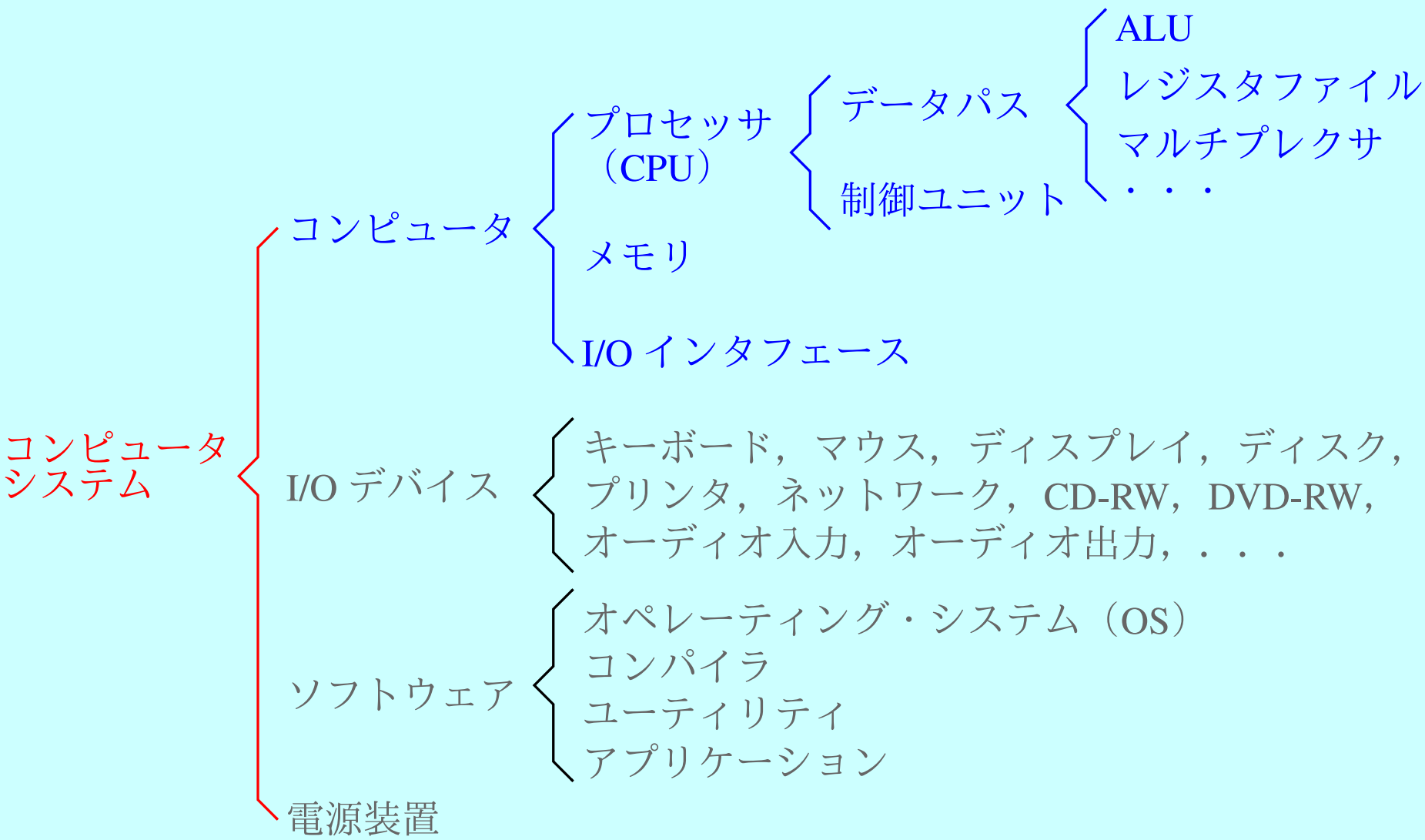
コンピュータ構成と設計（３）

CPU 構成要素の設計

李　亜民

2022 年 10 月 10 日 (月)

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
```

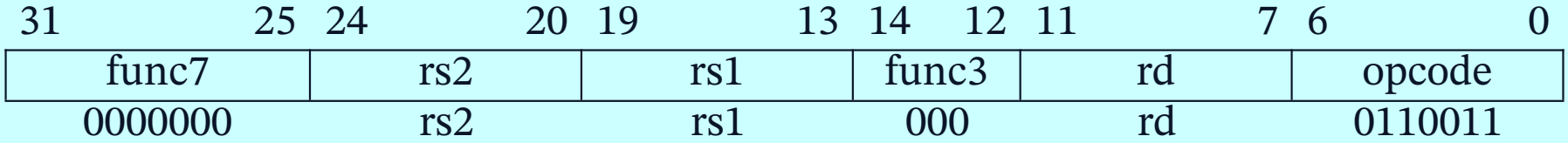
RV32I Base Instruction Set Encoding

0000000	rs2	rs1	000	rd	0110011
0100000	rs2	rs1	000	rd	0110011
0000000	rs2	rs1	010	rd	0110011
0000000	rs2	rs1	100	rd	0110011
0000000	rs2	rs1	110	rd	0110011
0000000	rs2	rs1	111	rd	0110011
0000000	shamt	rs1	001	rd	0010011
0000000	shamt	rs1	101	rd	0010011
0100000	shamt	rs1	101	rd	0010011
imm[11:0]		rs1	000	rd	1100111
imm[11:0]		rs1	000	rd	0010011
imm[11:0]		rs1	100	rd	0010011
imm[11:0]		rs1	110	rd	0010011
imm[11:0]		rs1	111	rd	0010011
imm[11:0]		rs1	010	rd	0000011
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011
imm[20|10:1|11|19:12]				rd	1101111
imm[31:12]				rd	0110111

1. add
2. sub
3. slt
4. xor
5. or
6. and
7. slli
8. srli
9. srai
10. jalr
11. addi
12. xori
13. ori
14. andi
15. lw
16. sw
17. beq
18. bne
19. jal
20. lui

マルチプレクサ (Multiplexer) の設計

add (Add) 命令




```
add  rd, rs1, rs2     # rd <- rs1 + rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値を加算して，その結果をレジスタ rd に格納する。

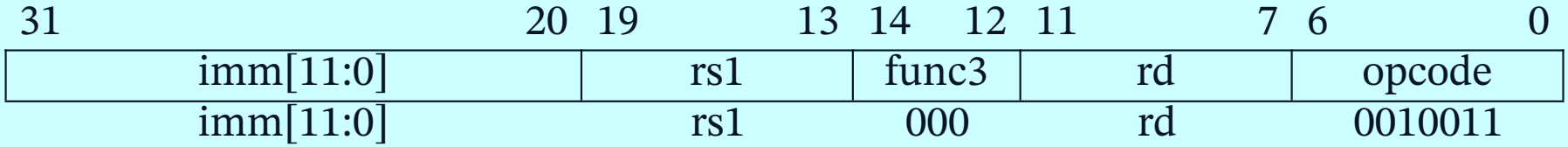
例:

```
add  x8, x6, x7
```

For example,

```
if   reg[6] = 2
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
            + 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0101  ( 5)
```

addi (Add Immediate) 命令




```
addi rd, rs1, imm    # rd <- rs1 + $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm を加算して，その結果をレジスタ rd に格納する。

例:

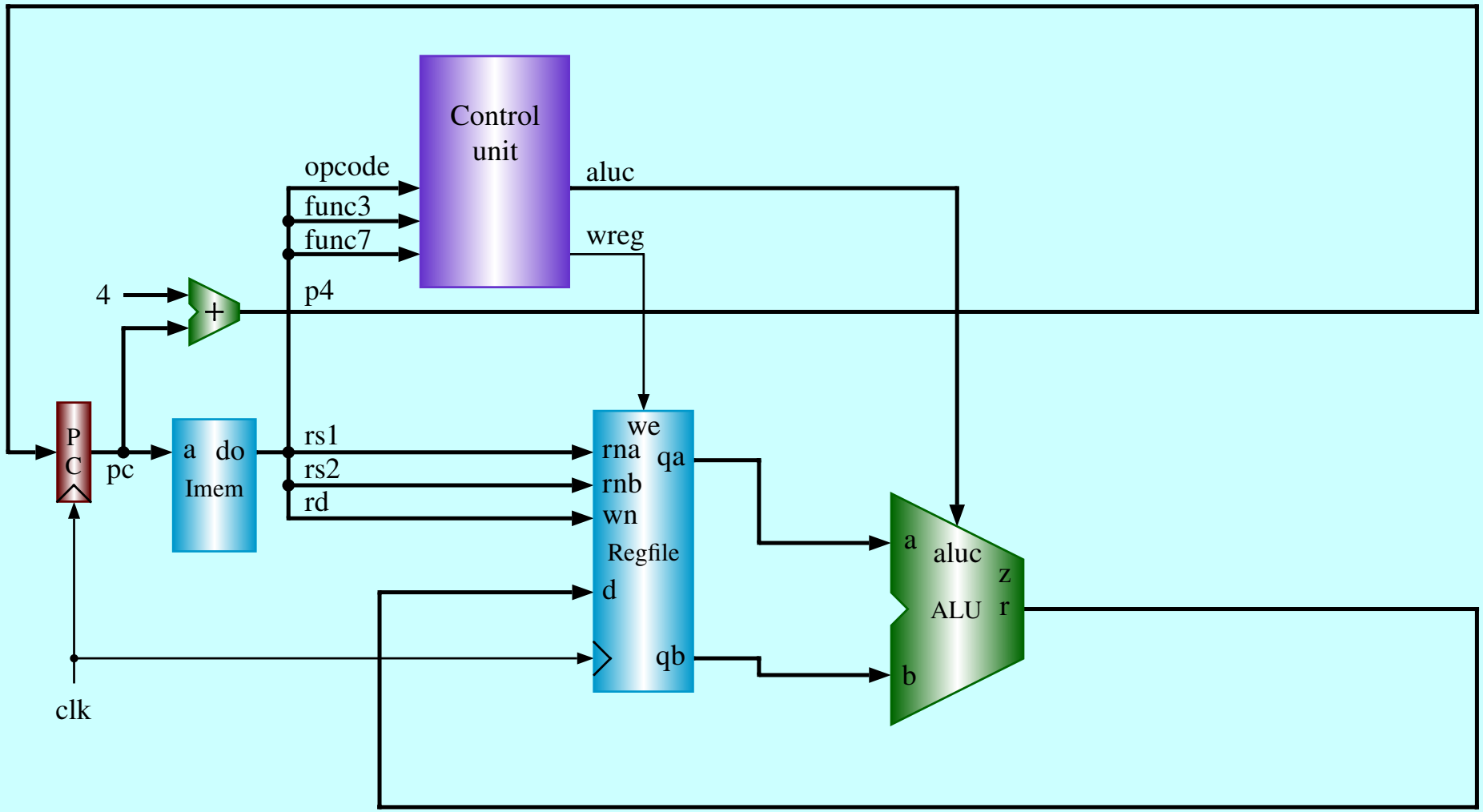
```
addi x8, x6, -1
```

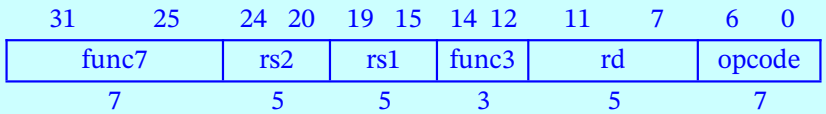
For example,

```
if   reg[6] = 2

then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
            + 1111 1111 1111 1111 1111 1111 1111 1111  (-1)
            = 0000 0000 0000 0000 0000 0000 0000 0001  ( 1)
```

RISC-V add の回路

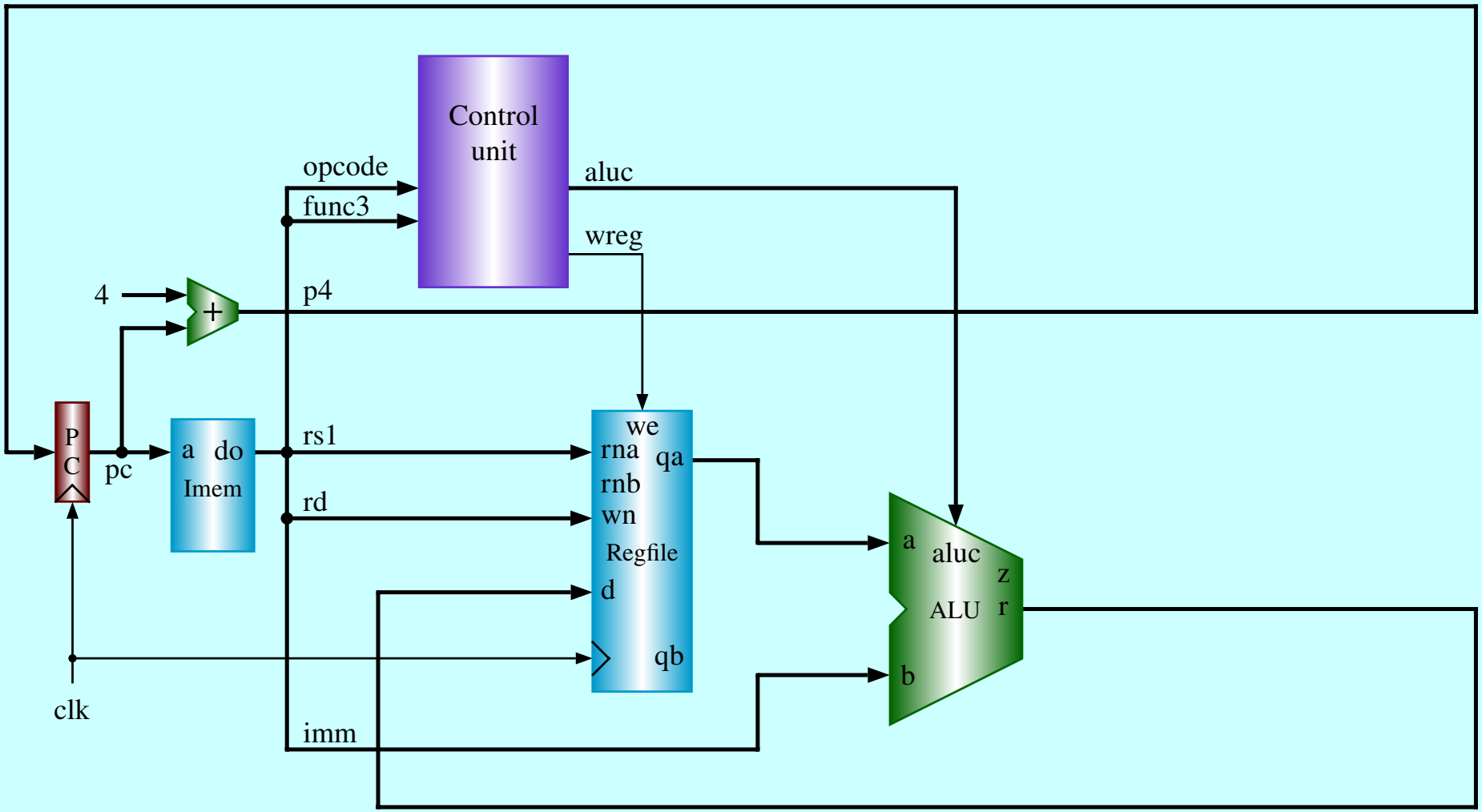




R-type

add rd, rs1, rs2 # rd ← rs1 + rs2

RISC-V addi の回路

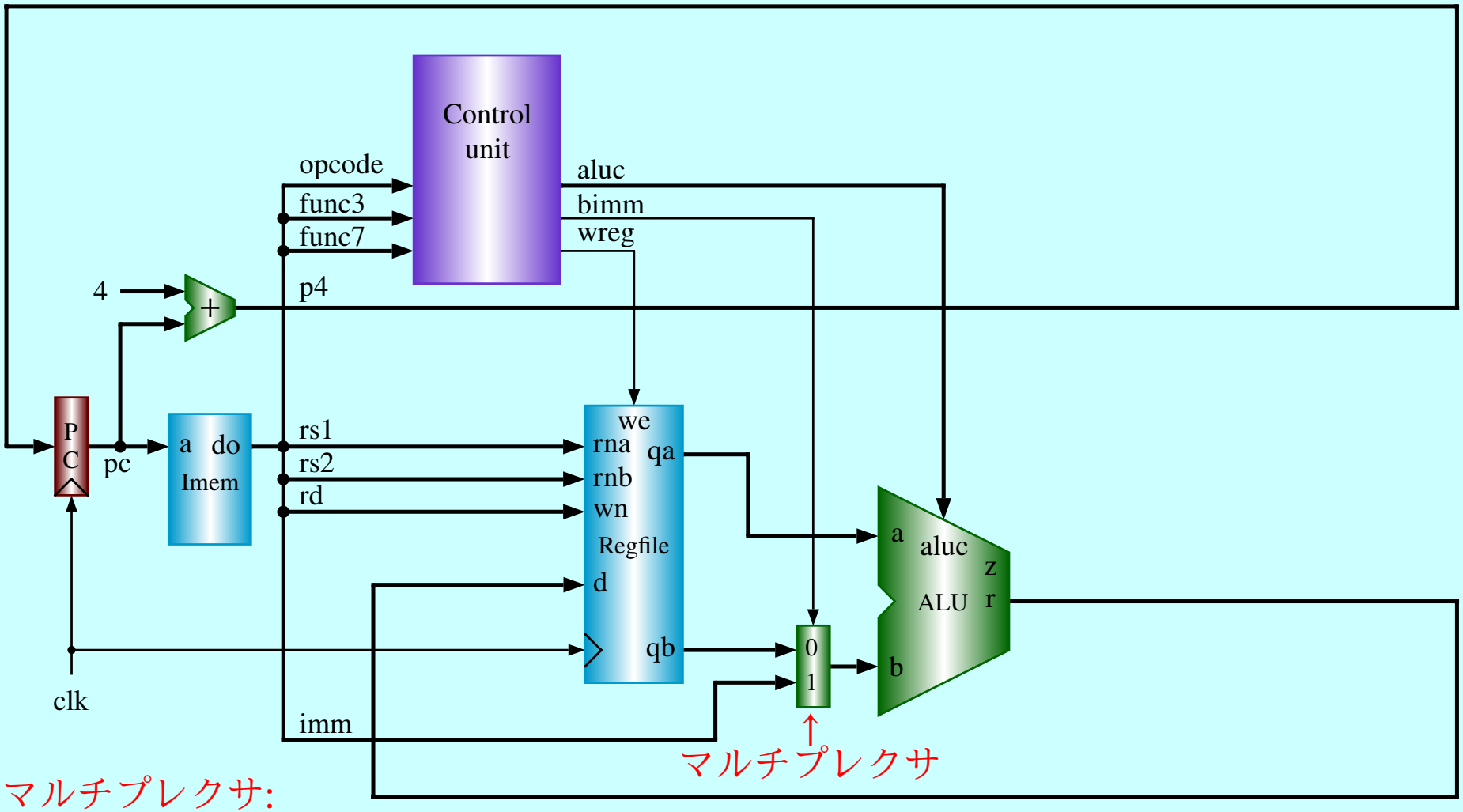


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	func3	rd	opcode
12	5	3	5	7

I-type

addi rd, rs1, imm　　# rd ← rs1 + $signed(imm[11:0])

RISC-V addi と add の回路



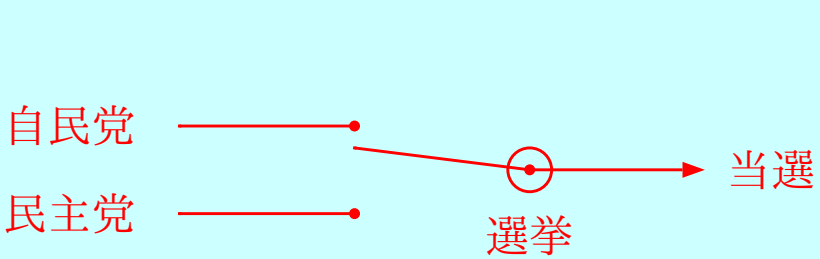
マルチプレクサ:

add 命令の場合: bimm = 0, ALU 入力 b = qb (レジスターデータを選択)

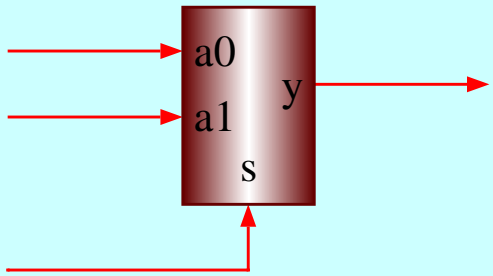
addi 命令の場合: bimm = 1, ALU 入力 b = imm (命令の中の即値を選択)

マルチプレクサの設計

マルチプレクサ (Multiplexer) は、ふたつ以上の入力をひとつの信号として出力する回路である。 (マルチプレクサ＝データ選択器)

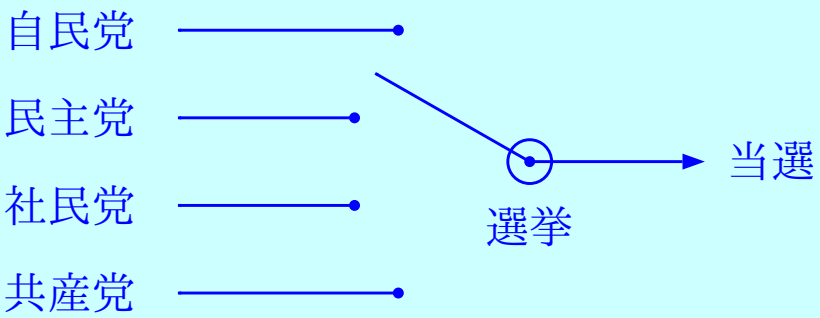


2-to-1 マルチプレクサ

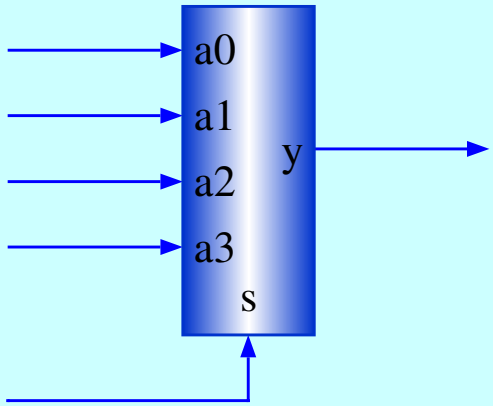


if s = 0, y = a0
if s = 1, y = a1

s: 1 bit



4-to-1 マルチプレクサ



if s = 00, y = a0
if s = 01, y = a1
if s = 10, y = a2
if s = 11, y = a3

s: 2 bits

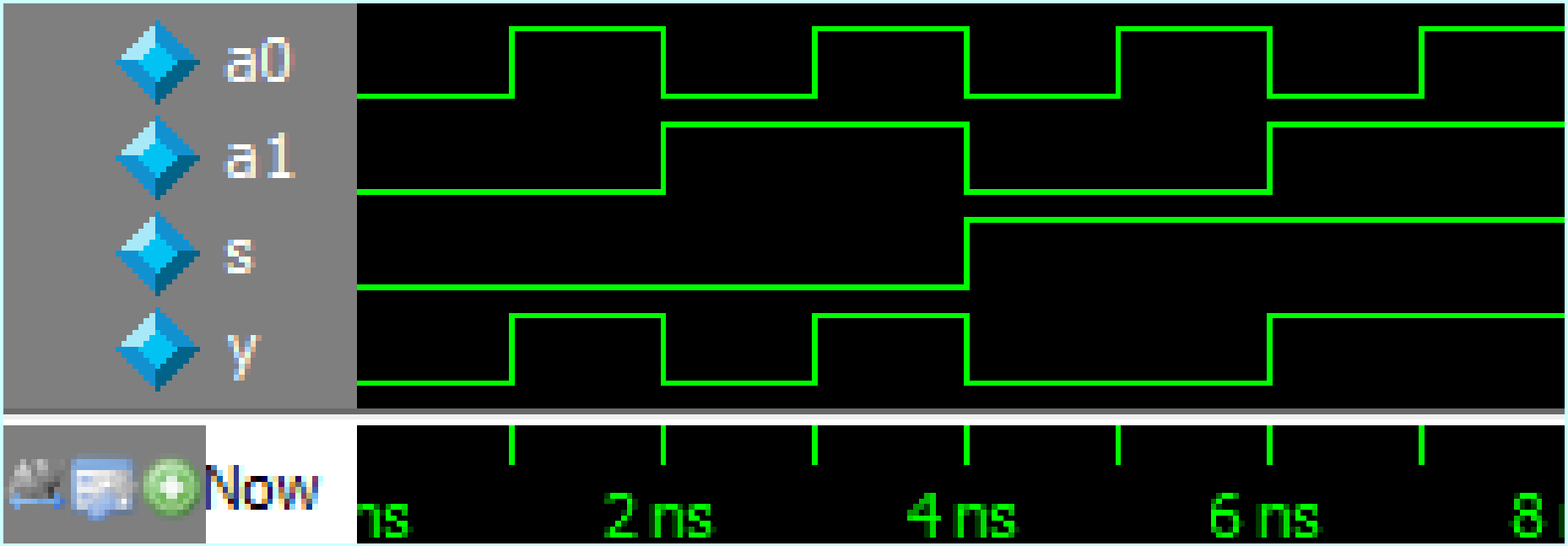
マルチプレクサの設計 mux2x1

```
module mux2x1_dataflow (a0,a1,s,y);
    input  a0, a1;
    input  s;
    output y;
    assign y = ~s & a0 | s & a1;
endmodule
```

マルチプレクサの設計 mux2x1

```
‘timescale 1ns/1ns
module mux2x1_dataflow_tb;
    reg  a0, a1;
    reg  s;
    wire y;
    mux2x1_dataflow i0 (a0,a1,s,y);
    initial begin
        #0 a0 = 0; a1 = 0; s = 0;
        #8 $stop;
    end
    always #1 a0 = ~a0;
    always #2 a1 = ~a1;
    always #4  s = ~s;
endmodule
```

マルチプレクサの設計 mux2x1



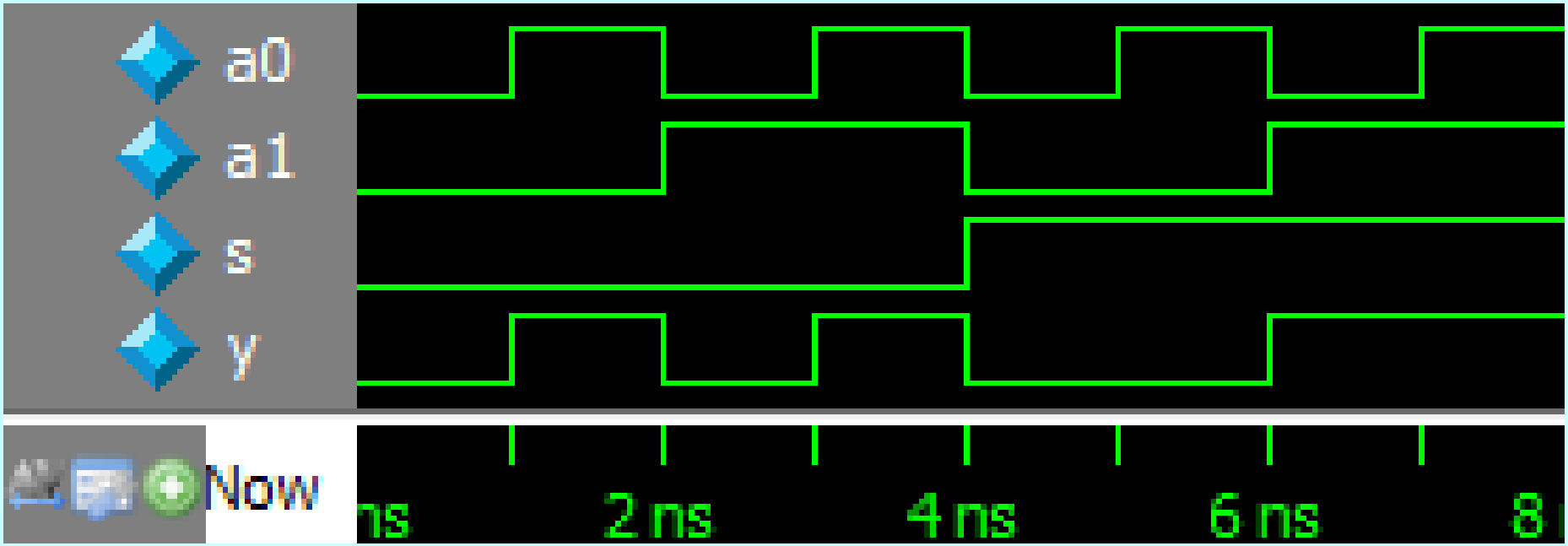
マルチプレクサの設計 mux2x1

```
module mux2x1 (a0,a1,s,y);
    input  a0, a1;
    input  s;
    output y;
    assign y = s ? a1 : a0;
endmodule
```

マルチプレクサの設計 mux2x1

```
‘timescale 1ns/1ns
module mux2x1_tb;
    reg  a0, a1;
    reg  s;
    wire y;
    mux2x1 i0 (a0,a1,s,y);
    initial begin
        #0 a0 = 0; a1 = 0; s = 0;
        #8 $stop;
    end
    always #1 a0 = ~a0;
    always #2 a1 = ~a1;
    always #4  s = ~s;
endmodule
```

マルチプレクサの設計 mux2x1



マルチプレクサの設計 mux2x1

```
module mux2x1  (a0,a1,s,y);
    input          a0, a1;
    input          s;
    output         y;
    assign         y = s ? a1 : a0;
endmodule
```

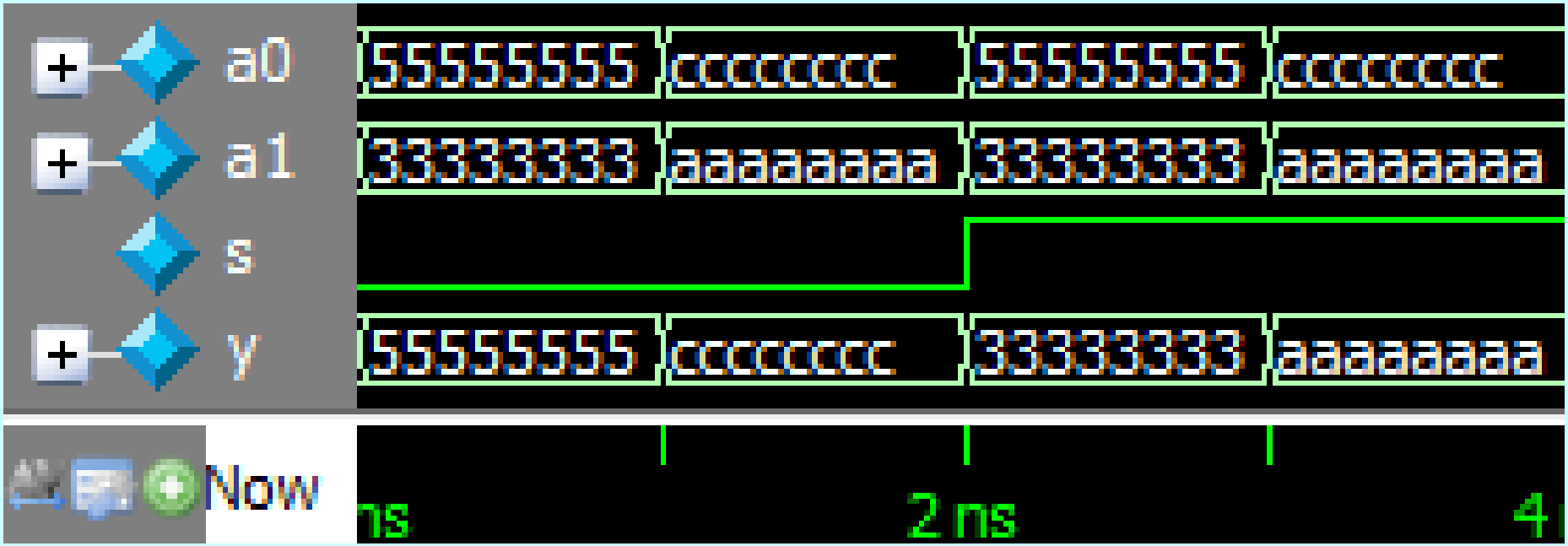
マルチプレクサの設計 mux2x32

```
module mux2x32 (a0,a1,s,y);
    input  [31:0] a0, a1;
    input         s;
    output [31:0] y;
    assign        y = s ? a1 : a0;
endmodule
```

マルチプレクサの設計 mux2x32

```
‘timescale 1ns/1ns
module mux2x32_tb;
    reg  [31:0] a0, a1;
    reg         s;
    wire [31:0] y;
    mux2x32 i0 (a0,a1,s,y);
    initial begin
        #0 a0 = 32’h55555555; a1 = 32’h33333333; s = 0;
        #1 a0 = 32’hcccccccc; a1 = 32’haaaaaaaa;
        #1 a0 = 32’h55555555; a1 = 32’h33333333; s = 1;
        #1 a0 = 32’hcccccccc; a1 = 32’haaaaaaaa;
        #1 $stop;
    end
endmodule
```

マルチプレクサの設計 mux2x32



beq (Branch on Equal) 命令

31 25	24 20	19 13	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011

```
beq  rs1, rs2, label # if (rs1 == rs2) pc <- pc + $signed({imm[12:1],0})
```

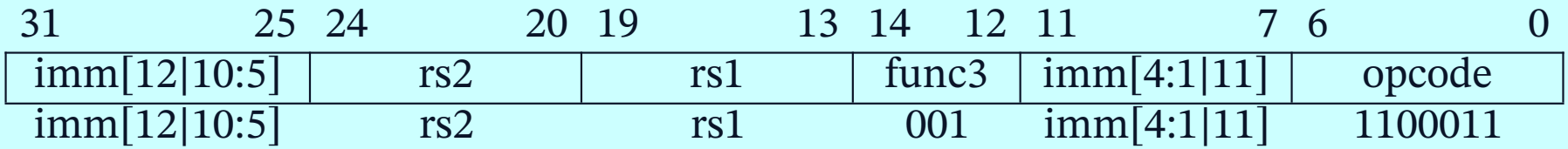
命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値が等しければ，分岐する。
分岐先アドレスは pc + $signed({imm[12:1],0}) である。例:

```
beq  x8, x0, label
```

For example,

```
if   reg[8] == 0 (equal, note: reg[0] = 0)
then pc = pc + $signed({imm[12:1],0})
else pc = pc + 4
```

bne (Branch on Not Equal) 命令



31 25	24 20	19 13	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011

```
bne  rs1, rs2, label # if (rs1 != rs2) pc <- pc + $signed({imm[12:1],0})
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値が等しくなければ，分岐する。

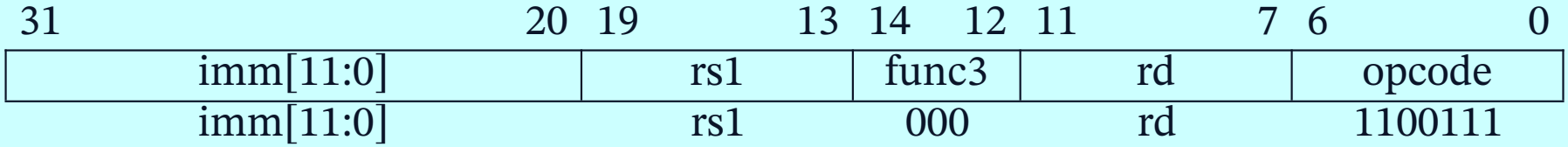
分岐先アドレスは pc + $signed({imm[12:1],0}) である。例:

```
bne  x8, x0, label
```

For example,

```
if   reg[8] != 0 (equal, note: reg[0] = 0)
then pc = pc + $signed({imm[12:1],0})
else pc = pc + 4
```

jalr (Jump And Link Register) 命令




```
jalr rd, rs1, imm # rd <- pc+4; pc <- (rs1+$signed(imm[11:0]))&0xfffffffe
```

命令の意味: pc + 4 の値をレジスタ rd に格納する。レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm を加算して，最下位ビットをゼロにして，その結果をジャンプ先アドレスとしてジャンプする。例:

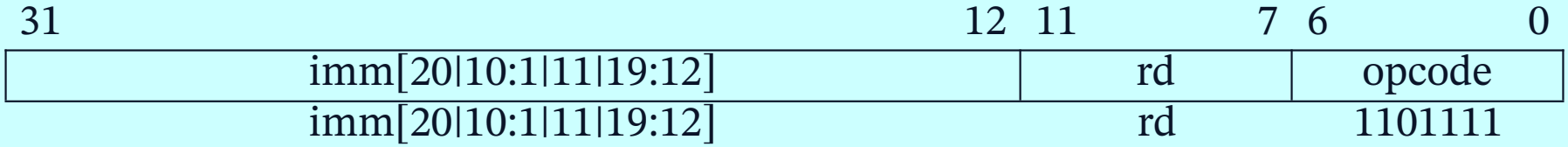
```
jalr x0, x1, 0          # = ret ra
```

For example,

```
if   reg[1] = 0000 0000 0000 0000 0000 0000 1111 0001

then pc     = 0000 0000 0000 0000 0000 0000 1111 0000
```

jal (Jump And Link) 命令




```
jal  rd, label        # rd <- pc + 4; pc <- pc + $signed({imm[20:1],0})
```

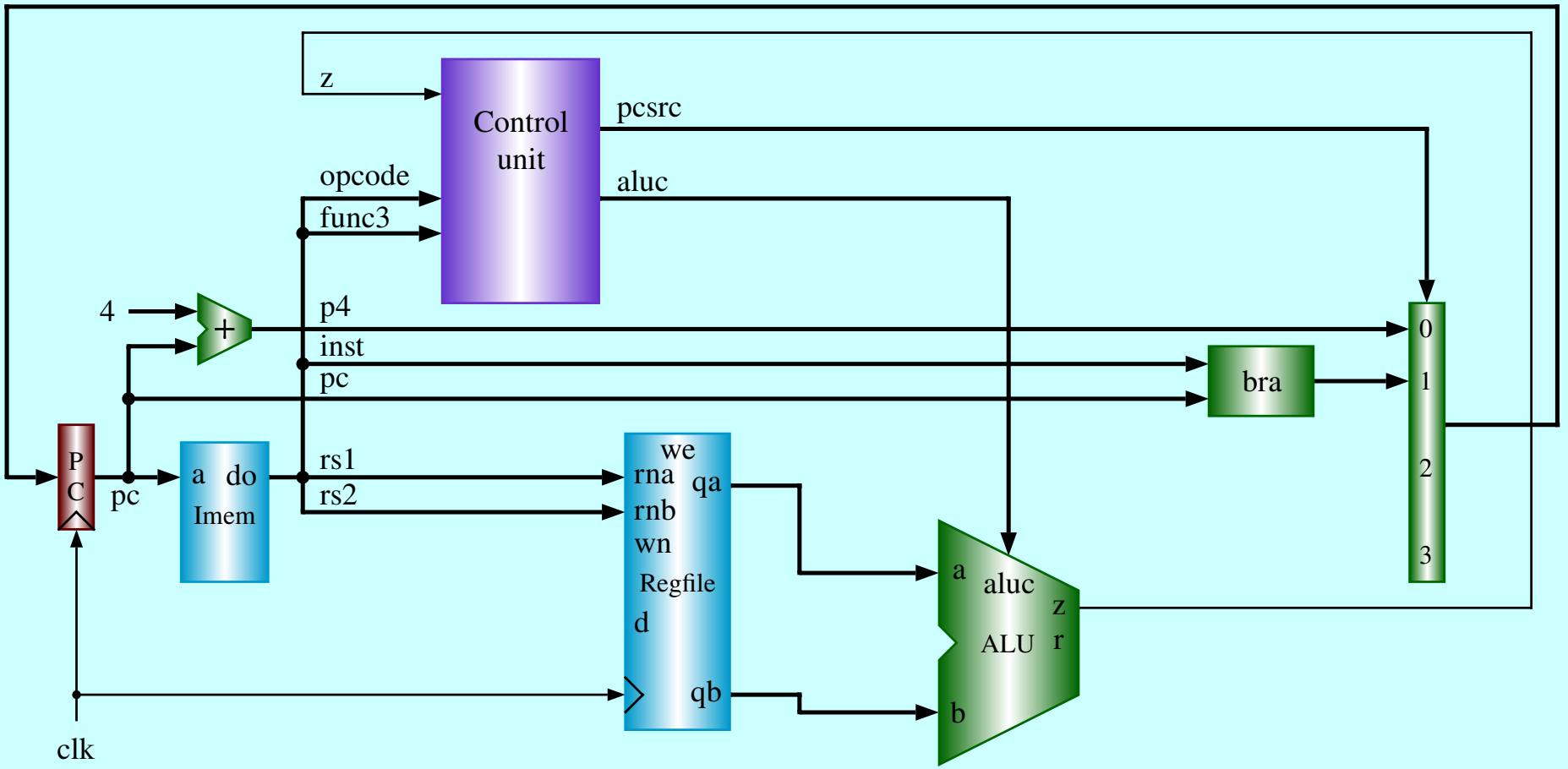
命令の意味: pc + 4 の値をレジスタ rd に格納する。pc に格納されている値と 2 の補数で表される即値 imm を加算して，その結果をジャンプ先アドレスとしてジャンプする。例:

```
jal  x1, subroutine  # = call subroutine
```

For example,

```
reg[1] = pc + 4       # save return address, use ret x1 to return
pc = subroutine       # jump to subroutine
```

RISC-V beq と bne の回路

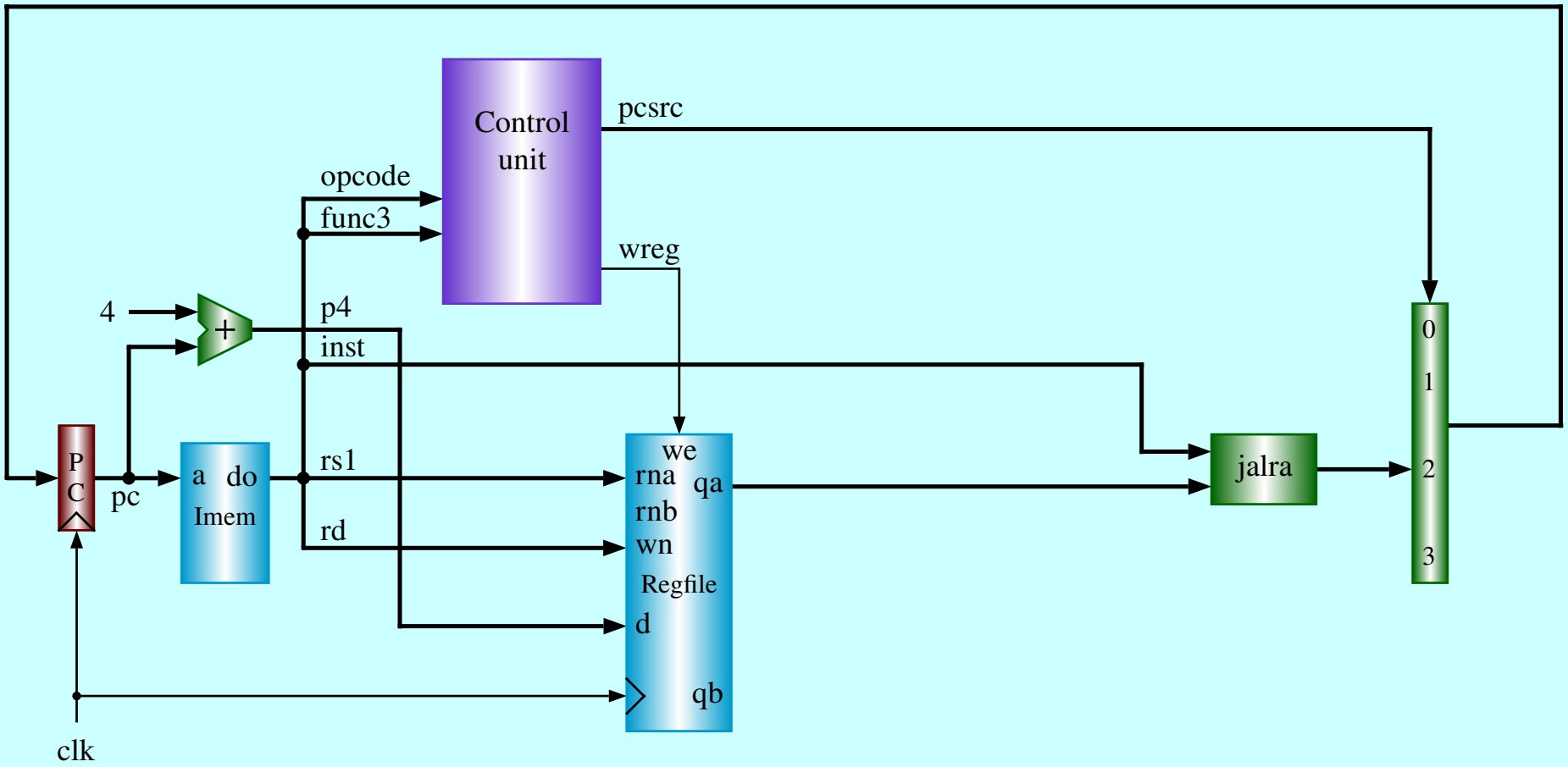


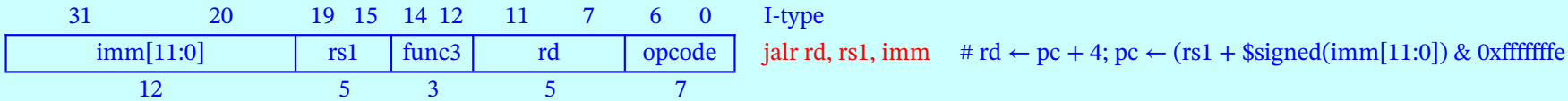
31 25	24 20	19 15	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
7	5	5	3	5	7

B-type

beq rs1, rs2, label　# if (rs1 = rs2) pc ← pc + $signed({imm[12:1],0})

RISC-V jalr の回路

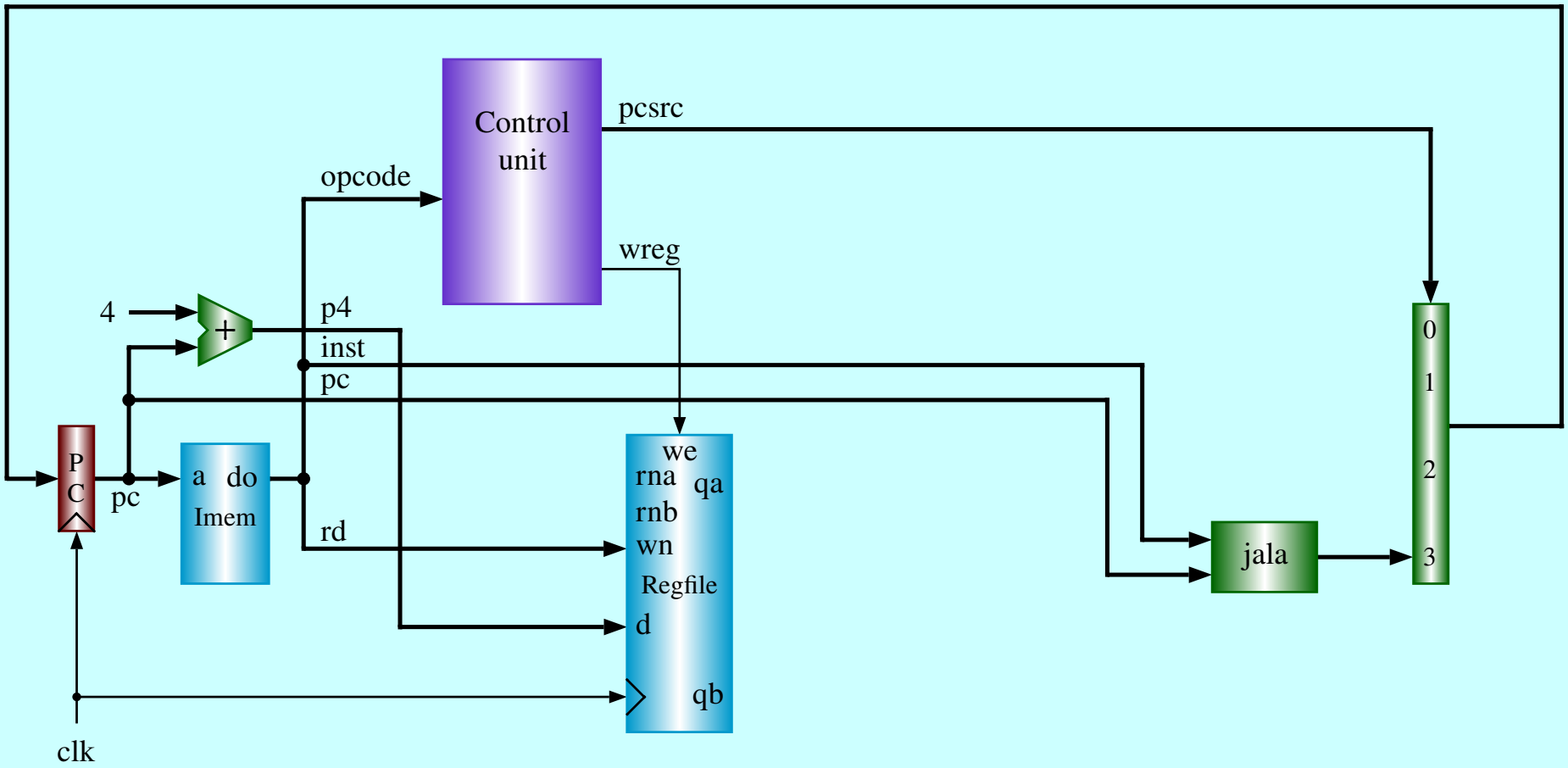


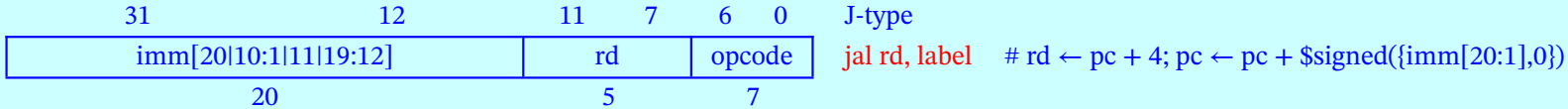


I-type

jalr rd, rs1, imm　# rd ← pc + 4; pc ← (rs1 + $signed(imm[11:0]) & 0xfffffffe

RISC-V jal の回路

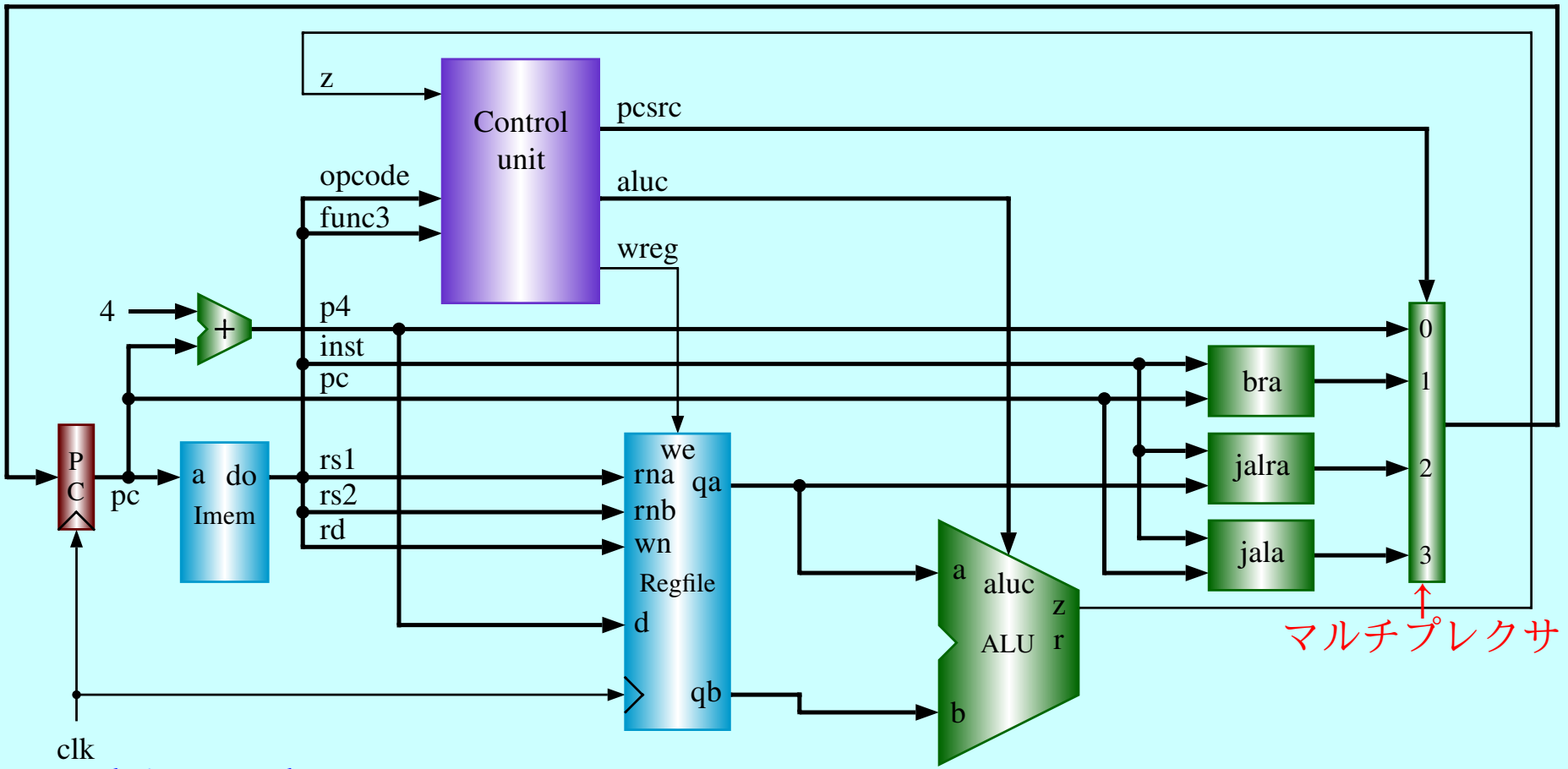




J-type

jal rd, label　# rd ← pc + 4; pc ← pc + $signed({imm[20:1],0})

RISC-V Branch と Jump の回路



Branch is not taken: pcsrc = 00, next pc = p4

Branch is taken: pcsrc = 01, next pc = bra

jalr: pcsrc = 10, next pc = jalra

jal: pcsrc = 11, next pc = jala

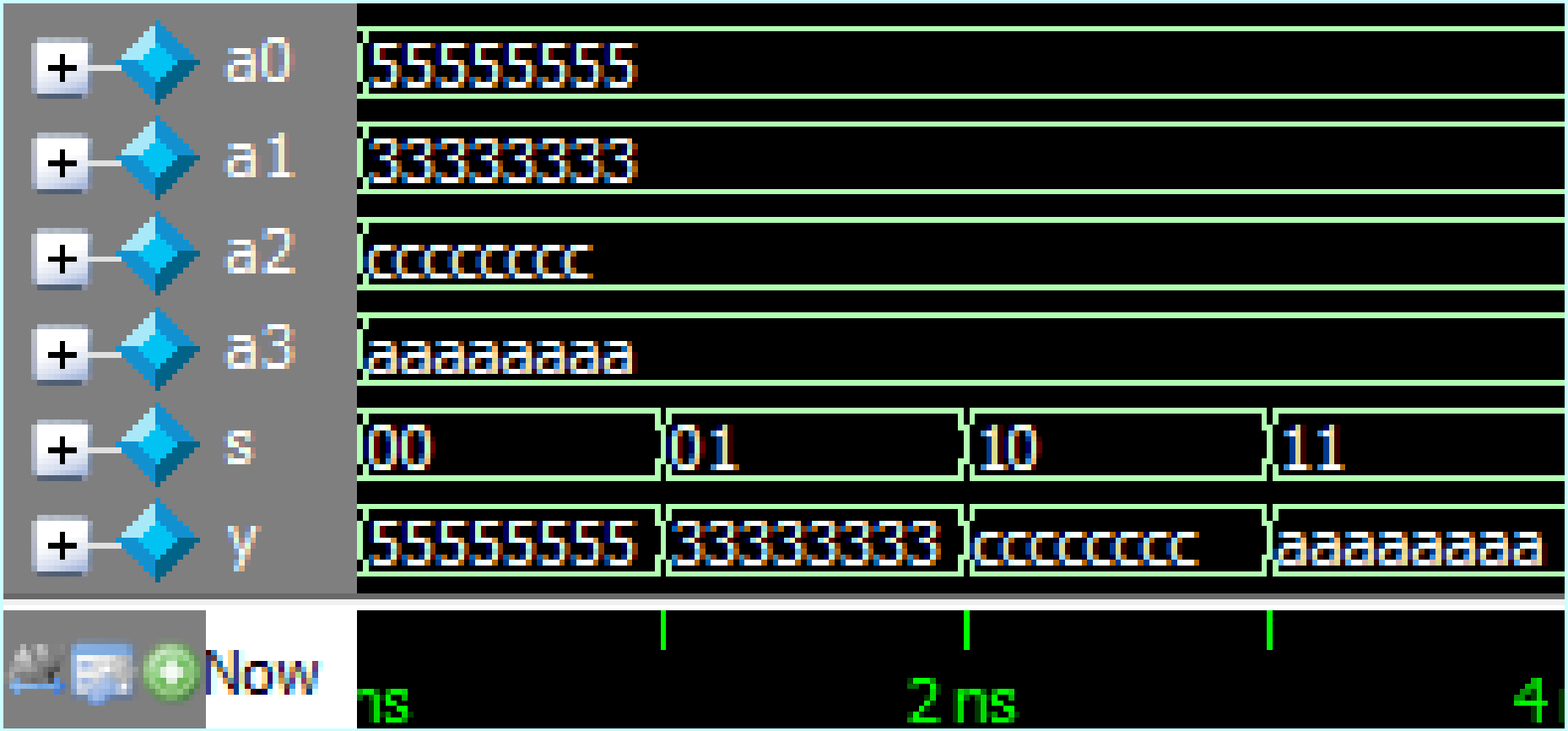
マルチプレクサの設計 mux4x32

```
module mux4x32 (a0,a1,a2,a3,s,y);
    input  [31:0] a0, a1, a2, a3;
    input   [1:0] s;
    output [31:0] y;
    assign y  = s[1] ? s[0] ? a3 : a2 : s[0] ? a1 : a0;
    //     s[1:0]:            11   10          01   00
endmodule
```

マルチプレクサの設計 mux4x32

```
‘timescale 1ns/1ns
module mux4x32_tb;
    reg  [31:0] a0, a1, a2, a3;
    reg   [1:0] s;
    wire [31:0] y;
    mux4x32 i0 (a0,a1,a2,a3,s,y);
    initial begin
        #0 a0 = 32’h55555555; a1 = 32’h33333333;
        #0 a2 = 32’hcccccccc; a3 = 32’haaaaaaaa;
        #0 s  =  2’b00;
        #4 $stop;
    end
    always #1 s = s + 2’d1;
endmodule
```

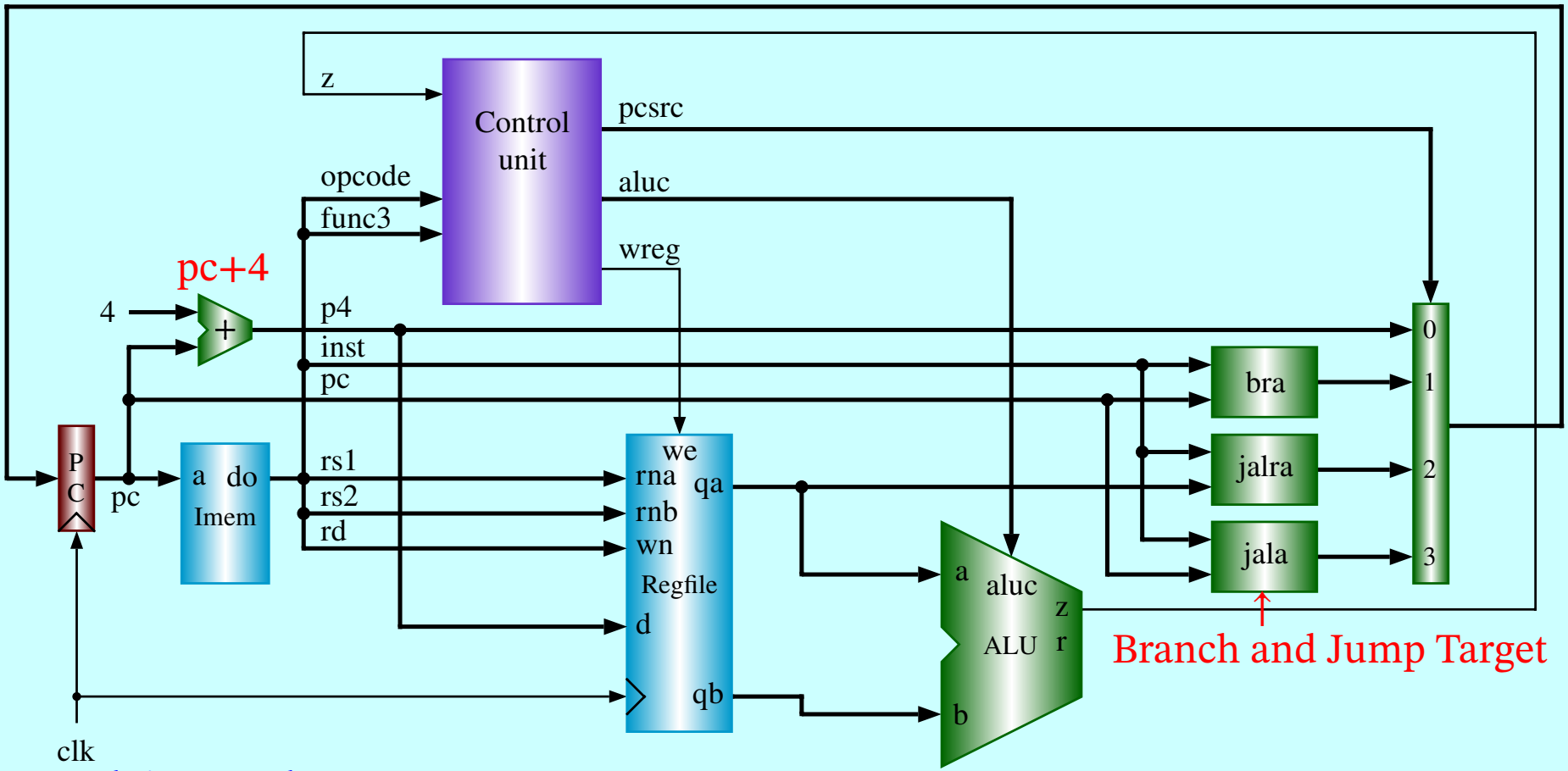
マルチプレクサの設計 mux4x32



NPCの設計

(Next Program Counter)

RISC-V Branch と Jump の回路



Branch is not taken: pcsrc = 00, next pc = p4

Branch is taken: pcsrc = 01, next pc = bra

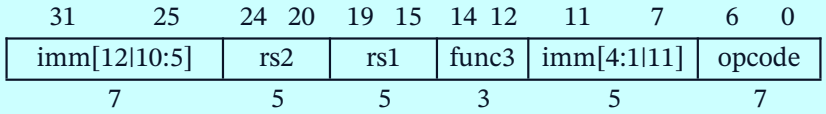
jalr: pcsrc = 10, next pc = jalra

jal: pcsrc = 11, next pc = jala

PC+4（p4）の設計

```
module pc4 (pc,p4);
    input  [31:0] pc;
    output [31:0] p4;
    assign p4 = pc + 32'h4;
endmodule
```

Branch Target（bra）の設計

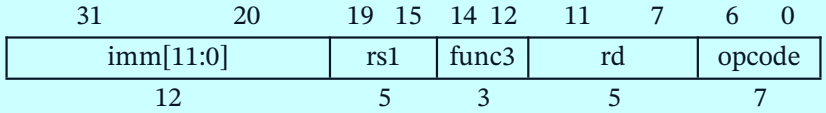


B-type

beq rs1, rs2, label　# if (rs1 = rs2) pc ← pc + $signed({imm[12:1],0})

```
module branch_addr (pc,inst,addr);
    input  [31:0] pc,inst;
    output [31:0] addr;                          // bits
    wire   [31:0] offset = {{20{inst[31]}},      // [31:12]
                            inst[7],             //    [11]
                            inst[30:25],         // [10:05]
                            inst[11:8],          // [04:01]
                            1'b0};               //    [00]
    assign addr = pc + offset;
endmodule
```

Jump Register Target（jalra）の設計

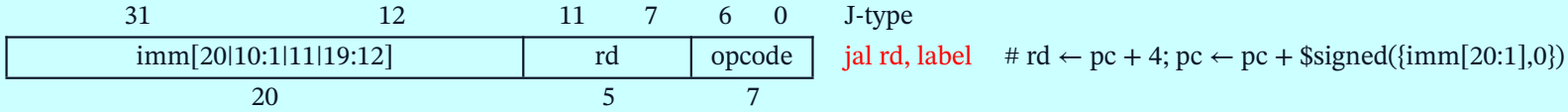


I-type

jalr rd, rs1, imm　# rd ← pc + 4; pc ← (rs1 + $signed(imm[11:0]) & 0xfffffffe

```
module jalr_addr (rs1,inst,addr);
    input  [31:0] rs1,inst;
    output [31:0] addr;                          // bits
    wire   [31:0] s_imm  = {{21{inst[31]}},   // [31:11]
                                inst[30:20]}; // [10:00]
    wire   [31:0] target = rs1 + s_imm;
    assign addr = {target[31:1],1'b0};
endmodule
```

Jump Target（jala）の設計



J-type

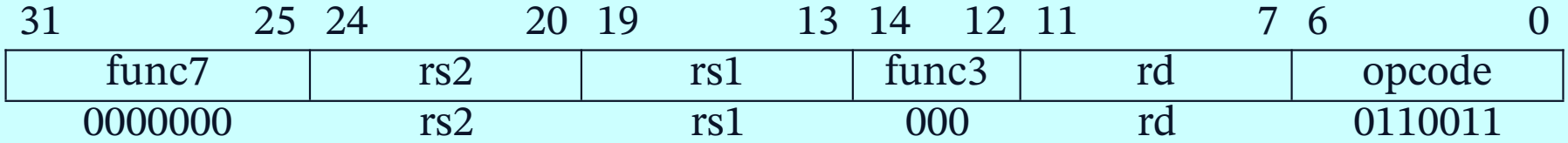
jal rd, label　# rd ← pc + 4; pc ← pc + $signed({imm[20:1],0})

```
module jal_addr (pc,inst,addr);
    input  [31:0] pc,inst;
    output [31:0] addr;                          // bits
    wire   [31:0] offset = {{12{inst[31]}},      // [31:20]
                            inst[19:12],         // [19:12]
                            inst[20],            //    [11]
                            inst[30:21],         // [10:01]
                            1'b0};               //    [00]
    assign addr = pc + offset;
endmodule
```

ALUの設計

(Arithmetic Logic Unit)

add (Add) 命令




```
add  rd, rs1, rs2     # rd <- rs1 + rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値を加算して，その結果をレジスタ rd に格納する。

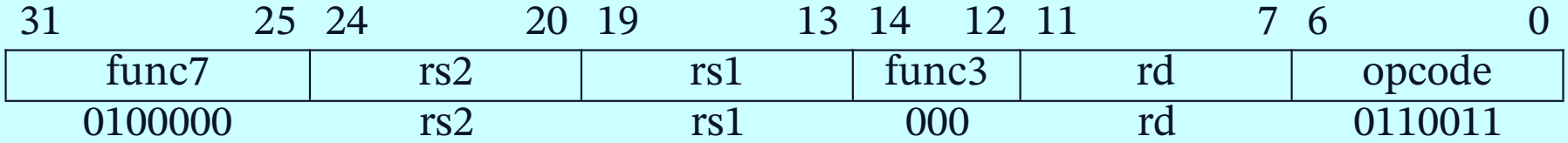
例:

```
add  x8, x6, x7
```

For example,

```
if   reg[6] = 2
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
            + 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0101  ( 5)
```

sub (Subtraction) 命令




```
sub  rd, rs1, rs2     # rd <- rs1 - rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値を減算して，その結果をレジスタ rs に格納する。

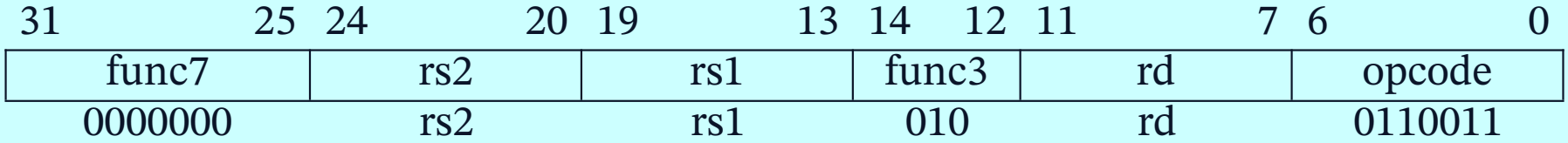
例:

```
sub  x8, x6, x7
```

For example,

```
if   reg[6] = 2
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
            - 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 1111 1111 1111 1111 1111 1111 1111 1111  (-1)
```

slt (Set on Less Than) 命令




```
slt  rd, rs1, rs2     # rd <- rs1 < rs2 (signed)
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値を比較して，その結果をレジスタ rd に格納する。

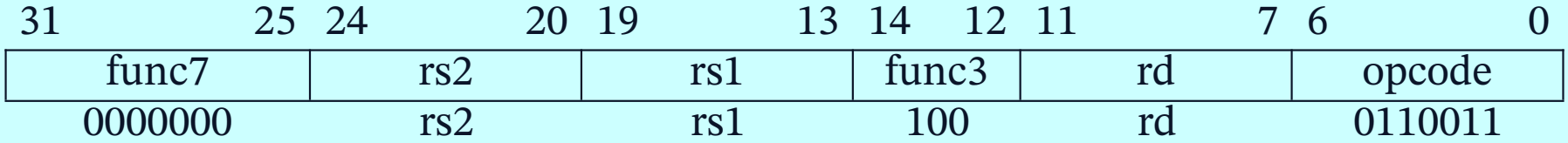
例:

```
slt  x8, x6, x7
```

For example,

```
if   reg[6] = 2
     reg[7] = 3
then reg[8] = 1 because 2 < 3 is true
if   reg[6] = 3
     reg[7] = 2
then reg[8] = 0 because 3 < 2 is false
```

xor (Exclusive Or) 命令




```
xor  rd, rs1, rs2     # rd <- rs1 ^ rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値の排他的論理和を取り，その結果をレジスタ rd に格納する。

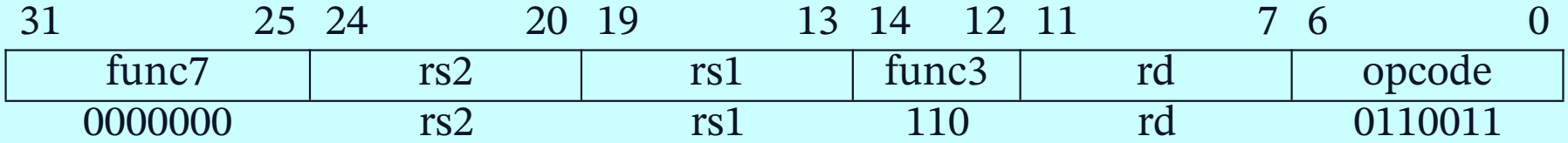
例:

```
xor  x8, x6, x7
```

For example,

```
if   reg[6] = 6
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0110  ( 6)
            ^ 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0101  ( 5)
```

or (Or) 命令




```
or   rd, rs1, rs2    # rd <- rs1 | rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値の論理和を取り，その結果をレジスタ rd に格納する。

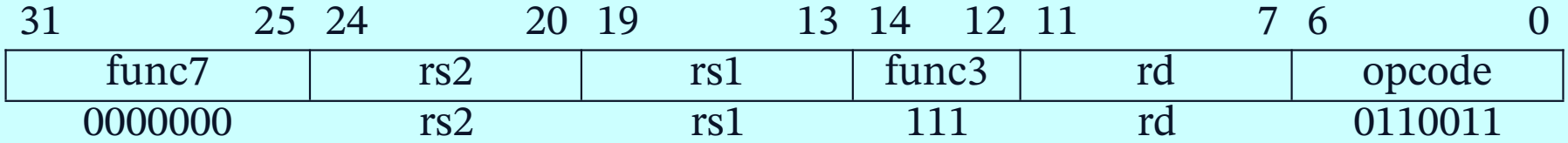
例:

```
or   x8, x6, x7
```

For example,

```
if   reg[6] = 6
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0110  ( 6)
            | 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0111  ( 7)
```

and (And) 命令




```
and  rd, rs1, rs2     # rd <- rs1 & rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値の論理積を取り，その結果をレジスタ rd に格納する。

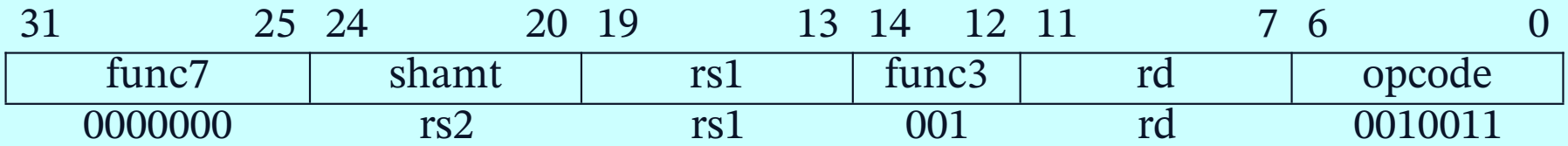
例:

```
and  x8, x6, x7
```

For example,

```
if   reg[6] = 6
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0110  ( 6)
            & 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
```

slli (Shift Left Logical Immediate) 命令




```
slli rd, rs1, shamt  # rd <- rs1 << shamt
```

命令の意味: レジスタ rs1 に格納されている値を shamt ビット左に論理シフトして，その結果をレジスタ rd に格納する。

例:

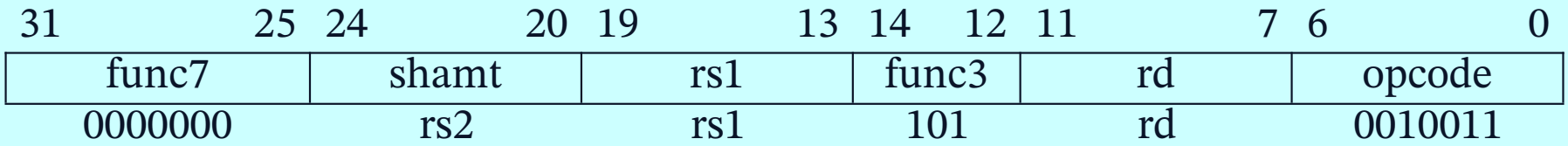
```
slli x8, x6, 4
```

For example,

```
if   reg[6] = 0000 0000 0000 0000 0000 0000 0000 0010

then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  << 4
            = 0000 0000 0000 0000 0000 0000 0010 0000
```

srli (Shift Right Logical Imm.) 命令




```
srli rd, rs1, shamt  # rd <- rs1 >> shamt
```

命令の意味: レジスタ rs1 に格納されている値を shamt ビット右に論理シフトして，その結果をレジスタ rd に格納する。

例:

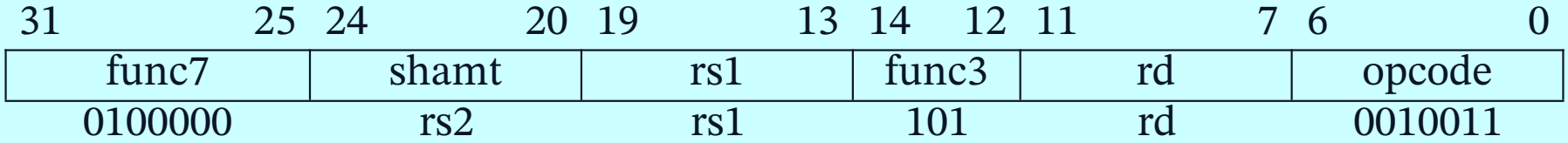
```
srli x8, x6, 4
```

For example,

```
if   reg[6] = 1111 0000 0000 0000 0000 0000 0000 0010

then reg[8] = 1111 0000 0000 0000 0000 0000 0000 0010  >> 4
            = 0000 1111 0000 0000 0000 0000 0000 0000
```

srai (Shift Right Arithmetic Imm.) 命令




```
srai rd, rs1, shamt  # rd <- rs1 >>> shamt
```

命令の意味: レジスタ rs1 に格納されている値を shamt ビット右に算術シフトして，その結果をレジスタ rd に格納する。

例:

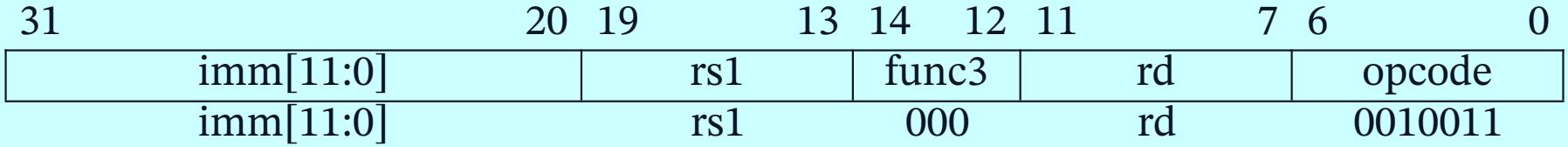
```
srai x8, x6, 4
```

For example,

```
if   reg[6] = 1111 0000 0000 0000 0000 0000 0000 0010

then reg[8] = 1111 0000 0000 0000 0000 0000 0000 0010 >>> 4
            = 1111 1111 0000 0000 0000 0000 0000 0000
```

addi (Add Immediate) 命令




```
addi rd, rs1, imm    # rd <- rs1 + $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm を加算して，その結果をレジスタ rd に格納する。

例:

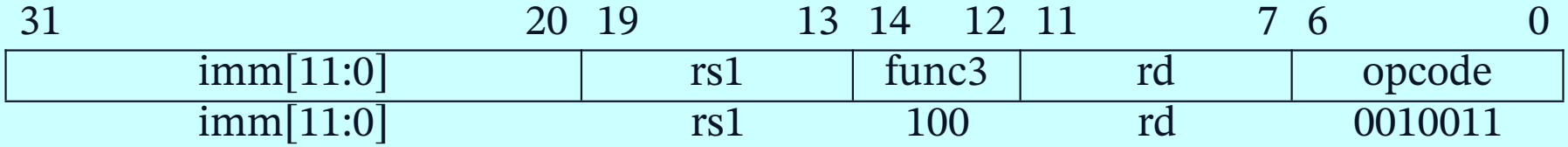
```
addi x8, x6, -1
```

For example,

```
if   reg[6] = 2

then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
            + 1111 1111 1111 1111 1111 1111 1111 1111  (-1)
            = 0000 0000 0000 0000 0000 0000 0000 0001  ( 1)
```

xori (Exclusive Or Immediate) 命令




```
xori rd, rs1, imm    # rd <- rs1 ^ $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の排他的論理和を取り，その結果をレジスタ rd に格納する。

例:

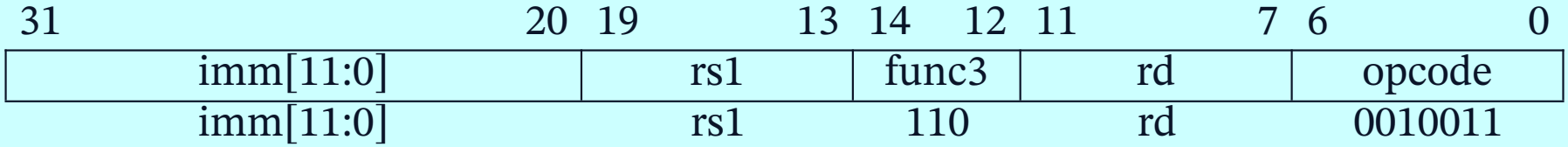
```
xori x8, x6, 0xf
```

For example,

```
if   reg[6] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1010

then reg[8] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1010
            ^ 0000 0000 0000 0000 0000 0000 0000 1111
            = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 0101
```

ori (Or Immediate) 命令




```
ori  rd, rs1, imm    # rd <- rs1 | $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の論理和を取り，その結果をレジスタ rd に格納する。

例:

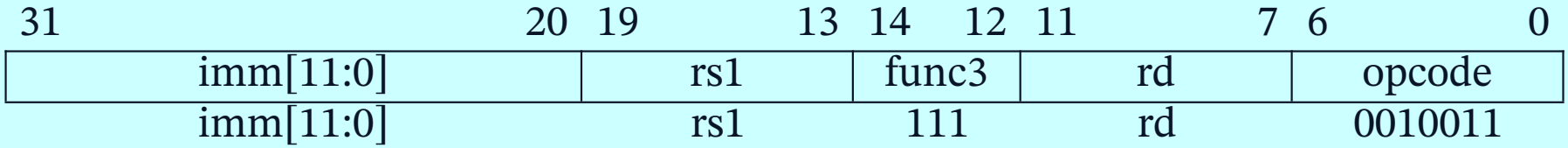
```
ori  x8, x6, 0xf
```

For example,

```
if   reg[6] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx

then reg[8] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
            | 0000 0000 0000 0000 0000 0000 0000 1111
            = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1111
```

andi (And Immediate) 命令




```
andi rd, rs1, imm    # rd <- rs1 & $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の論理積を取り，その結果をレジスタ rd に格納する。

例:

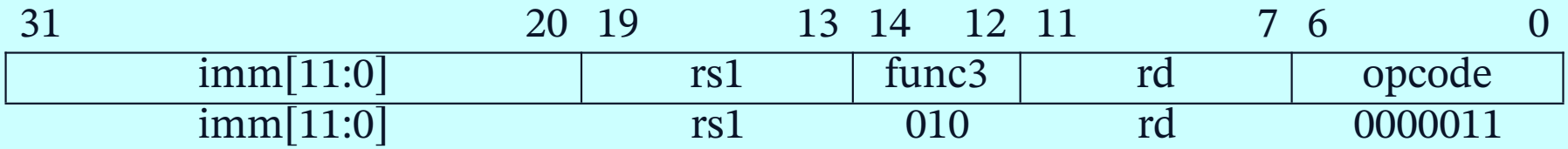
```
andi x8, x6, 0xf
```

For example,

```
if   reg[6] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1010

then reg[8] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1010
            & 0000 0000 0000 0000 0000 0000 0000 1111
            = 0000 0000 0000 0000 0000 0000 0000 1010
```

lw (Load Word) 命令




```
lw   rd,  imm(rs1)   # rd <- memory[rs1 + $signed(imm[11:0])]
```

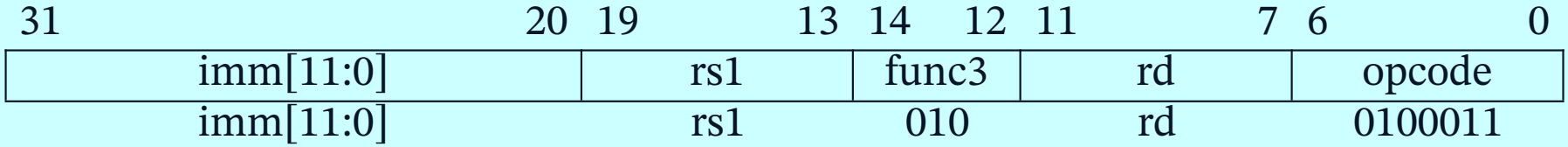
命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の和をメモリ・アドレスとし，そのアクセスしたメモリのデータをレジスタ rt に格納する。例:

```
lw   x8, 4(x6)
```

For example,

```
if    reg[6] = 12
     mem[16] = 3
then reg[8] = mem[12+4] = mem[16] = 3
```

sw (Store Word) 命令




```
sw   rs2, imm(rs1)   # memory[rs1 + $signed(imm[11:0])] <- rs2
```

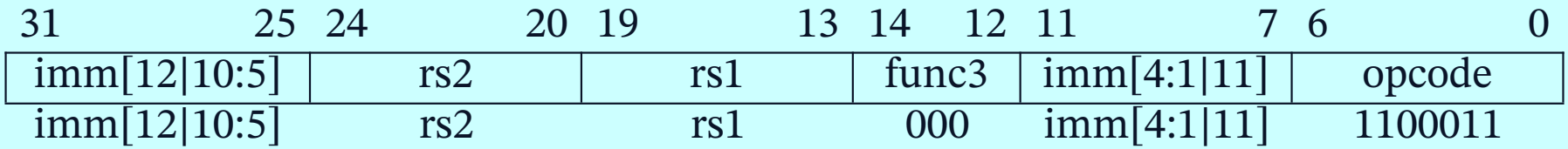
命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の和をメモリ・アドレスとし，そのアクセスしたメモリにレジスタ rs2 に格納されている値を書き込む。例:

```
sw   x8, 4(x6)
```

For example,

```
if   reg[6] = 12
     reg[8] = 3
then mem[12+4] = mem[16] = reg[8] = 3
```

beq (Branch on Equal) 命令



31 25	24 20	19 13	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011

```
beq  rs1, rs2, label # if (rs1 == rs2) pc <- pc + $signed({imm[12:1],0})
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値が等しければ，分岐する。
分岐先アドレスは pc + $signed({imm[12:1],0}) である。例:

```
beq  x8, x0, label
```

For example,

```
if   reg[8] == 0 (equal, note: reg[0] = 0)
then pc = pc + $signed({imm[12:1],0})
else pc = pc + 4
```

bne (Branch on Not Equal) 命令

31 25	24 20	19 13	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011

```
bne  rs1, rs2, label # if (rs1 != rs2) pc <- pc + $signed({imm[12:1],0})
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値が等しくなければ，分岐する。

分岐先アドレスは pc + $signed({imm[12:1],0}) である。例:

```
bne  x8, x0, label
```

For example,

```
if   reg[8] != 0 (equal, note: reg[0] = 0)
then pc = pc + $signed({imm[12:1],0})
else pc = pc + 4
```

lui (Load Upper Immediate) 命令


```
lui   rd,   imm          # rd <- {imm[31:12],000000000000}
```

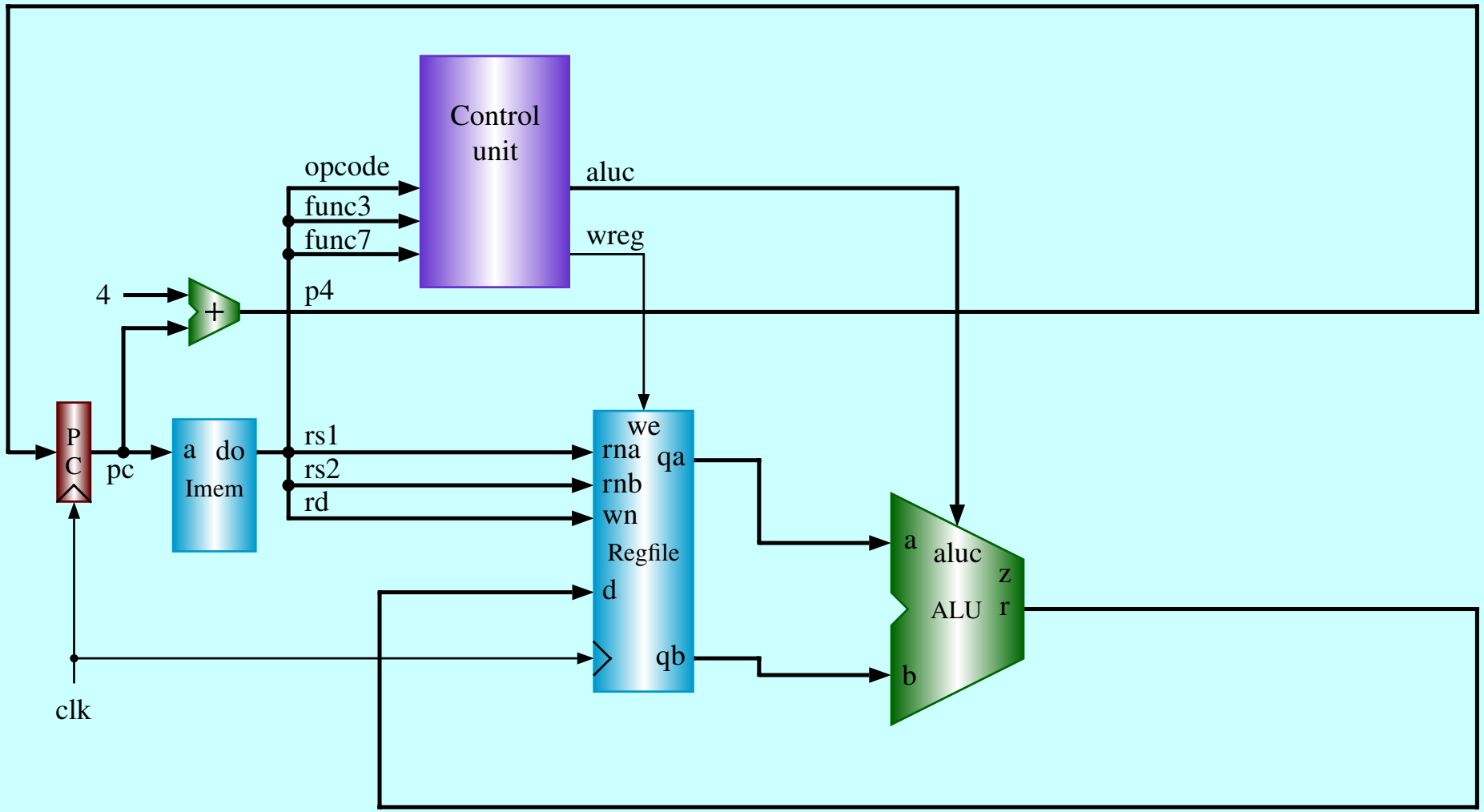
命令の意味: 即値 {imm[31:12],000000000000} をレジスタ rd に格納する。

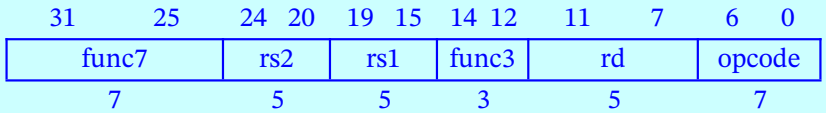
例:

```
lui   x8, 0xfffff
```

For example,
`reg[8] = 0xfffff000`

RISC-V add, sub, slt, xor, or, and の回路

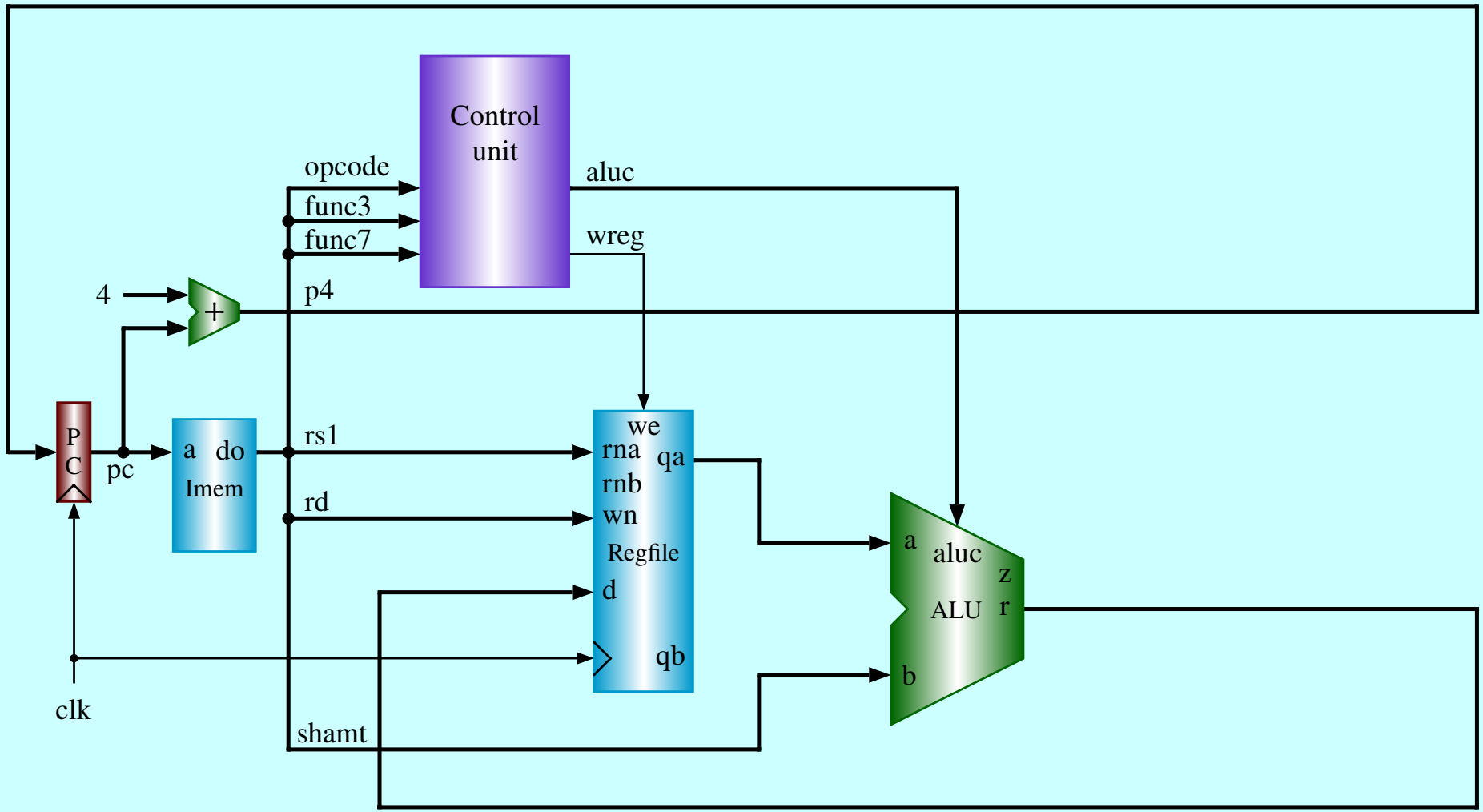




R-type

add rd, rs1, rs2 # rd ← rs1 + rs2

RISC-V slli, srli, srai の回路

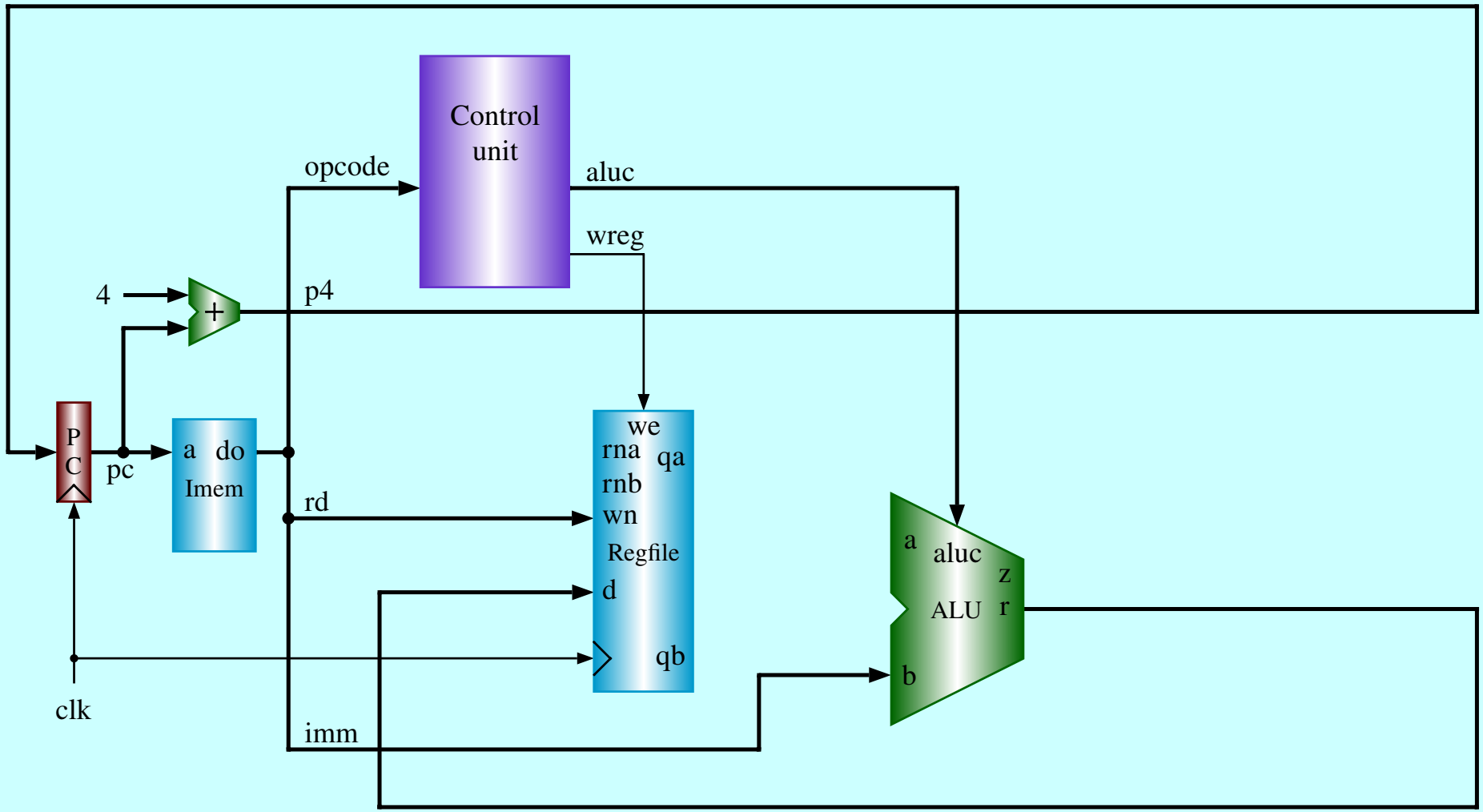


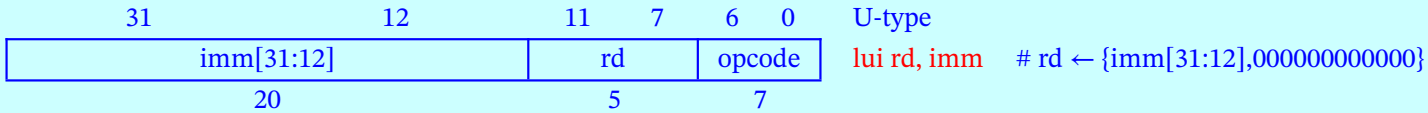
31 25	24 20	19 15	14 12	11 7	6 0
func7	shamt	rs1	func3	rd	opcode
7	5	5	3	5	7

H-type

slli rd, rs1, shamt　# rd ← rs1 << shamt

RISC-V lui の回路

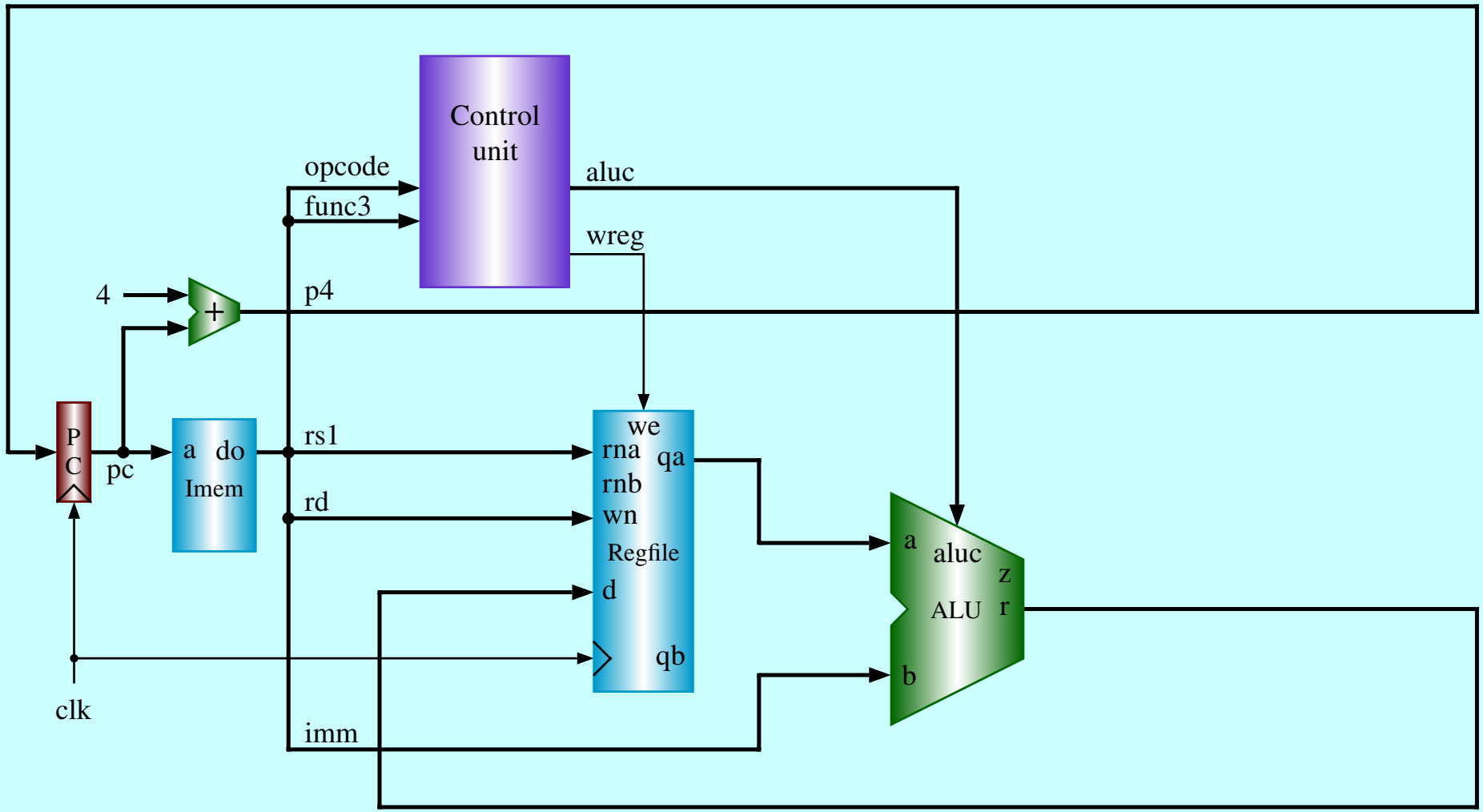




U-type

lui rd, imm　　# rd ← {imm[31:12],000000000000}

RISC-V addi, xori, ori, andi の回路

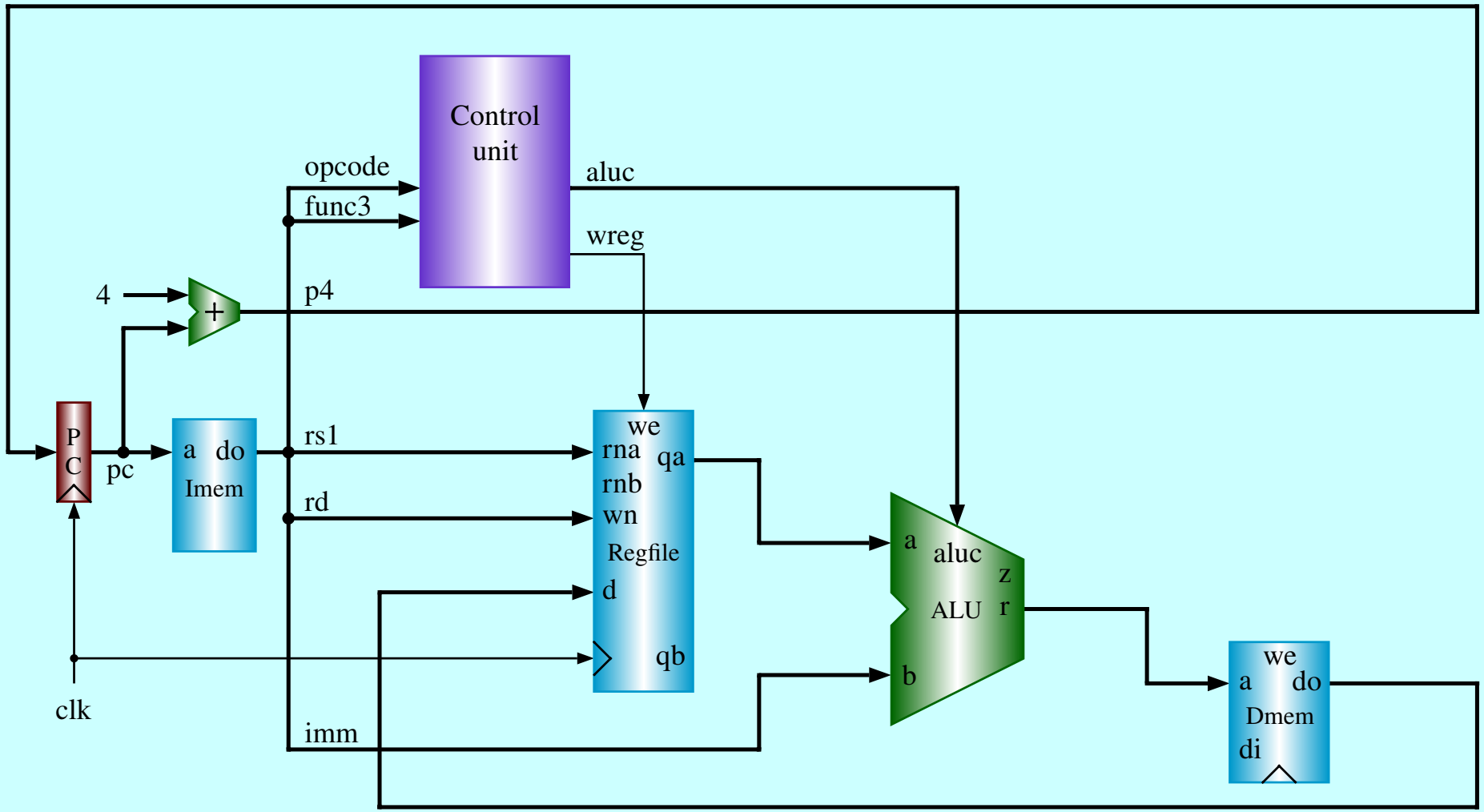


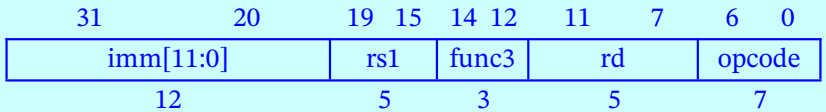
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
12	5	3	5	7

I-type

addi rd, rs1, imm　　# rd ← rs1 + $signed(imm[11:0])

RISC-V lw の回路

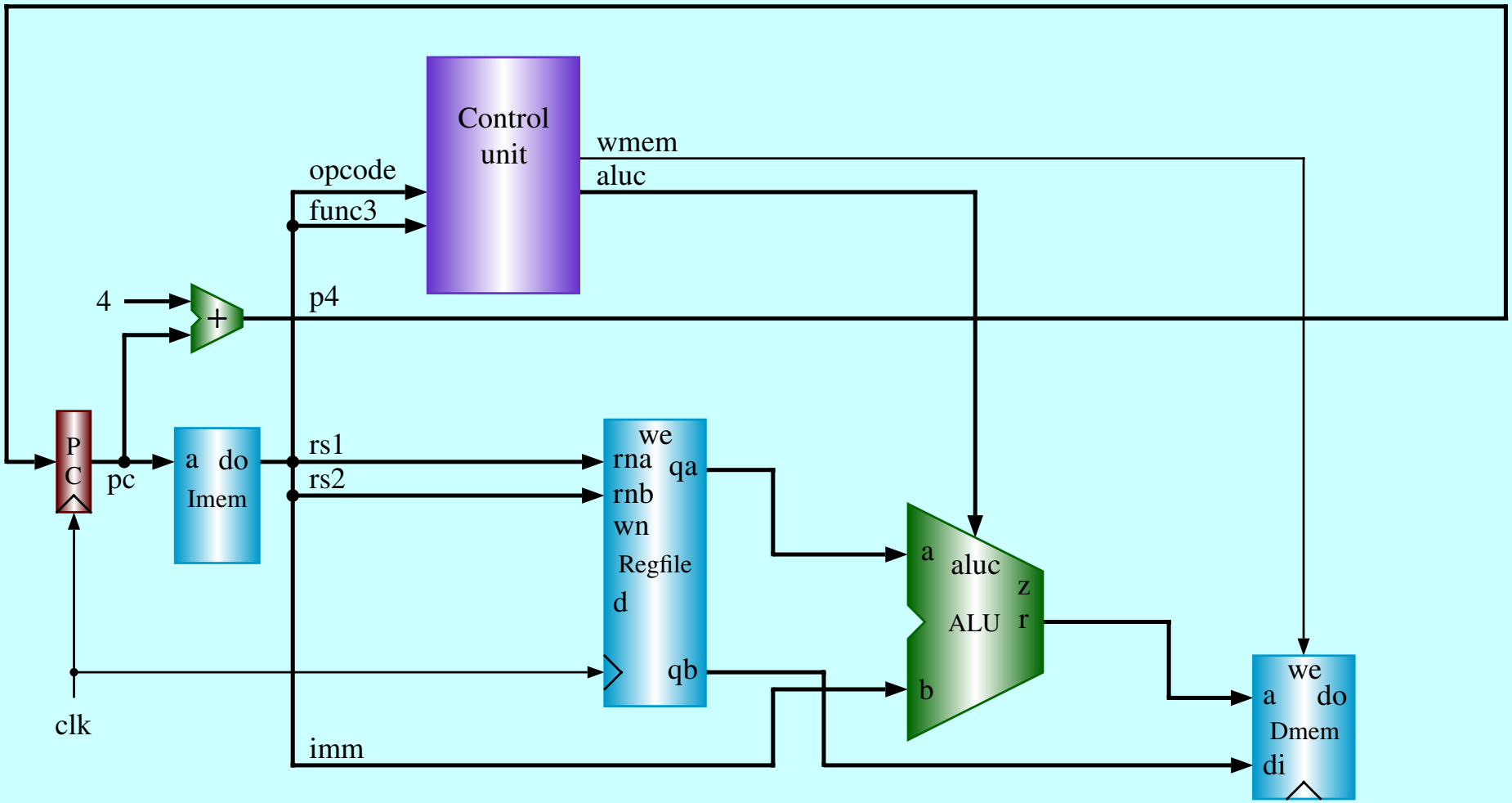




I-type

lw rd, imm(rs1) # rd ← memory[rs1+$signed(imm[11:0])]

RISC-V sw の回路

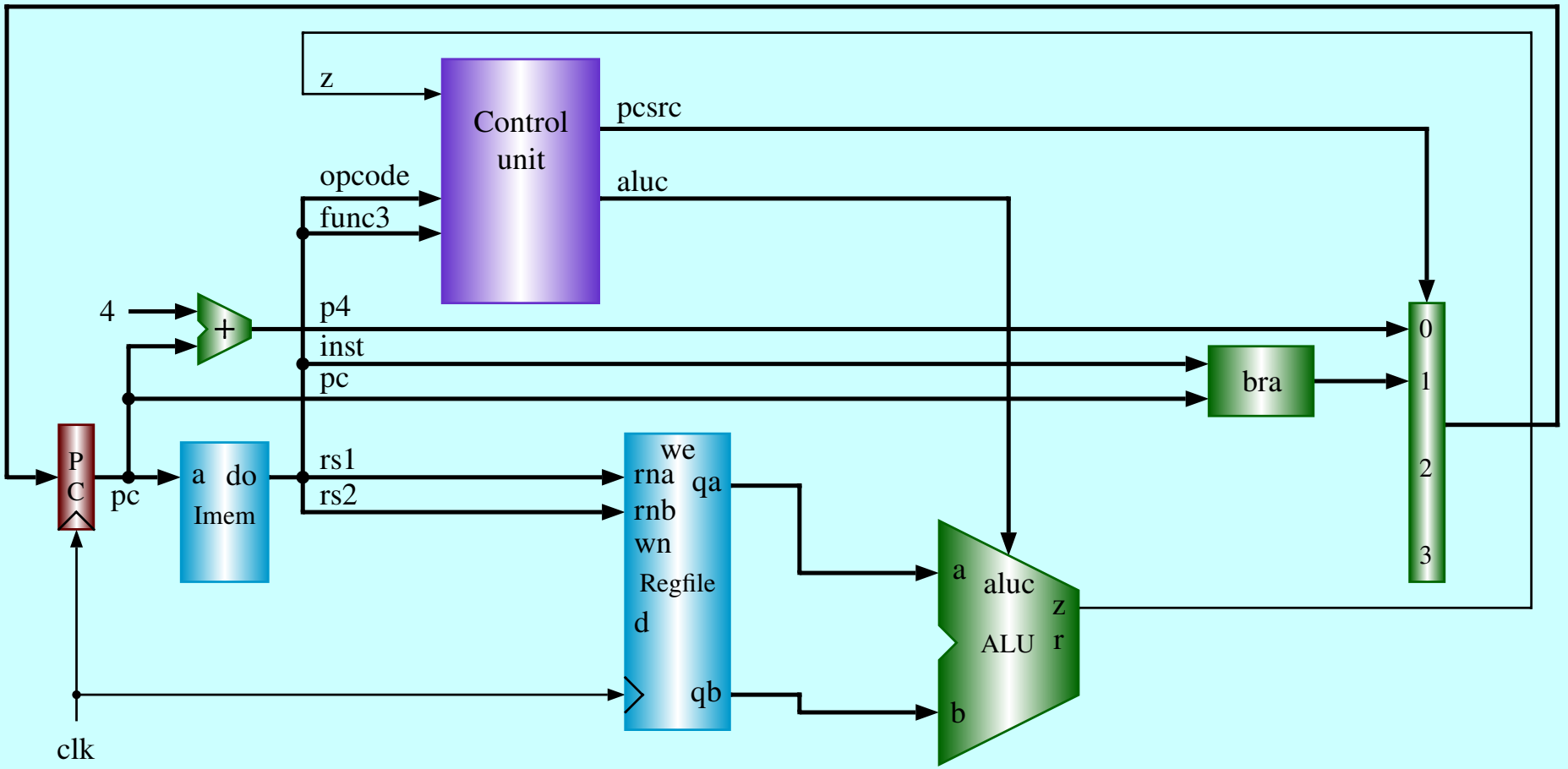


31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
7	5	5	3	5	7

S-type

sw rs2, imm(rs1) # memory[rs1+$signed(imm[11:0])] ← rs2

RISC-V beq と bne の回路



31 25	24 20	19 15	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
7	5	5	3	5	7

B-type

beq rs1, rs2, label　# if (rs1 = rs2) pc ← pc + $signed({imm[12:1],0})

20 RISC-V 命令の ALU 操作

```
 1. add  rd, rs1, rs2    # ADD ..................................... (1)
 2. sub  rd, rs1, rs2    # SUB ..................................... (2)
 3. slt  rd, rs1, rs2    # SLT ..................................... (3)
 4. xor  rd, rs1, rs2    # XOR ..................................... (4)
 5. or   rd, rs1, rs2    # OR ...................................... (5)
 6. and  rd, rs1, rs2    # AND ..................................... (6)
 7. slli rd, rs1, shamt  # SLL ..................................... (7)
 8. srli rd, rs1, shamt  # SRL ..................................... (8)
 9. srai rd, rs1, shamt  # SRA ..................................... (9)
10. jalr rd, rs1, imm    # no operation
11. addi rd, rs1, imm    # same as add
12. xori rd, rs1, imm    # same as xor
13. ori  rd, rs1, imm    # same as or
14. andi rd, rs1, imm    # same as and
15. lw   rd,  imm(rs1)   # same as add
16. sw   rs2, imm(rs1)   # same as add
17. beq  rs1, rs2, label # same as sub
18. bne  rs1, rs2, label # same as sub
19. jal  rd,  label      # no operation
20. lui  rd,  imm        # LUI ..................................(10)
```

ALU操作

	op	instructions
(1)	ADD	add, addi, lw, sw
(2)	SUB	sub, beq, bne
(3)	SLT	slt
(4)	XOR	xor, xori
(5)	OR	or, ori
(6)	AND	and, andi
(7)	SLL	slli
(8)	SRL	srli
(9)	SRA	srai
(10)	LUI	lui

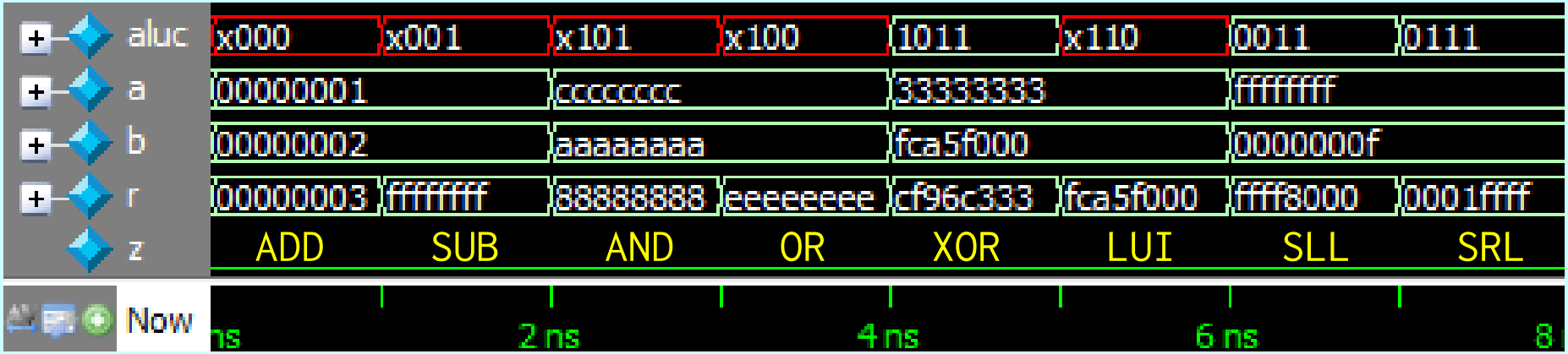
ALU操作

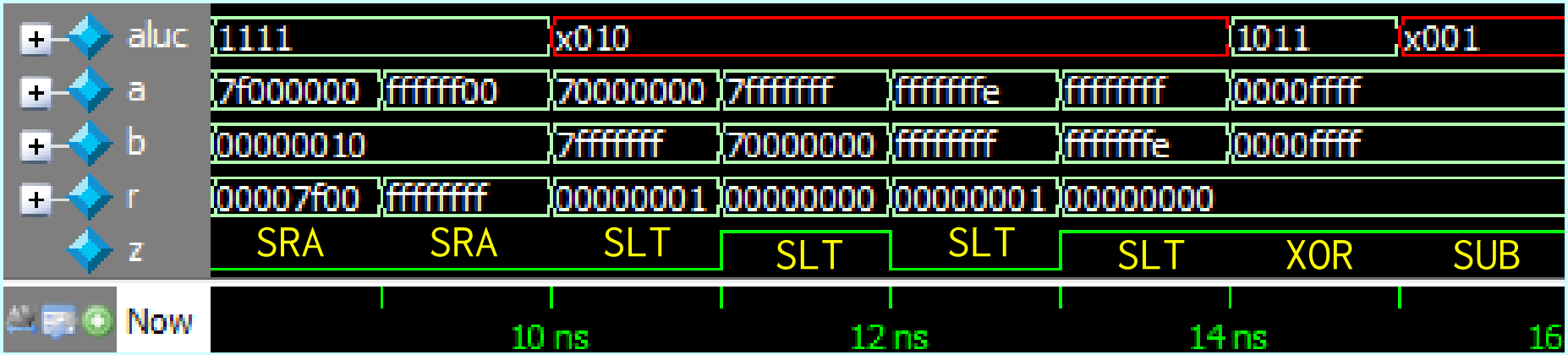
	op	aluc[3:0]	instructions
(1)	ADD	x 0 0 0	add, addi, lw, sw
(2)	SUB	x 0 0 1	sub, beq, bne
(3)	SLT	x 0 1 0	slt
(4)	XOR	1 0 1 1	xor, xori
(5)	OR	x 1 0 0	or, ori
(6)	AND	x 1 0 1	and, andi
(7)	SLL	0 0 1 1	slli
(8)	SRL	0 1 1 1	srli
(9)	SRA	1 1 1 1	srai
(10)	LUI	x 1 1 0	lui

ALUの設計

```
module alu (a,b,aluc,r,z);                                  // 32-bit alu with a zero flag
    input  [31:0] a, b;                                     // inputs: a, b
    input   [3:0] aluc;                                     // input:  alu control
    output [31:0] r;                                        // output: alu result
    output        z;                                        // output: zero flag
    assign r = calc(a,b,aluc);                              // call function
    assign z = ~|r;                                         // z = (r == 0)
    function  [31:0] calc;                                  // function
        input [31:0] a,b;                                   // input
        input  [3:0] aluc;                                  // input
        reg   [31:0] sub;                                   // local variable
        begin
            sub = a - b;                                    // for SUB and SLT
            casex(aluc)                                     // allowing don't care
                4'bx000: calc = a + b;                      // ADD, aluc: 0, 8
                4'bx001: calc = sub;                        // SUB, aluc: 1, 9
                4'bx010: calc = {31'b0,sub[31]};            // SLT, aluc: 2, a
                4'b1011: calc = a ^ b;                      // XOR, aluc: b,
                4'bx100: calc = a | b;                      // OR,  aluc: 4, c
                4'bx101: calc = a & b;                      // AND, aluc: 5, d
                4'b0011: calc = a << b[4:0];                // SLL, aluc: 3,
                4'b0111: calc = a >> b[4:0];                // SRL, aluc: 7,
                4'b1111: calc = $signed(a) >>> b[4:0];      // SRA, aluc: f,
                4'bx110: calc = b;                          // LUI, aluc: 6, e
            endcase
        end
    endfunction
endmodule
```

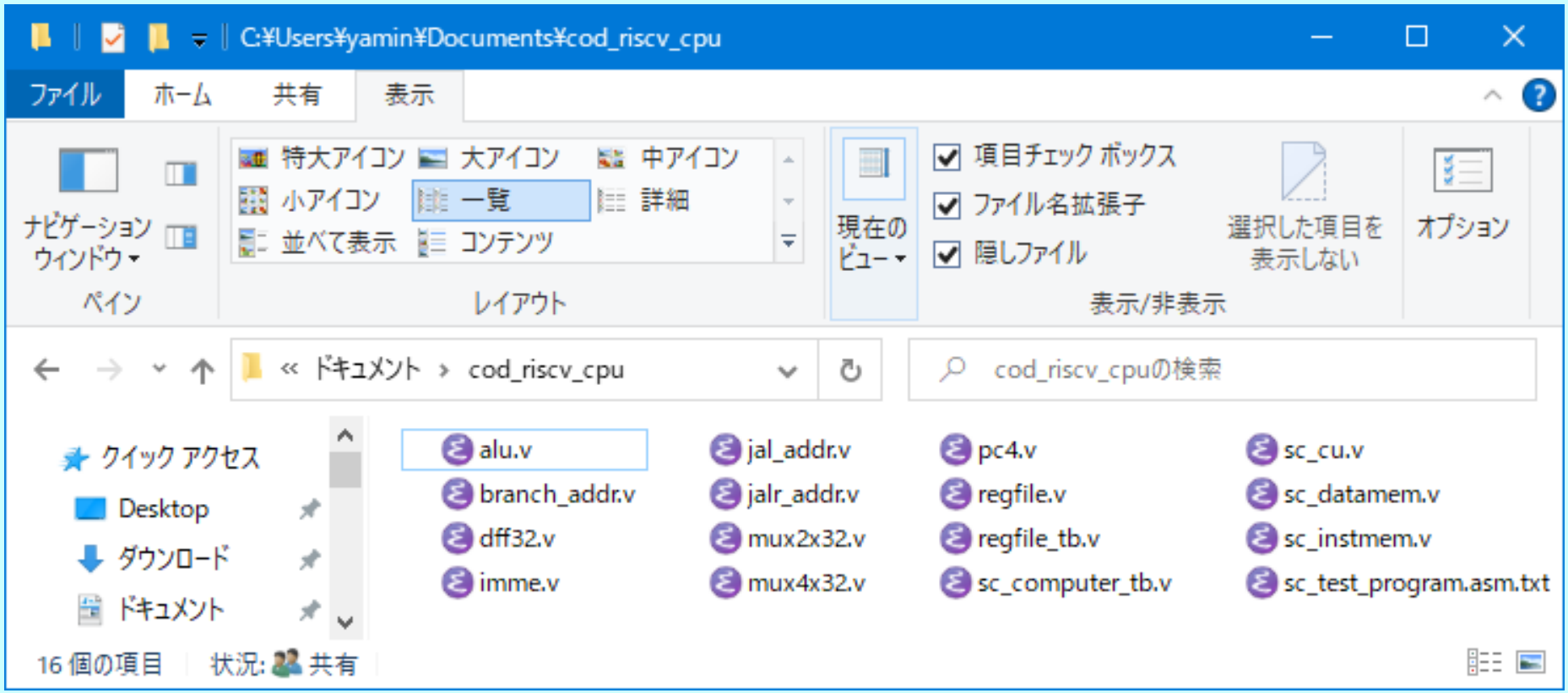
ALUのシミュレーション波形





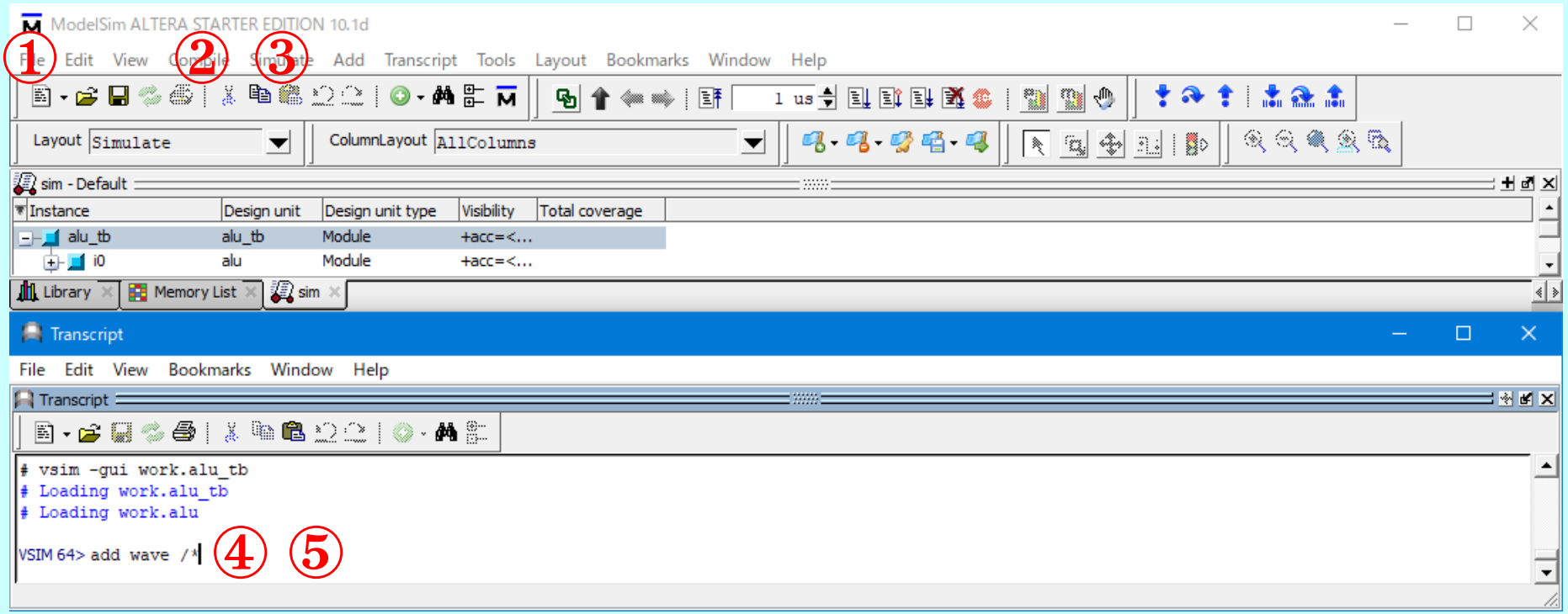
ModelSim の使い方

作業フォルダを準備する



1. 作業フォルダ cod_riscv_cpu を準備する
2. cod_riscv_cpu.zip 中のファイルをフォルダにコピーする

ModelSim の使い方



① Start ModelSim 10.1d. File ▶ Change Directory... ▶ cod_riscv_cpu

② Compile ▶ Compile... ▶ alu.v, alu_tb.v. Click Yes first time.

③ Simulate ▶ Start Simulation... ▶ work ▶ alu_tb

④ In Transcript window, type add wave /*

⑤ In Transcript window, type run 22ns

⑥ In Wave window, adjust waves and then File ▶ Export

課題 III (100 点 + 100 点)

1. ALU のシミュレーション波形を参照し、テストベンチ alu_tb.v を書き、ModelSim で alu.v をシミュレーションしなさい。また、出力波形の正しさ（計算結果）について説明しなさい。

2. オプション (+100 点): ModelSim で pc4.v、branch_addr.v、jal_addr.v、jalr_addr.v をそれぞれシミュレーションしなさい。