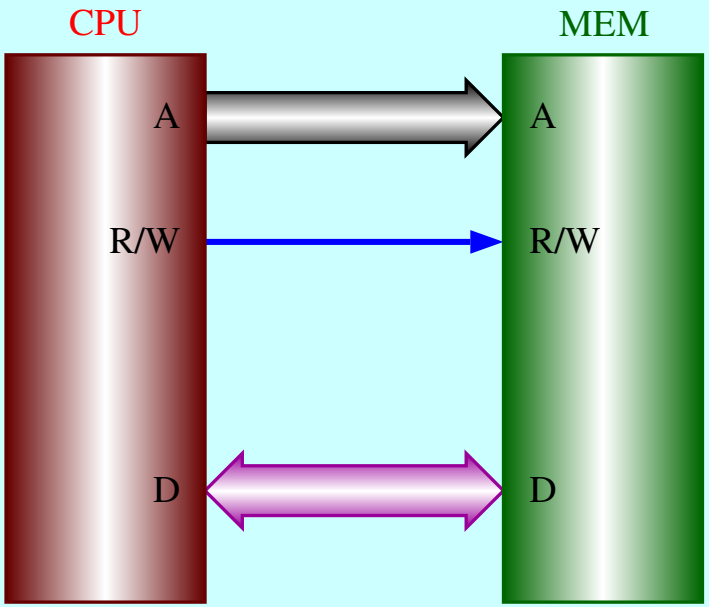
- メモリ: コンピュータ内でデータやプログラムを記憶する装置
- メモリ容量はバイトで表される。1バイトは8ビットに等しい
- 例えば，“メイン・メモリの容量は16ギガ・バイトである”という感じに使う

単位	10進法での意味	2進法での意味
K（キロ）	10^3	$2^{10} = 1\,024$
M（メガ）	10^6	$2^{20} = 1\,048\,576$
G（ギガ）	10^9	$2^{30} = 1\,073\,741\,824$
T（テラ）	10^{12}	$2^{40} = 1\,099\,511\,627\,776$

メモリとCPU



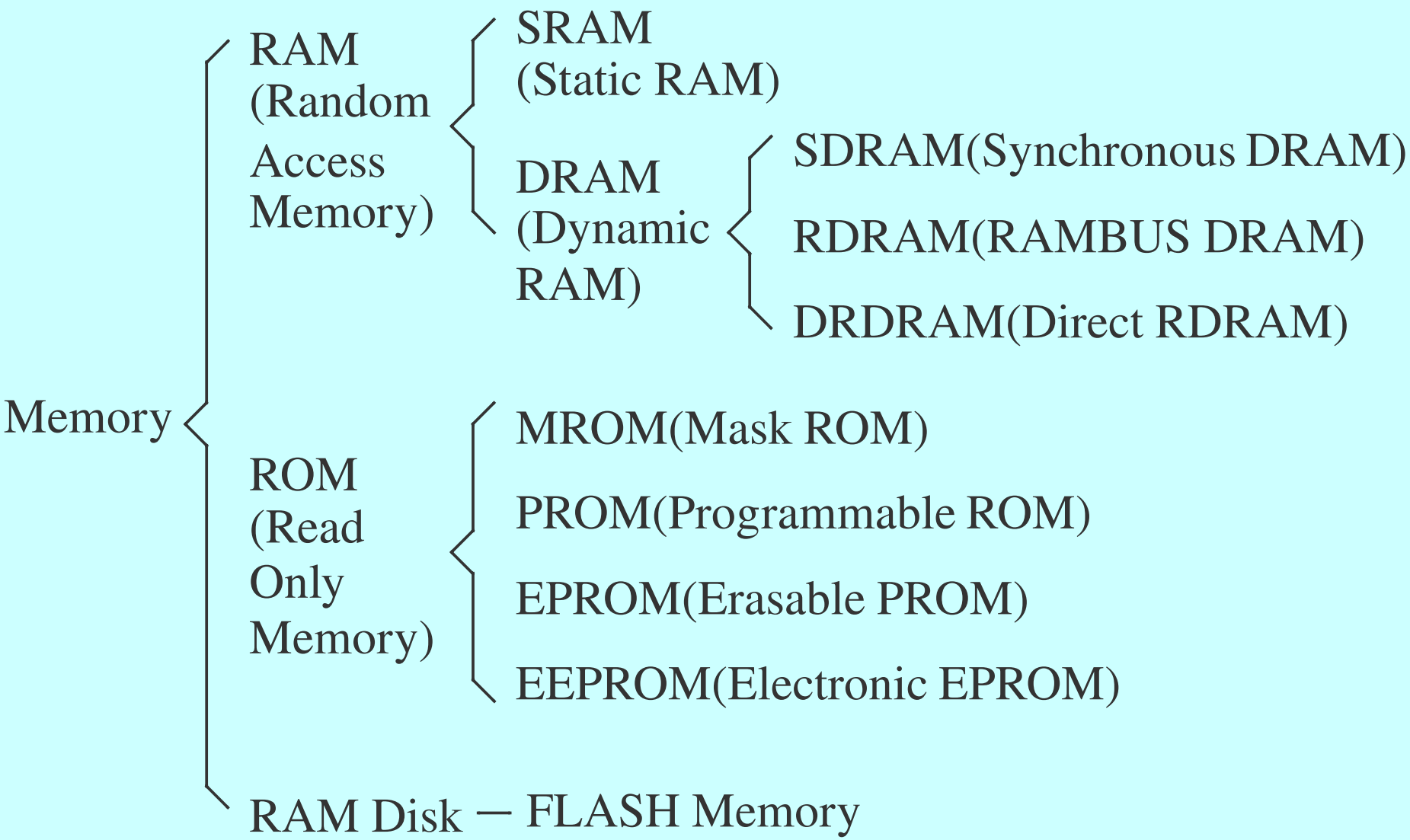
(a) 独立の単方向データバス



(b) 単一の双方向データバス

A (Address) : アドレスバス
D (Data) : データバス
DI (Data In) : データバス
DO (Data Out) : データバス
R/W (Read/Write) : リード/ライト信号

メモリの種類


Memory
RAM (Random Access Memory)
SRAM (Static RAM)
DRAM (Dynamic RAM)
SDRAM(Synchronous DRAM)
RDRAM(RAMBUS DRAM)
DRDRAM(Direct RDRAM)
ROM (Read Only Memory)
MROM(Mask ROM)
PROM(Programmable ROM)
EPROM(Erasable PROM)
EEPROM(Electronic EPROM)
RAM Disk ― FLASH Memory

命令メモリ (テスト・プログラム)

```
‘timescale 1ns/1ns
module sc_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’b00000101000000000111000001000011; // (04)         ori  x4, x1, 80    # x1 <- 80
    assign rom[5’h02] = 32’b00000000010000000000000010010011; // (08)         addi x5, x0,  4    # x5 <- 4
    assign rom[5’h03] = 32’b00000101110000000000000011101111; // (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
    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’b00000000000000000000001100110011; // (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
```

sc_instmem.s.txt

データメモリ (テスト・データ)

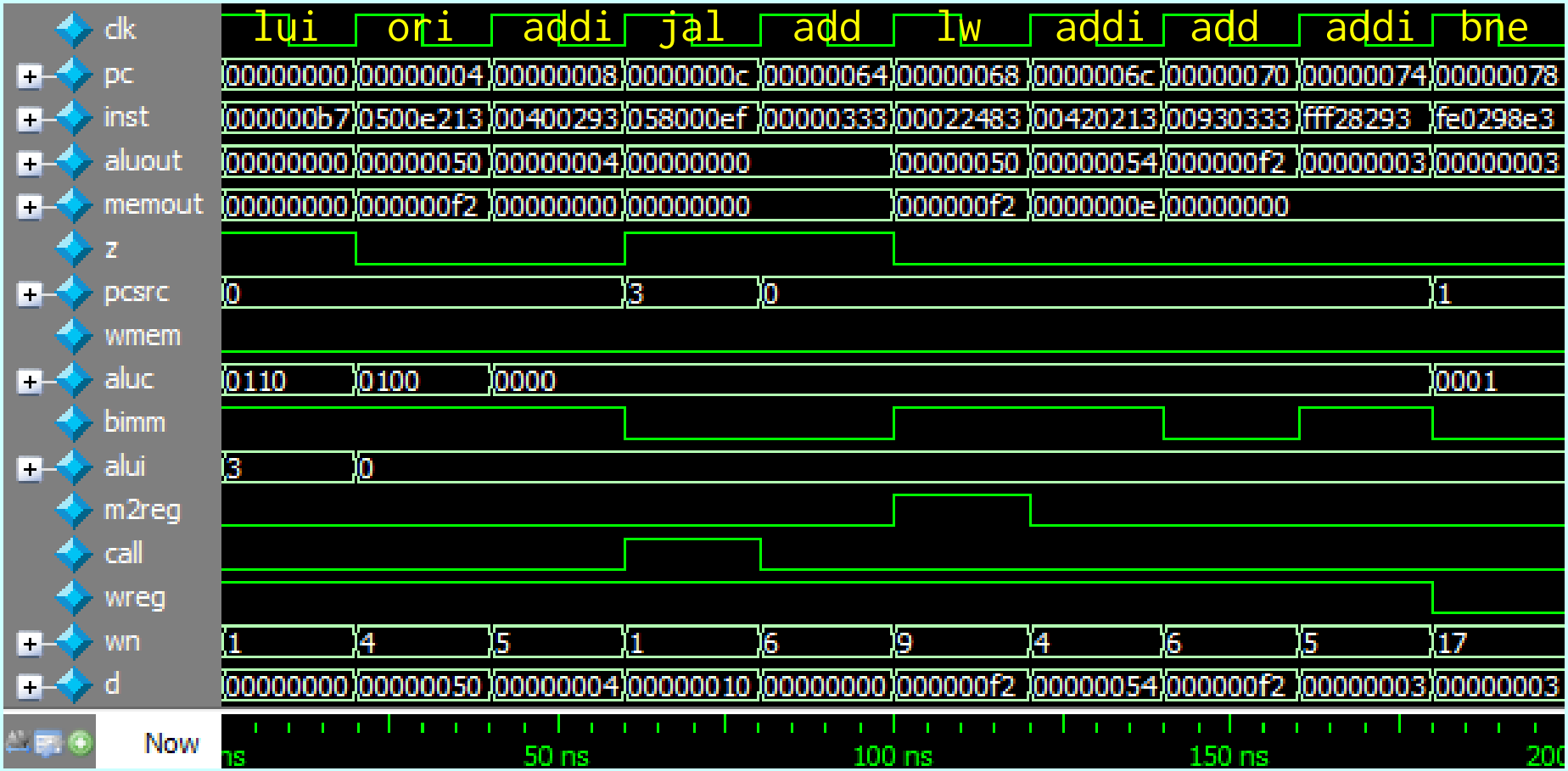
```
‘timescale 1ns/1ns
module sc_datamem (addr,datain,we,clk,dataout); // data memory, ram
    input         clk;                          // clock
    input         we;                           // write enable
    input  [31:0] datain;                       // data in (to memory)
    input  [31:0] 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
```

sc_instmem.s.txt

テストベンチ

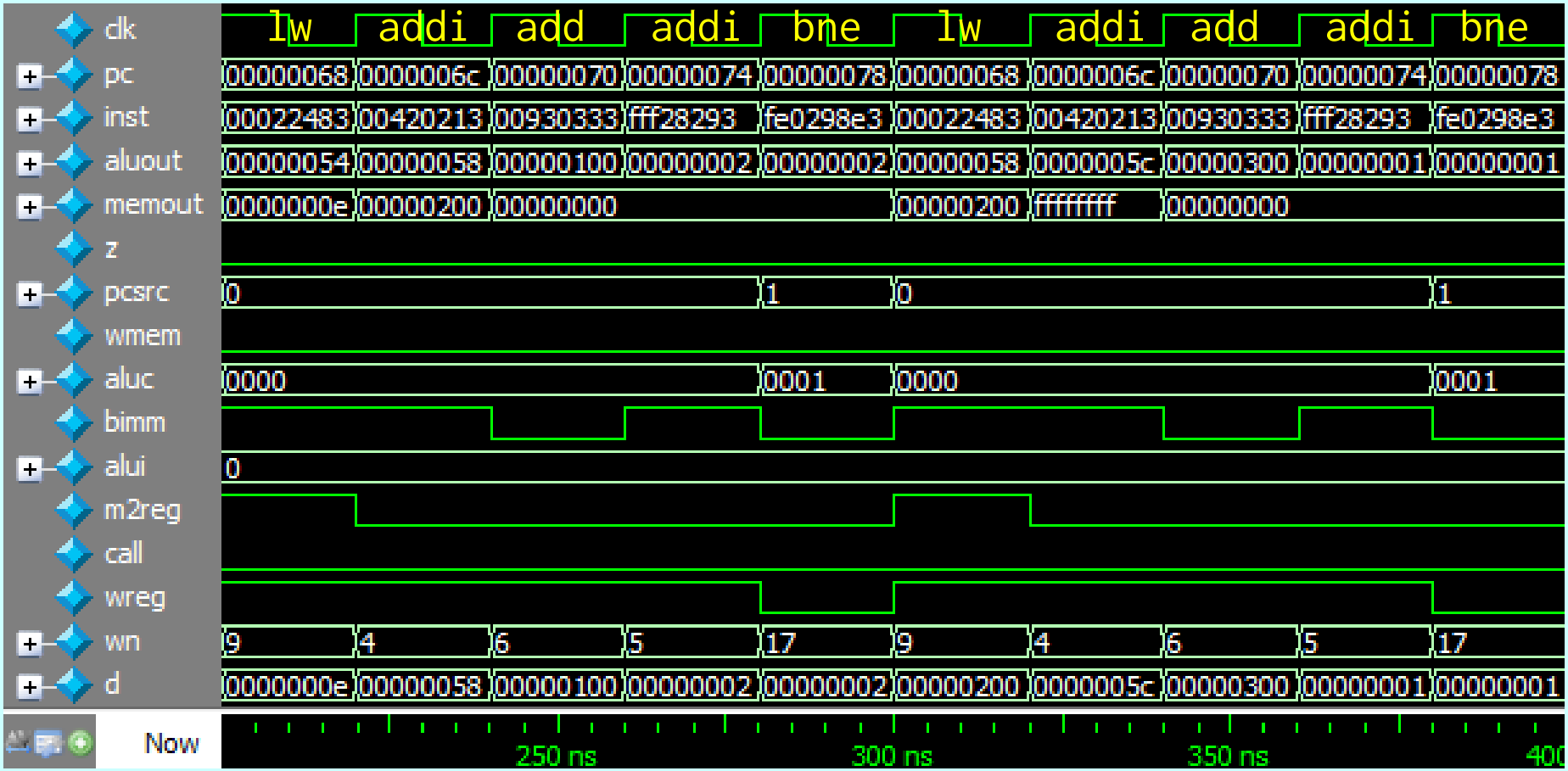
```
‘timescale 1ns/1ns
module sc_computer_tb;
    reg         clk, clrn;
    wire [31:0] inst, pc, aluout, memout;
    sc_computer cpu (.clk(clk),
                     .clrn(clrn),
                     .instr(inst),
                     .pc(pc),
                     .alu(aluout),
                     .mem(memout));
    initial begin
        #0    clrn = 0;
        #0    clk  = 1;
        #1    clrn = 1;
        #1399 $finish; // 1400 ns
    end
    always #10 clk = !clk;

    initial begin
        $dumpfile ("sc_computer.vcd");
        $dumpvars;
    end
endmodule
```

シミュレーション — 制御信号




```
(00) main:   lui  x1, 0                (68) loop:   lw   x9, 0(x4)
(04)         ori  x4, x1, 80           (6c)         addi x4, x4,  4
(08)         addi x5, x0,  4           (70)         add  x6, x6, x9
(0c) call:   jal  x1, sum              (74)         addi x5, x5, -1
(64) sum:    add  x6, x0, x0           (78)         bne  x5, x0, loop
```

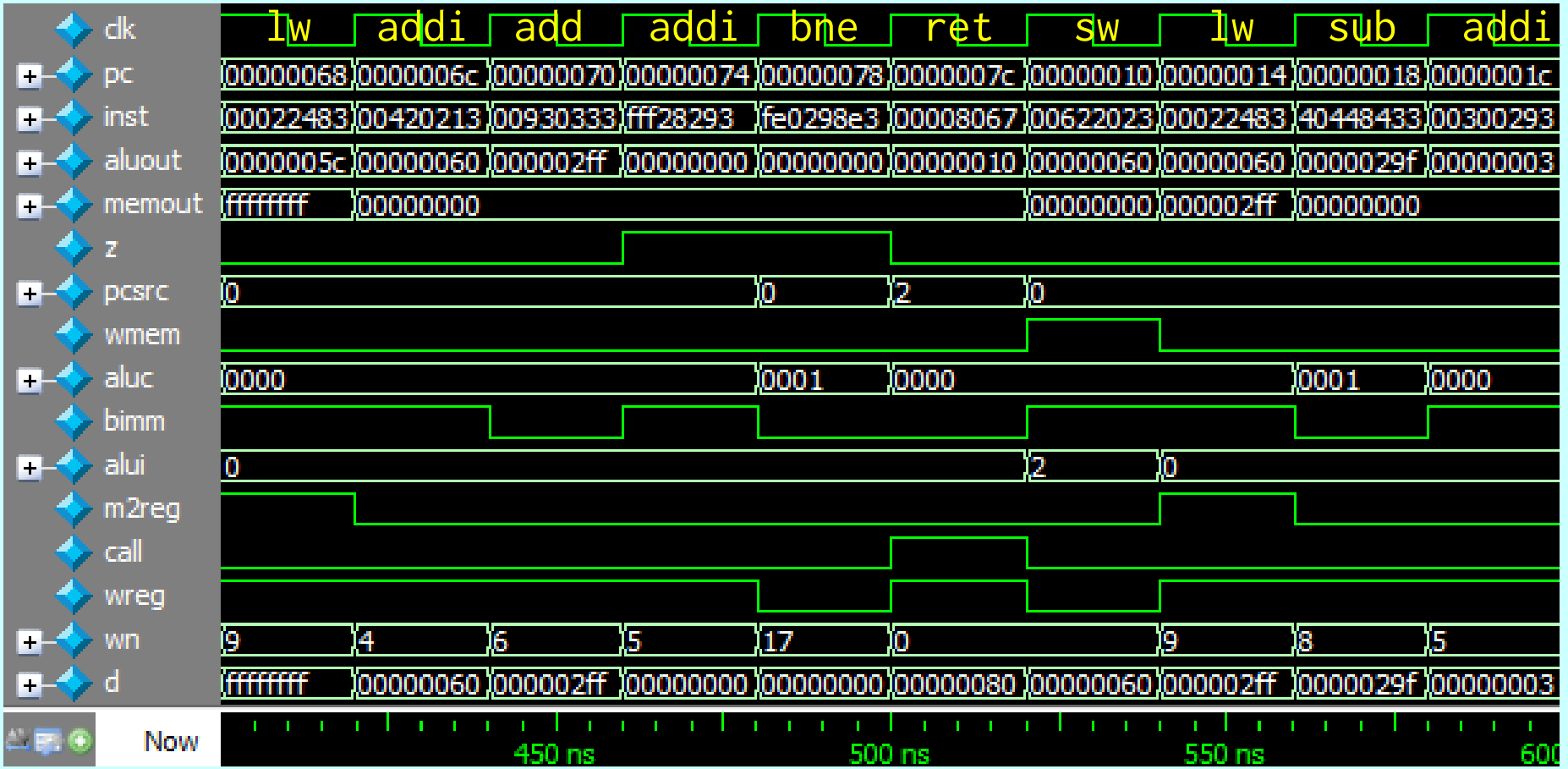
シミュレーション — 制御信号




```
(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
```

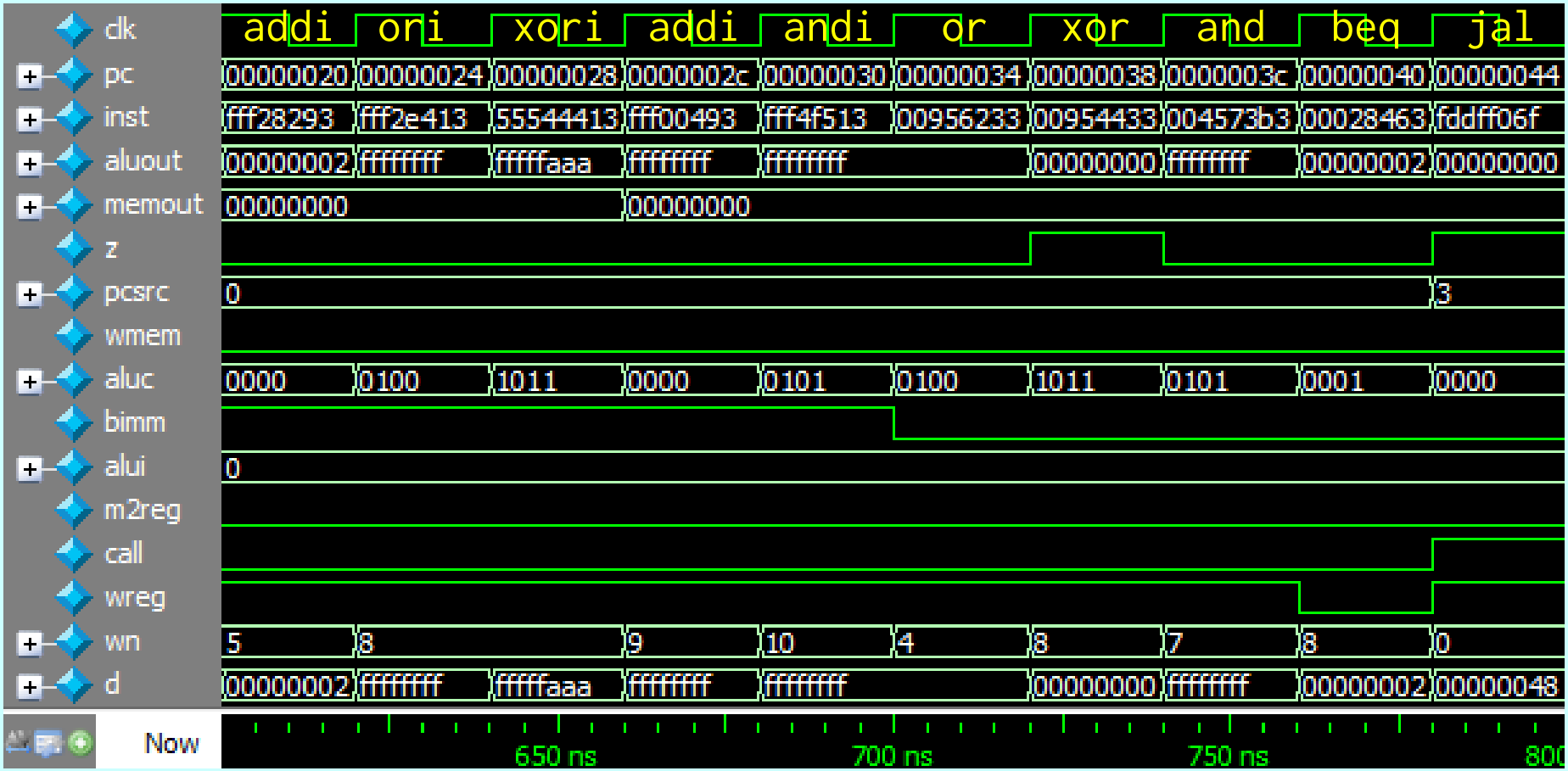
```
(68) loop:   lw   x9, 0(x4)
(6c)         addi x4, x4,  4
(70)         add  x6, x6, x9
(74)         addi x5, x5, -1
(78)         bne  x5, x0, loop
```

シミュレーション — 制御信号



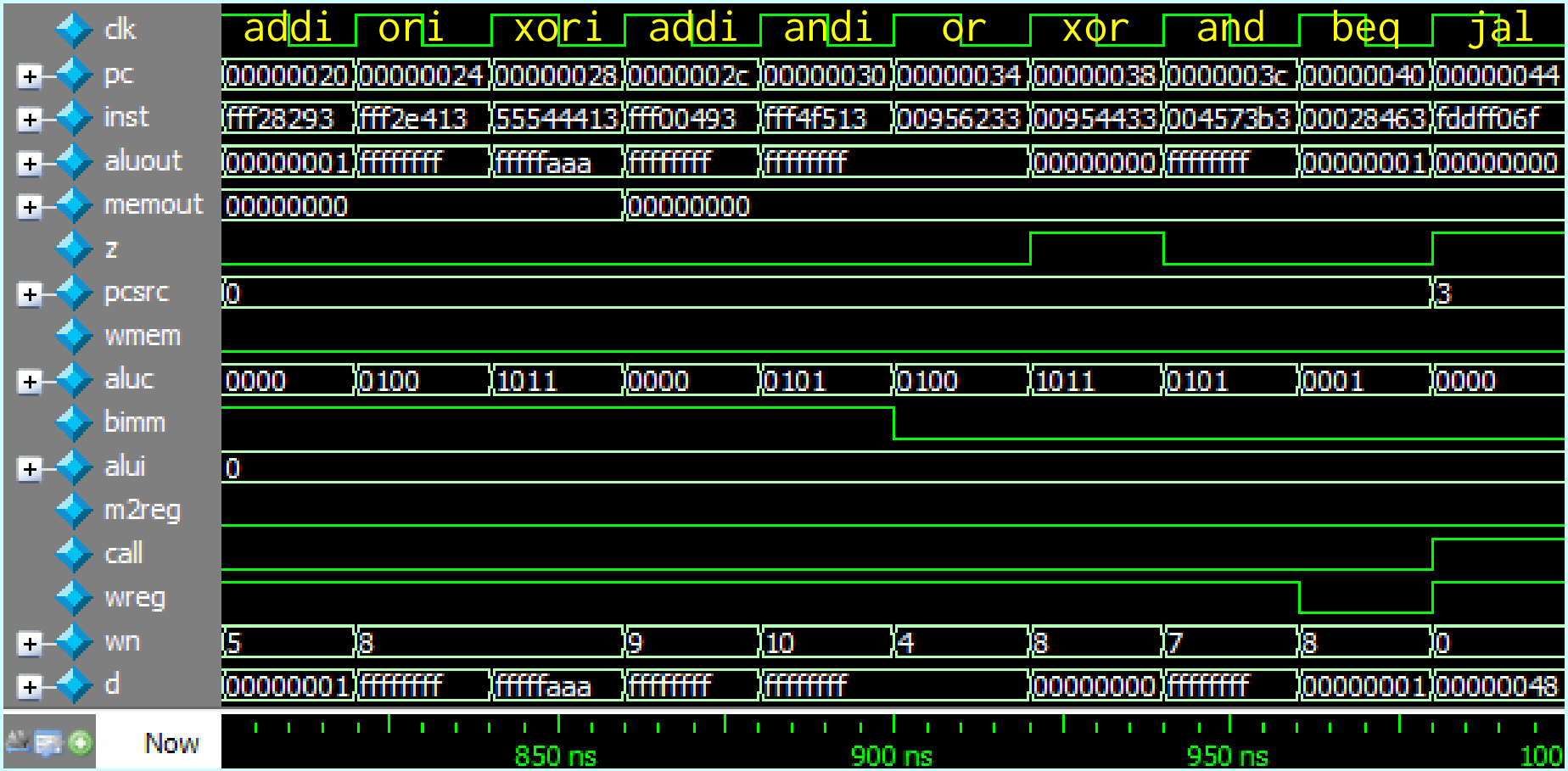

```
(68) loop:   lw   x9, 0(x4)          (7c)        ret  x1
(6c)         addi x4, x4,  4         (10)        sw   x6, 0(x4)
(70)         add  x6, x6, x9         (14)        lw   x9, 0(x4)
(74)         addi x5, x5, -1         (18)        sub  x8, x9, x4
(78)         bne  x5, x0, loop       (1c)        addi x5, x0,  3
```

シミュレーション — 制御信号



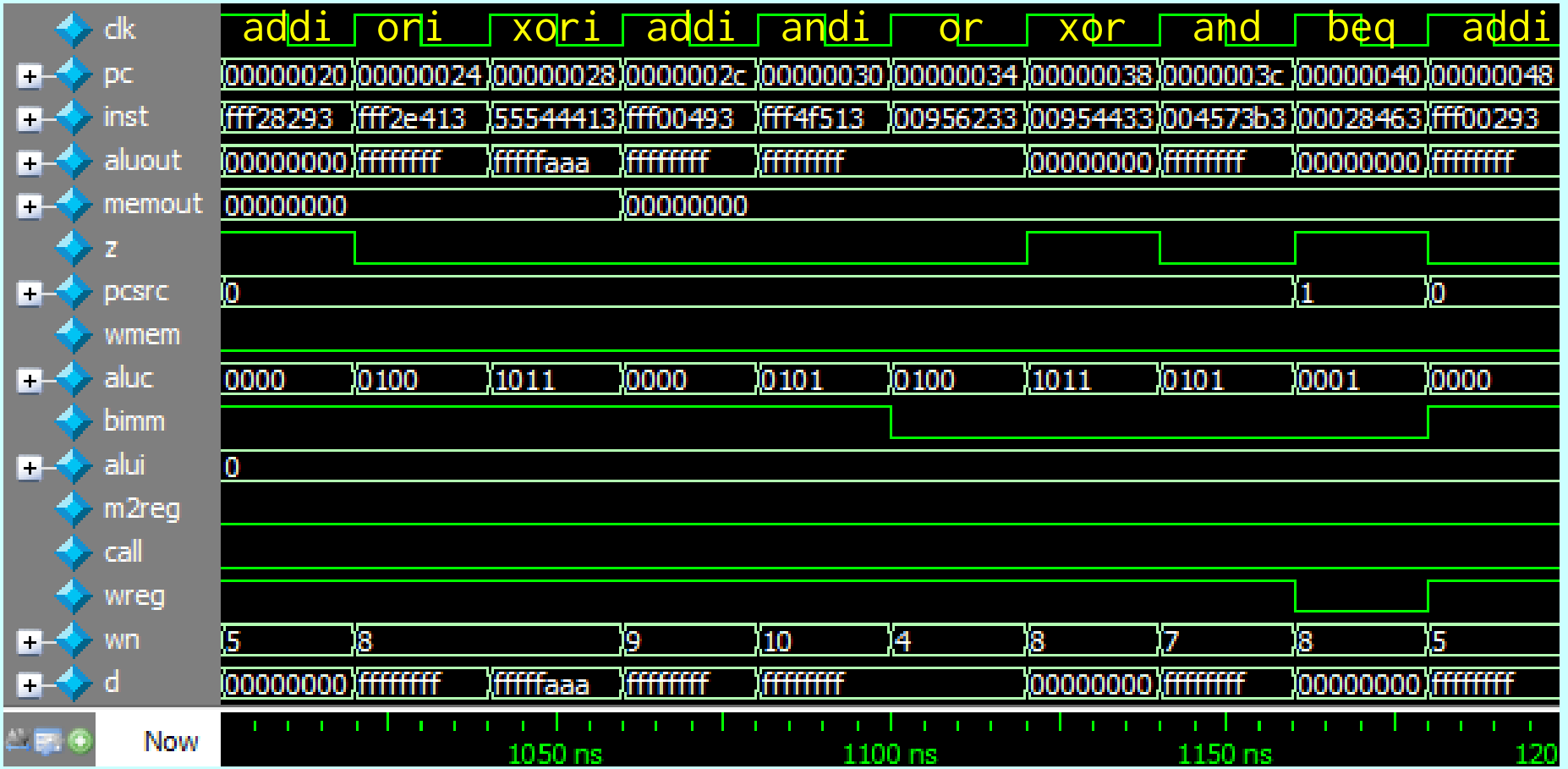

```
(20) loop2:  addi x5, x5, -1          (34)        or   x4, x10, x9
(24)         ori  x8, x5, -1          (38)        xor  x8, x10, x9
(28)         xori x8, x8, 0x555       (3c)        and  x7, x10, x4
(2c)         addi x9, x0, -1          (40)        beq  x5, x0, shift
(30)         andi x10,x9, -1          (44)        jal  x0, loop2
```

シミュレーション — 制御信号



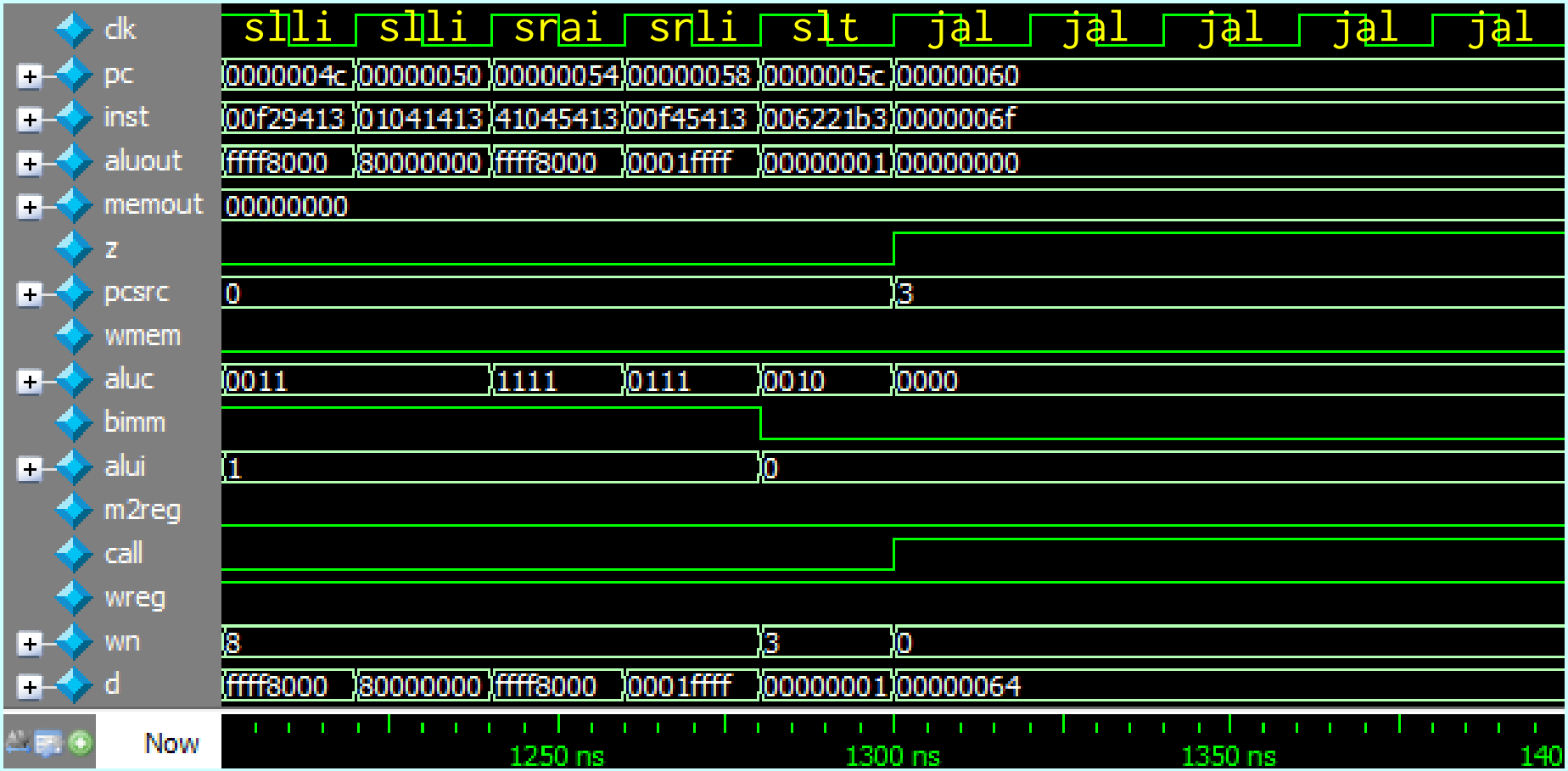

```
(20) loop2:  addi x5, x5, -1          (34)        or   x4, x10, x9
(24)         ori  x8, x5, -1          (38)        xor  x8, x10, x9
(28)         xori x8, x8, 0x555       (3c)        and  x7, x10, x4
(2c)         addi x9, x0, -1          (40)        beq  x5, x0, shift
(30)         andi x10,x9, -1          (44)        jal  x0, loop2
```

シミュレーション — 制御信号



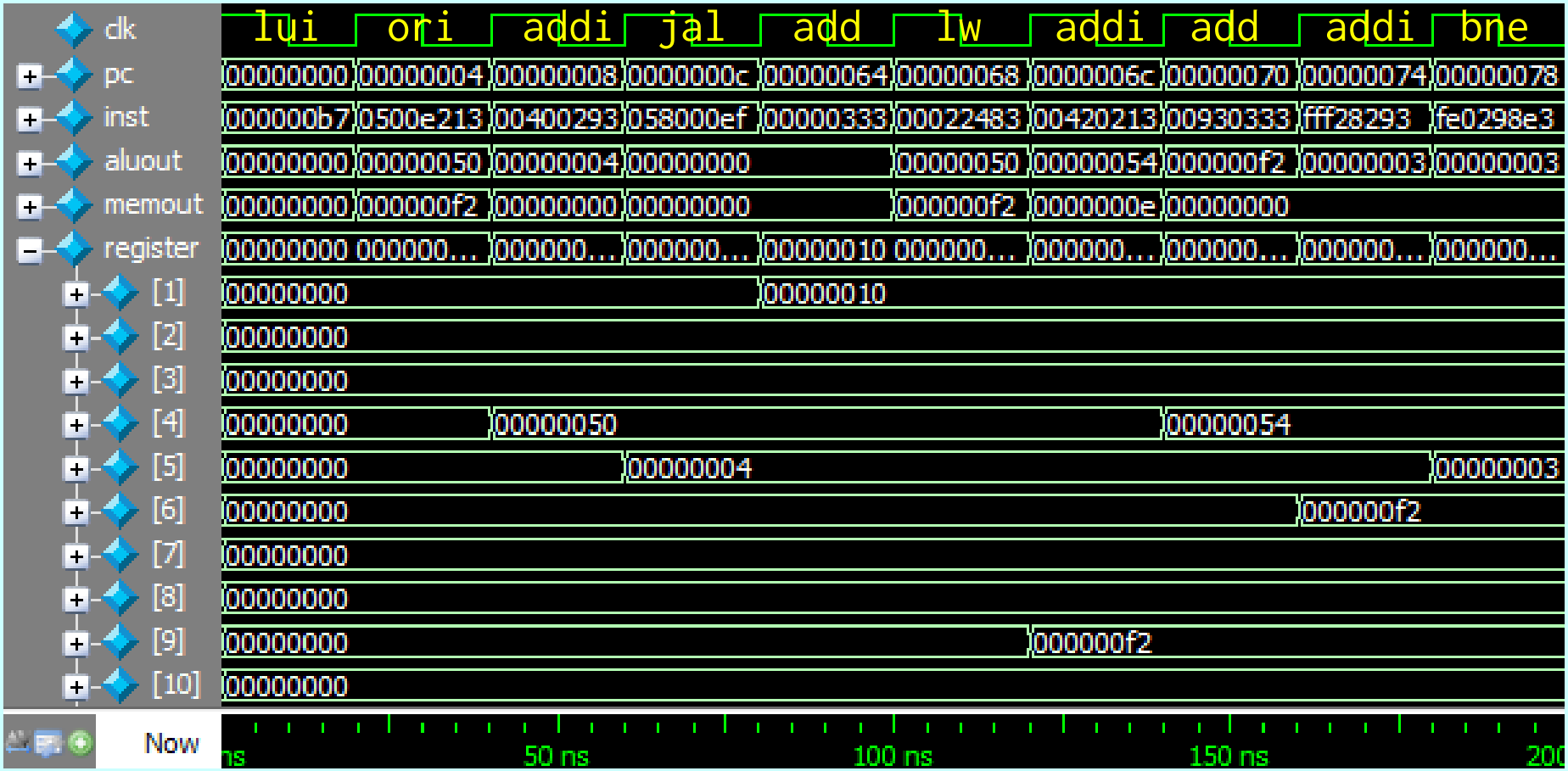

```
(20) loop2:  addi x5, x5, -1          (34)          or   x4, x10, x9
(24)         ori  x8, x5, -1          (38)          xor  x8, x10, x9
(28)         xori x8, x8, 0x555       (3c)          and  x7, x10, x4
(2c)         addi x9, x0, -1          (40)          beq  x5, x0, shift
(30)         andi x10,x9, -1          (48) shift:   addi x5, x0, -1
```

シミュレーション — 制御信号



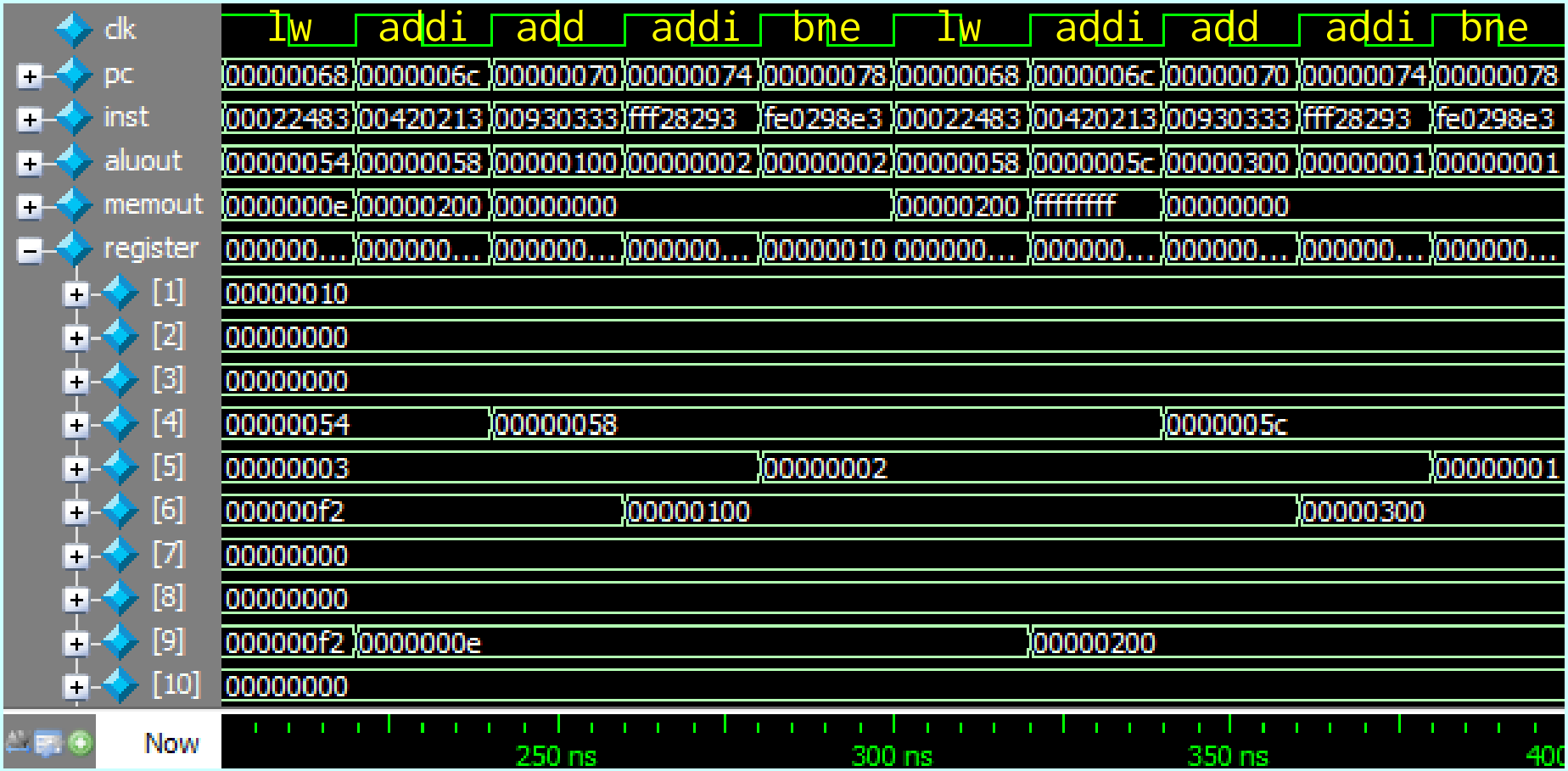

```
(4c)        slli x8, x5, 15                (60) finish: jal  x0, finish
(50)        slli x8, x8, 16                (60) finish: jal  x0, finish
(54)        srai x8, x8, 16                (60) finish: jal  x0, finish
(58)        srli x8, x8, 15                (60) finish: jal  x0, finish
(5c)        slt  x3, x4, x6                (60) finish: jal  x0, finish
```

シミュレーション — レジスタ




```
(00) main:   lui  x1, 0                (68) loop:   lw   x9, 0(x4)
(04)         ori  x4, x1, 80           (6c)         addi x4, x4,  4
(08)         addi x5, x0,  4           (70)         add  x6, x6, x9
(0c) call:   jal  x1, sum              (74)         addi x5, x5, -1
(64) sum:    add  x6, x0, x0           (78)         bne  x5, x0, loop
```

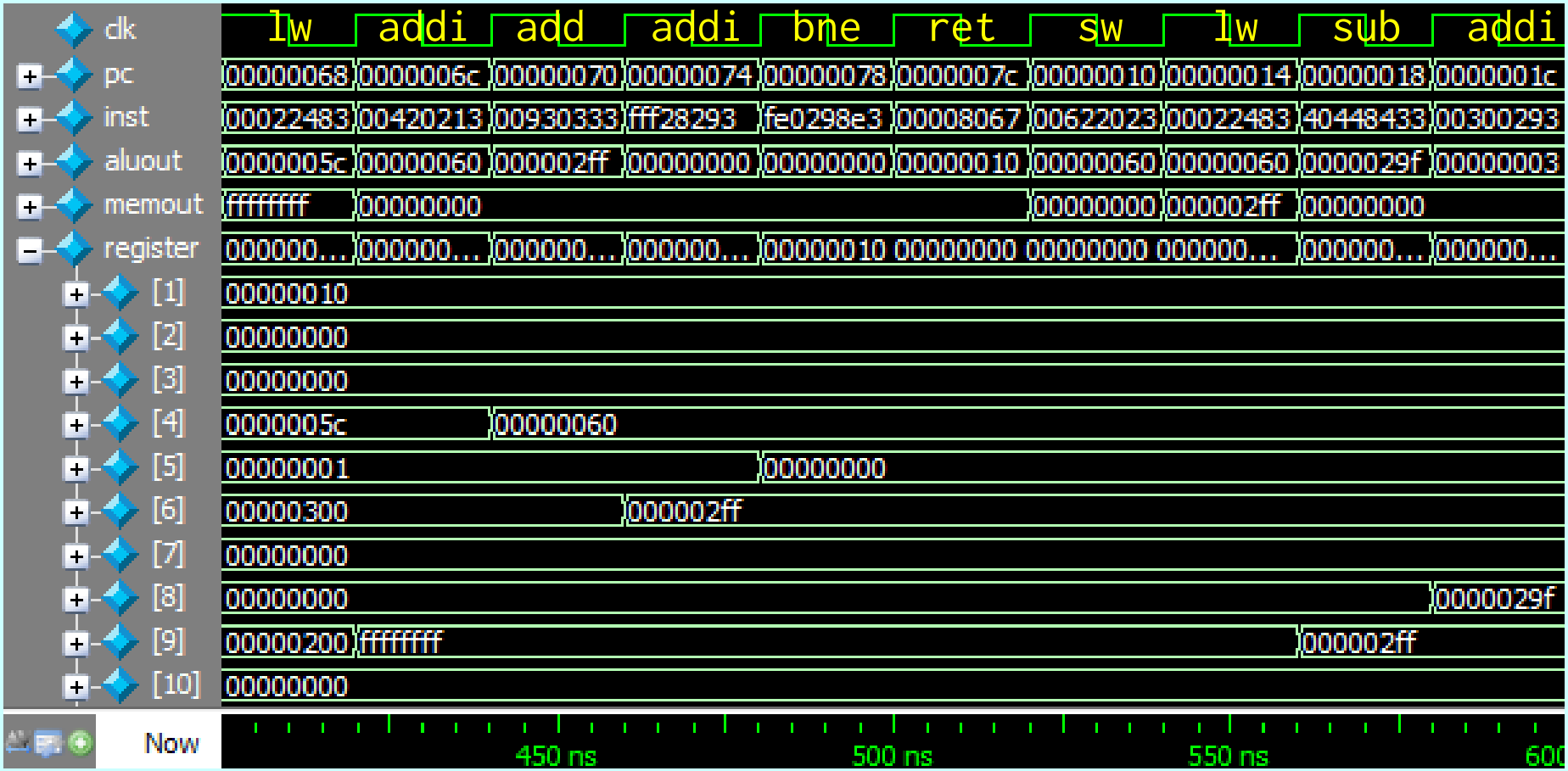
シミュレーション — レジスタ




```
(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
```

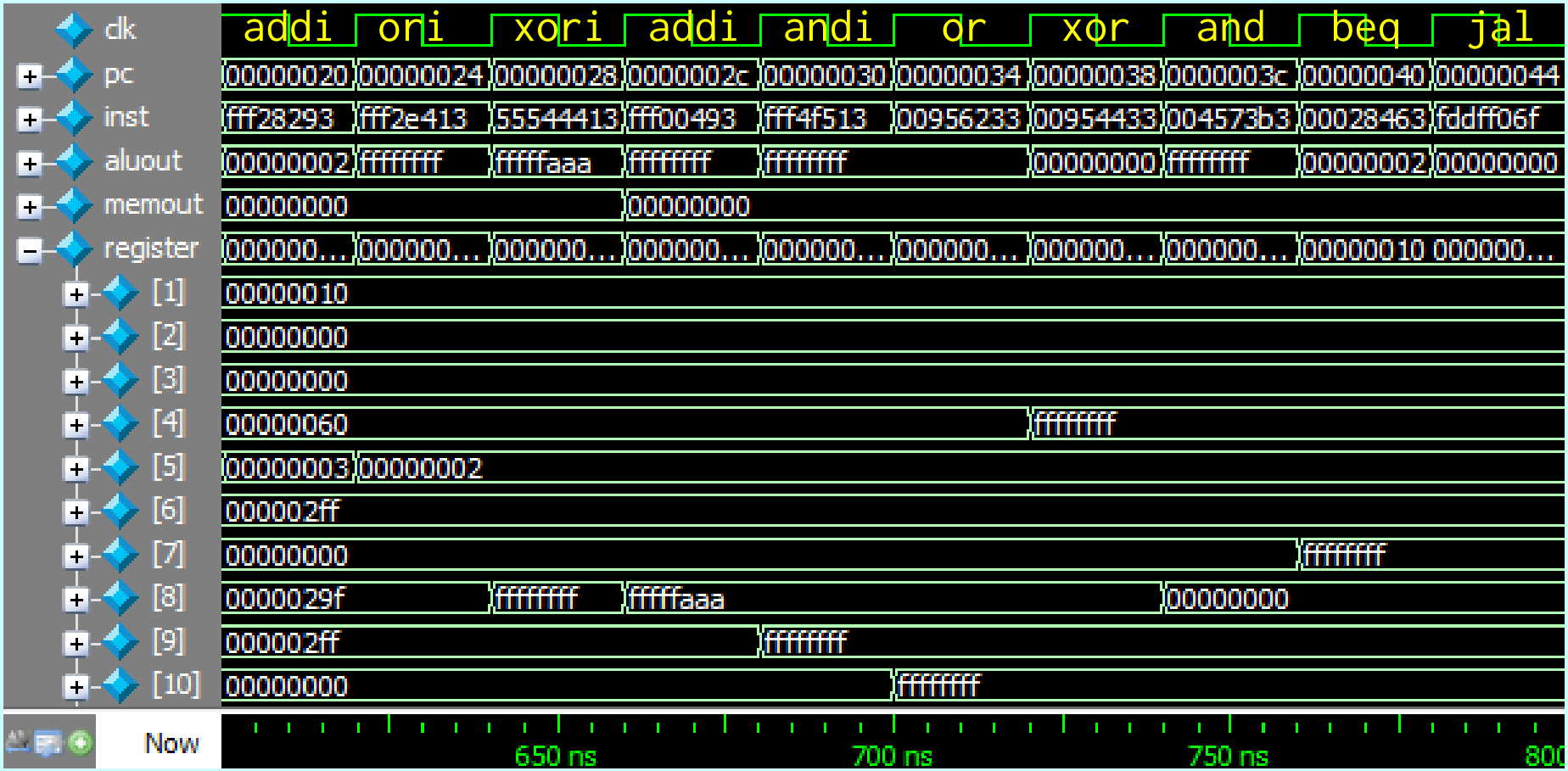
```
(68) loop:   lw   x9, 0(x4)
(6c)         addi x4, x4,  4
(70)         add  x6, x6, x9
(74)         addi x5, x5, -1
(78)         bne  x5, x0, loop
```

シミュレーション — レジスタ



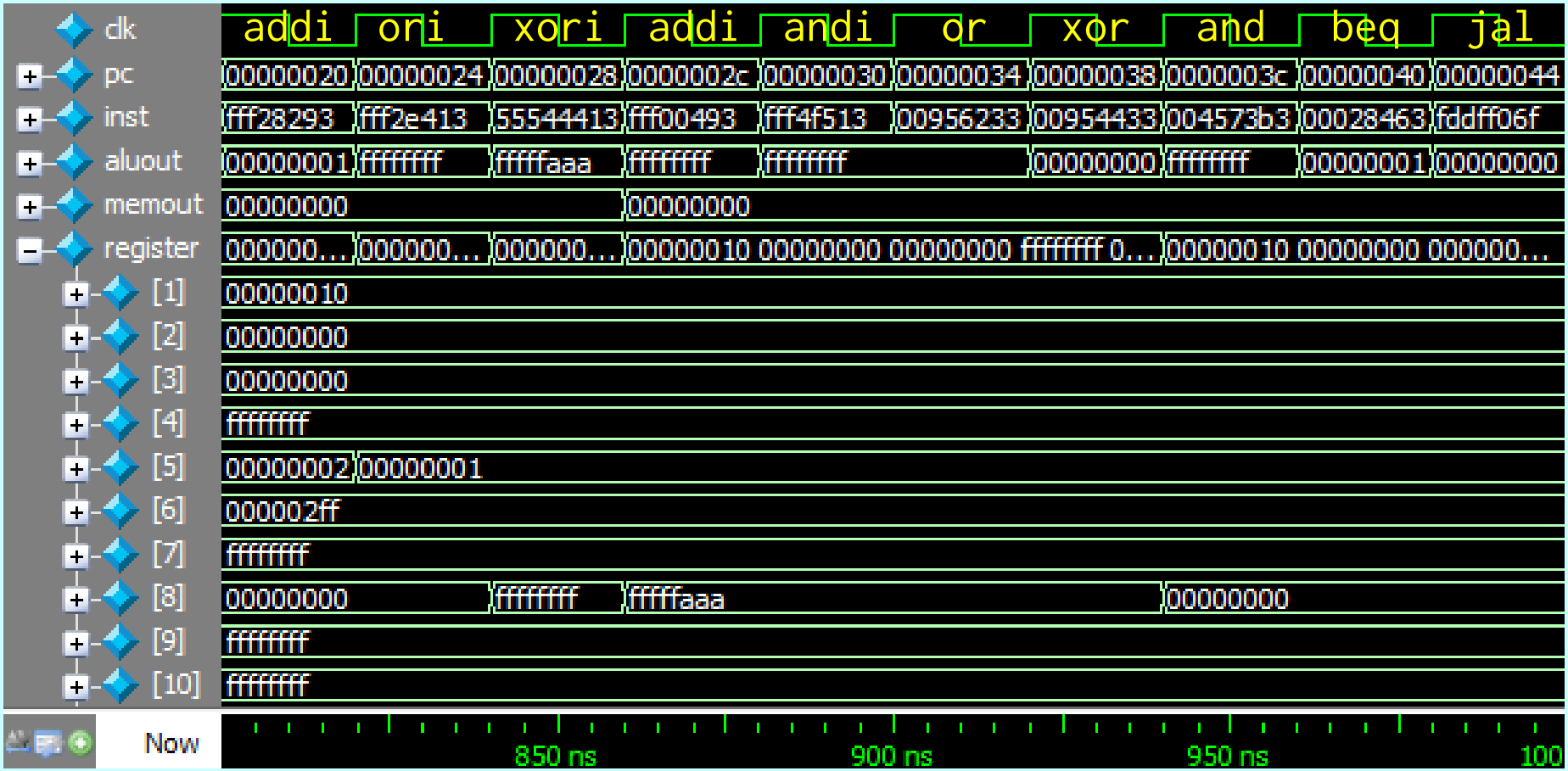

```
(68) loop:   lw   x9, 0(x4)          (7c)        ret  x1
(6c)         addi x4, x4,  4         (10)        sw   x6, 0(x4)
(70)         add  x6, x6, x9         (14)        lw   x9, 0(x4)
(74)         addi x5, x5, -1         (18)        sub  x8, x9, x4
(78)         bne  x5, x0, loop       (1c)        addi x5, x0,  3
```

シミュレーション — レジスタ



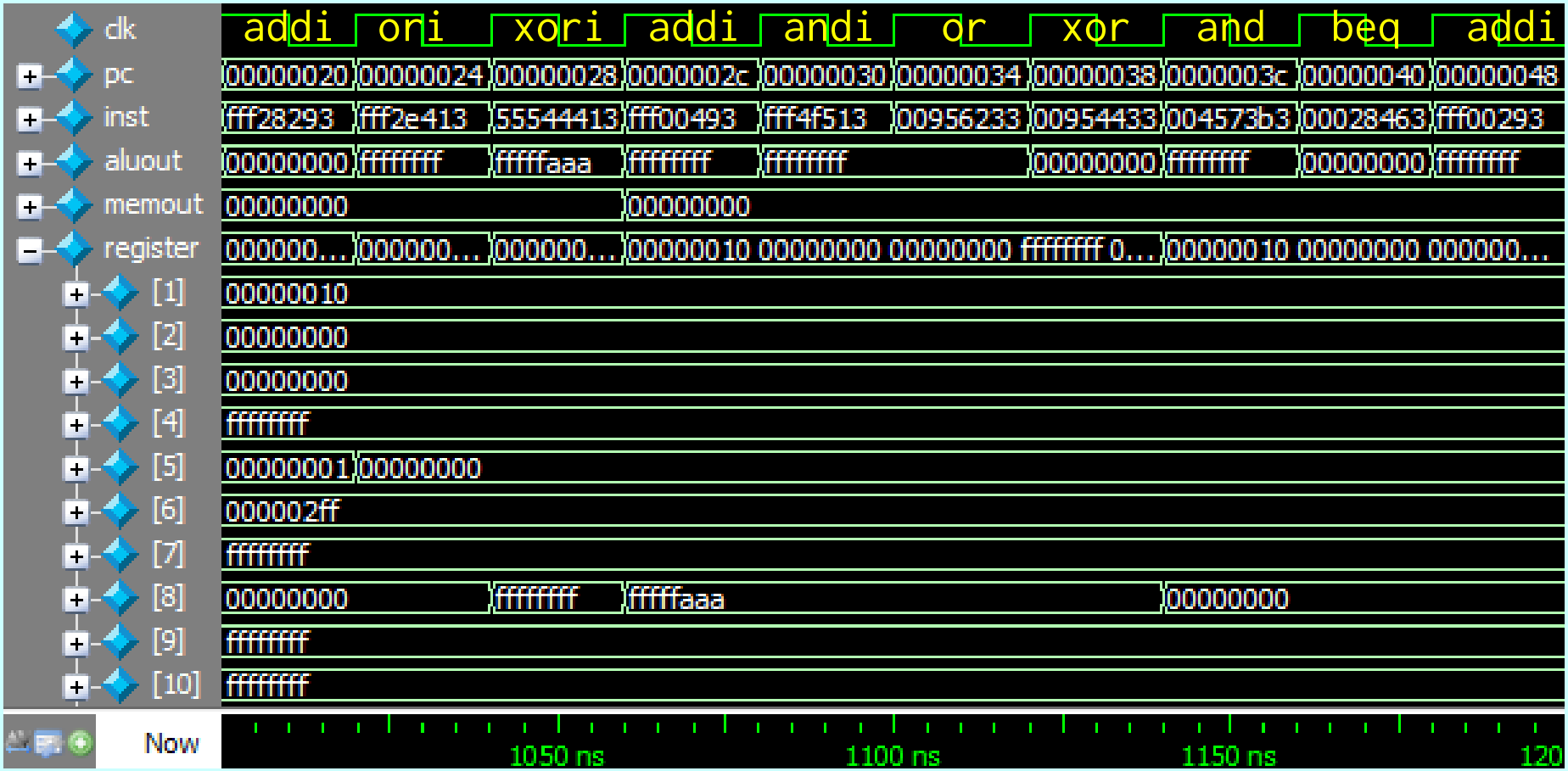

```
(20) loop2:  addi x5, x5, -1          (34)        or   x4, x10, x9
(24)         ori  x8, x5, -1          (38)        xor  x8, x10, x9
(28)         xori x8, x8, 0x555       (3c)        and  x7, x10, x4
(2c)         addi x9, x0, -1          (40)        beq  x5, x0, shift
(30)         andi x10,x9, -1          (44)        jal  x0, loop2
```

シミュレーション — レジスタ



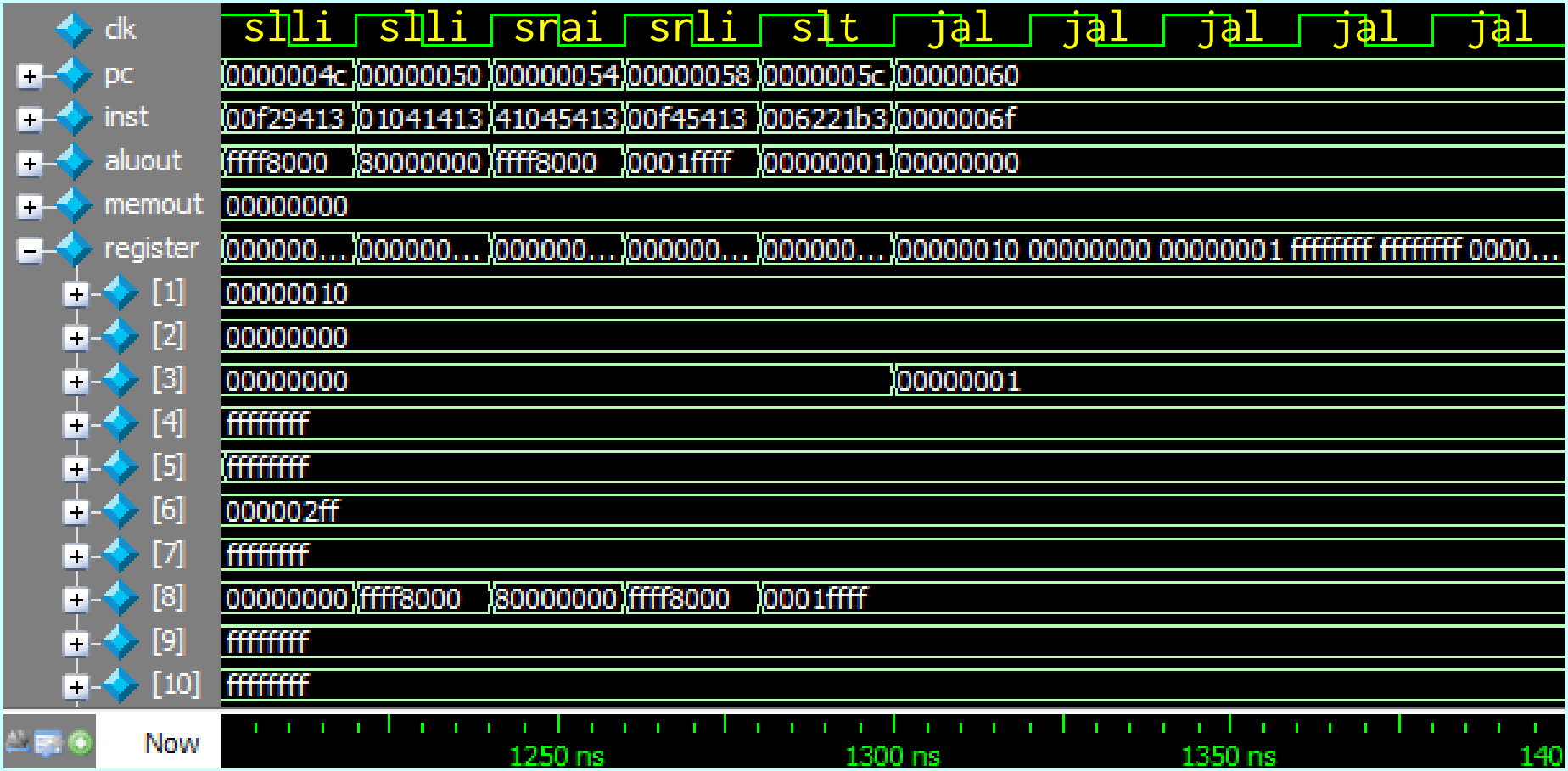

```
(20) loop2:  addi x5, x5, -1          (34)        or   x4, x10, x9
(24)         ori  x8, x5, -1          (38)        xor  x8, x10, x9
(28)         xori x8, x8, 0x555       (3c)        and  x7, x10, x4
(2c)         addi x9, x0, -1          (40)        beq  x5, x0, shift
(30)         andi x10,x9, -1          (44)        jal  x0, loop2
```

シミュレーション — レジスタ



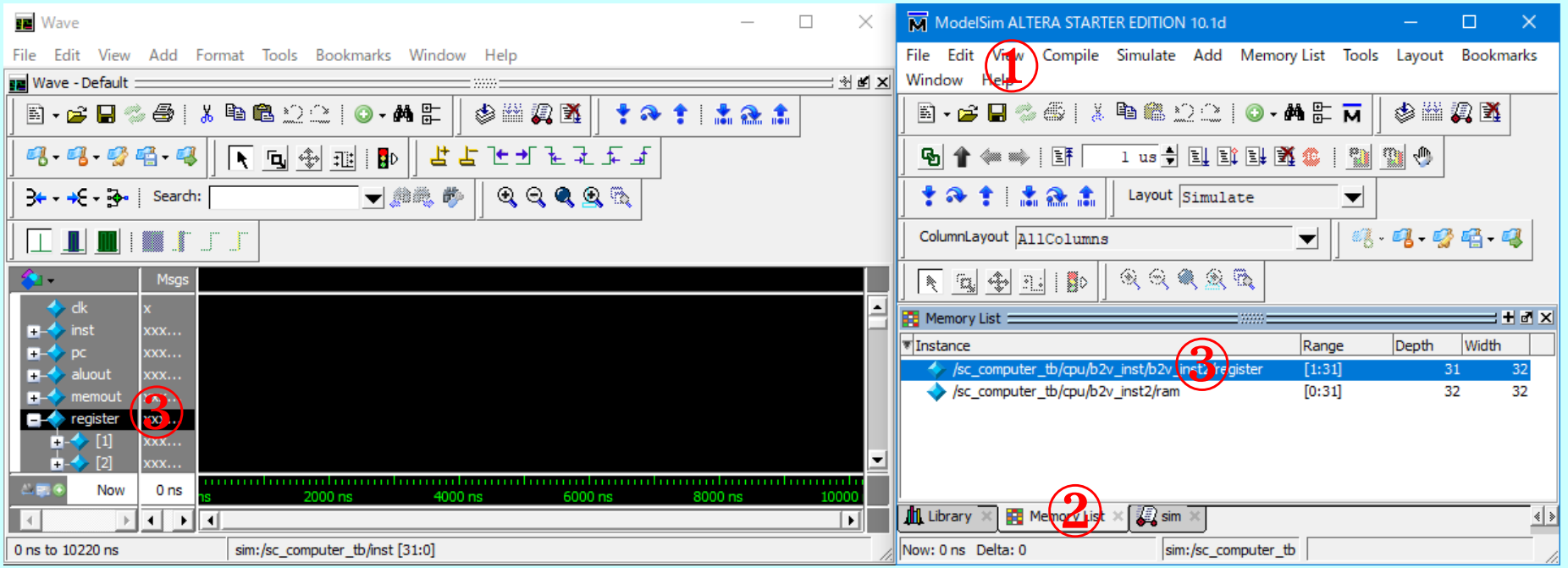

```
(20) loop2:  addi x5, x5, -1          (34)          or   x4, x10, x9
(24)         ori  x8, x5, -1          (38)          xor  x8, x10, x9
(28)         xori x8, x8, 0x555       (3c)          and  x7, x10, x4
(2c)         addi x9, x0, -1          (40)          beq  x5, x0, shift
(30)         andi x10,x9, -1          (48) shift:   addi x5, x0, -1
```

シミュレーション — レジスタ




```
(4c)        slli x8, x5, 15          (60) finish: jal  x0, finish
(50)        slli x8, x8, 16          (60) finish: jal  x0, finish
(54)        srai x8, x8, 16          (60) finish: jal  x0, finish
(58)        srli x8, x8, 15          (60) finish: jal  x0, finish
(5c)        slt  x3, x4, x6          (60) finish: jal  x0, finish
```

レジスタファイルの波形を表示する



① In the ModelSim main window, check View ▶ Memory List (w)

② In the ModelSim main window, select the Memory List window

③ Drag /sc_computer_tb/cpu/.../register and drop it to the Wave window

レジスタファイルの波形を表示する

On Mac PC:

```
iverilog -Wall -o sc_computer sc_computer_tb.v \
sc_computer.v sc_datamem.v sc_instmem.v \
sc_cpu.v dff32.v pc4.v sc_cu.v mux2x32.v alu.v \
regfile.v imme.v mux4x32.v branch_addr.v \
jalr_addr.v jal_addr.v

gtkwave sc_computer.vcd
```

ターミナルで gtkwave 起動後 SST の sc_computer_tb の左側にある三角形のボタンを押し、表示された cpu を選択すること...

2 進数の乗算

$$
\begin{array}{rcccccccl}
 & & & & 1 & 1 & 1 & 0 & (14_{10}) \\
 & & & \times & 1 & 0 & 1 & 0 & (10_{10}) \\
\hline
 & & & & 0 & 0 & 0 & 0 & \\
 & & & 1 & 1 & 1 & 0 & & \\
 & & 0 & 0 & 0 & 0 & & & \\
+ & 1 & 1 & 1 & 0 & & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & (140_{10})
\end{array}
$$

繰り返し乗算アルゴリズム

```
    c = 0 0 0 0 0 0 0 0          // 積
    a = 0 0 0 0 1 1 1 0          // 被乗数（かけられる数）(14)
    b = 0 0 0 0 1 0 1 0          // 乗数（かける数）(10)
1. 乗数 b 最下位ビットが 0
    c = 0 0 0 0 0 0 0 0          // 積
    a = 0 0 0 1 1 1 0 0          // 被乗数を左に1ビットシフトする
    b = 0 0 0 0 0 1 0 1          // 乗数を右に1ビットシフトする
2. 乗数 b 最下位ビットが 1
    c = c + a = 0 0 0 1 1 1 0 0  // 積
    a = 0 0 1 1 1 0 0 0          // 被乗数を左に1ビットシフトする
    b = 0 0 0 0 0 0 1 0          // 乗数を右に1ビットシフトする
3. 乗数 b 最下位ビットが 0
    c = 0 0 0 1 1 1 0 0          // 積
    a = 0 1 1 1 0 0 0 0          // 被乗数を左に1ビットシフトする
    b = 0 0 0 0 0 0 0 1          // 乗数を右に1ビットシフトする
4. 乗数 b 最下位ビットが 1
    c = c + a = 1 0 0 0 1 1 0 0  // 積 (128+12 = 140 = 14*10)
    a = 1 1 1 0 0 0 0 0          // 被乗数を左に1ビットシフトする
    b = 0 0 0 0 0 0 0 0          // 乗数を右に1ビットシフトする
```

繰り返し乗算アルゴリズム (C 言語)

```
# include <stdio.h>
int mul(int x, int y){
    int a, b, c;
    int i;                          // カウンタ
    a = x;                          // 被乗数 (かけられる数)
    b = y;                          // 乗数 (かける数)
    c = 0;                          // 積
    for (i = 0; i < 16; i++) {      // 16 回繰り返す:
        if ((b & 1) == 1) {         //       乗数最下位ビットが 1 ならば
            c += a;                 //             積 = 積 + 被乗数
        }                           //
        a = a << 1;                 //       被乗数を左に 1 ビットシフトする
        b = b >> 1;                 //       乗数を右に 1 ビットシフトする
    }                               //
    return(c);                      // 積を返す
}

int main(){
    int x = 0xc93f;
    int y = 0xe6c7;
    printf("0x%08x * 0x%08x = 0x%08x\n", x, y, mul(x, y));
    return 0;
}
```

繰り返し乗算アルゴリズム (C 言語)

yamin@localhost:/home/yamin/lectures/cod

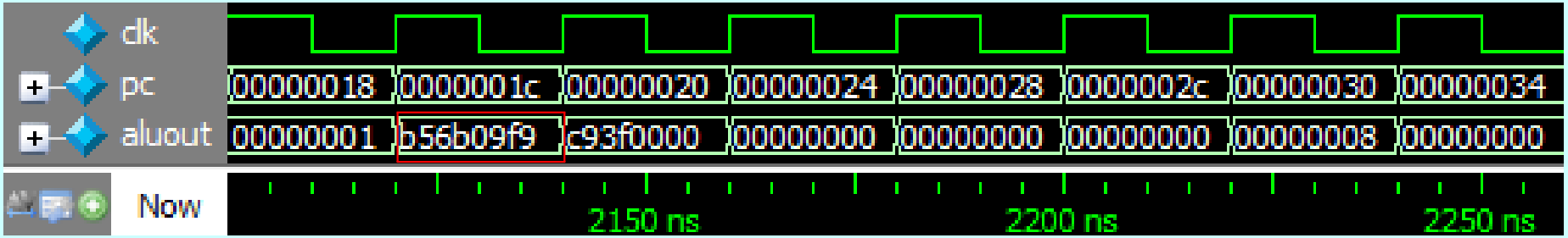
```
~/lectures/cod $ cat mul_by_shift_add.c
# include <stdio.h>
int mul(int x, int y){
    int a, b, c;
    int i;                      // カウンタ
    a = x;                      // 被乗数 (かけられる数)
    b = y;                      // 乗数 (かける数)
    c = 0;                      // 積
    for (i = 0; i < 16; i++) { // 16回繰り返す:
        if ((b & 1) == 1) {     //       乗数最下位ビットが 1 ならば
            c += a;             //             積 = 積 + 被乗数
        }                       //
        a = a << 1;             //       被乗数を左に 1 ビットシフトする
        b = b >> 1;             //       乗数を右に 1 ビットシフトする
    }                           //
    return(c);                  // 積を返す
}

int main(){
    int x = 0xc93f;
    int y = 0xe6c7;
    printf("0x%08x * 0x%08x = 0x%08x\n", x, y, mul(x, y));
    return 0;
}
~/lectures/cod $ gcc mul_by_shift_add.c -o mul_by_shift_add
~/lectures/cod $ ./mul_by_shift_add
0x0000c93f * 0x0000e6c7 = 0xb56b09f9
~/lectures/cod $
```

乗算プログラム

次の乗算プログラムを完成させ、Rivasm に実行させ、Verilog HDL に変換させ、さらに自分の CPU に実行させなさい。

```
.data 0
x: .word 0xc93f, 0xe6c7, 0  # 被乗数, 乗数, 積
.text 0
main:   la   x10, x         # (00) data address
        lw   x14, 0(x10)    # (04) 被乗数
        lw   x15, 4(x10)    # (08) 乗数
mul:    add  x16, x0, x0    # (0c) 積
        addi x12, x0, 16    # (10) counter
loop:   ...                 # (14) 乗数の最下位ビット
        ...                 # (18) 0 なら go to shift
        ...                 # (1c) 積 = 積 + 被乗数
shift:  ...                 # (20) 被乗数をシフト
        ...                 # (24) 乗数をシフト
        addi x12, x12,-1    # (28) counter - 1
        bne  x12, x0, loop  # (2c) 条件分岐
        sw   x16, 8(x10)    # (30) 積をメモリに保存
finish: j    finish         # (34)
.end
```

乗算プログラム

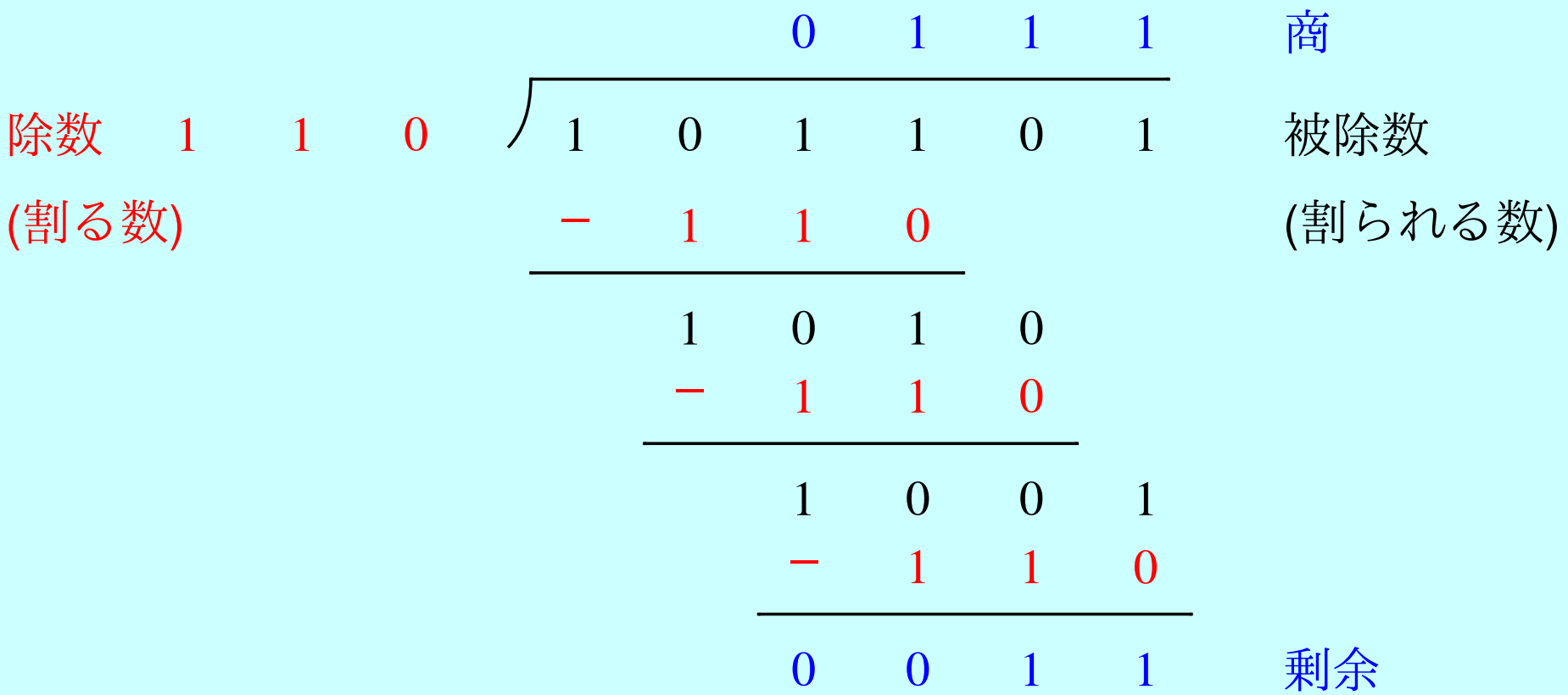


⇑

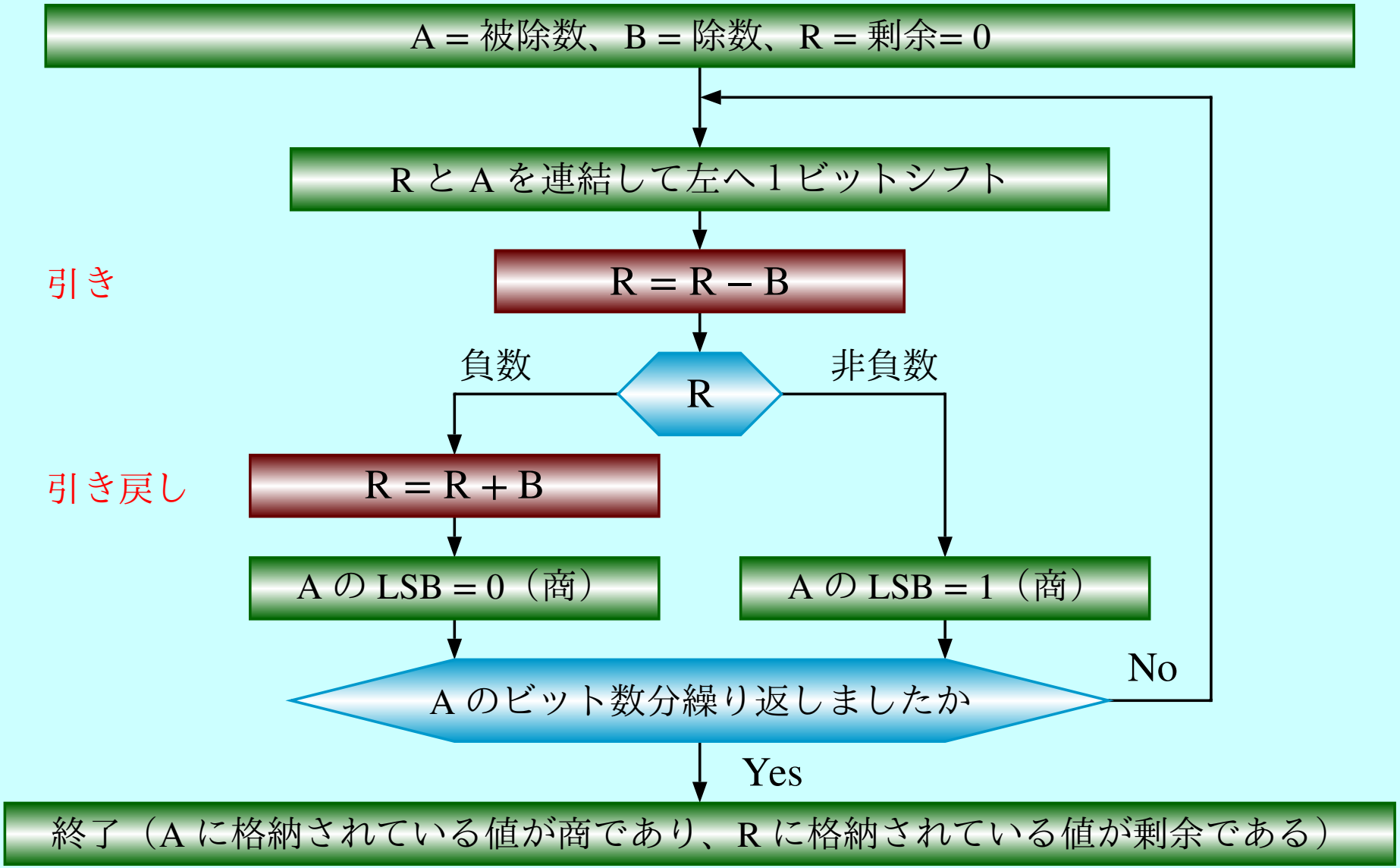
pc = 0x0000001c: 0xc93f × 0xe6c7 = 0xb56b09f9

2 進数の除算

45 / 6 = 7 … 3



除算 — 引き戻し法（符号なし）



引き戻し例: A = 1101, B = 0011

繰返し回数	操作	除数	剰余	被除数, 商
		B	R	A
0	初期化	0011	00000	1101
1	RA を左シフト	0011	00001	101
	R = R − B	0011	11110	101
	R は負数，商 0	0011	11110	1010
	R を元に戻す	0011	00001	1010
2	RA を左シフト	0011	00011	010
	R = R − B	0011	00000	010
	R は負数でない，商 1	0011	00000	0101
3	RA を左シフト	0011	00000	101
	R = R − B	0011	11101	101
	R は負数，商 0	0011	11101	1010
	R を元に戻す	0011	00000	1010
4	RA を左シフト	0011	00001	010
	R = R − B	0011	11110	010
	R は負数，商 0	0011	11110	0100
	R を元に戻す	0011	00001	0100

除算 — 引き戻し法（符号付き）

inputs: signed [31:0] x, y;
if x is negative, a = −x;
if y is negative, b = −y;
do restoring division on a and b to get q and r;
if the signs of x and y are different, q = −q;
if x is negative, r = −r;

Examples:

```
   7 /   2 =   3 ...   1
   7 / −2 = −3 ...   1
 −7 /   2 = −3 ... −1
 −7 / −2 =   3 ... −1
```

除算 — 引き放し法（符号なし）

- 引き戻し法では剰余が負になった場合，$R = R + B$ を実行して剰余を元に戻す。
- その後に，元に戻した剰余を 1 ビット左にシフトし，B を引く。
- これらの操作を次のように簡単にできる：
  $(R + B) \times 2 - B = 2R + B$
- つまり，剰余が負数の場合，元に戻し，そこから B を引く代わりに，剰余を 1 ビット左にシフトし，それに B を足すことよっても同じ効果が得られる。

R
- 非負数: 商 $= 1$, R と A を連結して左へ 1 ビットシフト, $R = R - B$
- 負数: 商 $= 0$, R と A を連結して左へ 1 ビットシフト, $R = R + B$

引き放し例: A = 1101, B = 0011

繰返し回数	操作	除数	剰余	被除数, 商
		B	R	A
0	初期化	0011	00000	1101
1	RA を左シフト	0011	00001	101
	R = R − B	0011	11110	101
	R は負数, 商 0	0011	11110	1010
2	RA を左シフト	0011	11101	010
	R = R + B	0011	00000	010
	R は負数でない, 商 1	0011	00000	0101
3	RA を左シフト	0011	00000	101
	R = R − B	0011	11101	101
	R は負数, 商 0	0011	11101	1010
4	RA を左シフト	0011	11011	010
	R = R + B	0011	11110	010
	R は負数, 商 0	0011	11110	0100
End	R は負数, 剰余 +B	0011	00001	0100

除算 — 引き放し法（符号付き）

inputs: signed [31:0] x, y;
if x is negative, a = −x;
if y is negative, b = −y;
do non-restoring division on a and b to get q and r;
if the signs of x and y are different, q = −q;
if x is negative, r = −r;

Examples:

```
   7 /   2 =   3 ...   1
   7 / −2 = −3 ...   1
 −7 /   2 = −3 ... −1
 −7 / −2 =   3 ... −1
```

RV32M (Mul/Div/Rem) Instructions

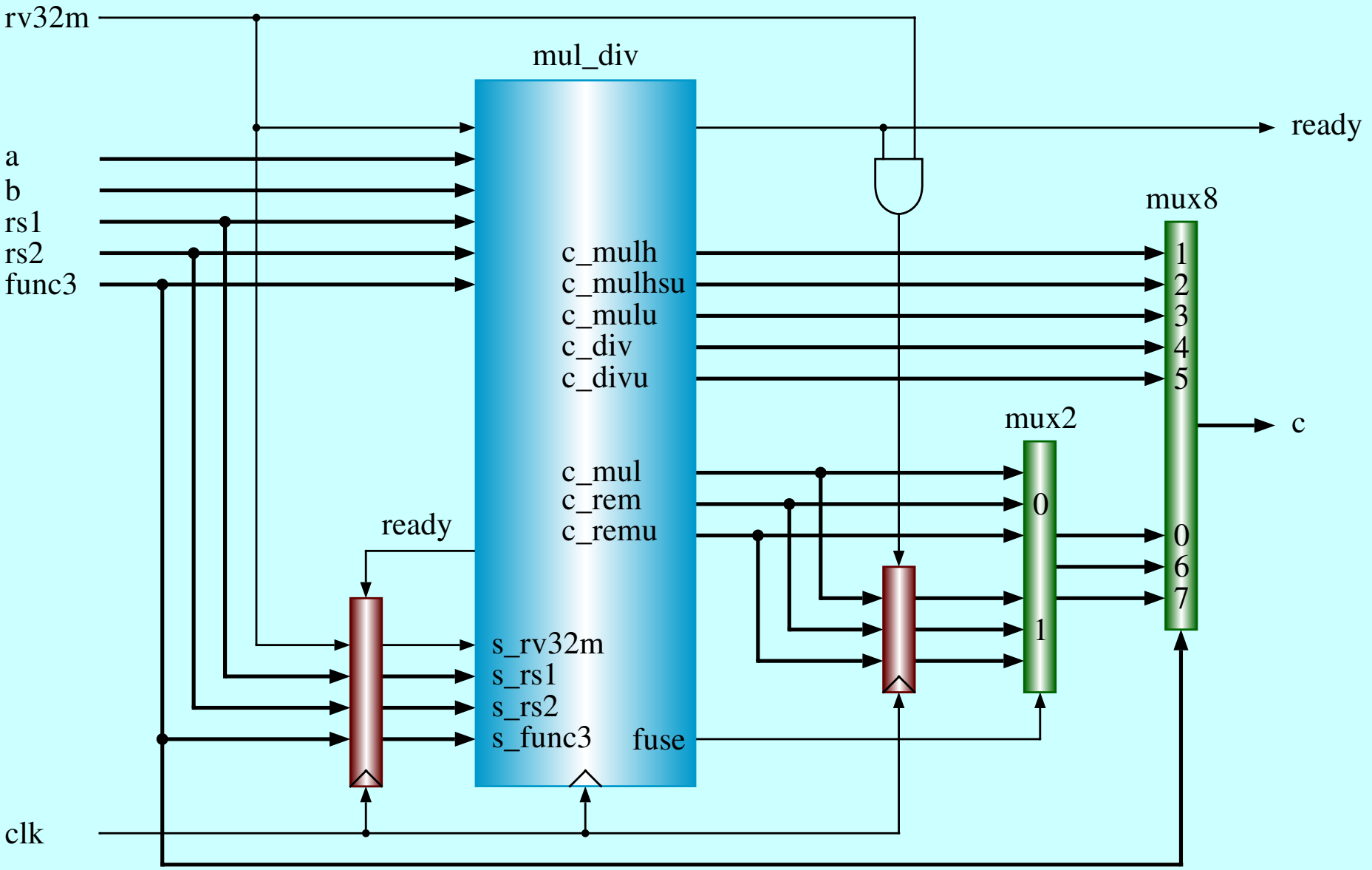
0000001	rs2	rs1	000	rd	0110011	mul
0000001	rs2	rs1	001	rd	0110011	mulh
0000001	rs2	rs1	010	rd	0110011	mulhsu
0000001	rs2	rs1	011	rd	0110011	mulhu
0000001	rs2	rs1	100	rd	0110011	div
0000001	rs2	rs1	101	rd	0110011	divu
0000001	rs2	rs1	110	rd	0110011	rem
0000001	rs2	rs1	111	rd	0110011	remu

```
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)
```

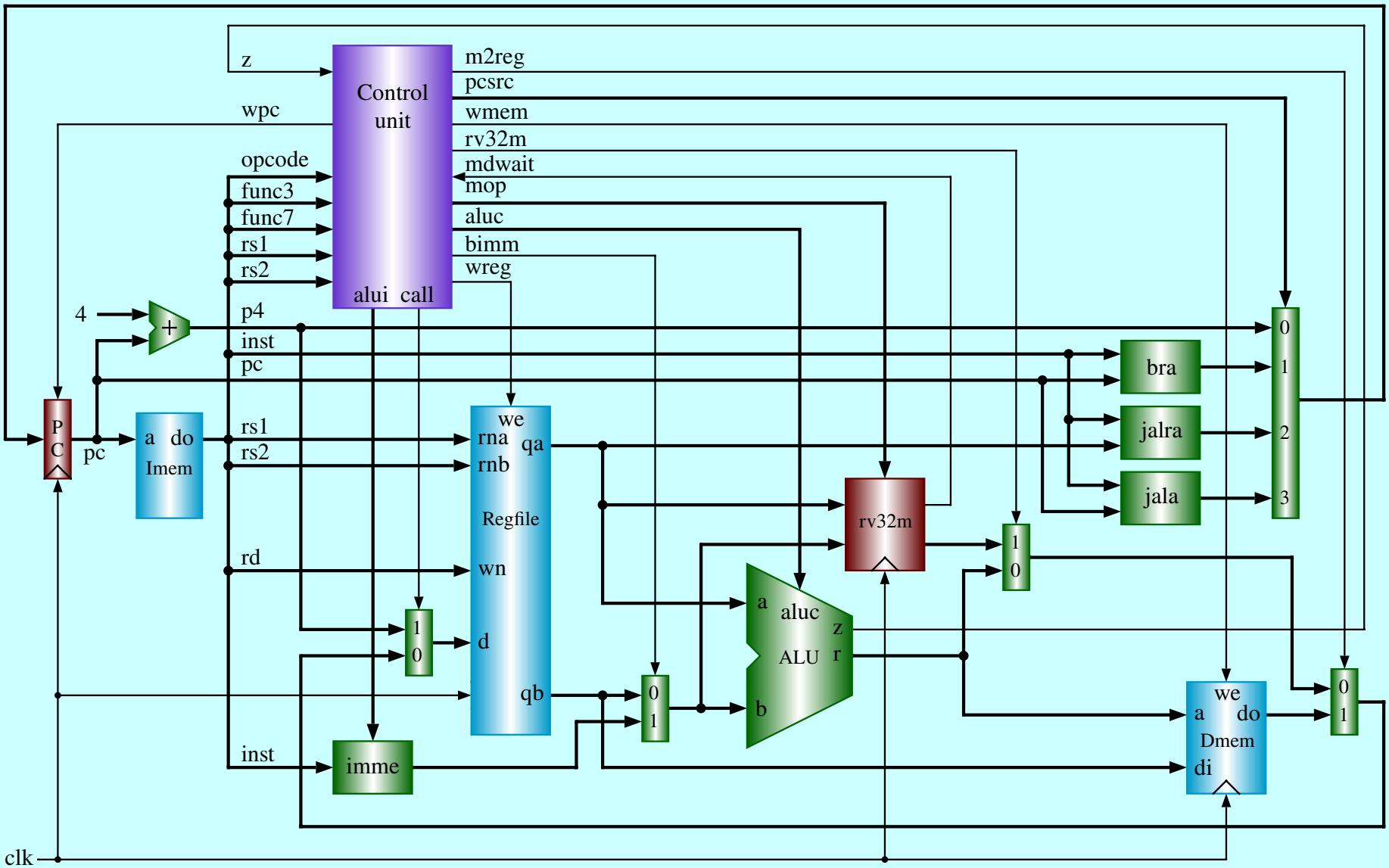
RISC-V RV32M Fuse

- “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 RV32M Fuse



RISC-V RV32IM



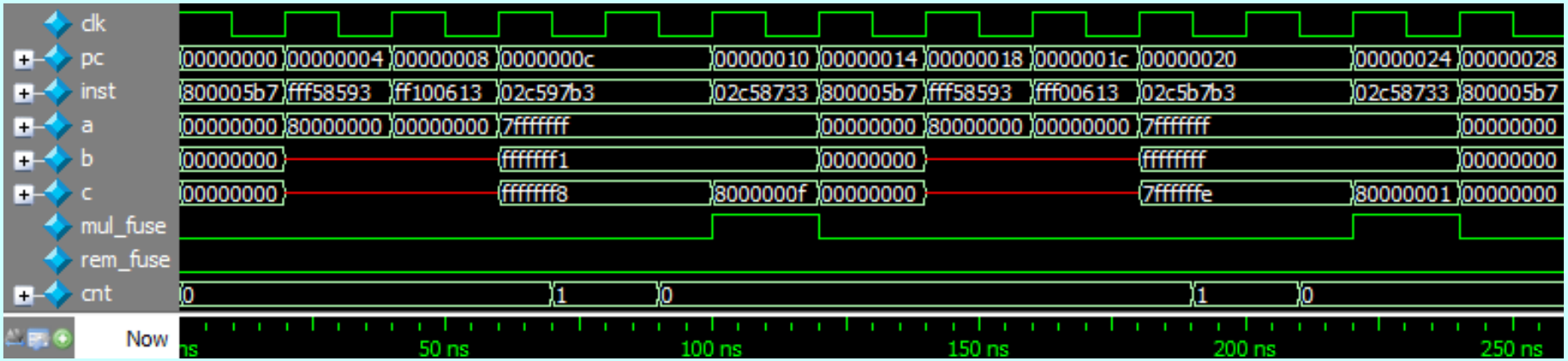
RISC-V RV32M Test Program

```
.text
main:
s_x_s:  li     a1, 0x7fffffff # (00,04) a            H = 0xfffffff8
        li     a2, -15        # (08) b              L = 0x8000000f
        mulh   a5,  a1, a2    # (0c) product high
        mul    a4,  a1, a2    # (10) product low, fused with mulh
u_x_u:  li     a1, 0x7fffffff # (14,18) a           H = 0x7ffffffe
        li     a2, -1         # (1c) b              L = 0x80000001
        mulhu  a5,  a1, a2    # (20) product high
        mul    a4,  a1, a2    # (24) product low, fused with mulhu
s_x_u:  li     a1, 0x7fffffff # (28,2c) a           H = 0x7ffffffe
        li     a2, -1         # (30) b              L = 0x80000001
        mulhsu a5,  a1, a2    # (34) product high
        mul    a4,  a1, a2    # (38) product low, fused with mulhsu
```

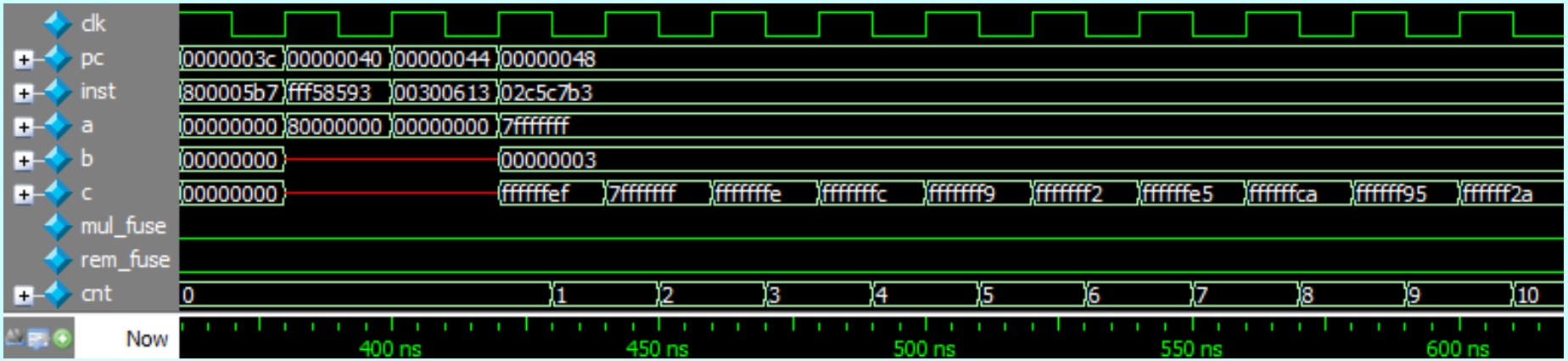
RISC-V RV32M Test Program

```
sign0:  li     a1, 0x7fffffff # (3c,40) a              Q = 0x2aaaaaaa
        li     a2,   3        # (44) b                  R = 0x00000001
        div    a5,  a1, a2    # (48) div signed
        rem    a4,  a1, a2    # (4c) rem signed, fused with div
sign1:  li     a1, 0xfffffff2 # (50) a                  Q = 0xfffffffc
        li     a2,   3        # (54) b                  R = 0xfffffffe
        div    a5,  a1, a2    # (58) div signed
        rem    a4,  a1, a2    # (5c) rem signed, fused with div
unsign: li     a1, 0xfffffff2 # (60) a                  Q = 0x55555550
        li     a2,   3        # (64) b                  R = 0x00000002
        divu   a5,  a1, a2    # (68) div unsigned
        remu   a4,  a1, a2    # (6c) rem unsigned, fused with divu
finish: jal    x0, finish     # (70) dead loop
.end
```

Simulation Waveform

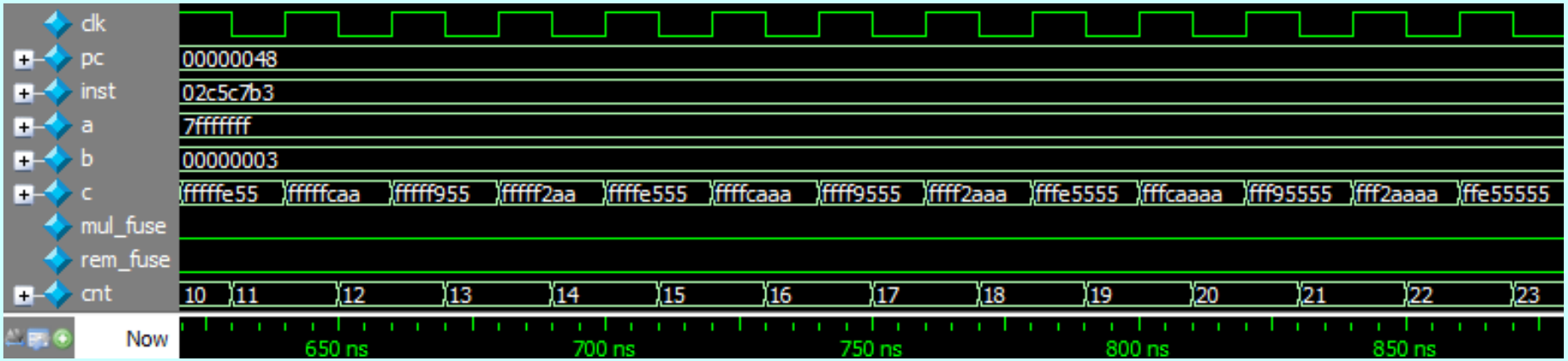


mulh-mul: 0xffffffff88000000f; mulhu-mul: 0x7ffffffe80000001

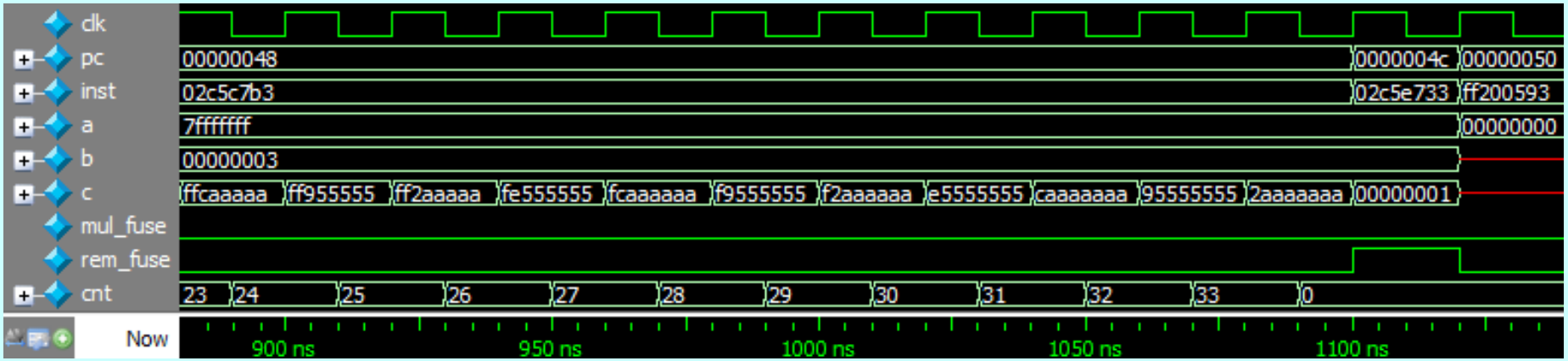


div-rem: 0x7fffffff / 3

Simulation Waveform

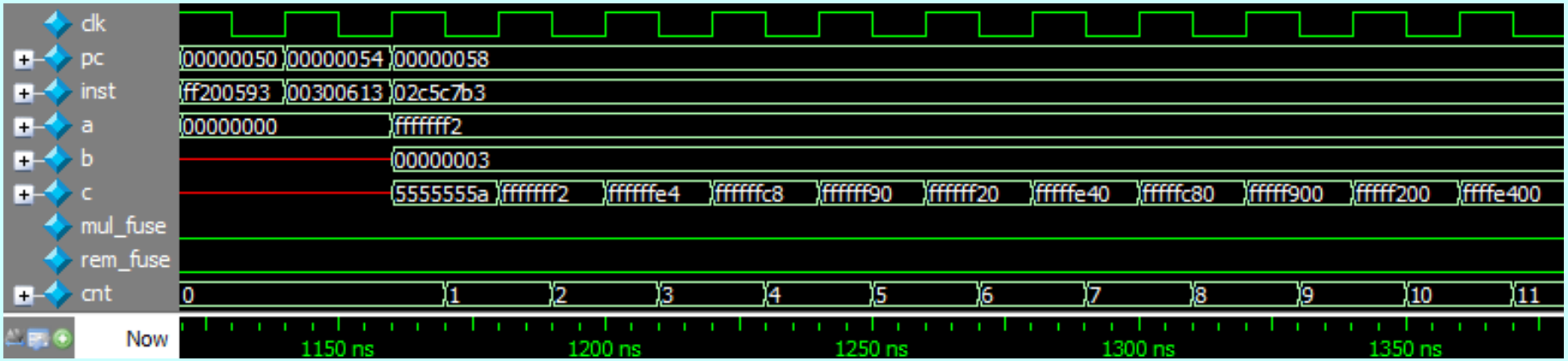


div-rem: 0x7fffffff / 3

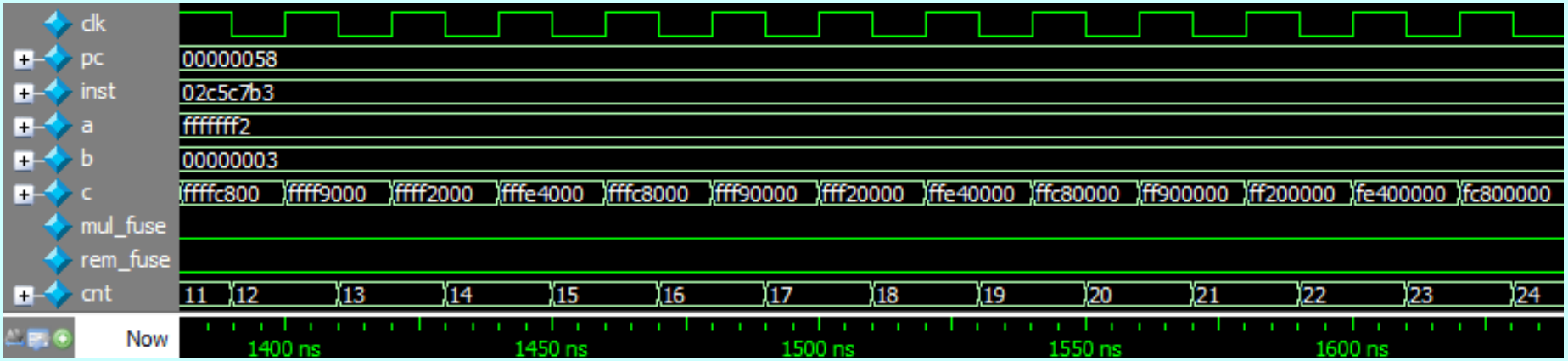


div q = 0x2aaaaaaa, rem r = 0x00000001 (rem fuse)

Simulation Waveform

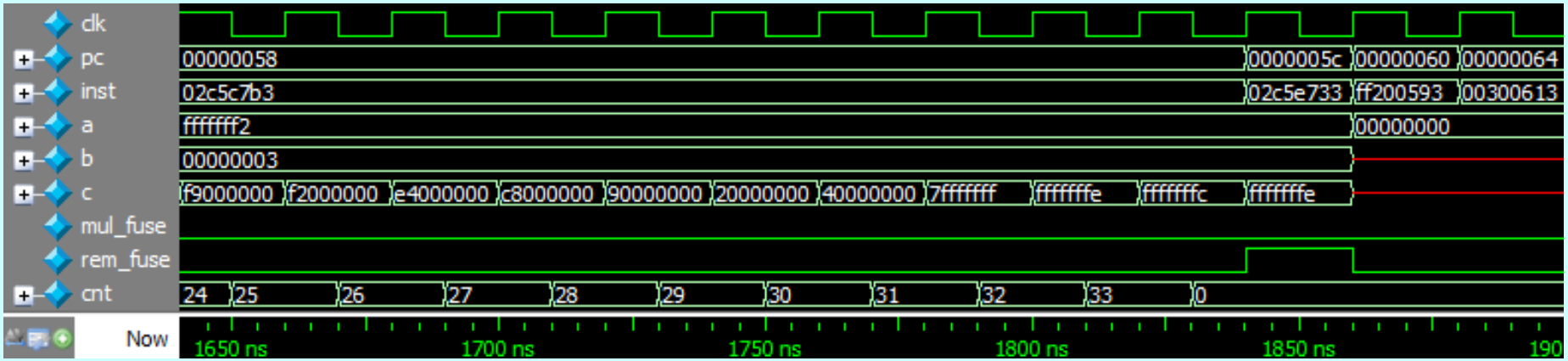


div-rem: 0xfffffff2 / 3

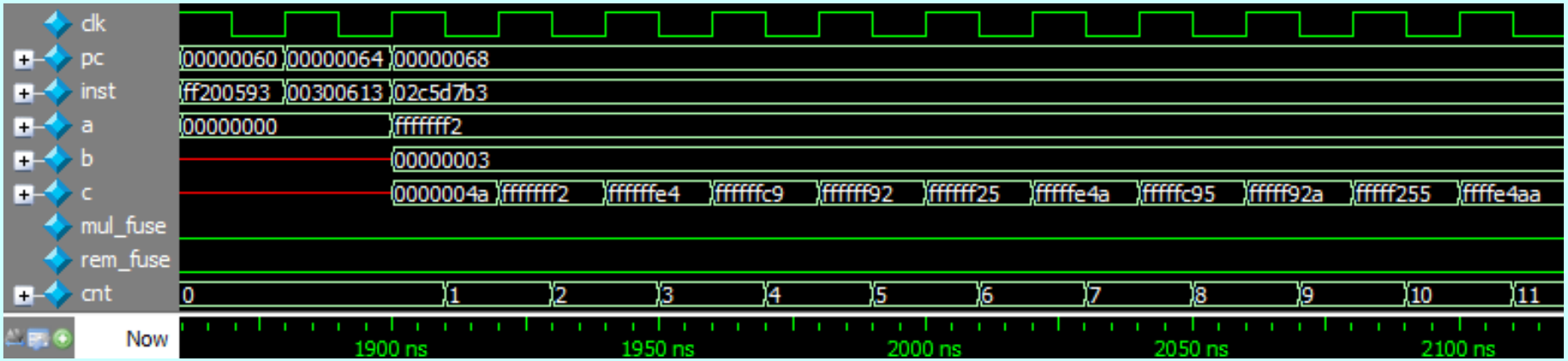


div-rem: 0xfffffff2 / 3

Simulation Waveform

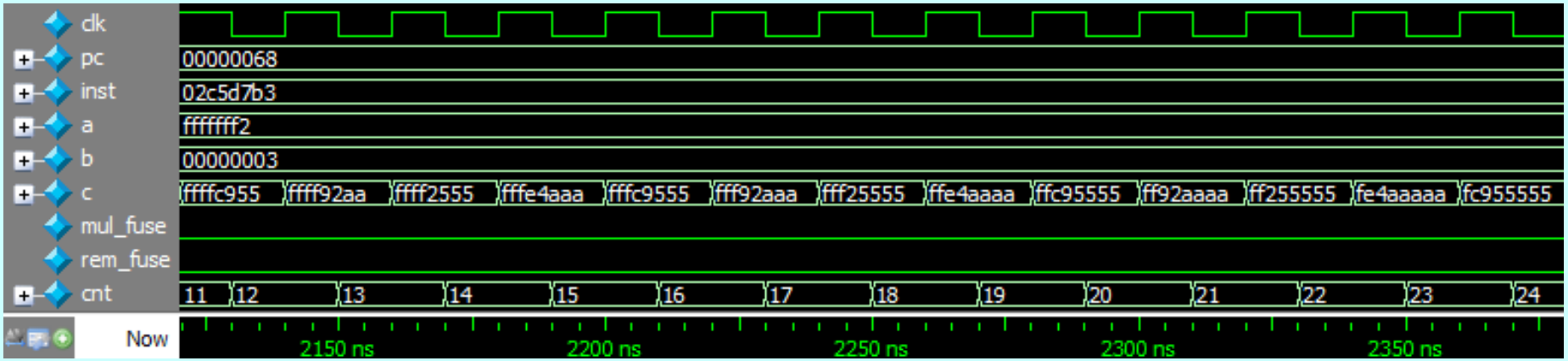


div q = 0xfffffffc, rem r = 0xfffffffe (rem fuse)

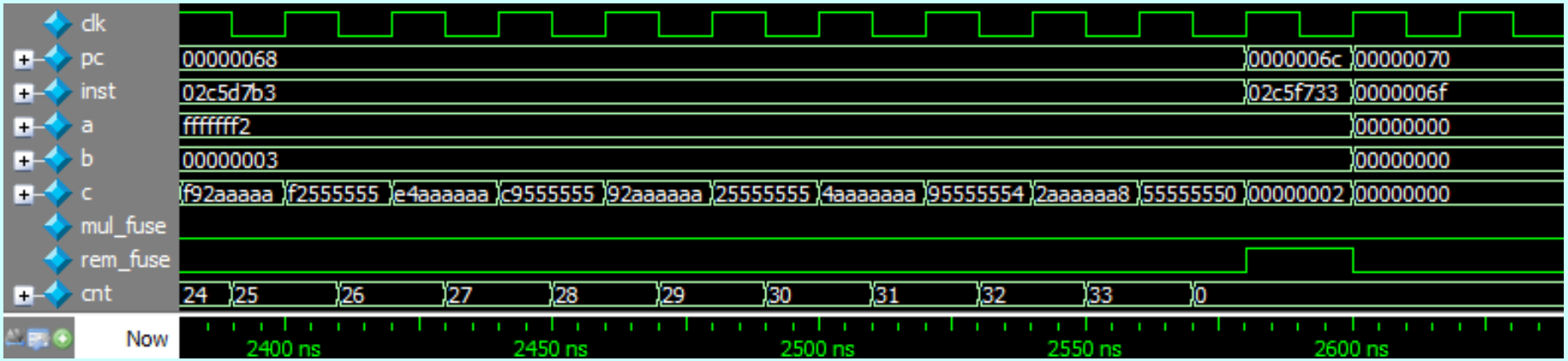


divu-remu: 0xfffffff2 / 3

Simulation Waveform



divu-remu: 0xffffffff2 / 3



divu q = 0x55555550, remu r = 0x00000002 (remu fuse)

課題 VI (500 点 + 100 点)

単一サイクルコンピュータ sc_computer を設計とシミュレーションしなさい。シミュレーション時の出力に関して、どんな命令を実行しているのか、どのような演算がされているかを自分で計算した結果と比較して波形の説明を入れること (500 点)。

オプション (+100 点):
Design and simulate an RISC-V CPU RV32IM that can execute RISC-V integer instructions as well eight multiplication and division instructions (see P47-P57).