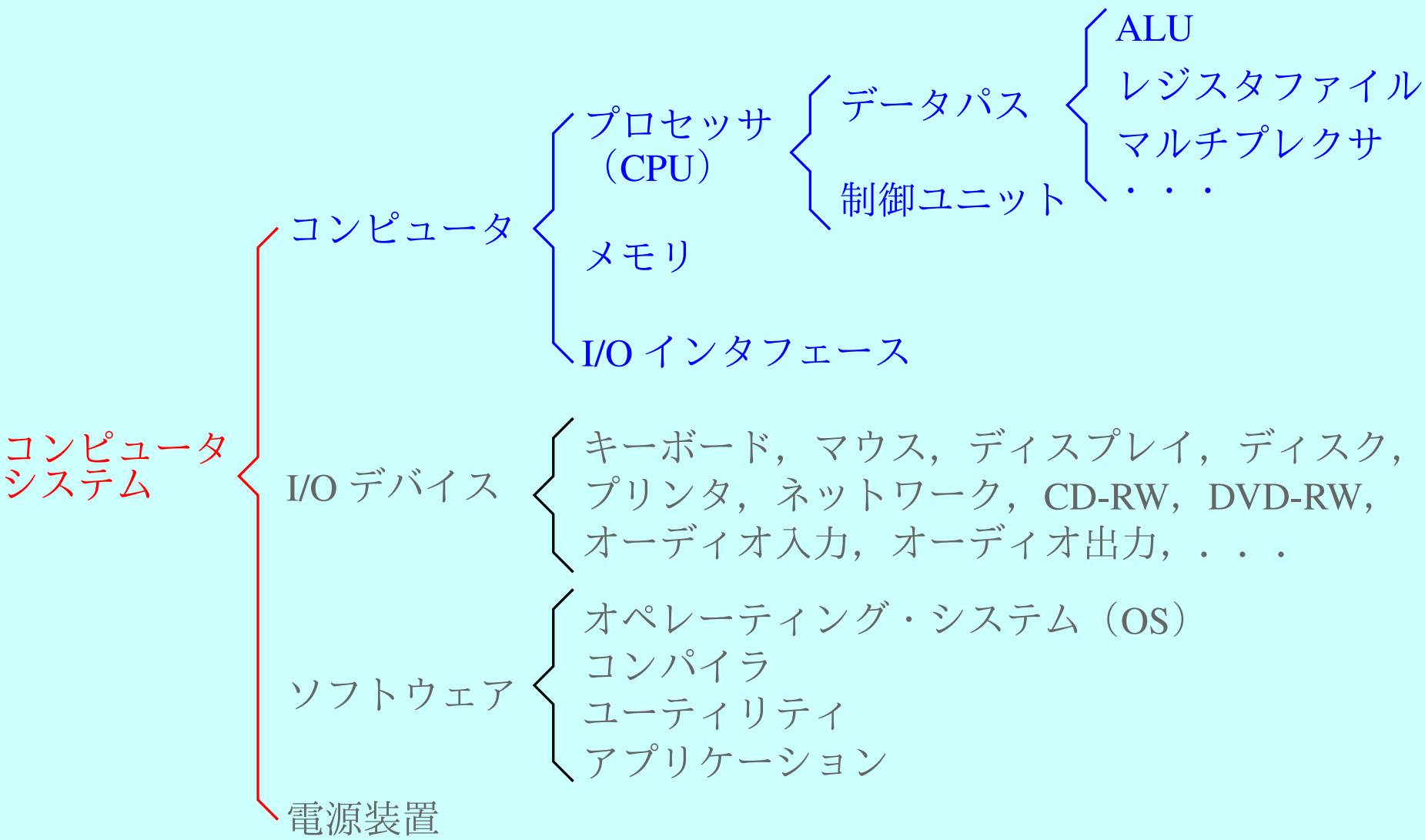
コンピュータ構成と設計（４）

CPU 構成要素の設計

李　亜民

2022 年 10 月 17 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
```

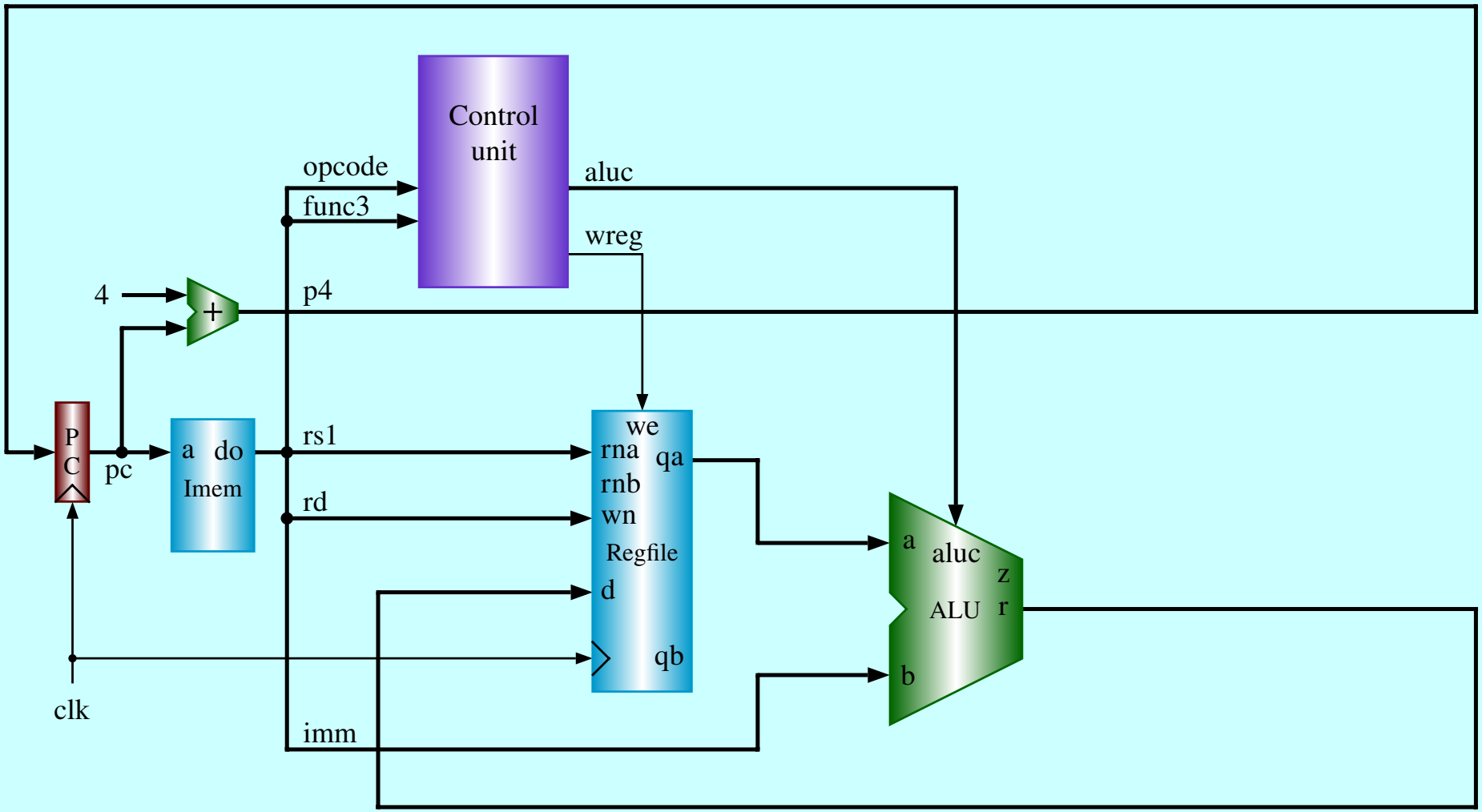
RV32I Base Instruction Set Encoding

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

即値 imme 回路 (Immediate) の設計

RISC-V addi, xori, ori, andi

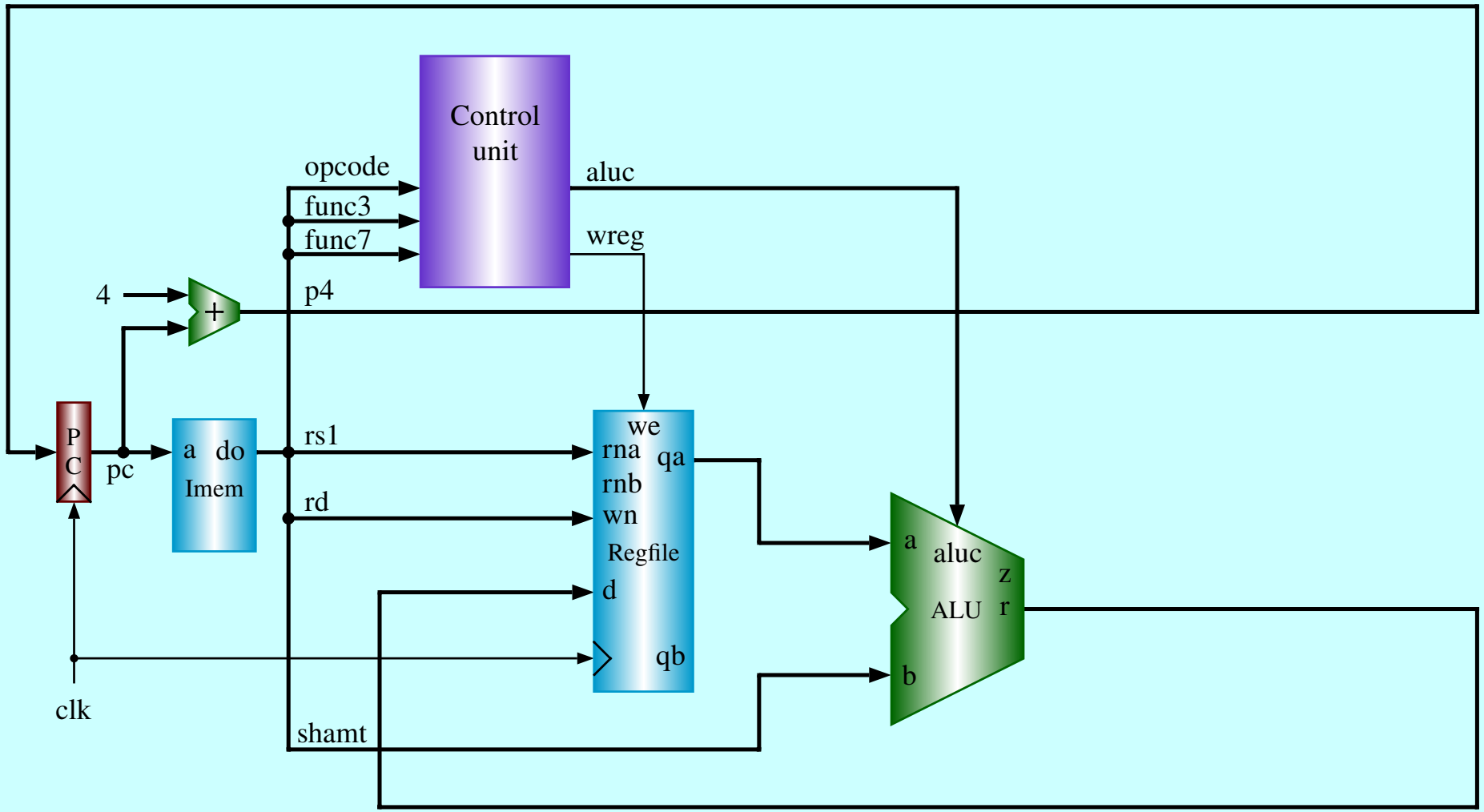


31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
12	5	3	5	7

I-type

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

RISC-V slli, srli, srai

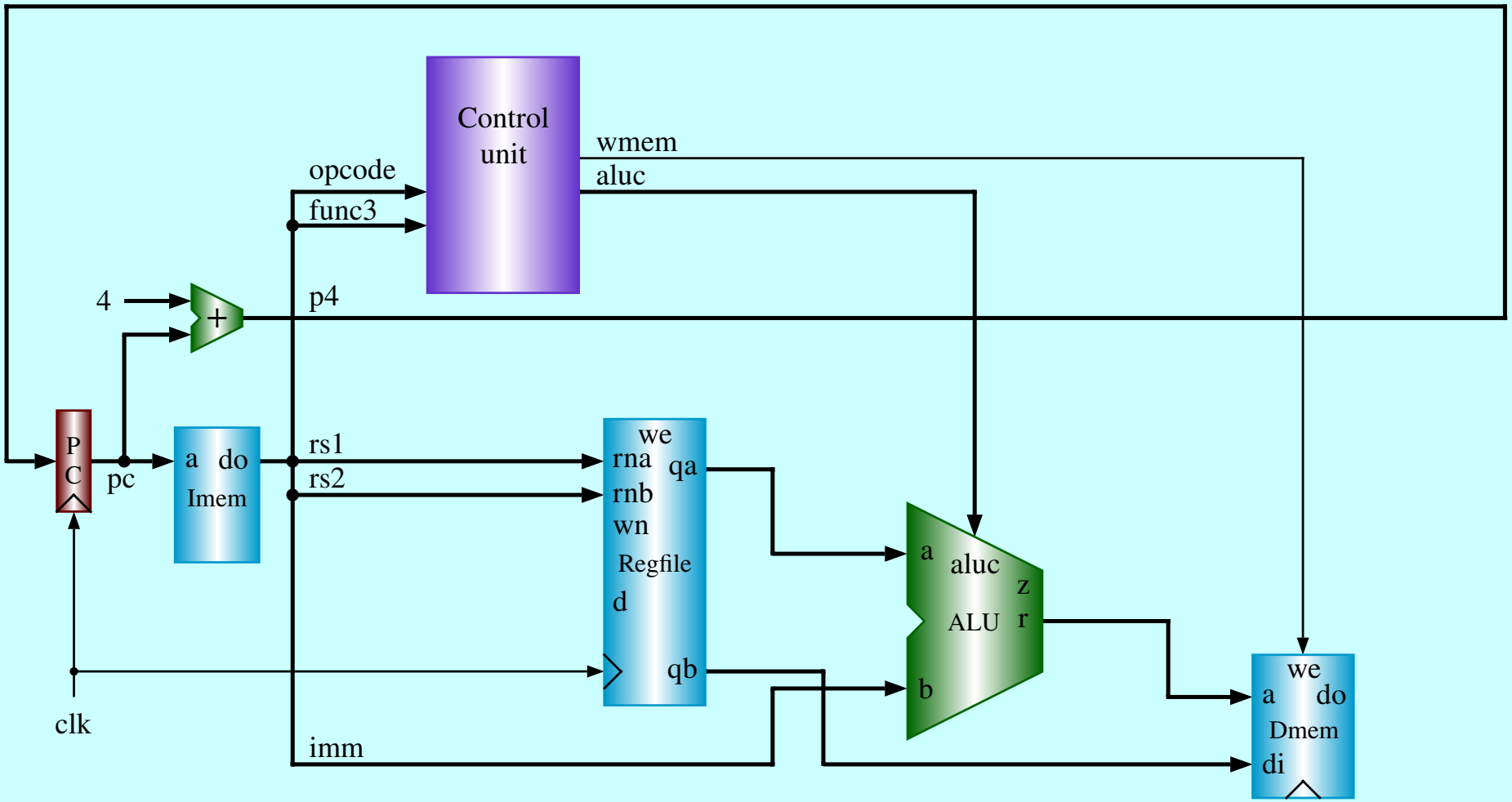


31 25	24 20	19 15	14 12	11 7	6 0
func7	shamt	rs1	func3	rd	opcode
7	5	5	3	5	7

H-type

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

RISC-V sw

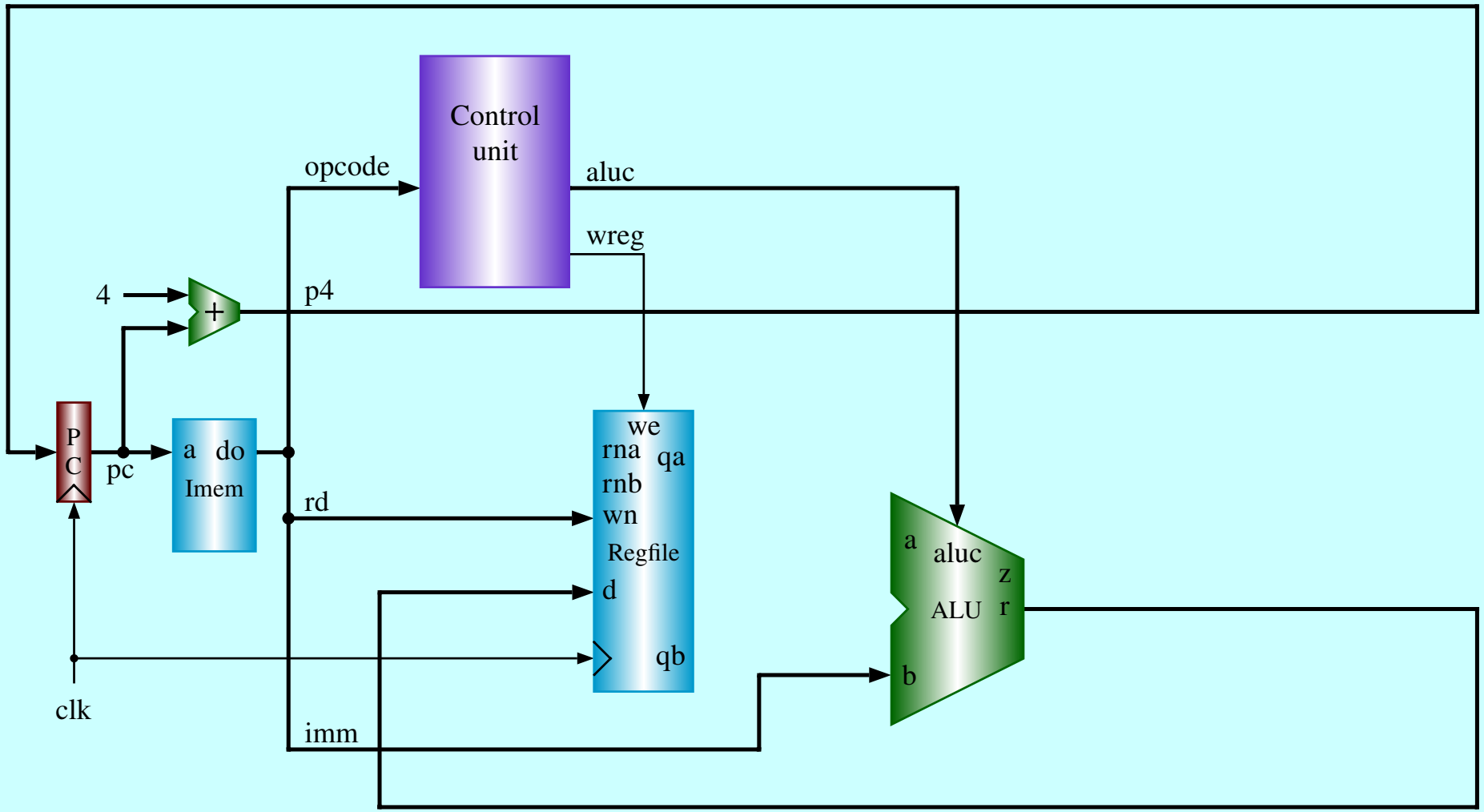


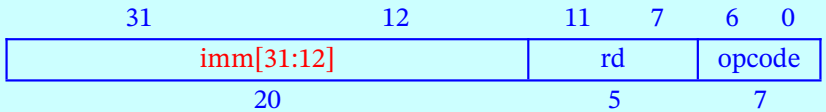
31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
7	5	5	3	5	7

S-type

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

RISC-V lui

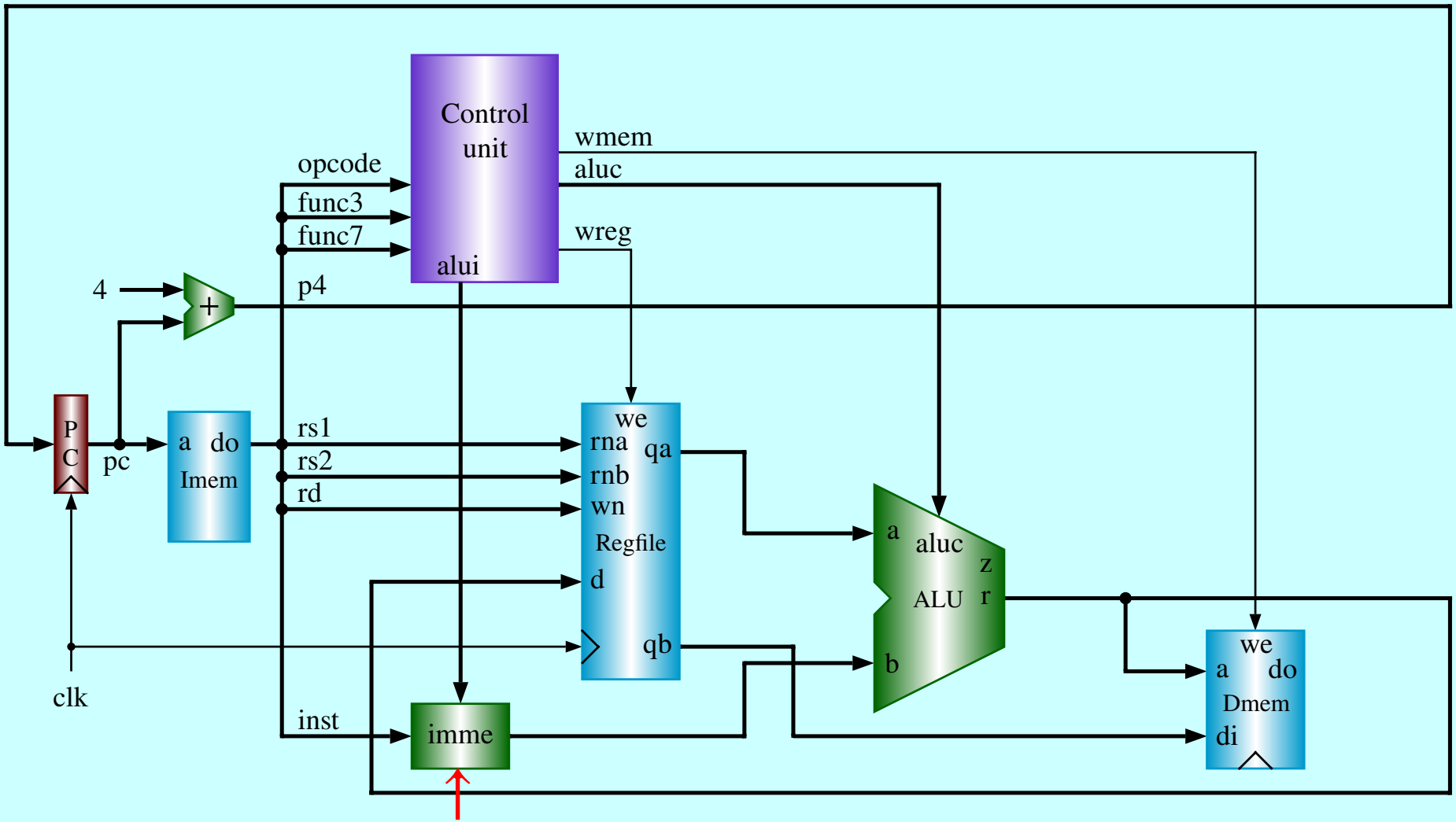




U-type

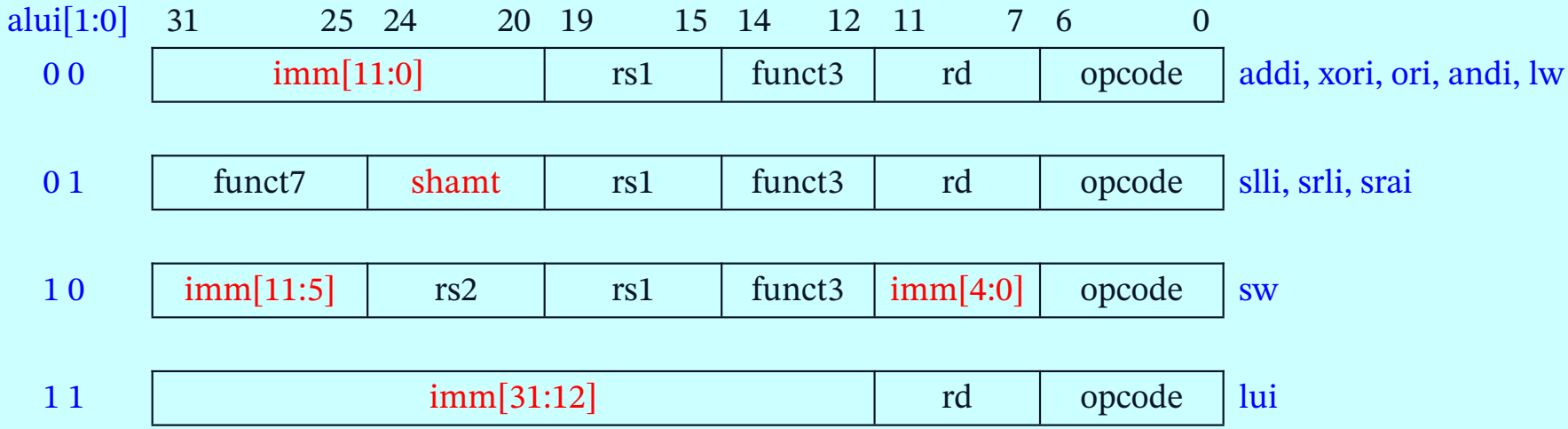
lui rd, imm # rd ← {imm[31:12],000000000000}

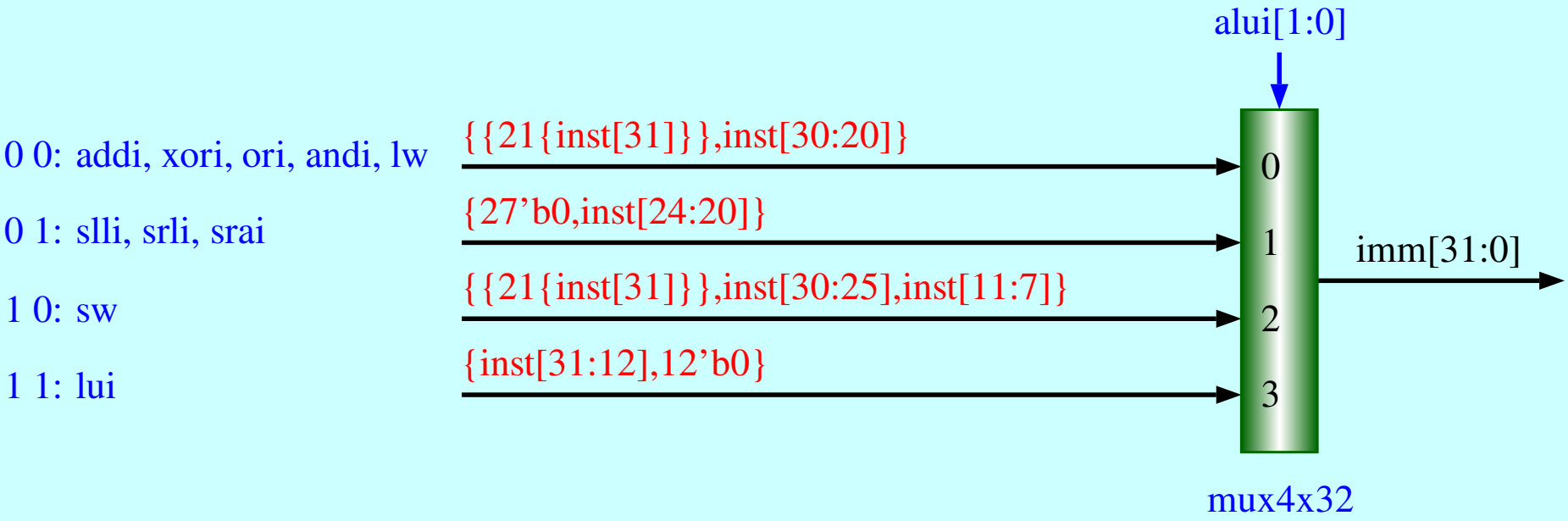
RISC-V 即値 imme



imme for addi, xori, ori, andi, slli, srli, srai, sw, lui

RISC-V 即値 imme の回路





RISC-V 即値 imme の回路

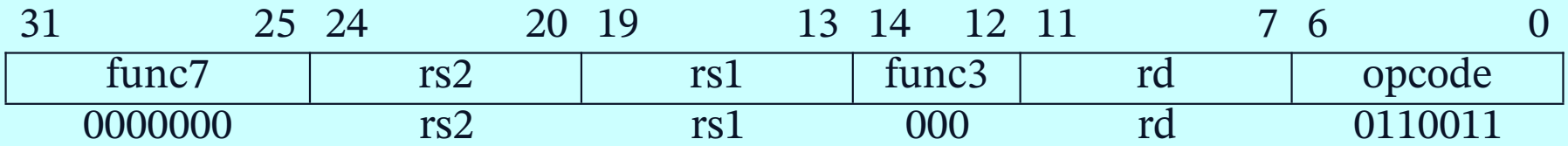
```
module imme (inst,alui,imm); // 32-bit immediate
    input   [1:0] alui;
    input  [31:0] inst;
    output [31:0] imm;

    assign imm = get_imm(inst,alui);

    function [31:0] get_imm;
        input [31:0] inst;
        input  [1:0] alui;
        case(alui)
            2'b00: get_imm = {{21{inst[31]}},inst[30:20]}; // addi,xori,ori,andi,lw
            2'b01: get_imm = {27'b0,         inst[24:20]}; // slli,srli,srai
            2'b10: get_imm = {{21{inst[31]}},inst[30:25],inst[11:7]}; // sw
            2'b11: get_imm = {inst[31:12],12'b0};                     // lui
        endcase
    endfunction
endmodule
```

レジスタファイル (Register File) の設計

add (Add) 命令




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

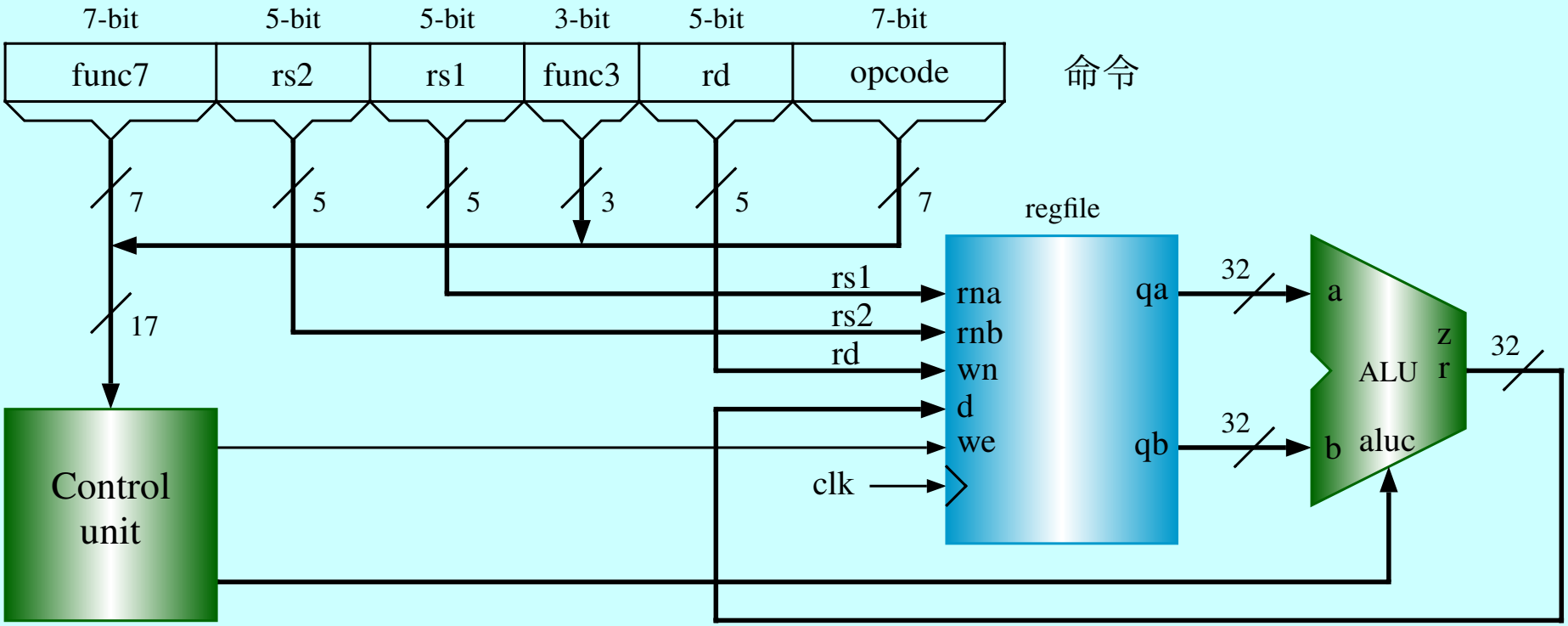
命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値を加算して，その結果をレジスタ rd に格納する。

例:

```
add  x8, x6, x7 # x8 <- x6 + x7
```

レジスタファイルが必要

レジスタファイルが必要

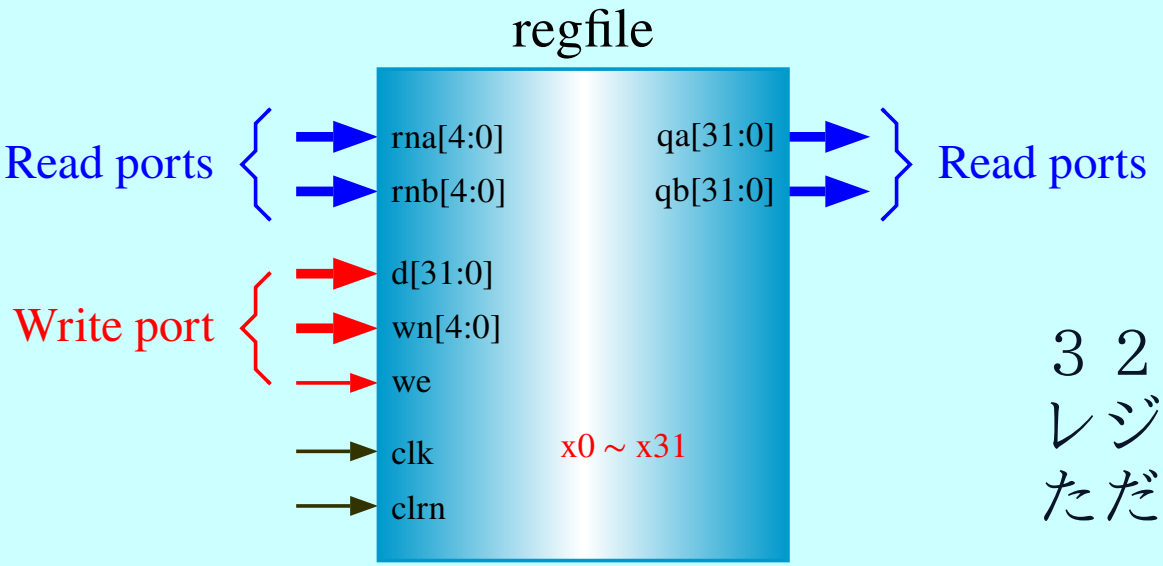


命令の例

```
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。

レジスタファイル

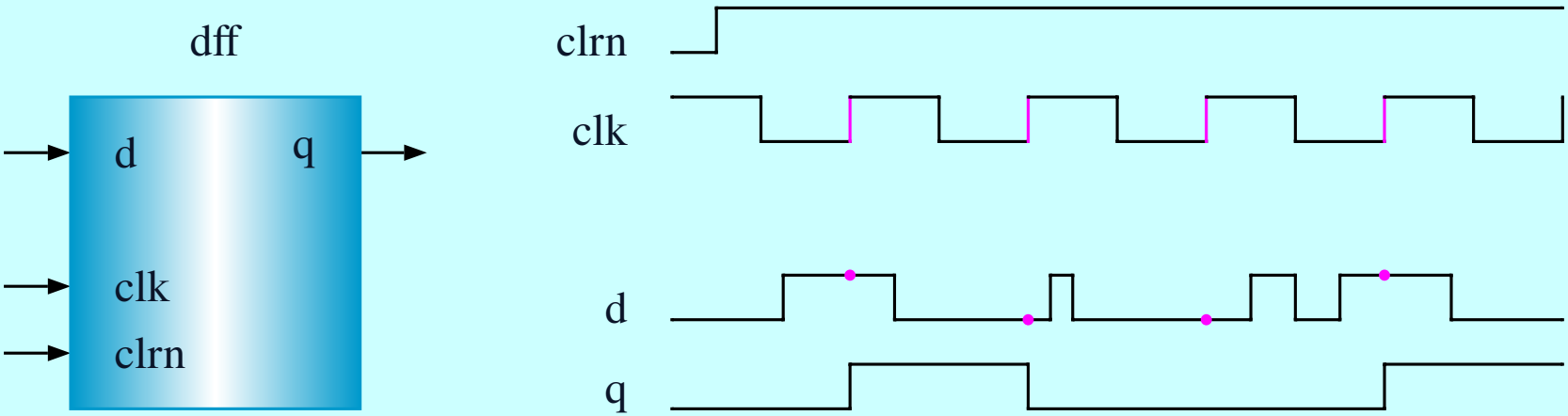


３２ビットの
レジスタ３２本(x0 ~ x31)
ただし、常に x0 = 0

rna[4:0]	—	register number of read port A
rnb[4:0]	—	register number of read port B
qa[31:0]	—	data output of read port A
qb[31:0]	—	data output of read port B
wn[4:0]	—	register number of write port
d[31:0]	—	data input of write port
we	—	write enable

dff の回路

- D フリップフロップ (D Flip Flop — DFF)
  - クロック信号の立ち上がり (0 から 1 への遷移) の直前のデータ入力 (d) の値が記憶され、出力 (q) に出力される。
- シンボルと波形

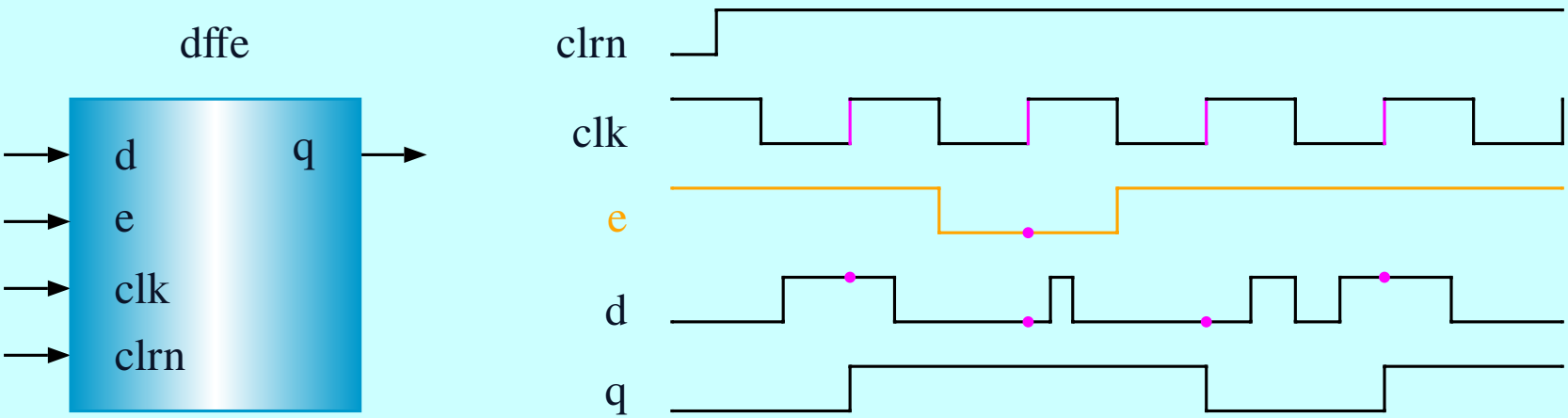


clrn: 0 のとき dff をクリアします (出力 q = 0)

dffe の回路

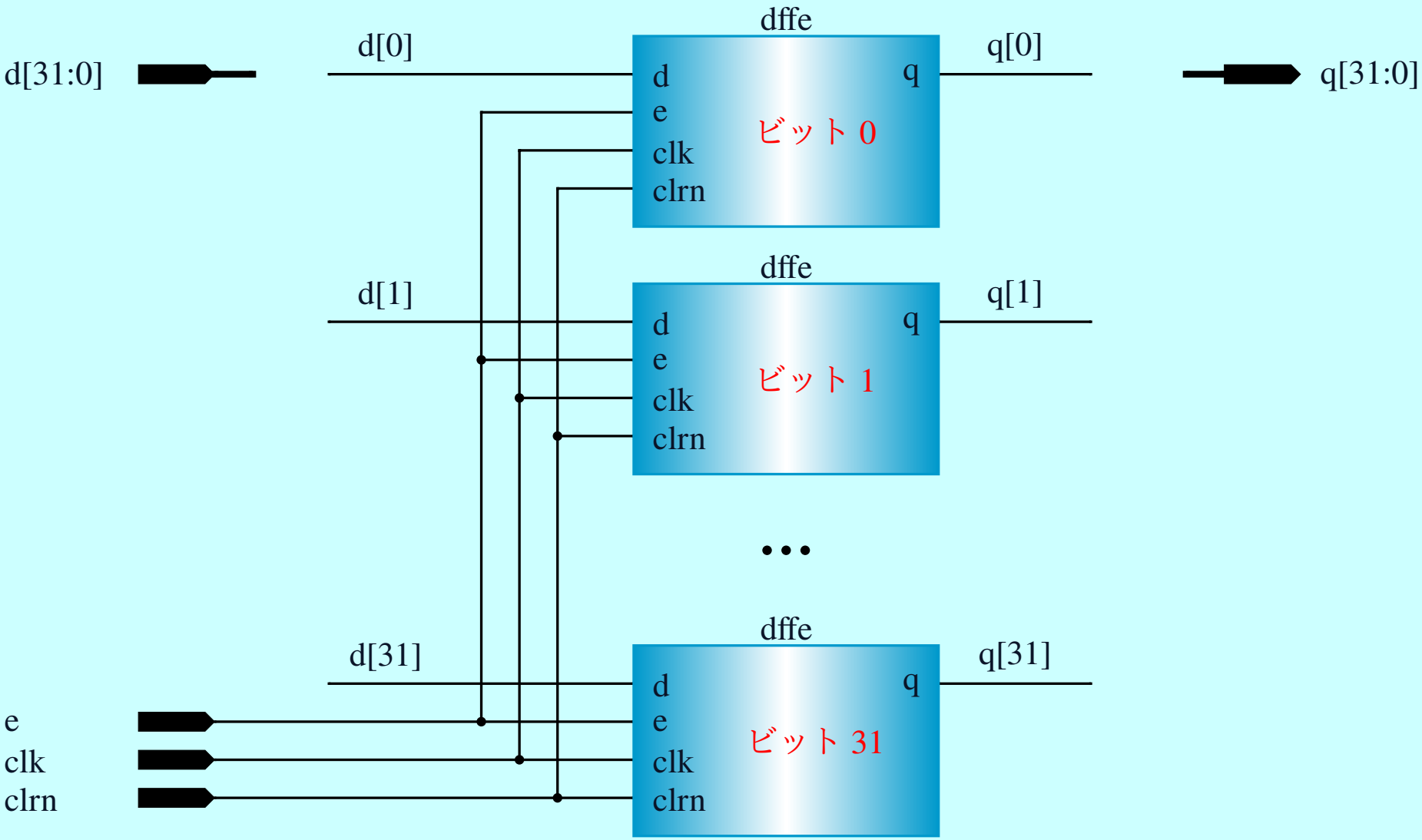
- DFFE — イネーブル (E) 付き DFF
  - E = 1: DFFE = DFF
  - E = 0: 保存禁止

- シンボルと波形

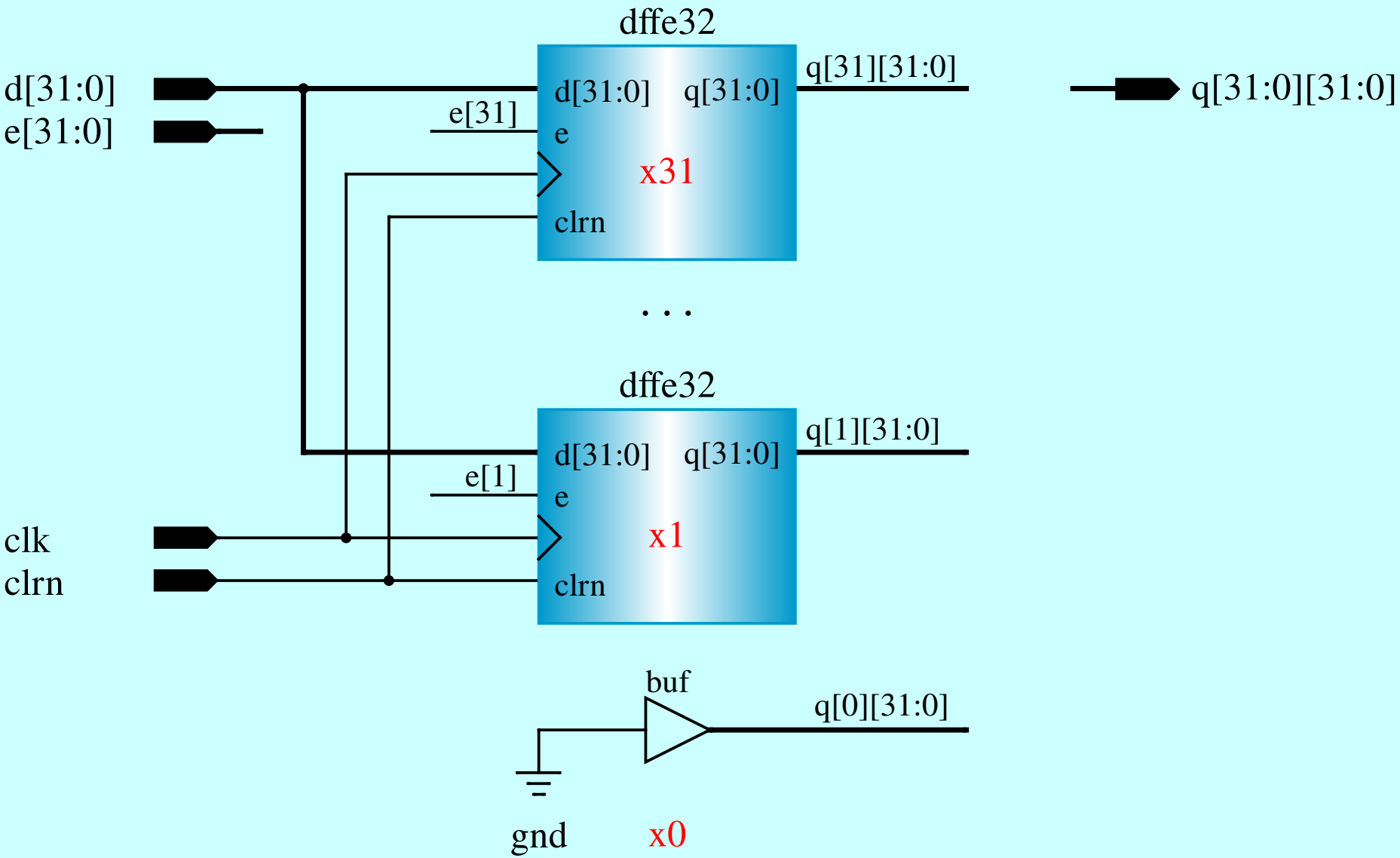


clrn: 0 のとき dffe をクリアします (出力 q = 0)

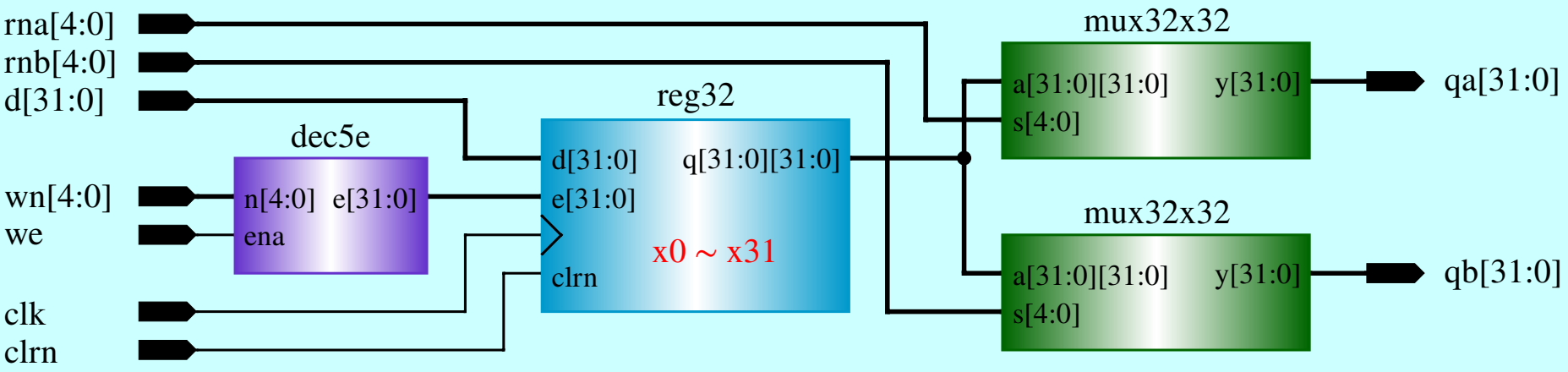
dffe32 の回路


d[31:0]
d[0]
dffe
d
e
clk
clrn
q
q[0]
ビット 0
q[31:0]
d[1]
dffe
d
e
clk
clrn
q
q[1]
ビット 1
d[31]
dffe
d
e
clk
clrn
q
q[31]
ビット 31
e
clk
clrn

reg32 の回路


dffe32
d[31:0]
e[31:0]
d[31:0]
q[31:0]
e[31]
e
x31
clrn
q[31][31:0]
q[31:0][31:0]
. . .
dffe32
d[31:0]
q[31:0]
e[1]
e
x1
clrn
q[1][31:0]
clk
clrn
buf
q[0][31:0]
gnd
x0

レジスタファイルの回路



rna[4:0]	—	register number of read port A
rnb[4:0]	—	register number of read port B
qa[31:0]	—	data output of read port A
qb[31:0]	—	data output of read port B
wn[4:0]	—	register number of write port
d[31:0]	—	data input of write port
we	—	write enable

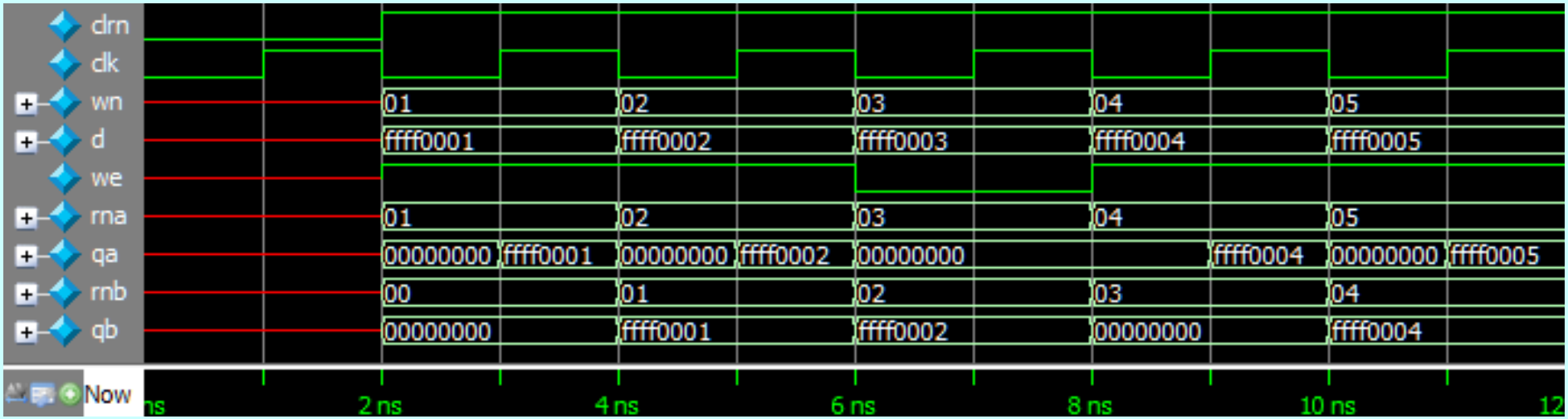
レジスタファイルの回路

```
module regfile (rna,rnb,d,wn,we,clk,clrn,qa,qb); // 32x32 regfile
    input  [31:0] d;                             // data of write port
    input   [4:0] rna;                           // reg # of read port A
    input   [4:0] rnb;                           // reg # of read port B
    input   [4:0] wn;                            // reg # of write port
    input         we;                            // write enable
    input         clk, clrn;                     // clock and reset
    output [31:0] qa, qb;                        // read ports A and B
    reg    [31:0] register [1:31];               // 31 32-bit registers
    assign qa = (rna == 0)? 0 : register[rna];   // read port A
    assign qb = (rnb == 0)? 0 : register[rnb];   // read port B
    always @(posedge clk or negedge clrn)        // write port
        if (!clrn)
            register[01]  <= 0;                  // reset
            ... ... // clear registers 02 - 30
            register[31]  <= 0;                  // reset
        else
            if ((wn != 0) && we)                 // not reg[0] & enabled
                register[wn] <= d;               // write d to reg[wn]
endmodule
```

レジスタファイル回路のテストベンチ

```
‘timescale 1ns/1ns
module regfile_tb;
    reg   [4:0] rna,rnb,wn;
    reg  [31:0] d;
    reg         we,clk,clrn;
    wire [31:0] qa,qb;
    regfile rf (rna,rnb,d,wn,we,clk,clrn,qa,qb);
    initial begin
           clk  = 0; clrn = 0;
        #2 clrn = 1; we = 1; d = 32’hffff0000; wn = 0; rna = 0; rnb = 5’d31;
        #4 we   = 0;
        #2 we   = 1;
       #58 $stop;
    end
    always #1 clk = !clk;
    always #2 d   = d + 1;
    always #2 wn  = wn + 1;
    always #2 rna = rna + 1;
    always #2 rnb = rnb + 1;
endmodule
```

レジスタファイル回路の波形




```
initial begin
        clk = 0;   clrn = 0;
    #2 we  = 1;   d    = 32'hffff0000;
        wn  = 0;   rna  = 0;   rnb = 5'd31;
    #4 we  = 0;
    #2 we  = 1;
   #58 $stop;
end
always #1 clk = !clk;
always #2 d   = d + 1;
always #2 wn  = wn + 1;
always #2 rna = rna + 1;
always #2 rnb = rnb + 1;
```

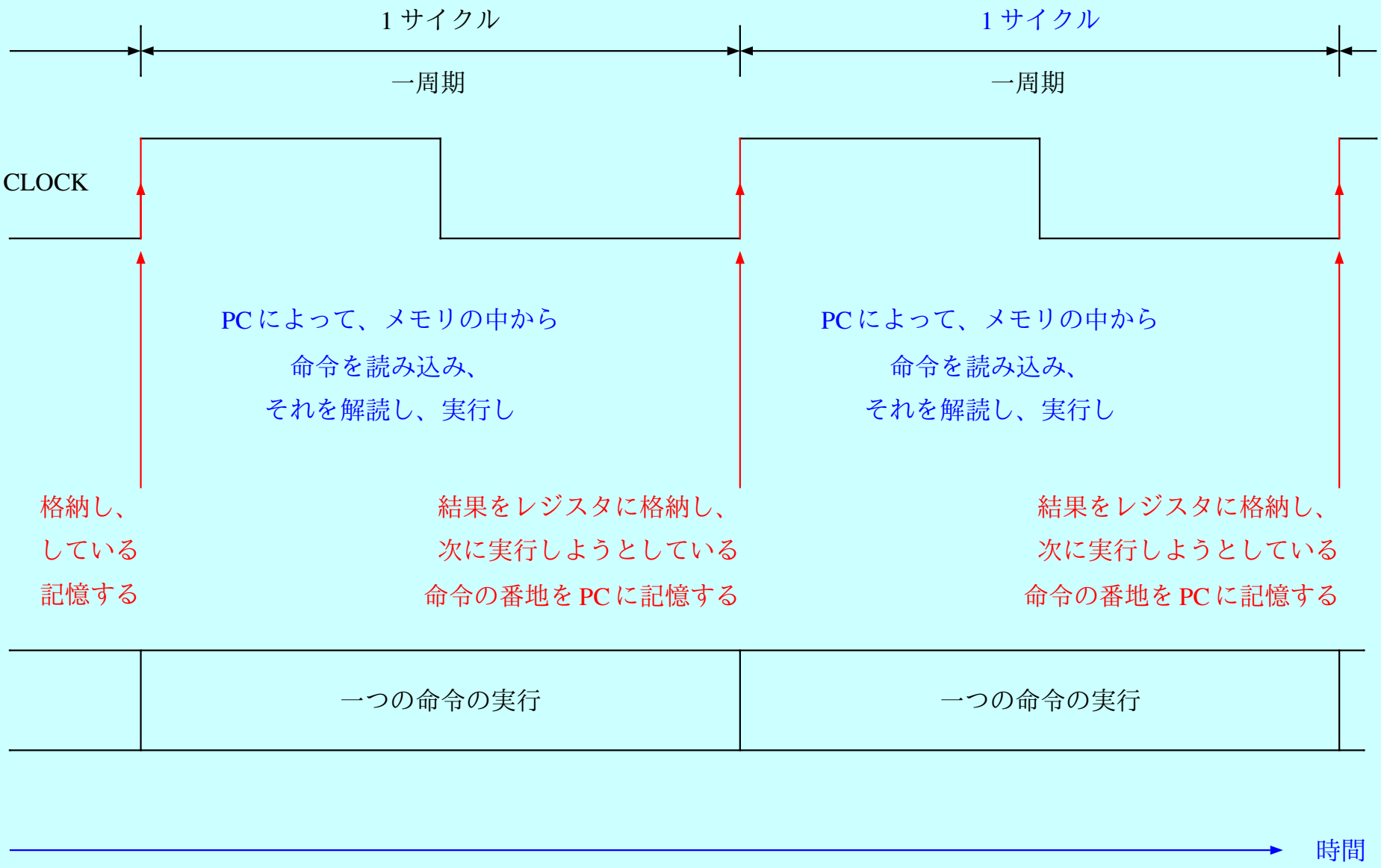
20 RISC-V 命令のまとめ

```
 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
```

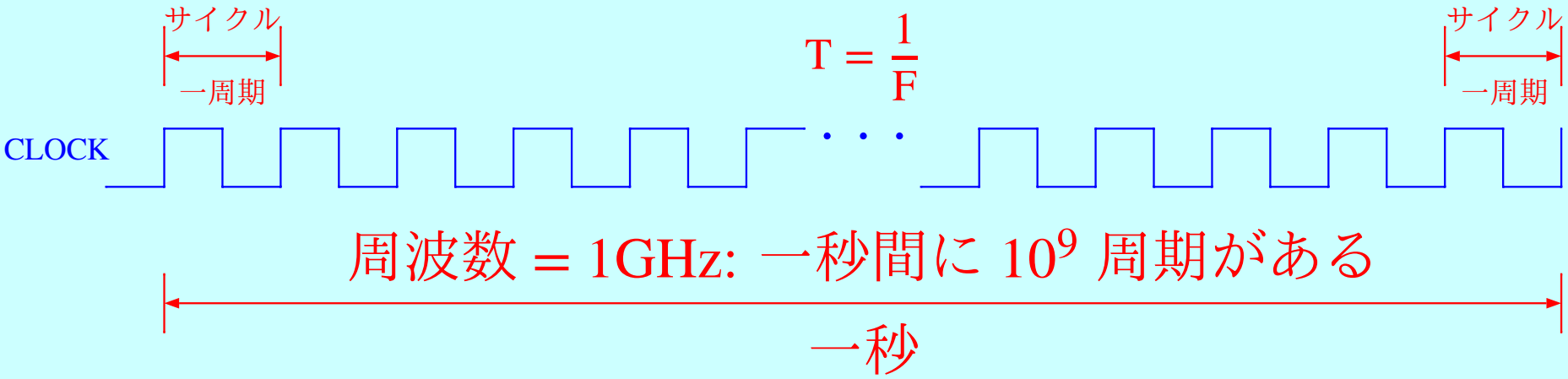
命令の実行手順

- プログラムカウンタ (Program Counter)
  - 現在実行しようとしている命令が入っているメモリのアドレスを 示す「プログラムカウンタ (PC)」と呼ばれる
- 命令の実行手順
  - メモリの中から命令を読み込み、 それを解読し、実行し、次に実行しようとしている命令の番地を PC に記憶する、という手順を繰り返すことになります
- 例 : add x3, x1, x2 # x3 ← x1 + x2
  1. PC によって、メモリから命令をフェッチ (Fetch) する
  2. レジスタ 1 と 2 に格納されている値を読出す
  3. その 2 つの値を加算する
  4. 結果をレジスタ 3 に格納し、PC + 4 を PC に記憶する

単一サイクル CPU


1 サイクル
一周期
1 サイクル
一周期
CLOCK
PC によって、メモリの中から
命令を読み込み、
それを解読し、実行し
PC によって、メモリの中から
命令を読み込み、
それを解読し、実行し
格納し、
している
記憶する
結果をレジスタに格納し、
次に実行しようとしている
命令の番地を PC に記憶する
結果をレジスタに格納し、
次に実行しようとしている
命令の番地を PC に記憶する
一つの命令の実行
一つの命令の実行
時間

周波数と一サイクルの時間の関係



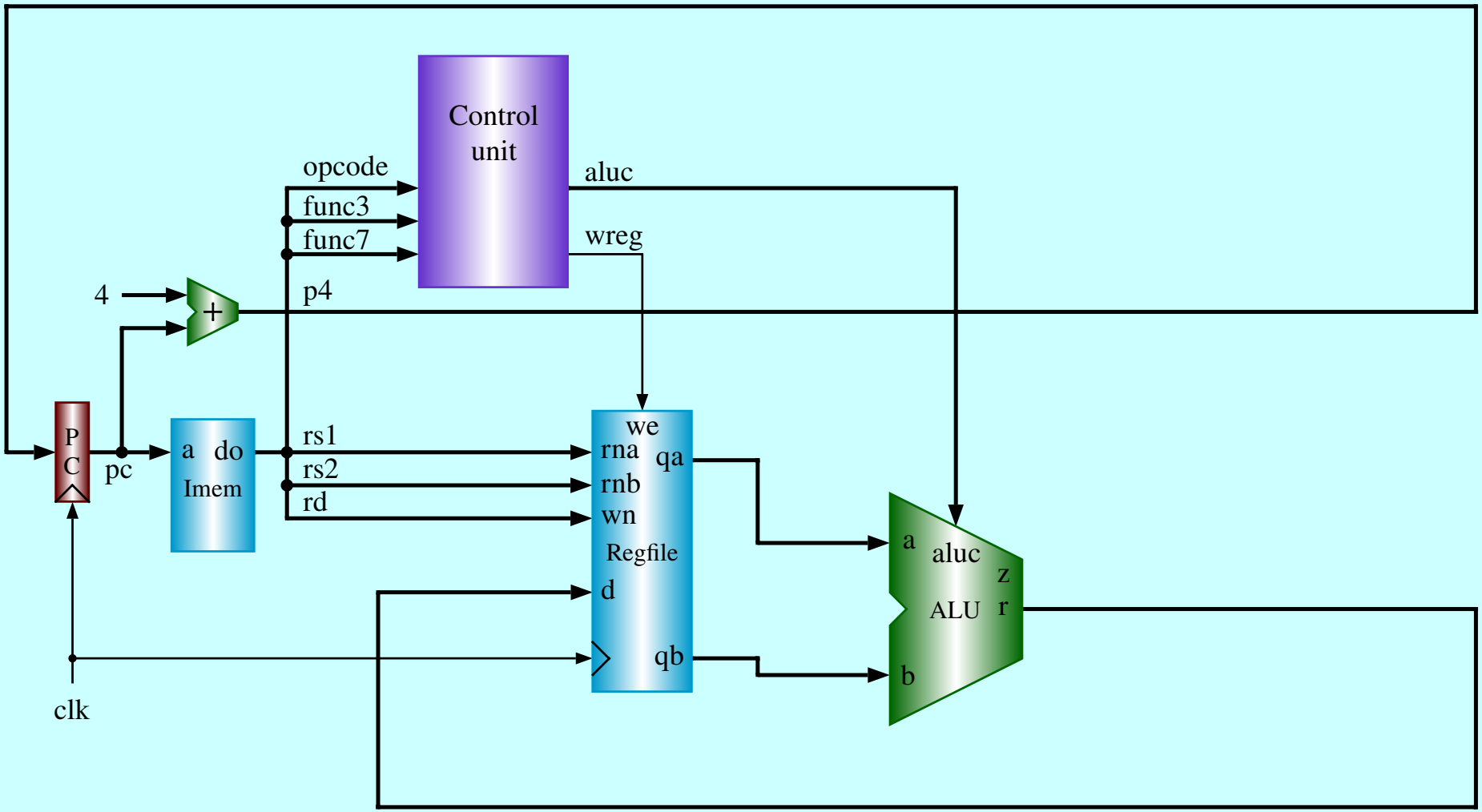
周波数 F = 1GHz: 一周期の時間 $T = \frac{1}{1 \times 10^9} = 10^{-9}\mathrm{s} = 1\mathrm{ns}$

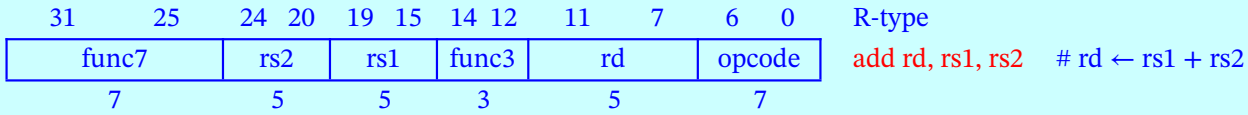
周波数 F = 1MHz: 一周期の時間 $T = \frac{1}{1 \times 10^6} = 10^{-6}\mathrm{s} = 1\mu\mathrm{s}$

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)

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

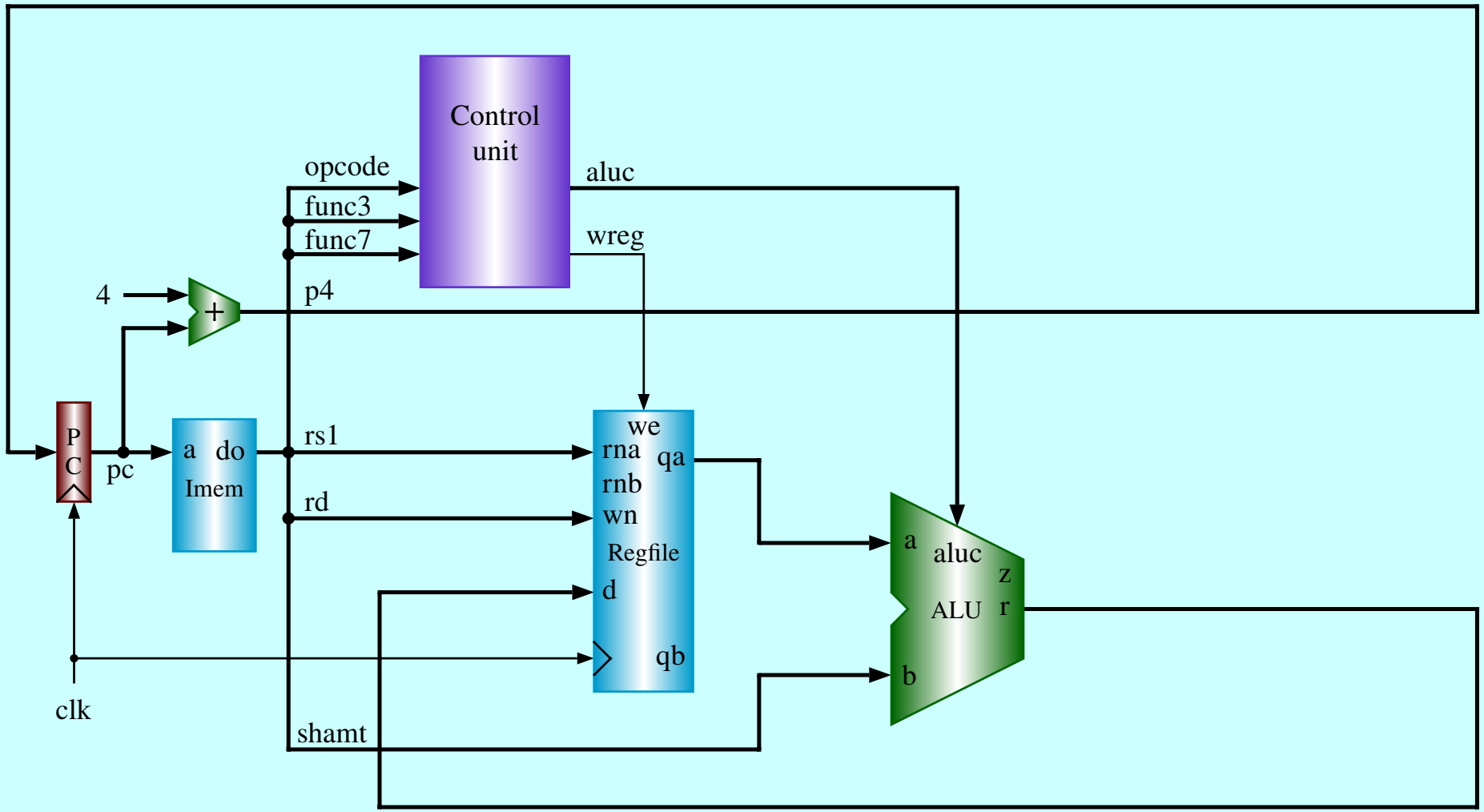




R-type

add rd, rs1, rs2 # rd ← rs1 + rs2

RISC-V slli, srli, srai

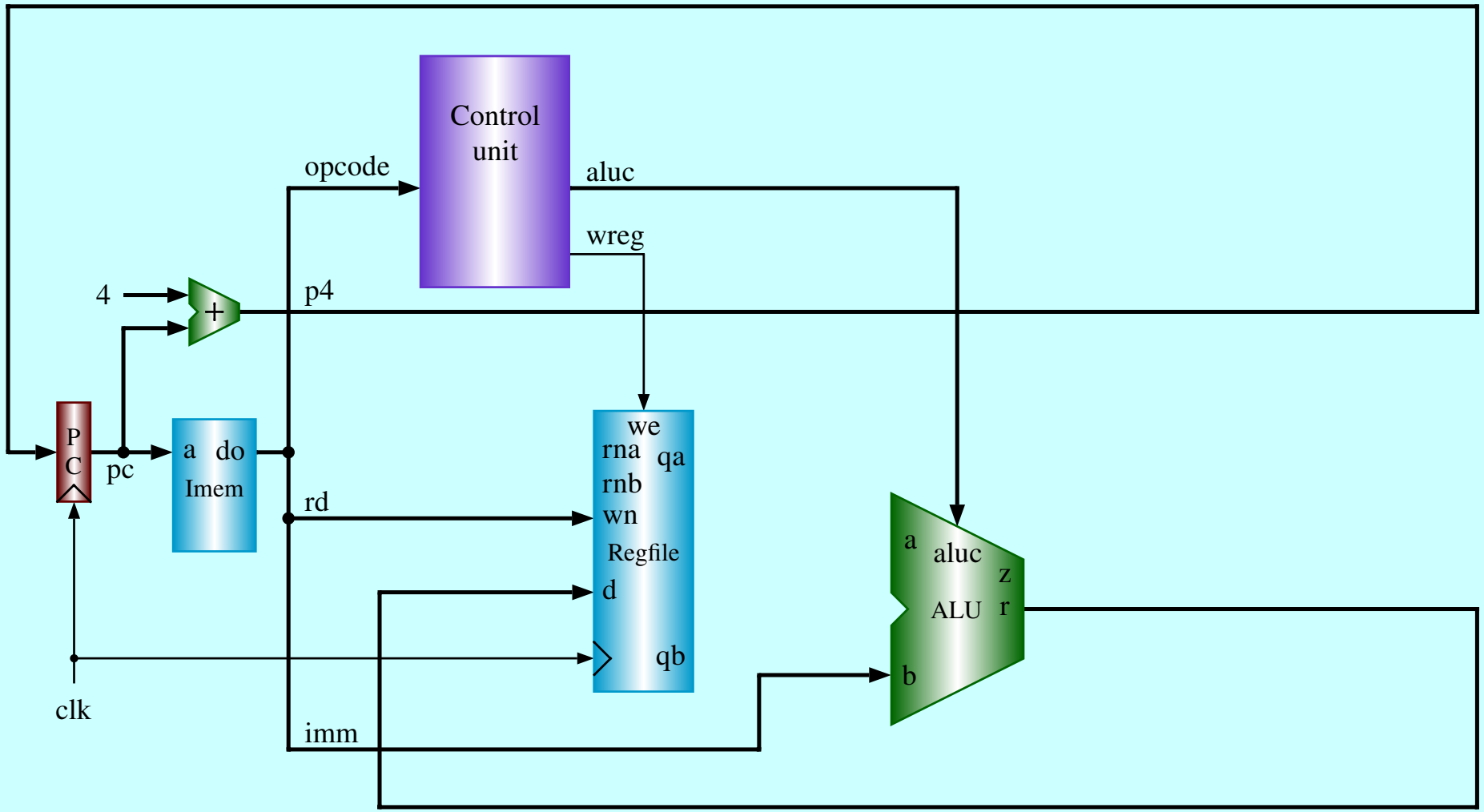


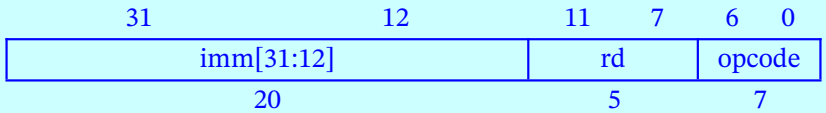
31 25	24 20	19 15	14 12	11 7	6 0
func7	shamt	rs1	func3	rd	opcode
7	5	5	3	5	7

H-type

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

RISC-V lui

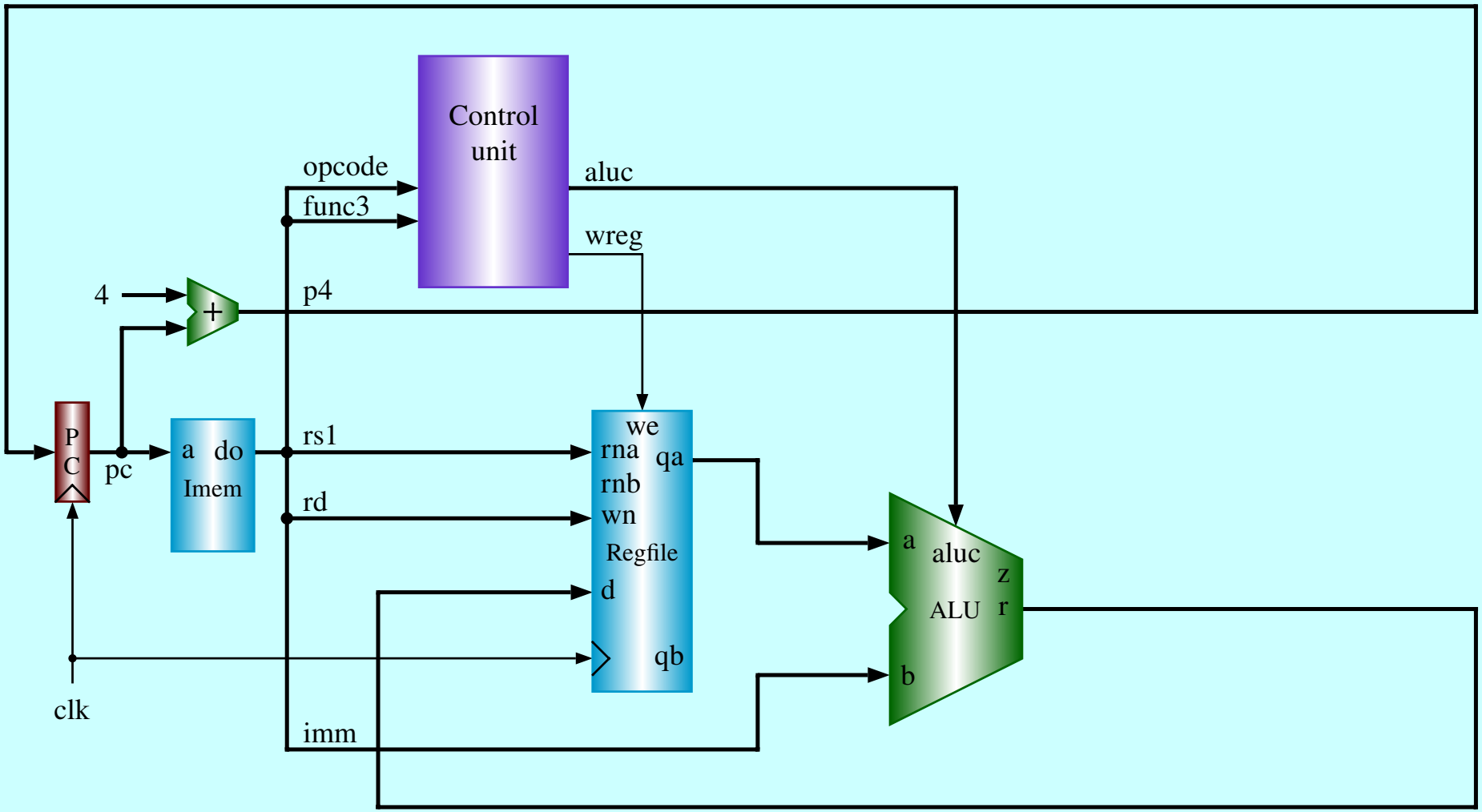




U-type

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

RISC-V addi, xori, ori, andi

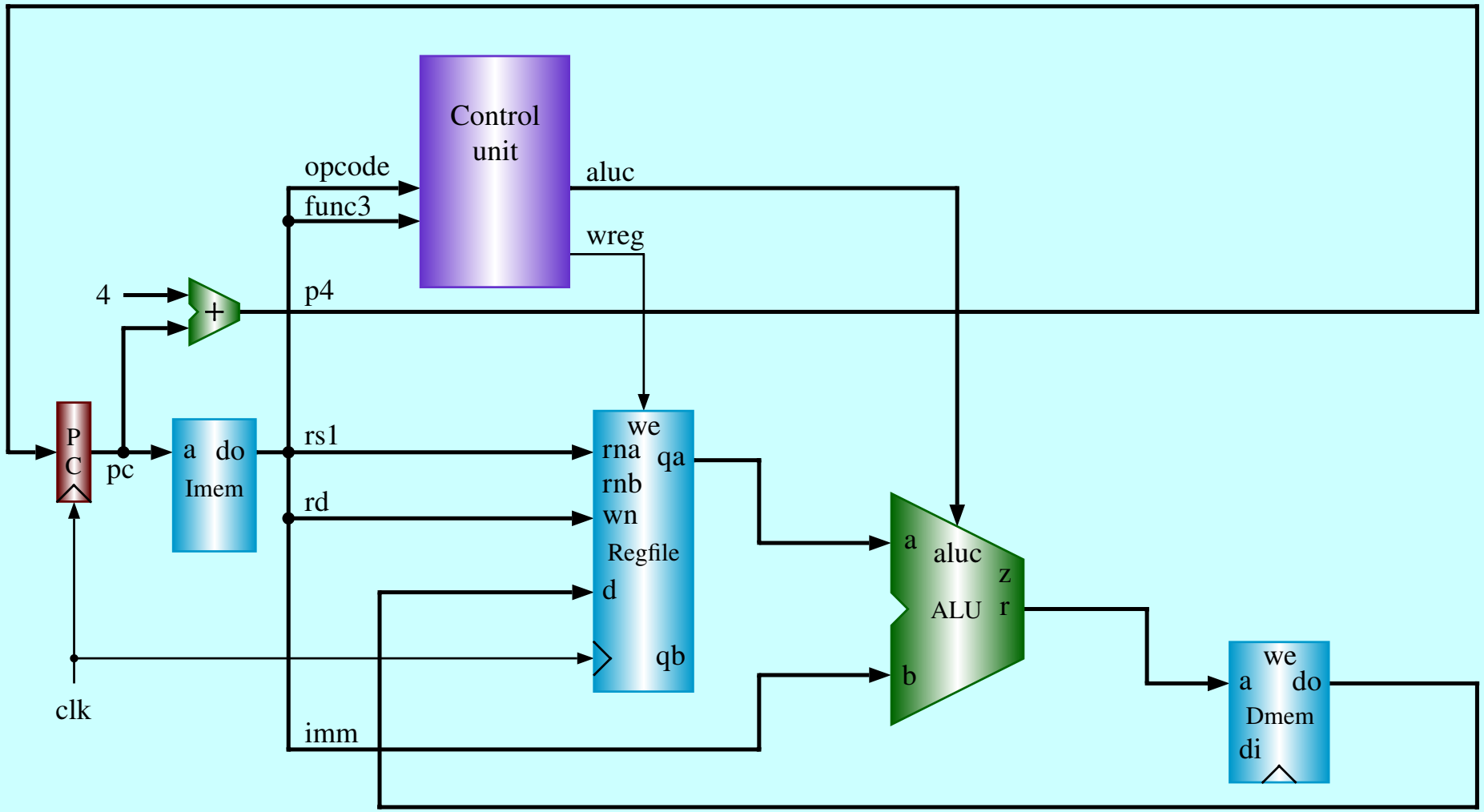


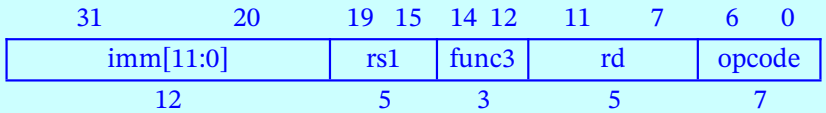
31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	func3	rd	opcode
12	5	3	5	7

I-type

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

RISC-V lw

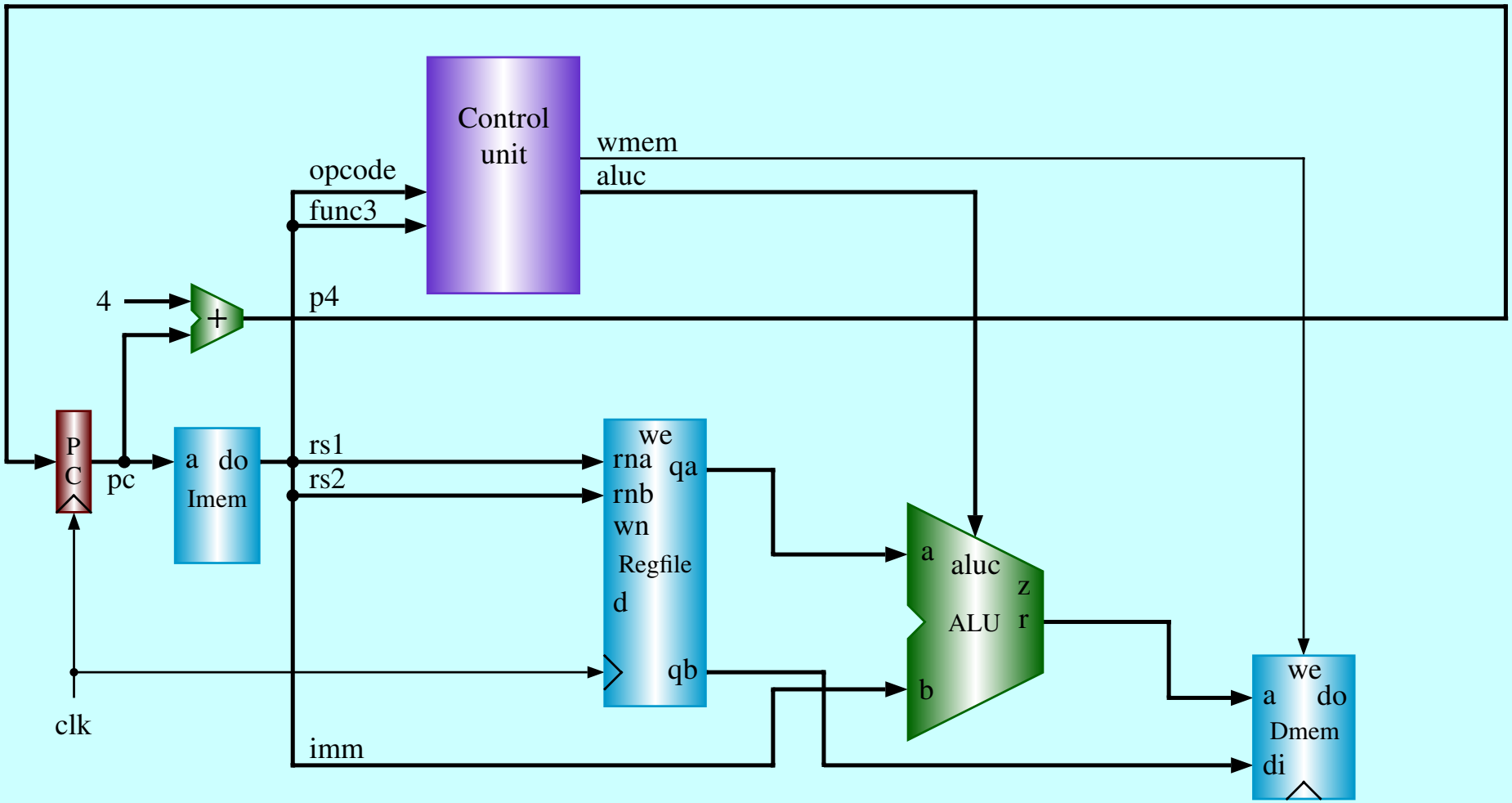




I-type

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

RISC-V sw

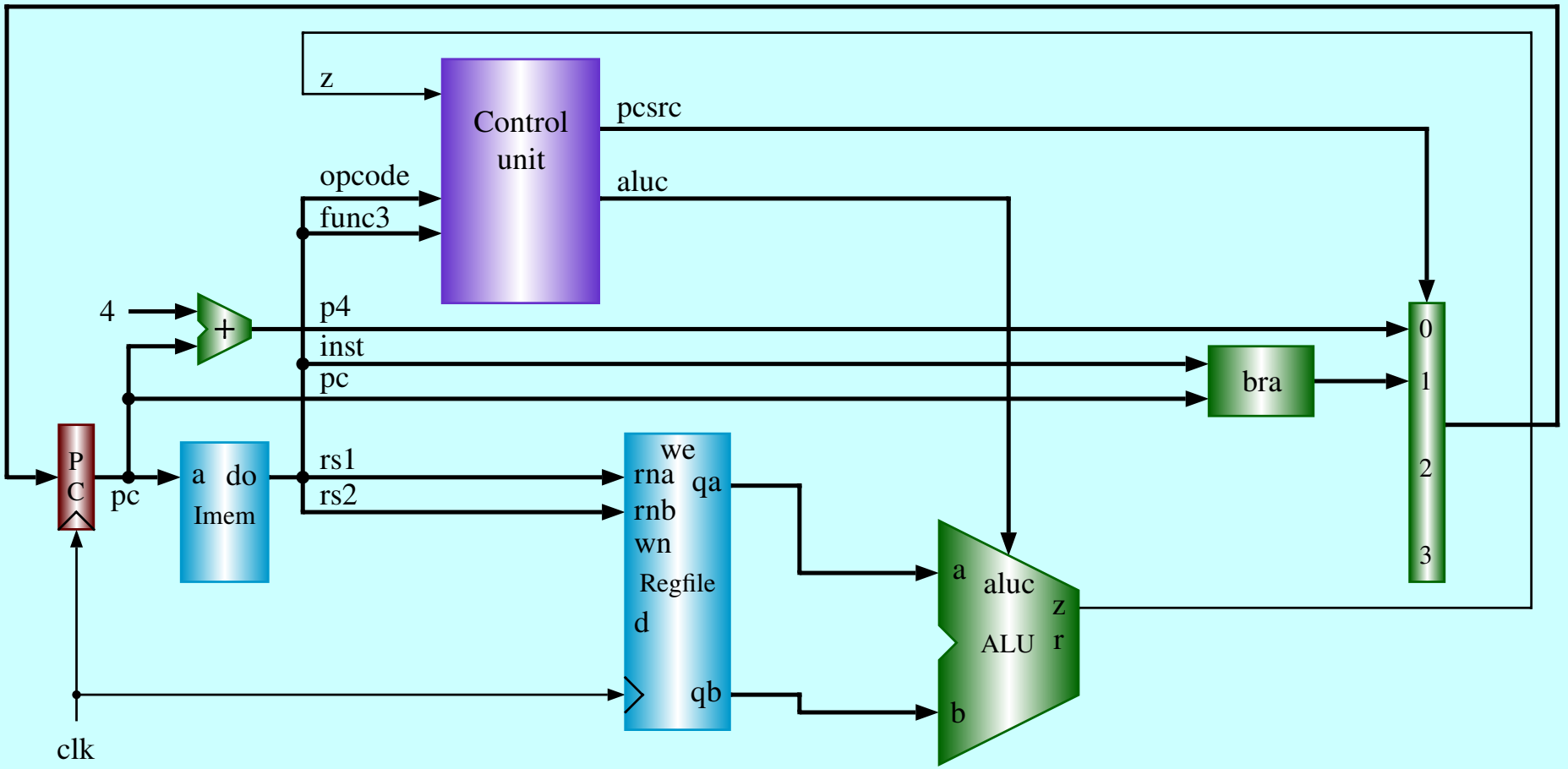


31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
7	5	5	3	5	7

S-type

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

RISC-V beq, bne

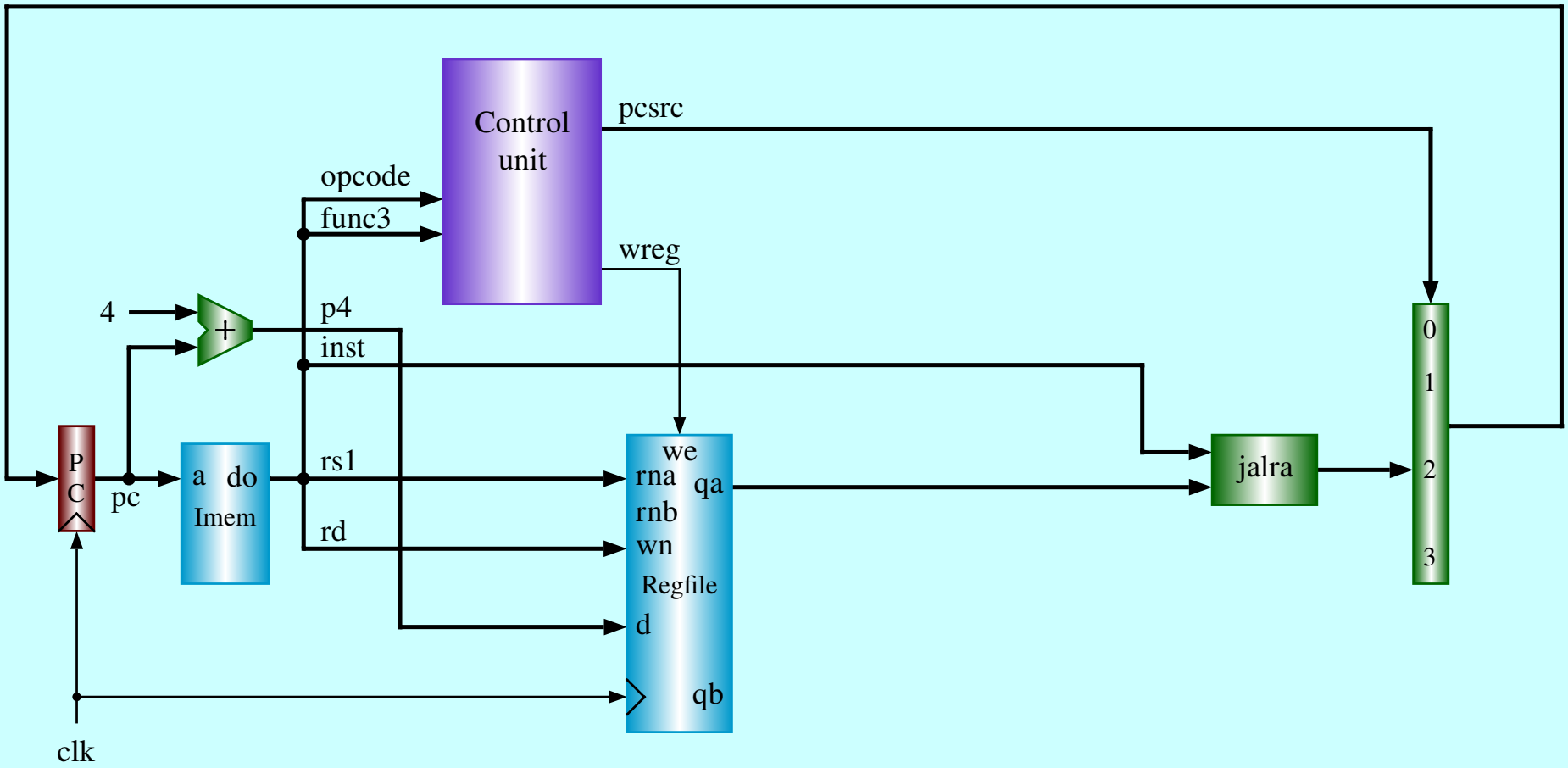


31 25	24 20	19 15	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
7	5	5	3	5	7

B-type

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

RISC-V jalr

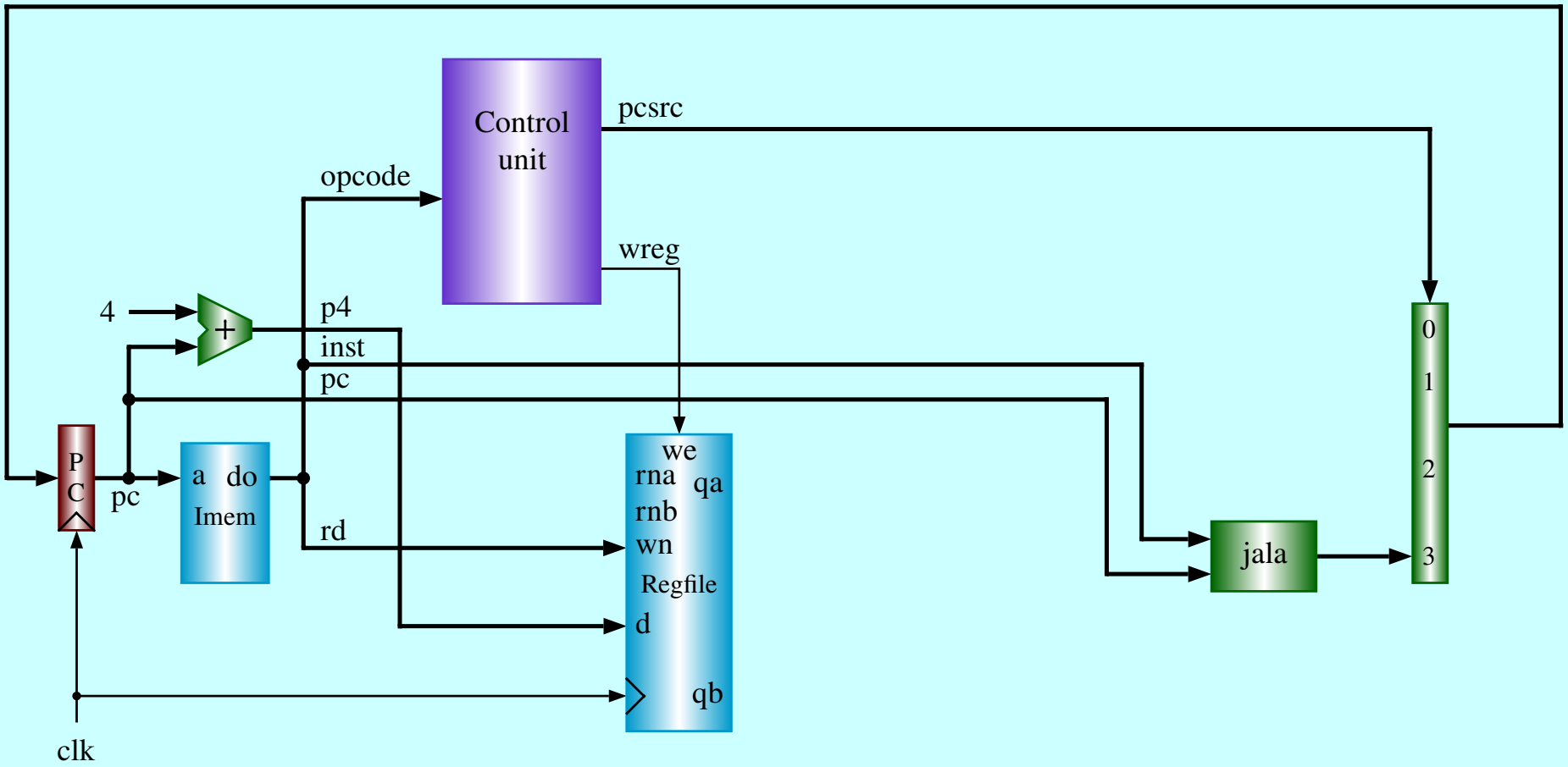


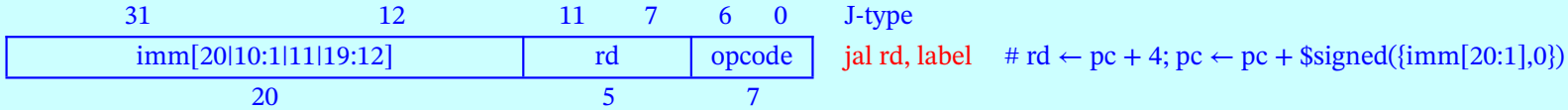
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
12	5	3	5	7

I-type

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

RISC-V jal

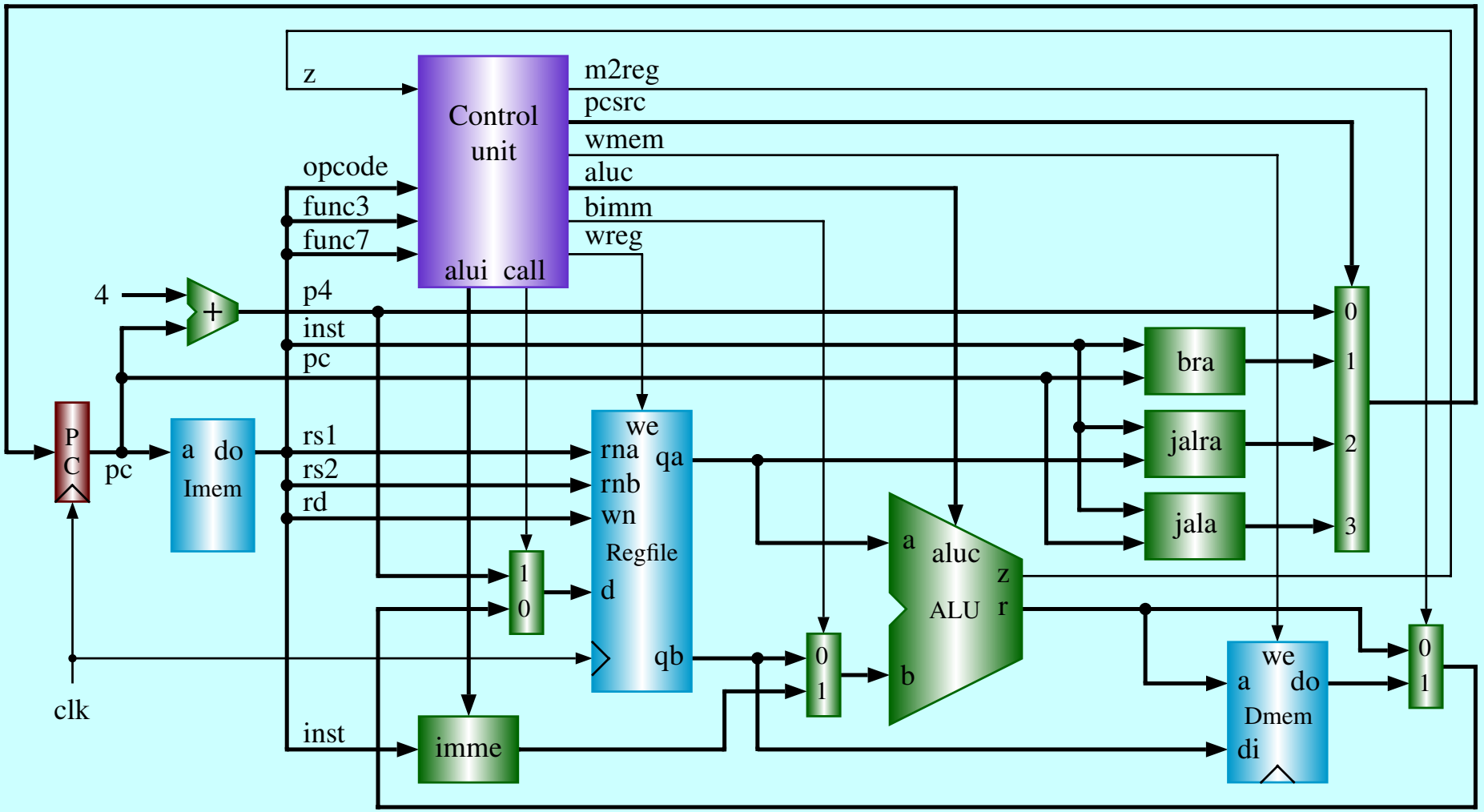




J-type

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

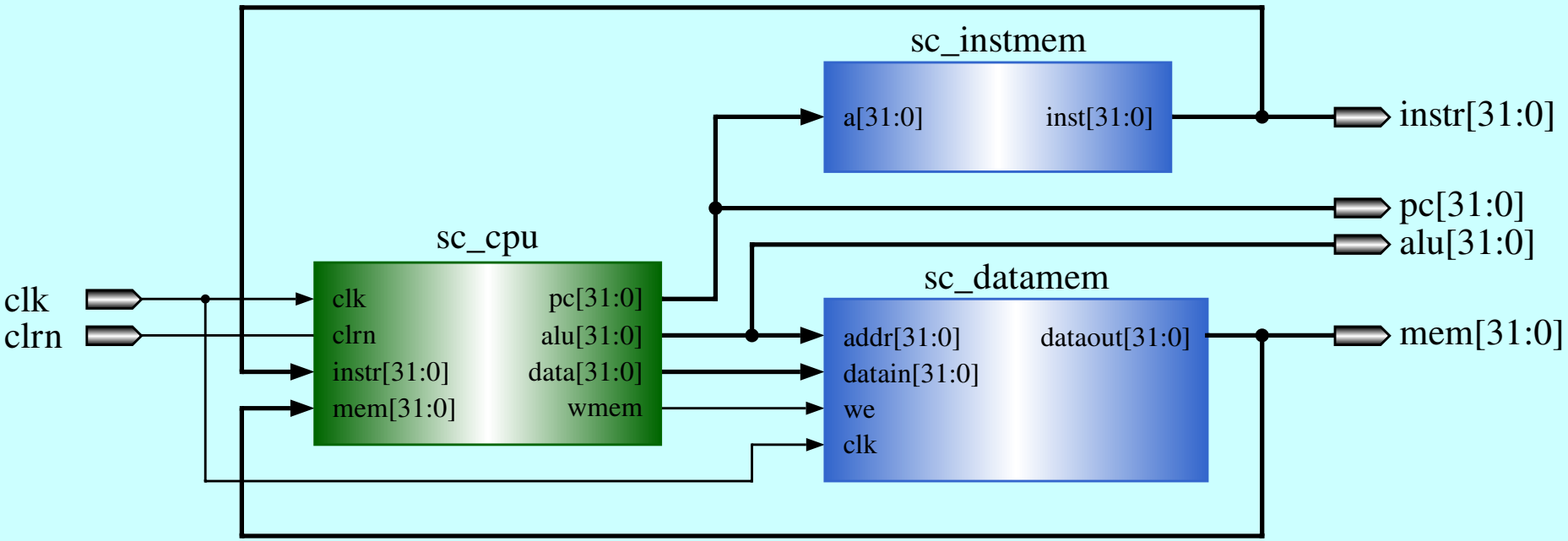
RISC-V コンピュータ



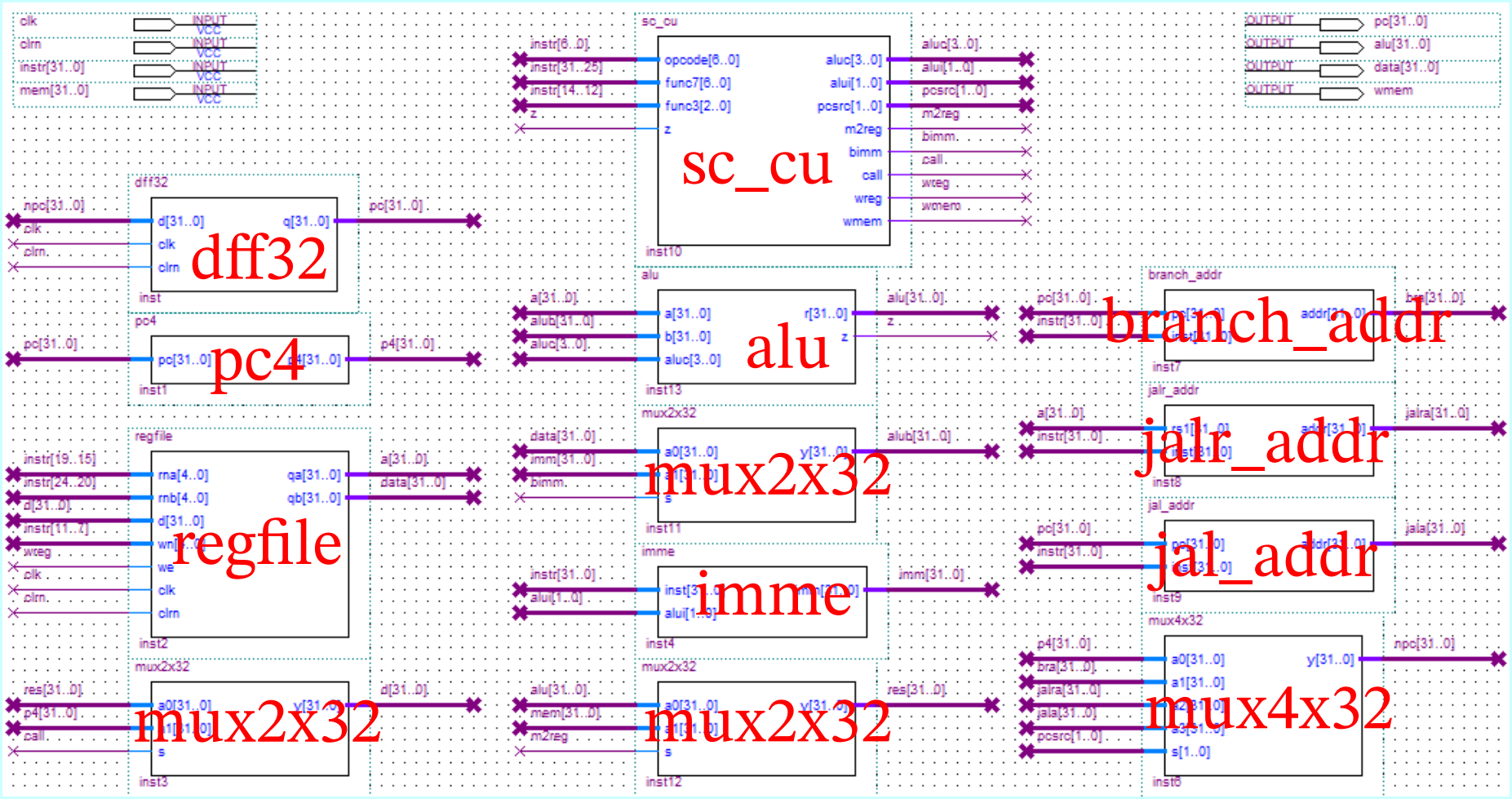
CPU + 命令メモリ Imem + データメモリ Dmem

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

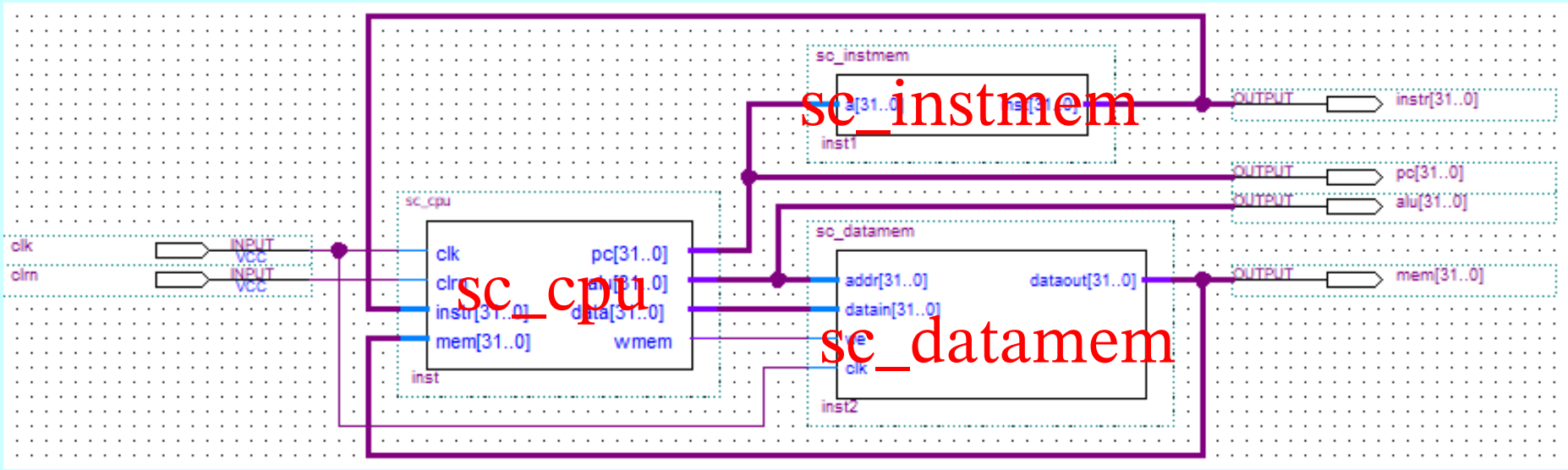


RISC-V CPU



sc_cpu の回路

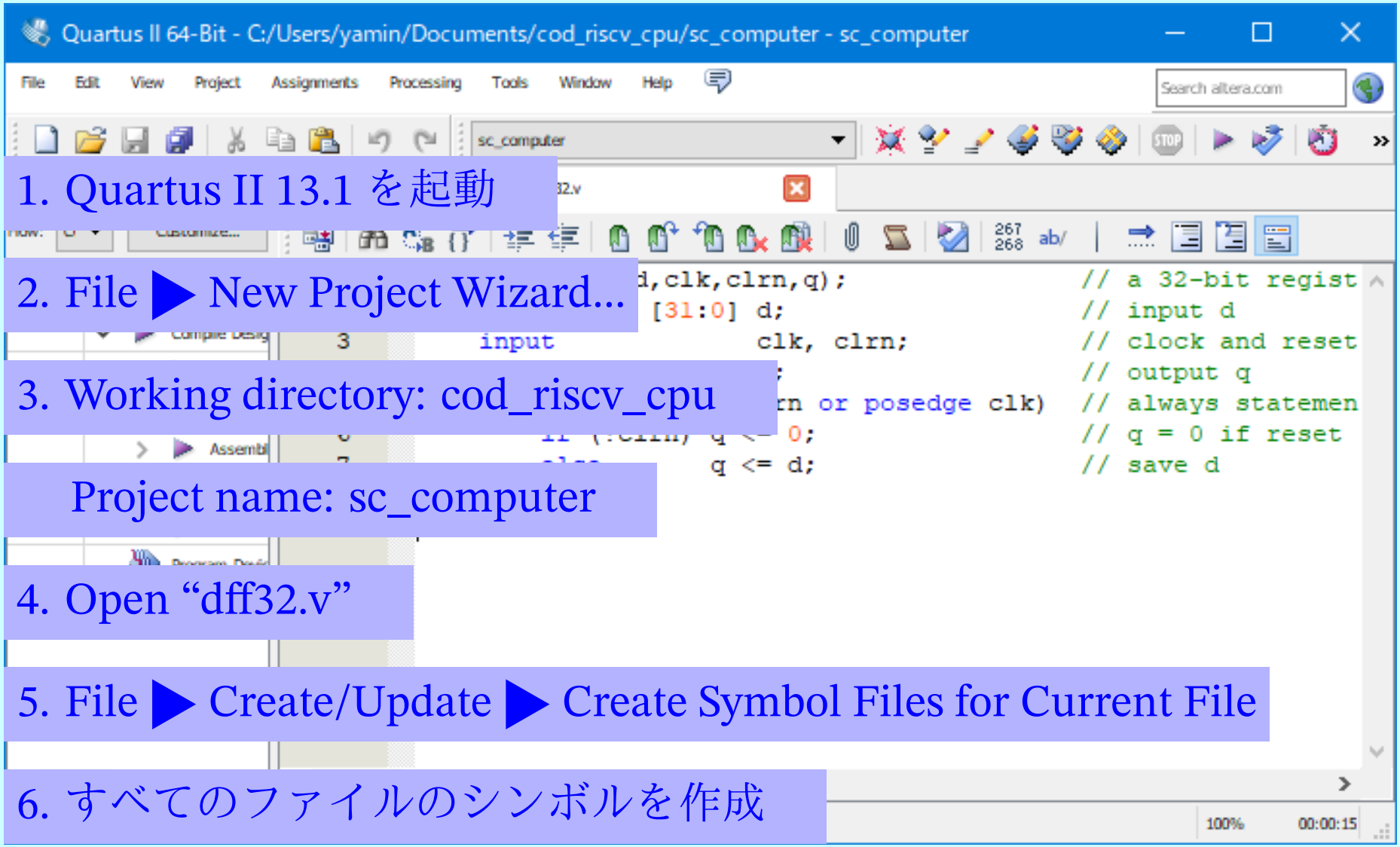
RISC-V コンピュータ



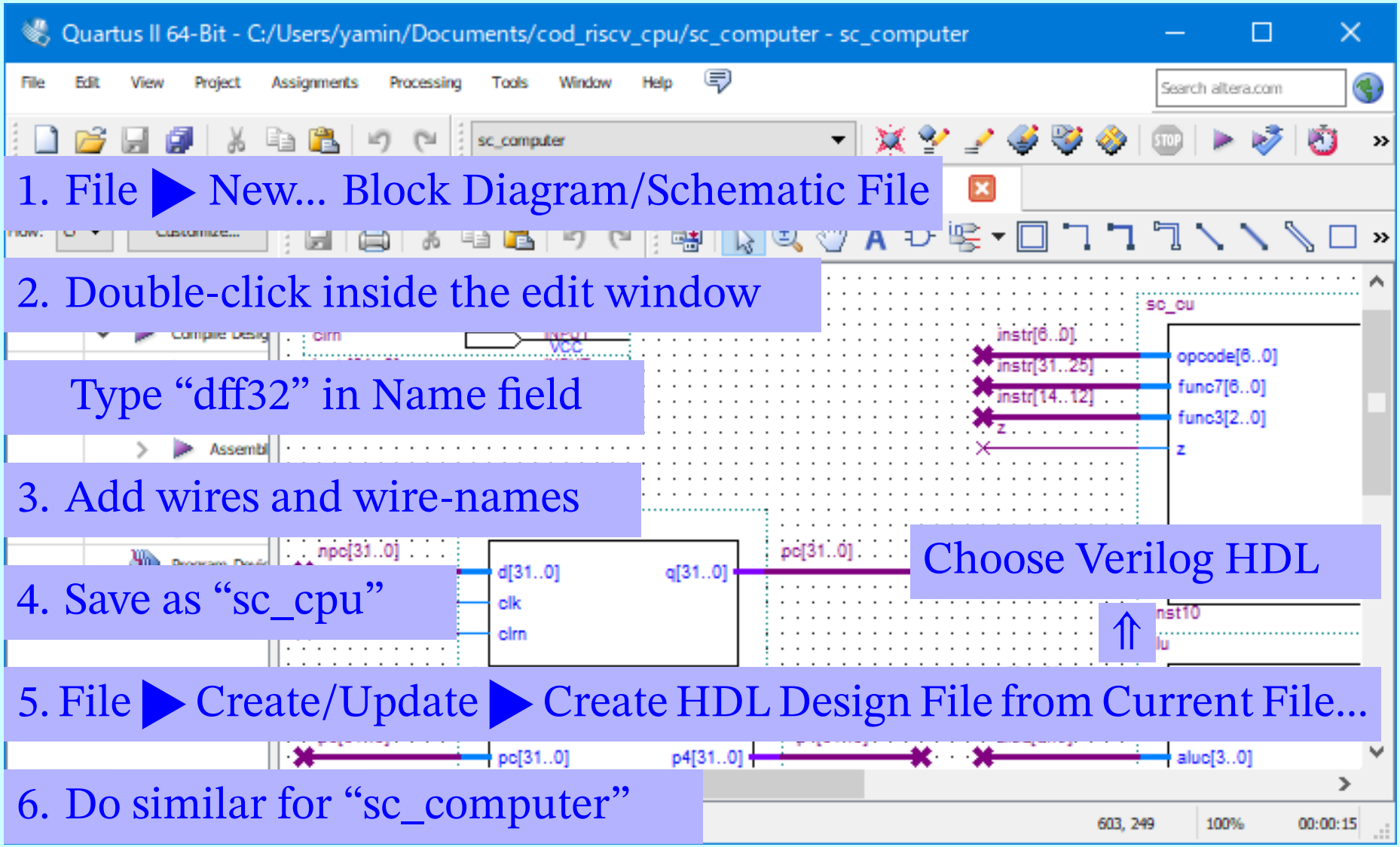
sc_computer の回路

Quartus II の使い方

シンボルファイルを作成

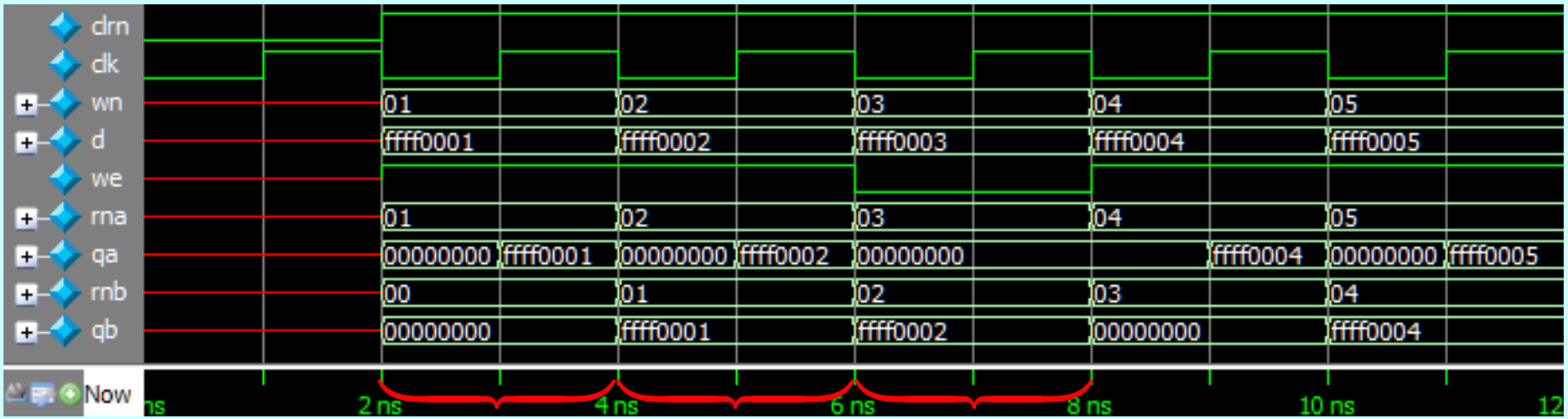


sc_cpu と sc_computer を作成



課題 IV (100 点)

1. テストベンチ regfile_tb.v を使って、ModelSim で regfile.v をシミュレーションしなさい。



また、2~4ns、4~6ns、6~8ns、62~64ns、64~66ns ところの波形について、それぞれ説明しなさい。

2. 回路図で sc_cpu を設計しなさい。