

Lab 7

Single-Cycle CPU Lab

Video Link : <https://youtu.be/1YRKz1B4Ed8>



Outline

1. Introduction
2. ISA / Verilog Review
3. CPU Architecture Review
4. Example of CPU Micro-Architecture



Pework of implementing a CPU

You need to **be familiar with** ...

- | | |
|---|----------------------------|
| 1. ISA | (Lab2 ~ Lab3 – RISC-V ISA) |
| 2. HDL (Verilog / SystemVerilog / VHDL) | (Lab4 ~ Lab6 – Verilog) |
| 3. Concept of CPU Architecture | (Lab7) |



ISA / Verilog Review



ISA (Instruction Set Architecture)

Software

```
000000000000100000000001010010011
0000000000001000000000001100010011
000000000000100101001001110010011
000000000000100110001111000010011
00000001110000111000111010110011
```

Machine Instructions

ISA

ISA is an abstraction of hardware

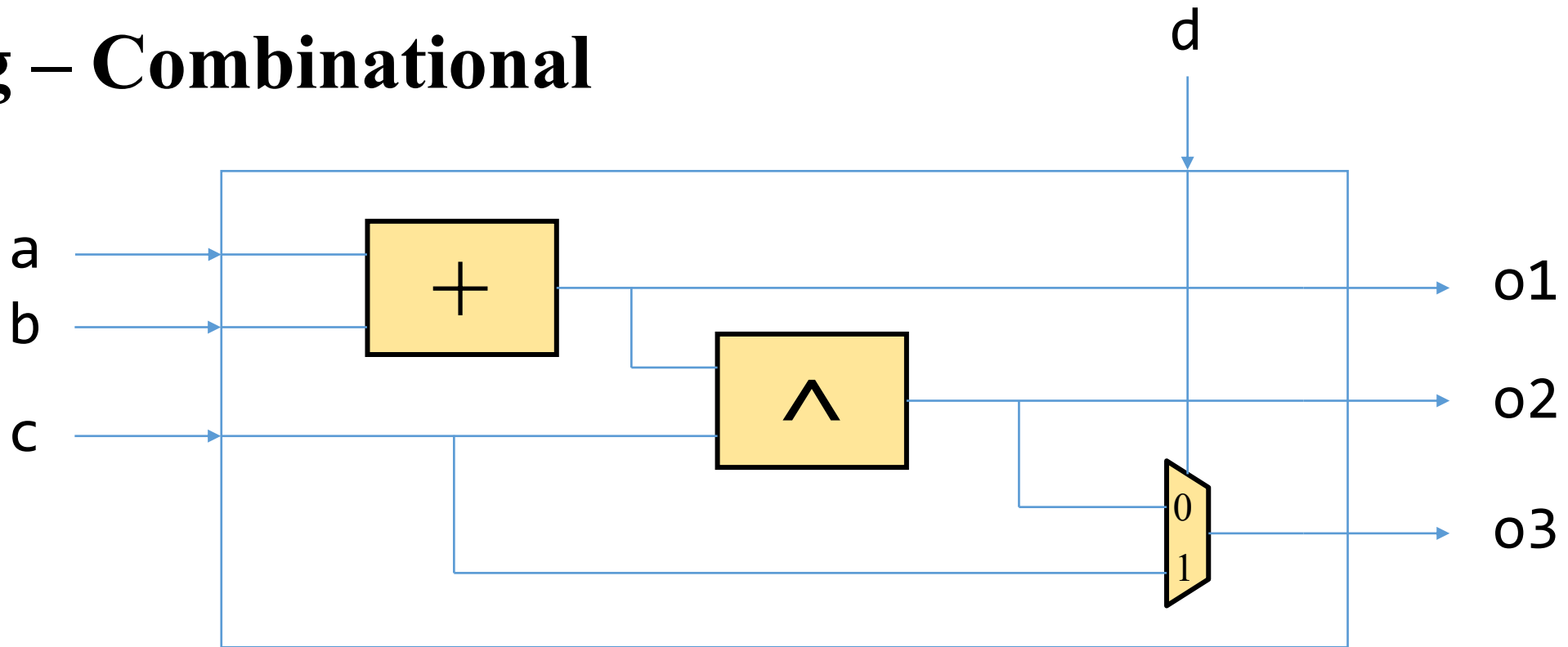
ISA is an implementation target

Hardware

HDL

(Verilog / SystemVerilog / VHDL)

Verilog – Combinational



Recommand

```
wire o1, o2, o3;

assign o1 = a + b;
assign o2 = o1 ^ c;
assign o3 = (d) ? c : o2;
```

```
reg o1, o2, o3;

always @ (*) begin
    o1 = a + b;
    o2 = o1 ^ c;
    o3 = (d) ? c : o2;
end
```

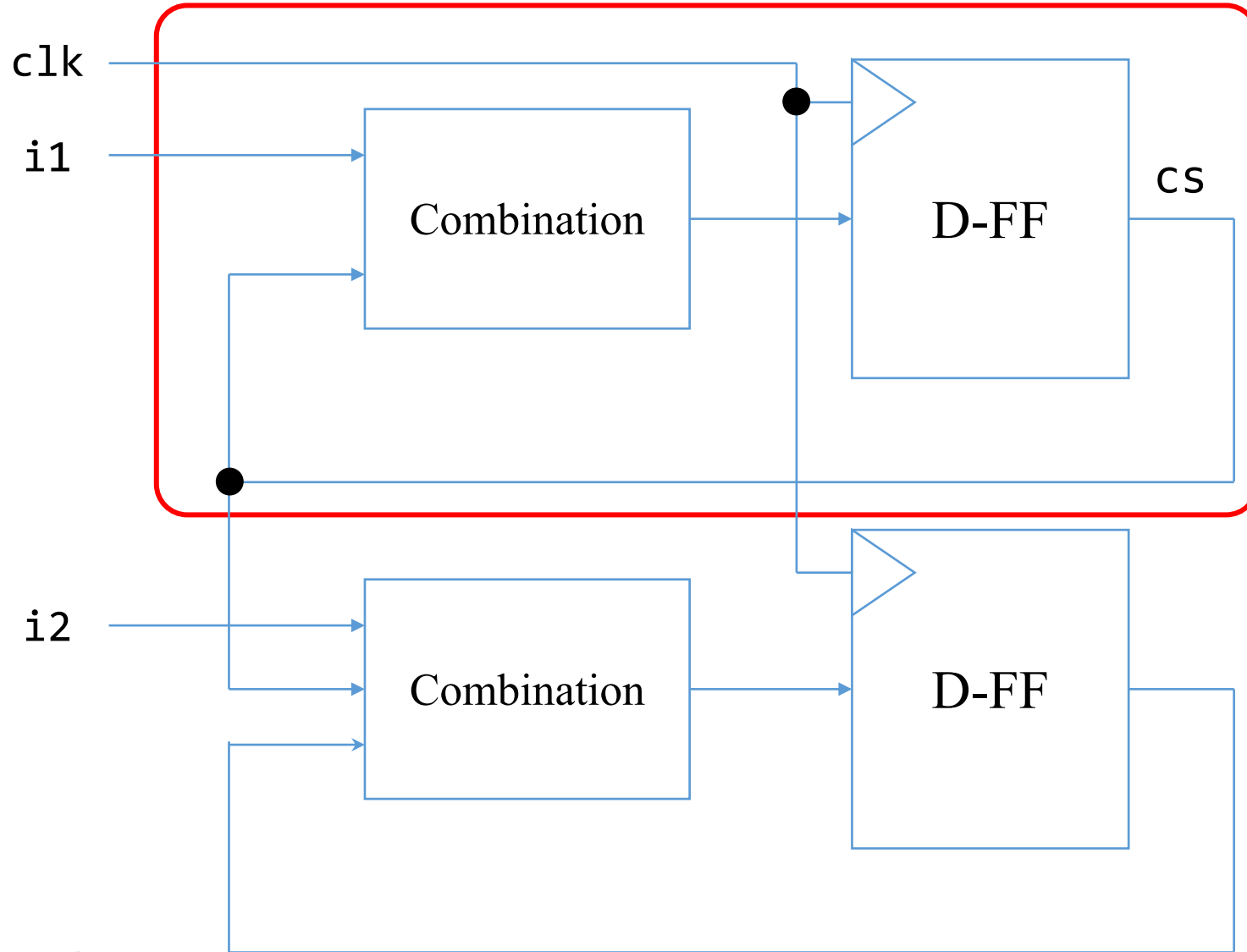
```
reg o1, o2, o3;

always @ (*) begin
    o2 = o1 ^ c;
    o1 = a + b;
    o3 = (d) ? c : o2;
end
```

```
reg o1, o2, o3;

always @ (*) begin
    o1 = a + b;
    o2 = o1 ^ c;
    if (d)
        o3 = c;
    else
        o3 = o2;
end
```

Verilog – Sequential

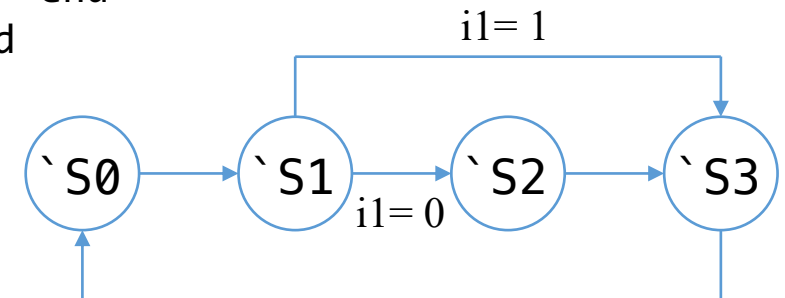


2 phases

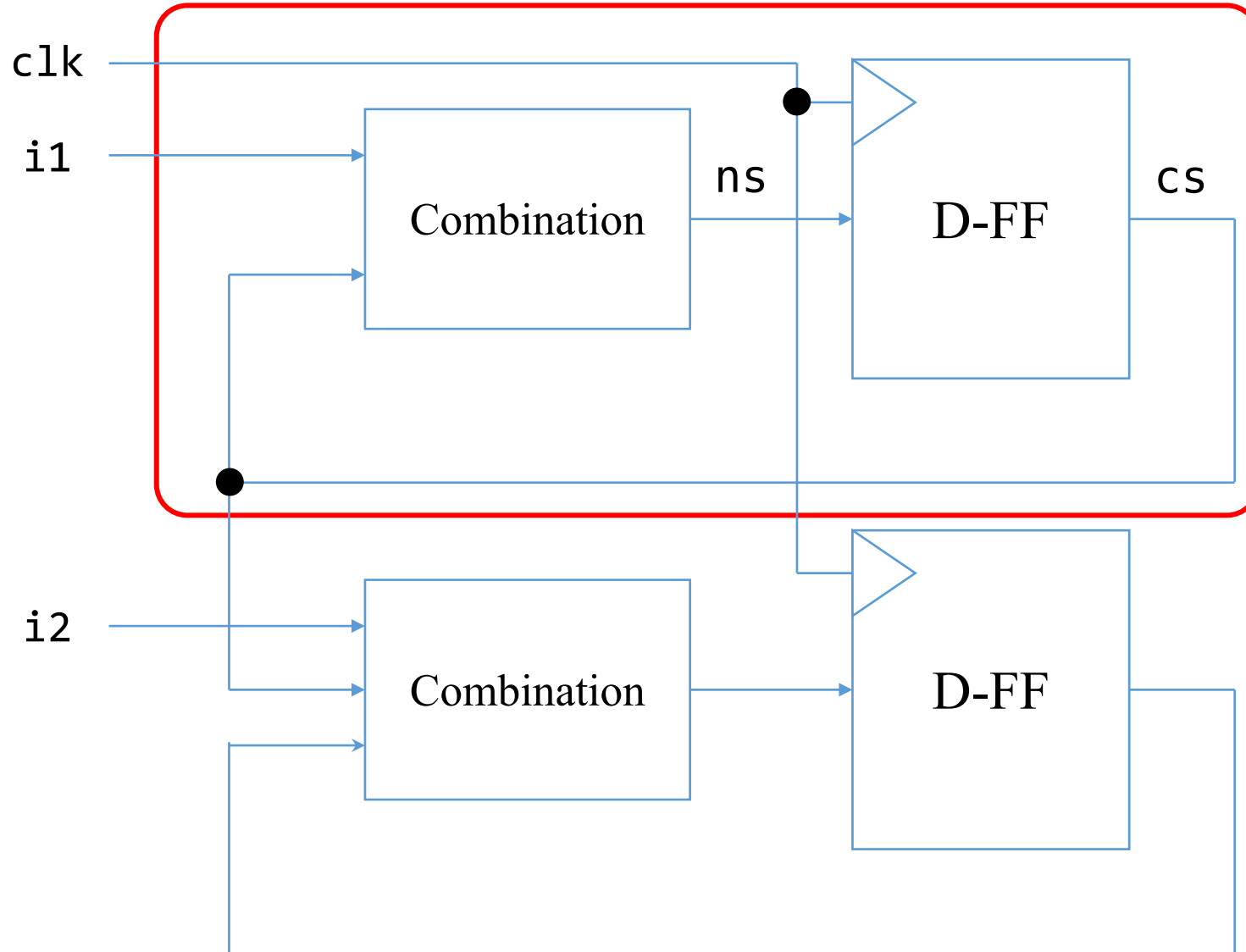
1. Evaluate (Before Clock Edge)
2. Update (When Clock Edge)

Method 1 :

```
always@(posedge clk) begin
    if (rst)
        cs <= `S0;
    else begin
        case(cs)
            `S0 : cs <= `S1;
            `S1 : cs <= (i1) ? `S3 : `S2;
            `S2 : cs <= `S3;
            `S3 : cs <= `S0;
            default : cs <= `S0;
        endcase
    end
end
```



Verilog – Sequential



2 phases

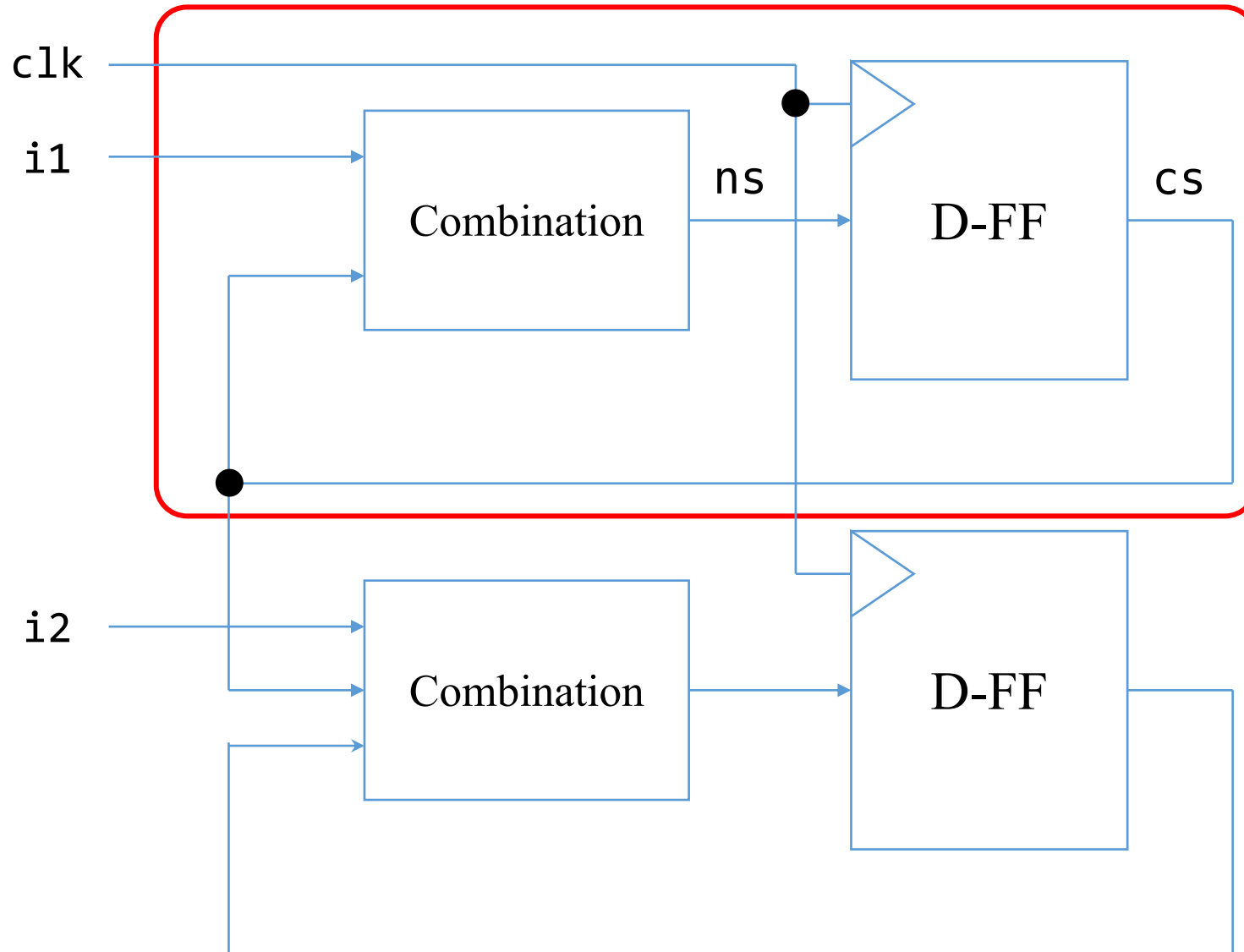
1. Evaluate (Before Clock Edge)
2. Update (When Clock Edge)

Method 2 (Recommend) :

```
always@(posedge clk) begin
    if (rst)
        cs <= `S0;
    else
        cs <= ns;
end
```

```
always@(*) begin
    case(cs)
        `S0: ns = `S1;
        `S1: ns = (i1) ? `S3 : `S2;
        `S2: ns = `S3;
        `S3: ns = `S0;
        default : ns = `S0;
    endcase
end
```


Verilog – Sequential



2 phases

1. Evaluate (Before Clock Edge)
2. Update (When Clock Edge)

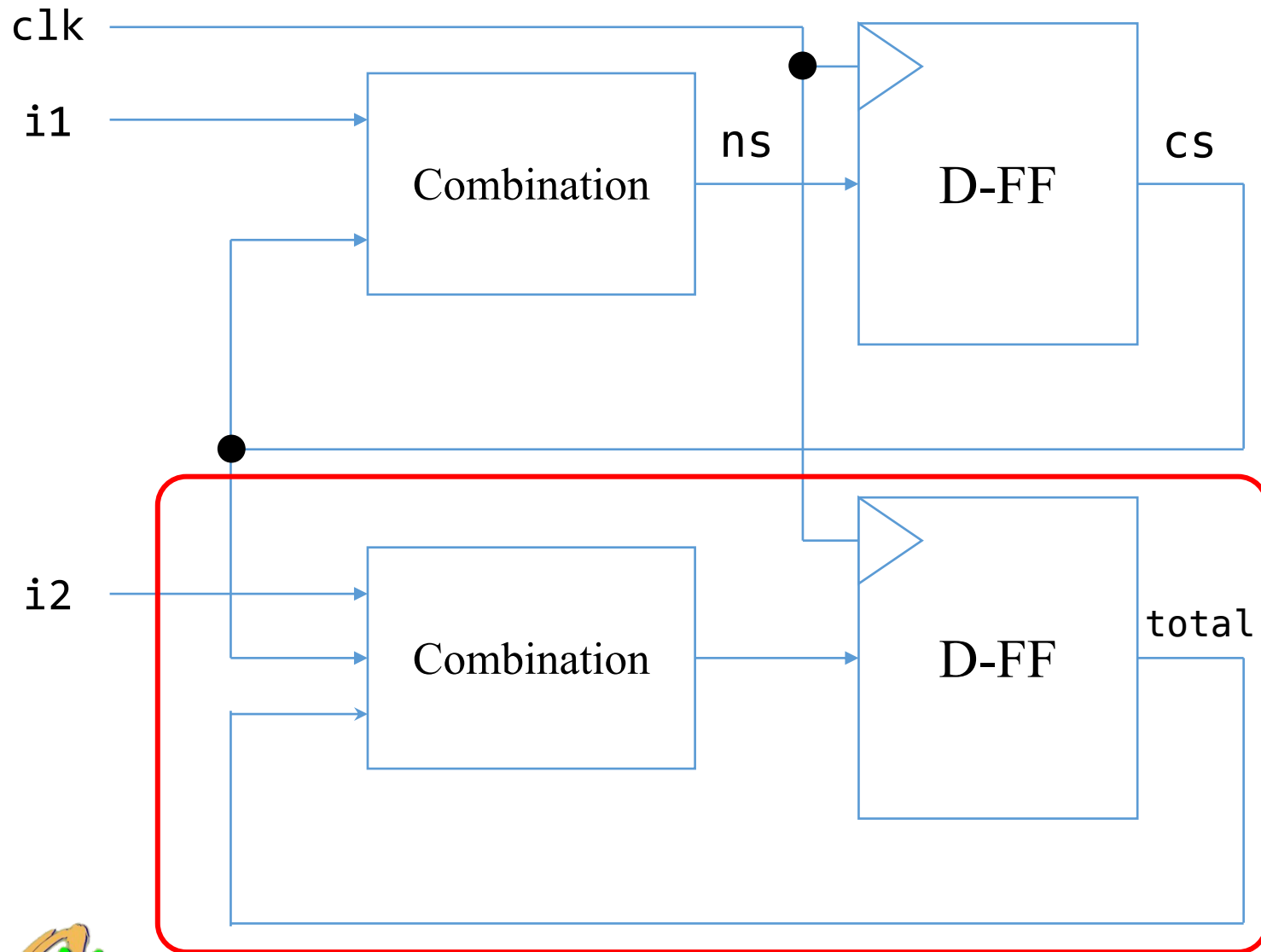
Method 2 (Recommend) :

```
always@(posedge clk or posedge rst) begin
    if (rst)
        cs <= `S0;
    else
        cs <= ns;
end
```

Asynchronous

```
always@(*) begin
    case(cs)
        `S0: ns = `S1;
        `S1: ns = (i1) ? `S3 : `S2;
        `S2: ns = `S3;
        `S3: ns = `S0;
        default : ns = `S0;
    endcase
end
```

Verilog – Sequential

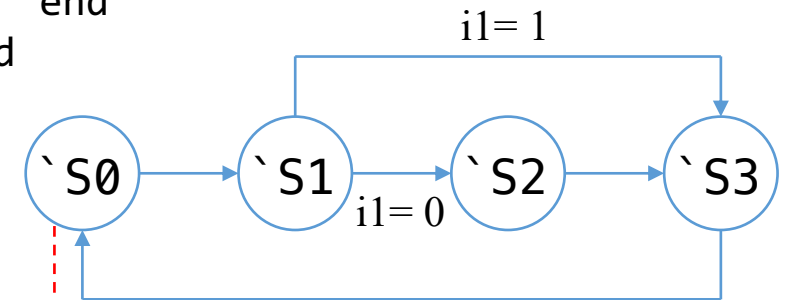


2 phases

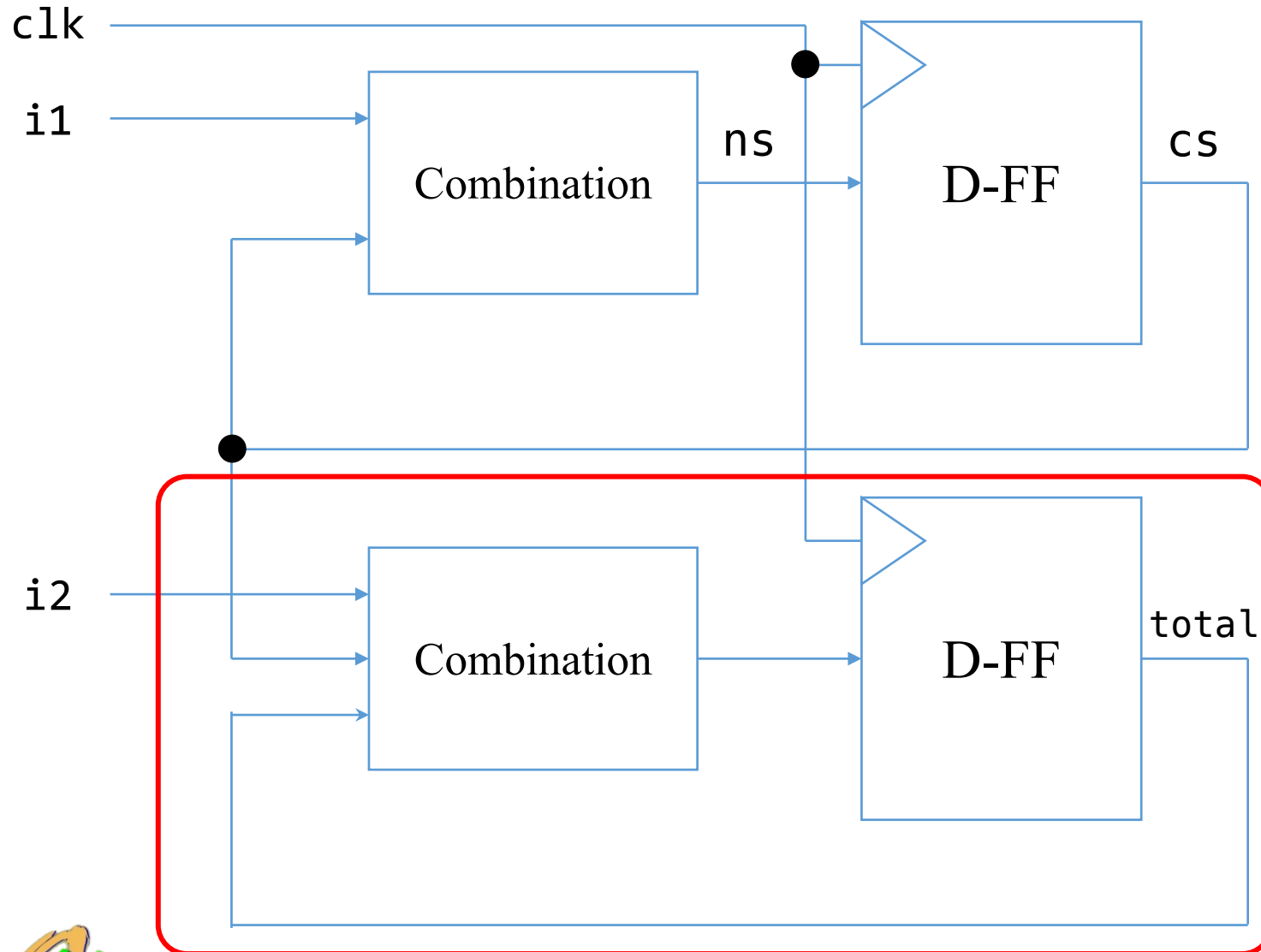
1. Evaluate (Before Clock Edge)
2. Update (When Clock Edge)

Method 1 :

```
always@(posedge clk) begin
    if (rst) begin
        total <= 0;
    end
    else if (cs == `S3) begin
        total <= 0;
    end
    else begin
        total <= total + i2;
    end
end
```



Verilog – Sequential



2 phases

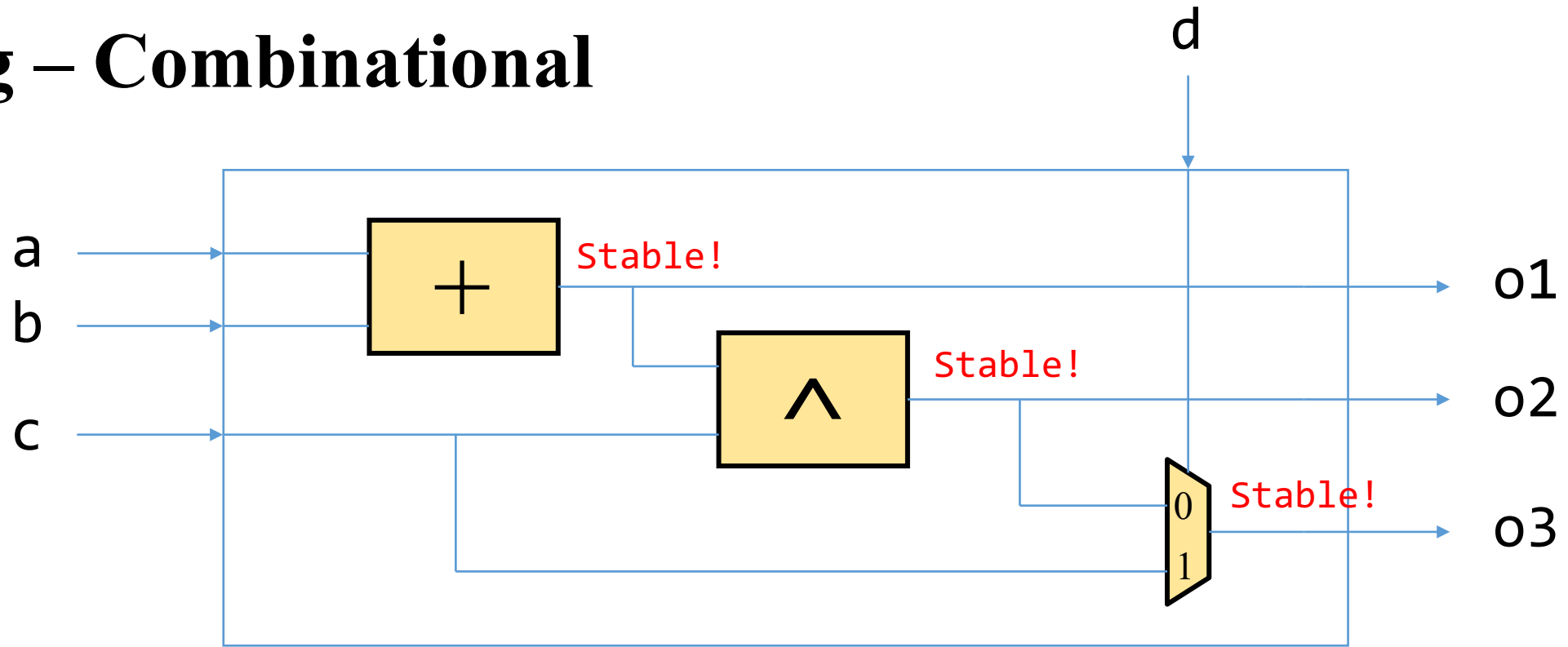
1. Evaluate (Before Clock Edge)
2. Update (When Clock Edge)

Method 2 :

```
always@(posedge clk) begin
    if (rst) begin
        total <= 0;
    end
    else if (cs == `S3) begin
        total <= 0;
    end
    else begin
        total <= total_tmp;
    end
end

assign total_tmp = total + i2;
```

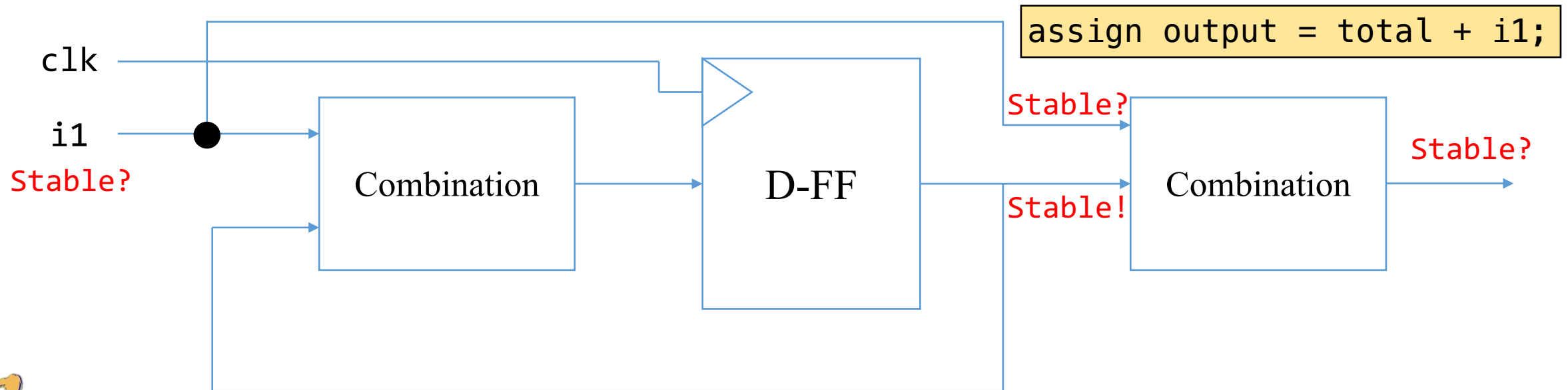
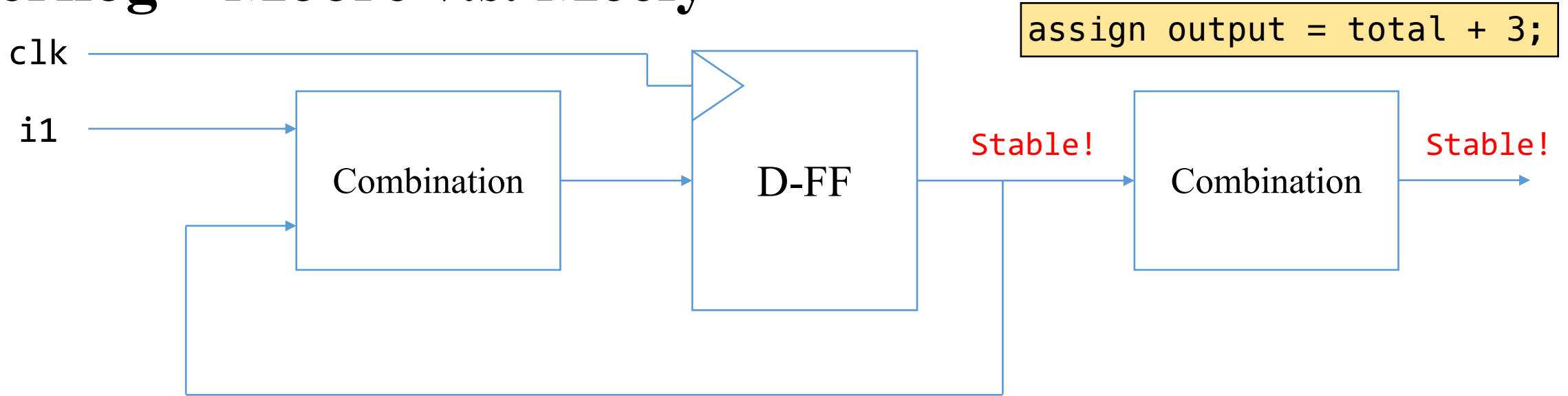
Verilog – Combinational



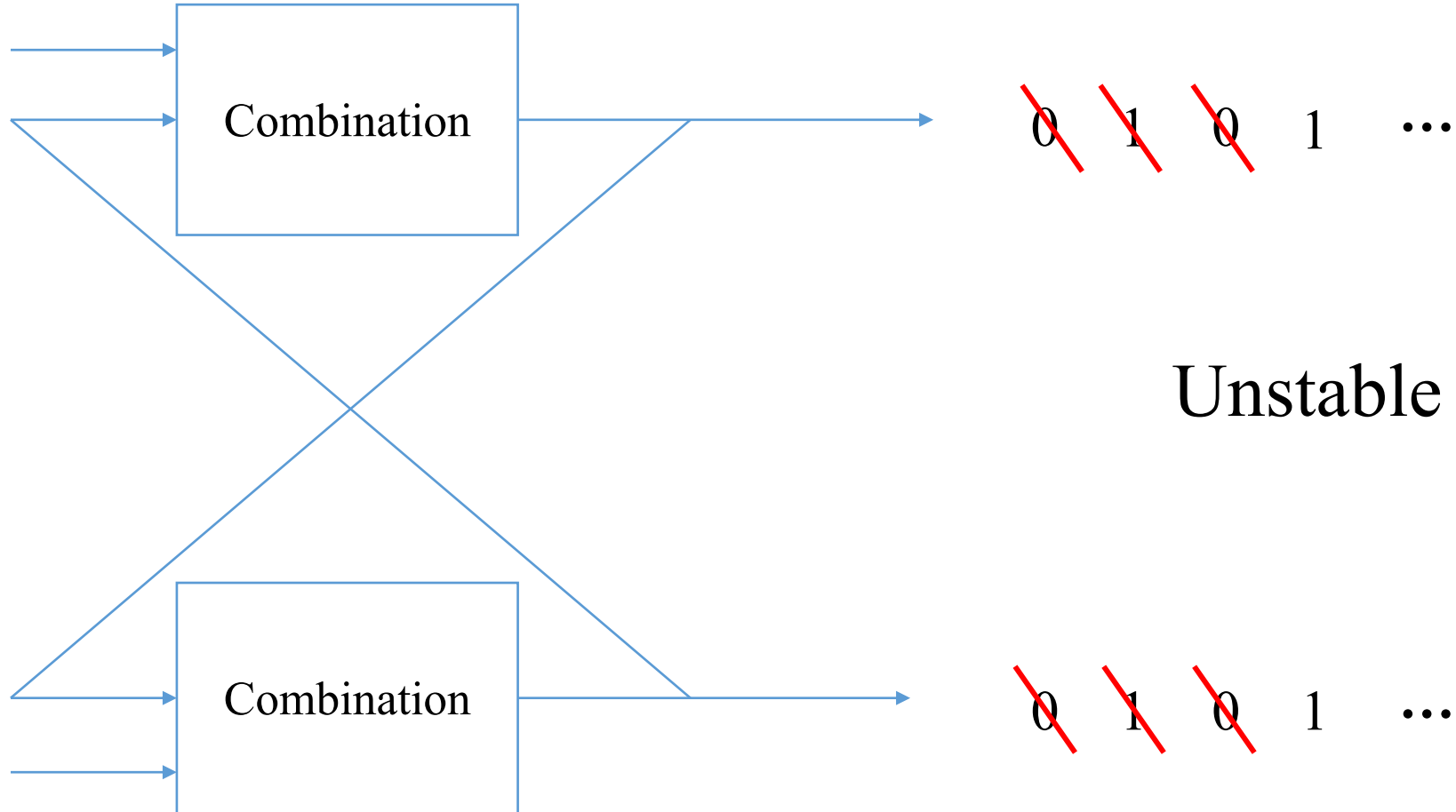
Stable!

~~Stable?~~
Stable!

Verilog – Moore v.s. Mealy



Verilog – Attention !

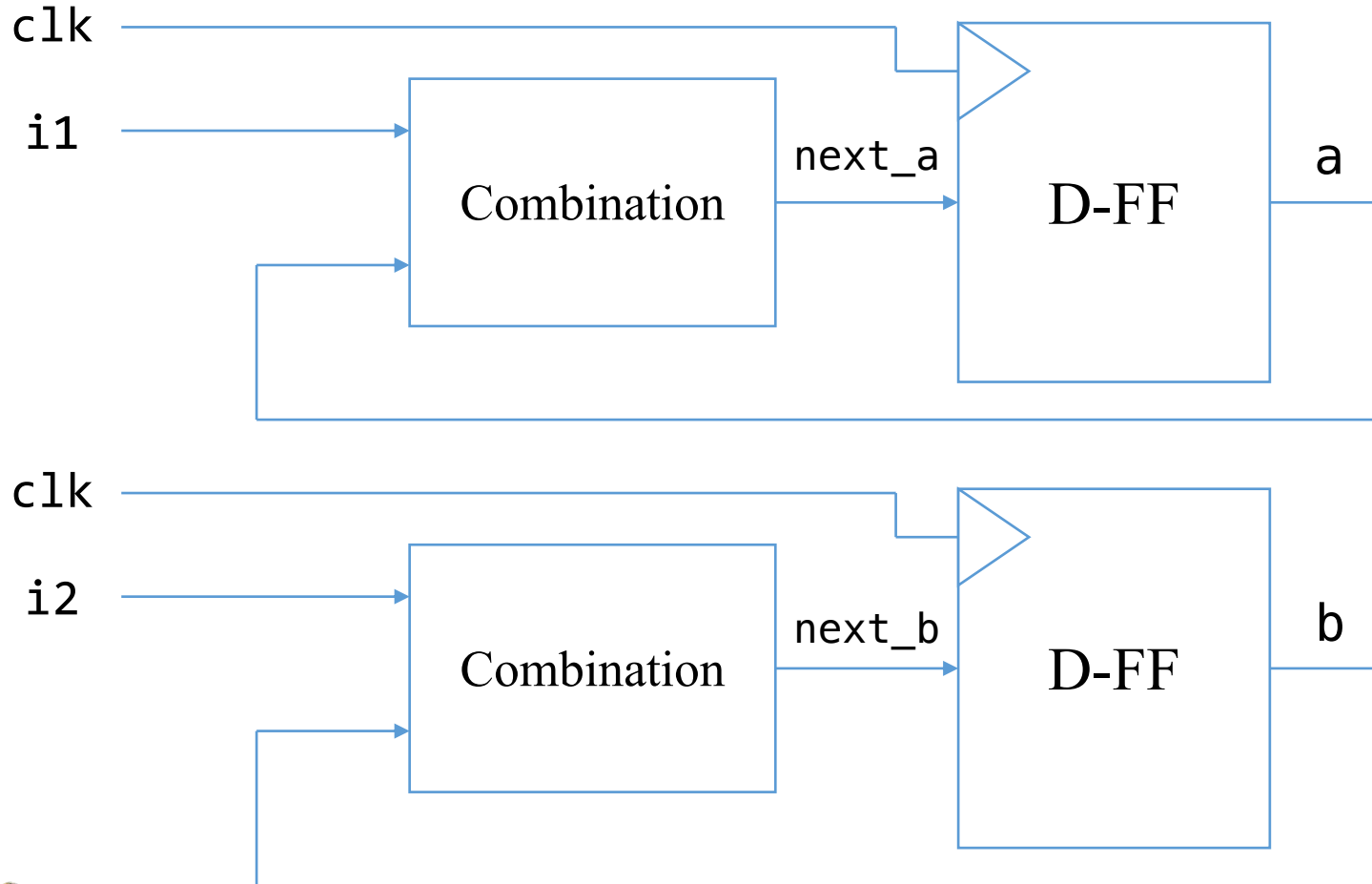


Unstable ,Oscillating Signal

Verilog – Attention !

- **Sequential circuit**

always @ (posedge clk) : **Always use non-blocking assignment (<=)**



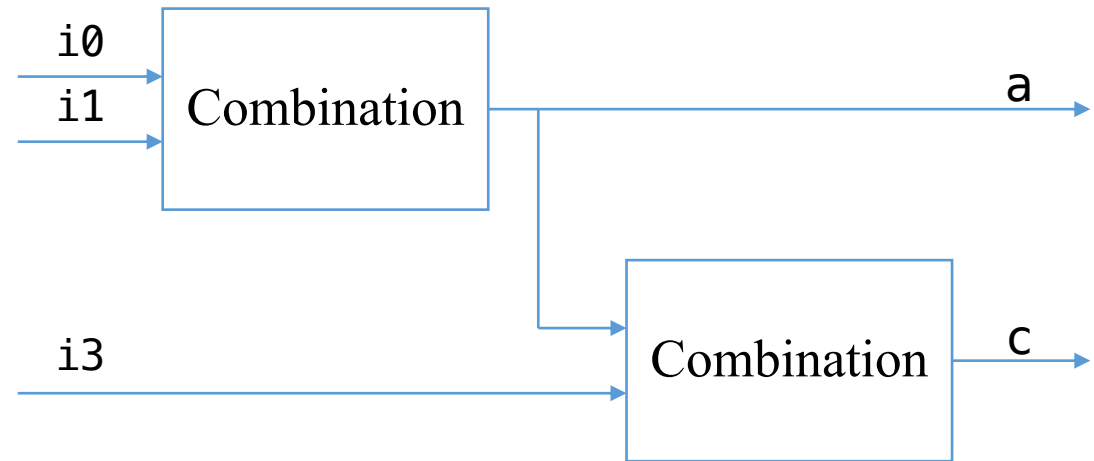
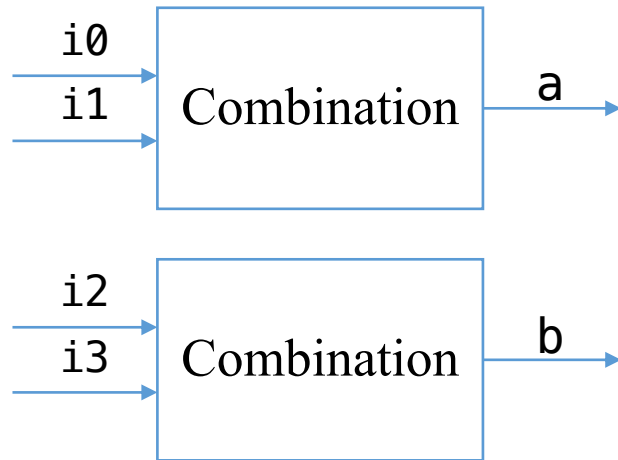
```
always@(posedge clk) begin
    if (rst) begin
        a <= 0;
        b <= 0;
    end
    else begin
        a <= next_a;
        b <= next_b;
    end
end
```

```
assign next_a = ... ;
assign next_b = ... ;
```

Verilog – Attention !

- **Combinational circuit**

always @ (*) : **Can use blocking (=) or non-blocking (<=) , but can't mix**



Don't mix

```
always @ (*) begin
    a <= i0 + i1;
    b <= i2 + i3;
end
```

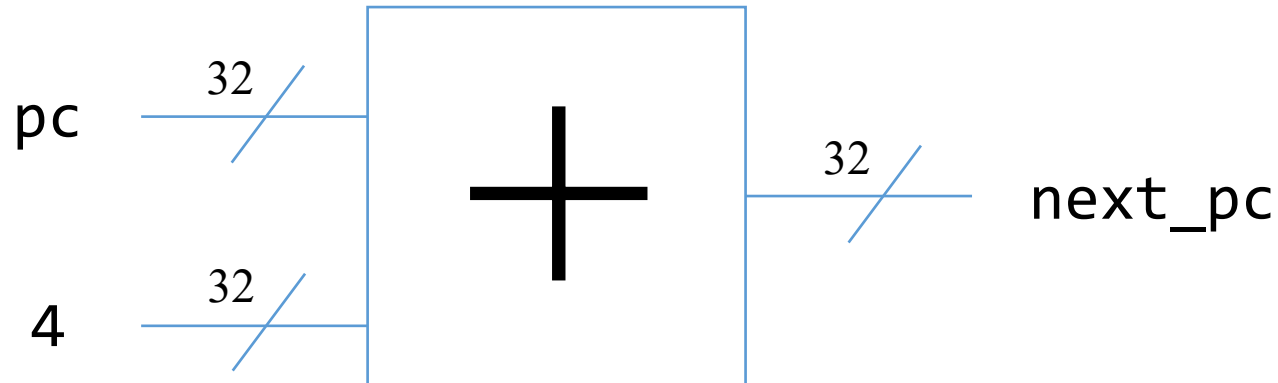
~~```
always @ (*) begin
 a <= i0 + i1;
 b <= i2 + i3;
 c = a + i3;
end
```~~

```
always @ (*) begin
 a = i0 + i1;
 c = a + i3;
end
```



# Verilog – Attention !

- Try to add “**bit length**” when write **constant**



```
assign next_pc = pc + 32'd4;
```

# Verilog – Attention !

- A signal can't be updated in different always block or continuous assignment

- **Sequential**

E.g.

- when `cs == `S1`  
    `a = 2`
- when `cs == `S2`  
    `a = 1`
- else  
    `a = 0`

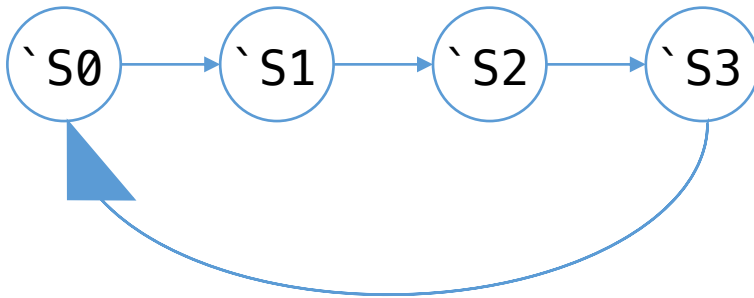
```
always@(posedge clk) begin
 if (cs == `S0)
 a <= 2;
 else if (cs == `S1)
 a <= 1;
 else
 a <= 0;
end
```

- **Combinational (Method 1)**

```
always@(*) begin
 if (cs == `S1)
 a = 2;
 else if (cs == `S2)
 a = 1;
 else
 a = 0;
end
```

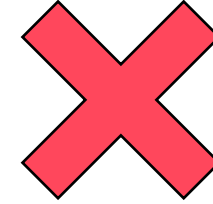
- **Combinational (Method 2)**

```
assign a = (cs == `S1) ? 2 :
 (cs == `S2) 1 : 0;
```



# Verilog – Attention !

- **Don't assign initial value when declaring it**
- **Combinational : No need to consider initial value**

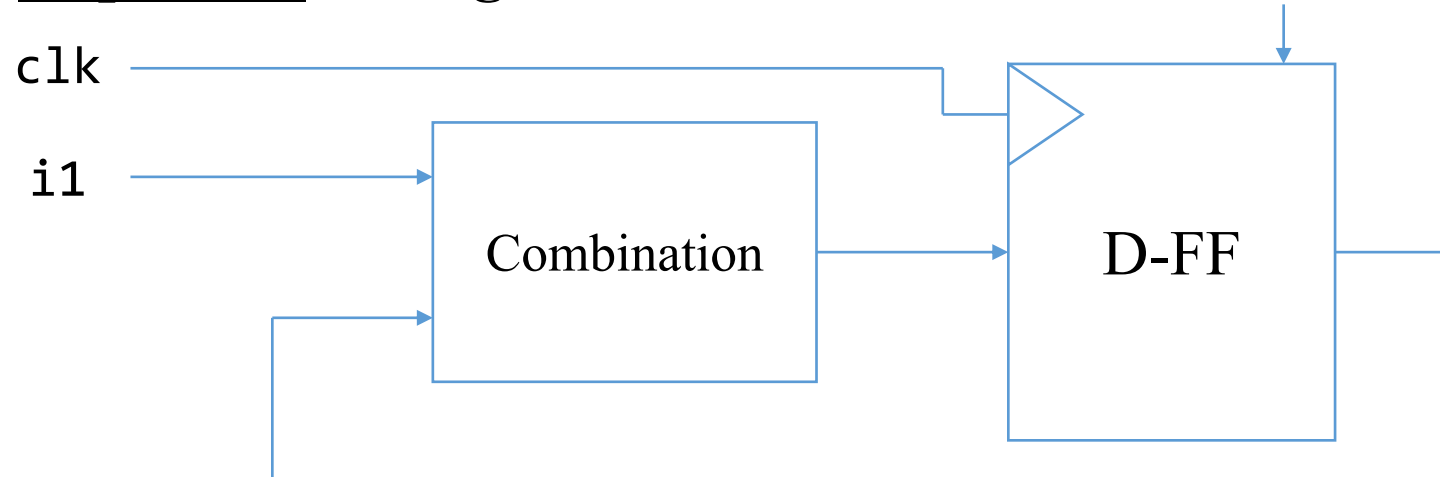


```
wire [3:0] a = 4'b1010;
reg [2:0] b = 3'b010;
```



output is always calculated by input

- **Sequential : Assign initial value when reset**



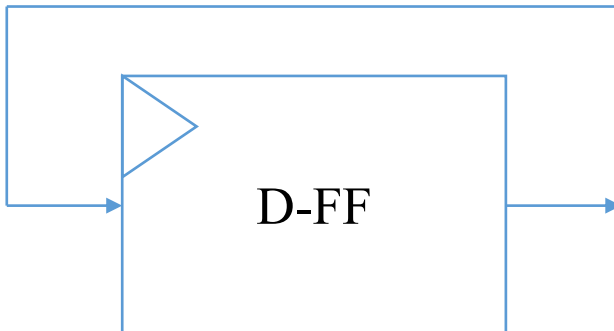
```
always@(posedge clk) begin
 if (rst)
 cs <= `S0;
 else
 cs <= ns;
end
```

# Verilog – Attention !

- In always block, it will **implicit execute (signal = signal)**, when there is no value assigned to the signal

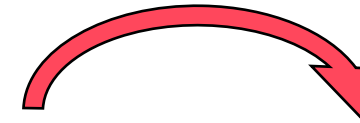
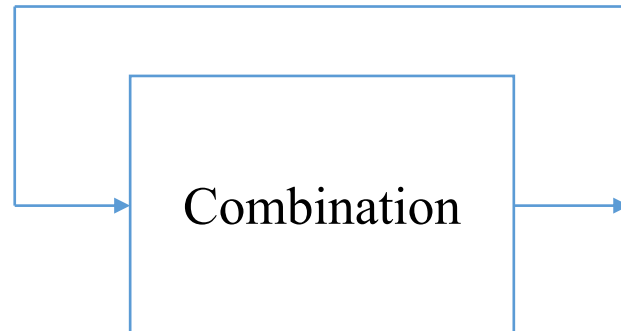
```
always@(posedge clk) begin
 if (rst)
 total <= 0;
 else if (cs == `S3)
 total <= 0;
end
```

**total <= total;**



```
always@(*) begin
 if (cs == `S1)
 total = input;
 else if (cs == `S3)
 total = 0;
end
```

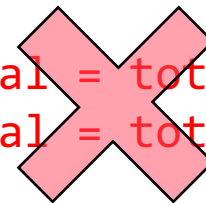
**total = total;**



**!!! Combinational need to fill up all situations !!!**

```
always@(*) begin
 if (cs == `S1)
 total = input;
 else if (cs == `S3)
 total = 0;
 else
 total = 0;
end
```

**total = total;**  
**total = total + 3;**

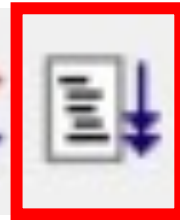


# ModelSim

Restart



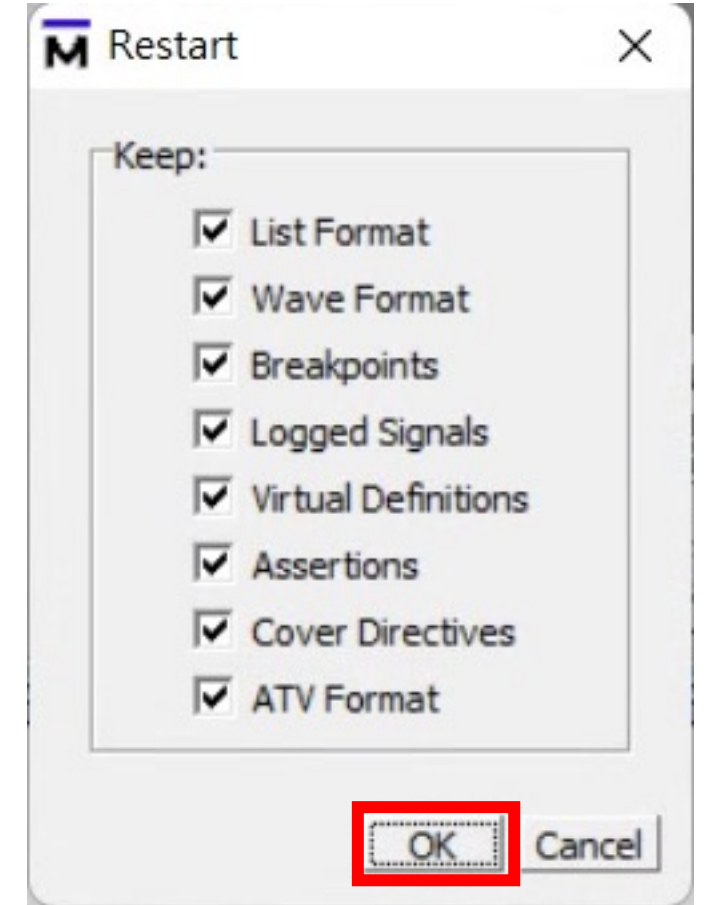
Run all



Stop

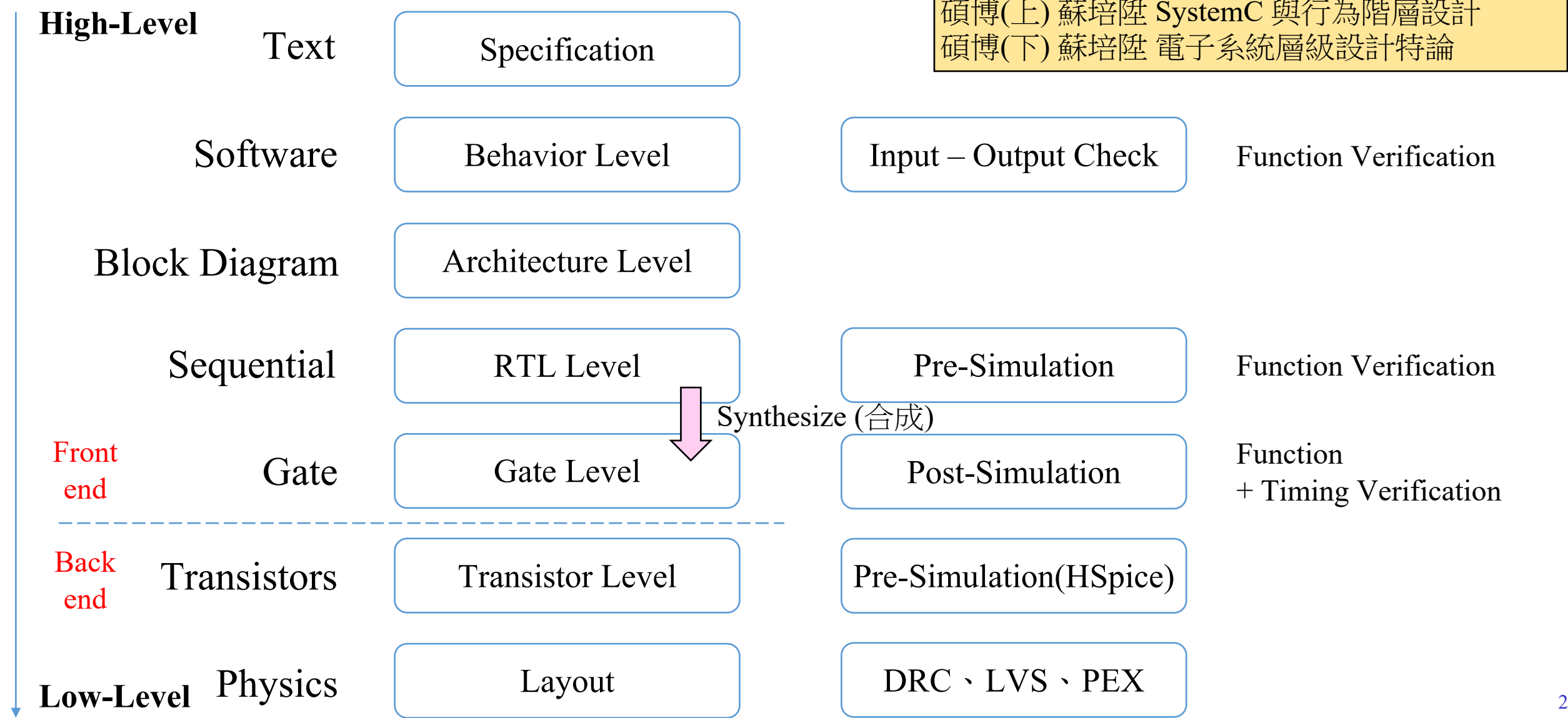


- If you add wave after run all, you need to restart first
- If you found the “run all” button was gray after you clicked it
  1. You forgot to click restart first
  2. The program was falling in infinite loop



# IC Design Level

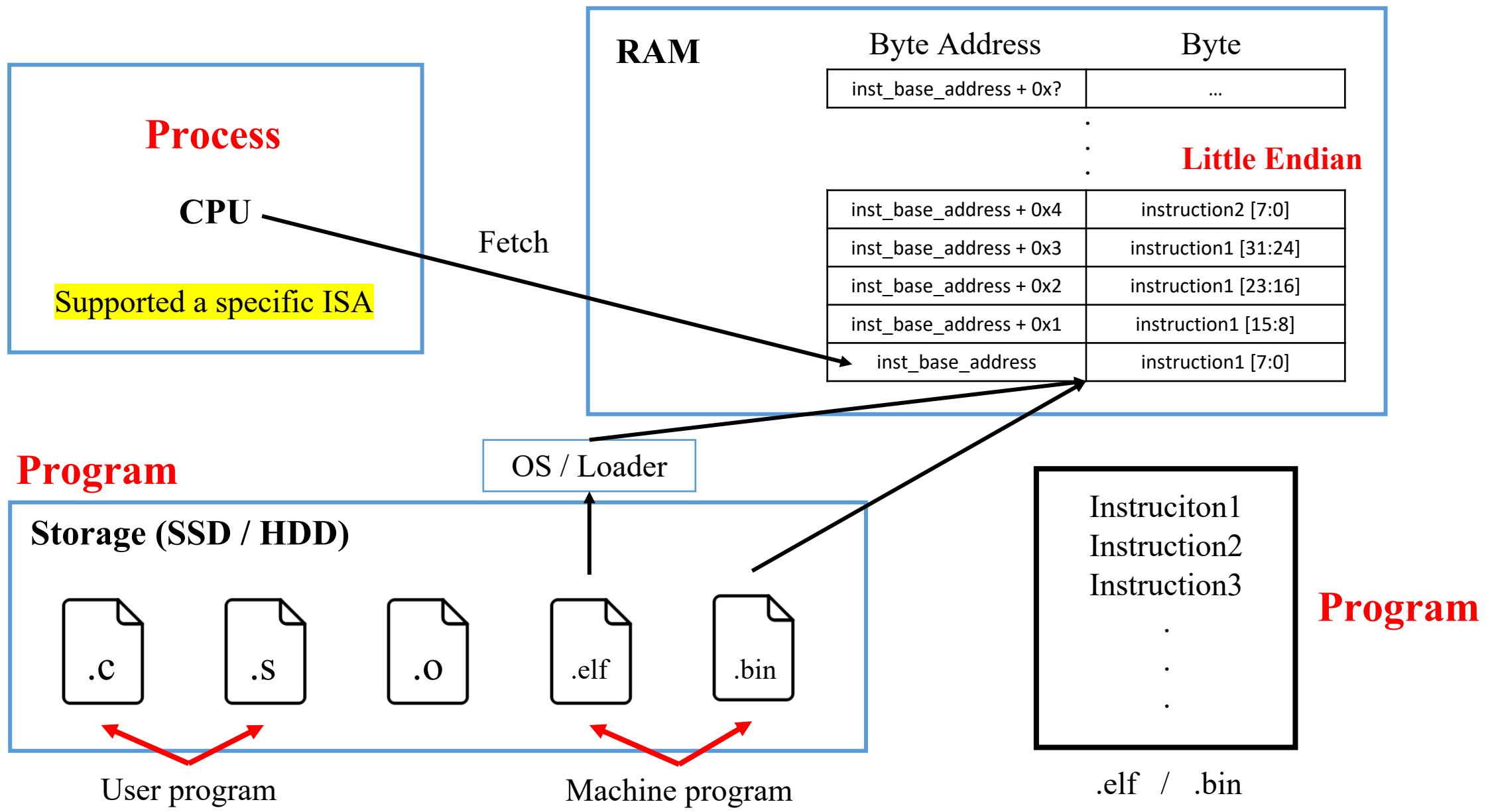
二下 邱瀝毅 超大型積體電路電腦輔助設計概論  
三下 張順志 VLSI 電路設計  
碩博(上) 蘇培陞 SystemC 與行為階層設計  
碩博(下) 蘇培陞 電子系統層級設計特論



# CPU Architecture Review

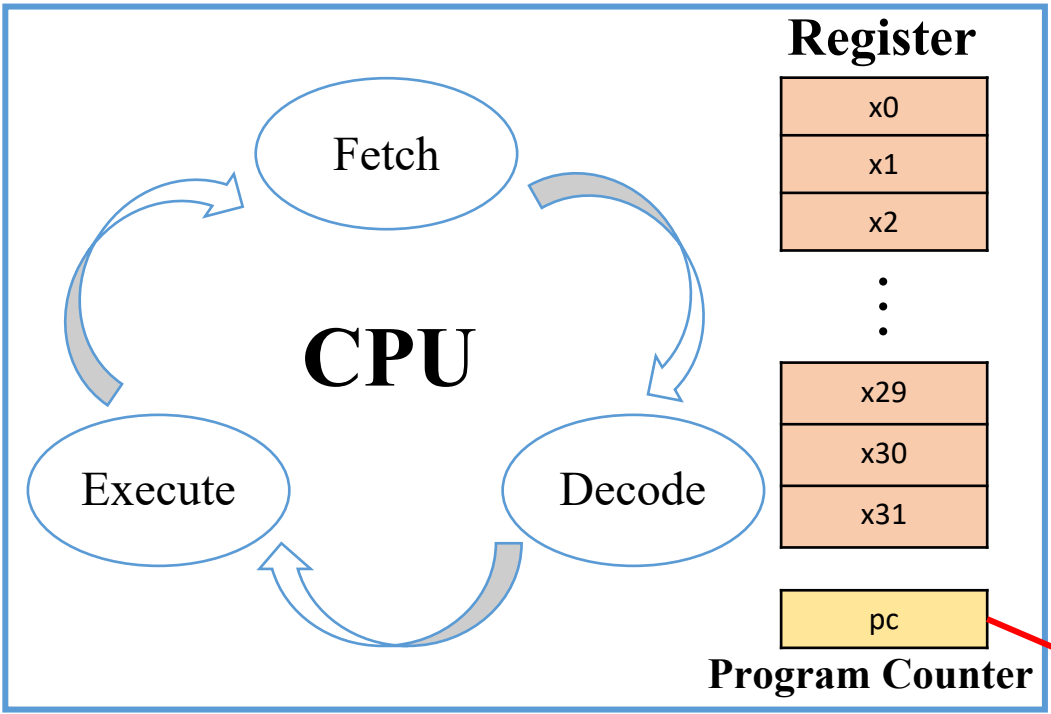


# How a CPU execute a program ?





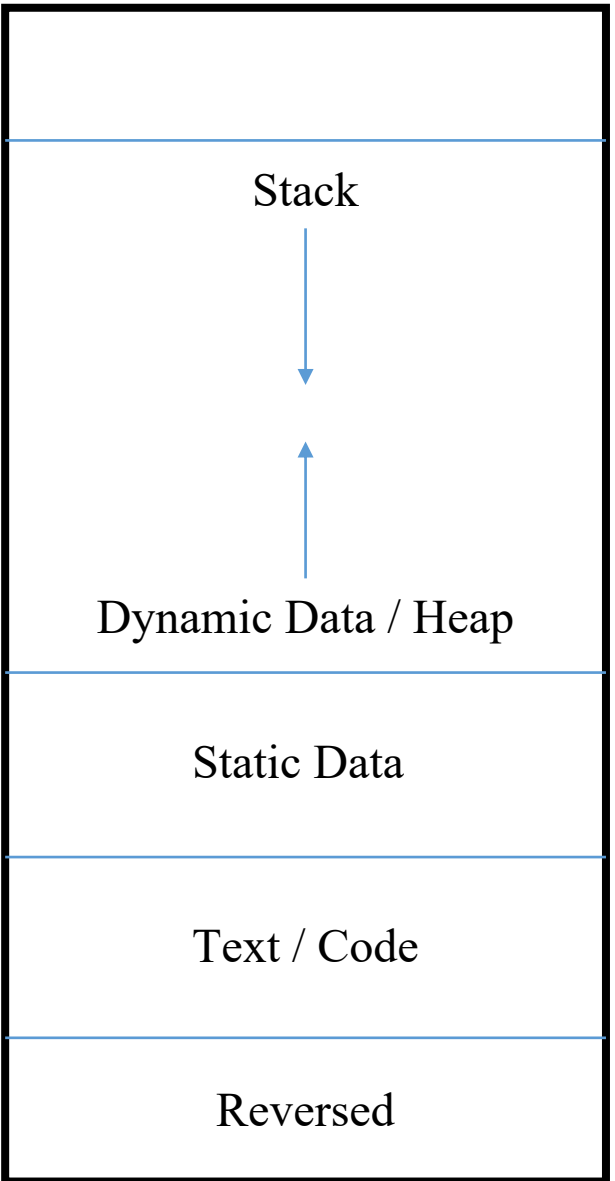
# Register & Main Memory



**Register is faster than RAM**

So, RISC need to load data from memory to registers before computing

**RAM**  
0xbffffff0



### 3 stages :

- **Fetch** : Fetch an **instruction that pc points to** from RAM
- **Decode** : Decode the fetched instruction to know what it mean
- **Execute** : Execute the instruction according the decode result

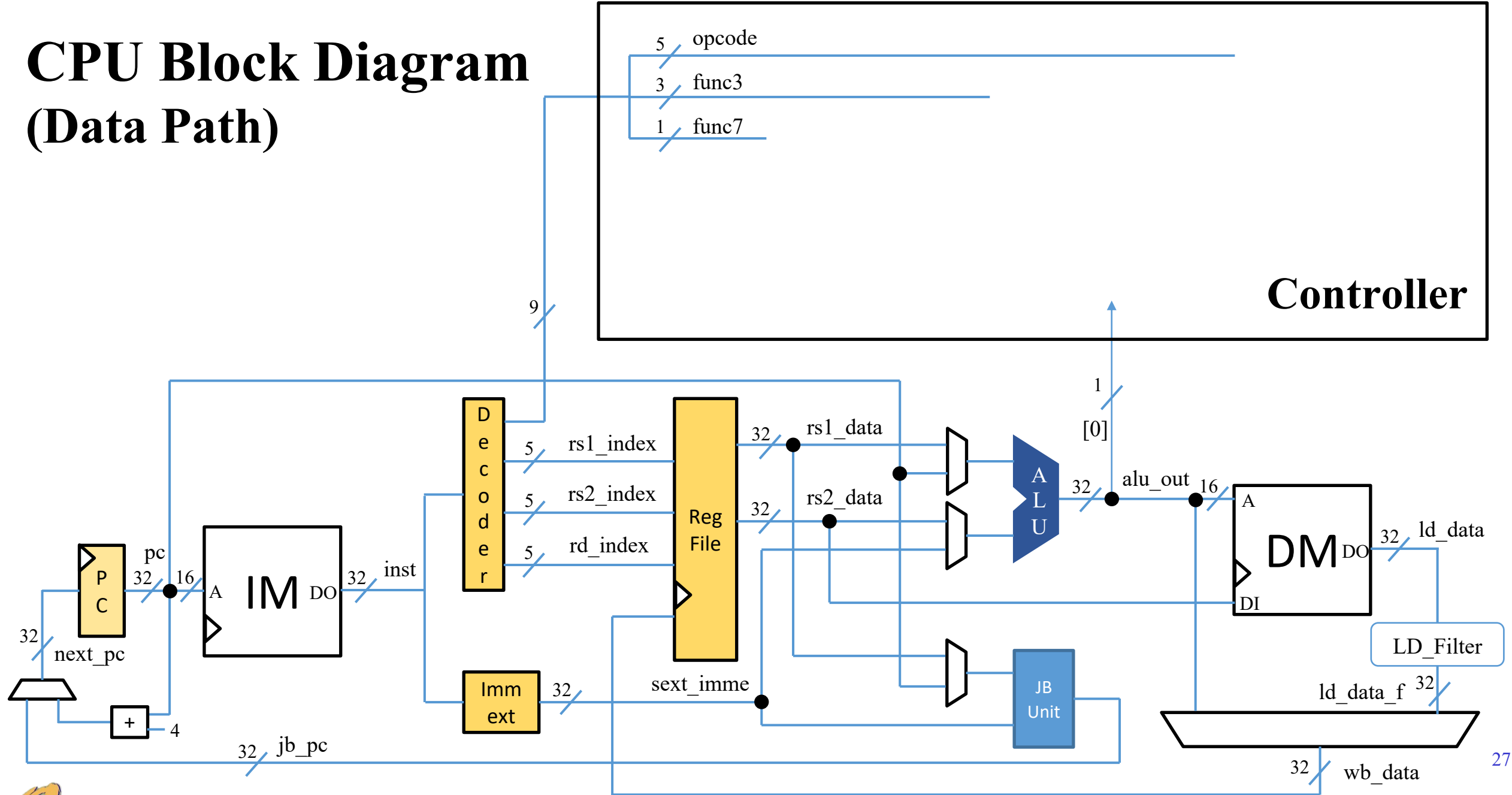
0x00010000  
0x00000000



# Example of CPU Micro-Architecture



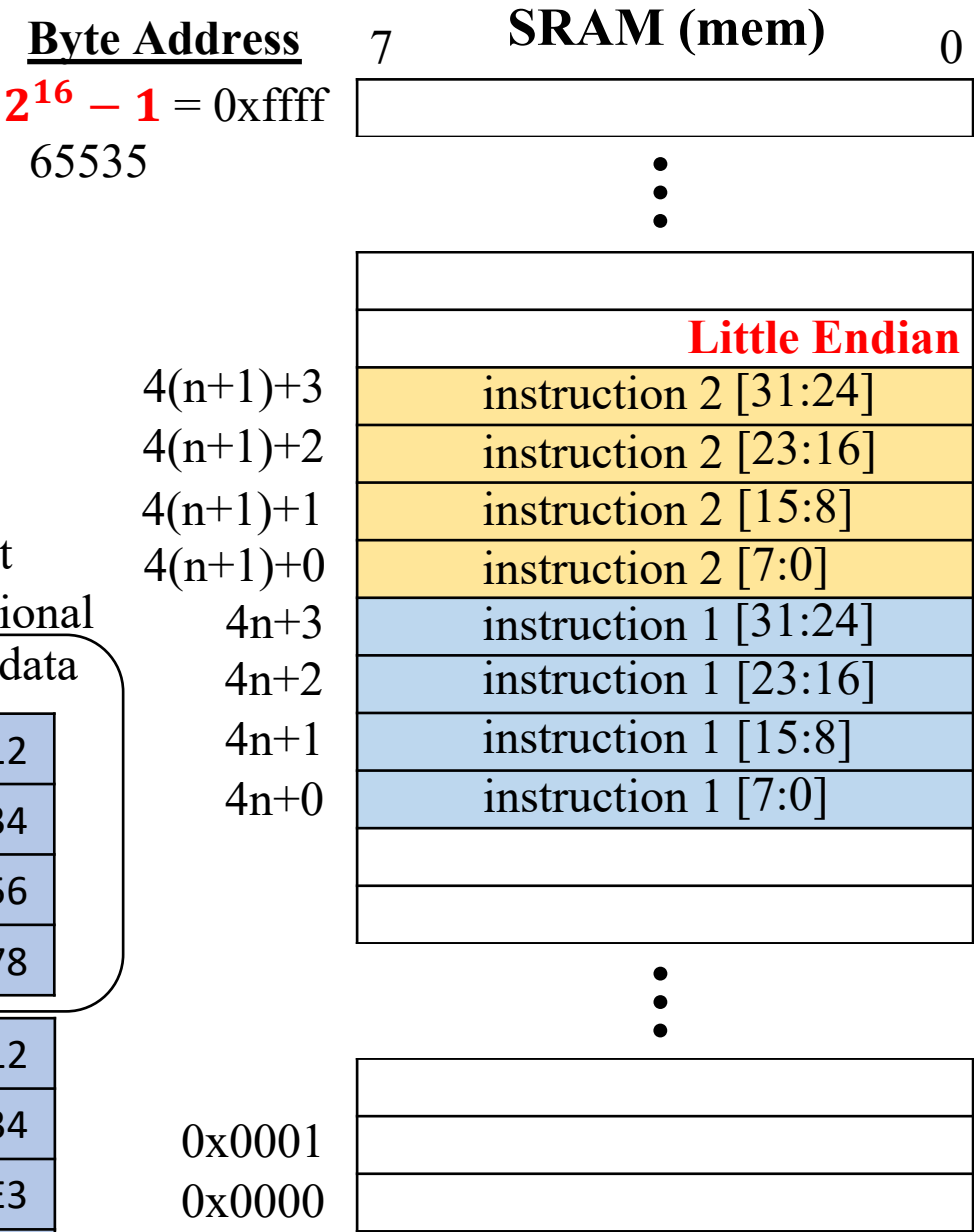
# CPU Block Diagram (Data Path)



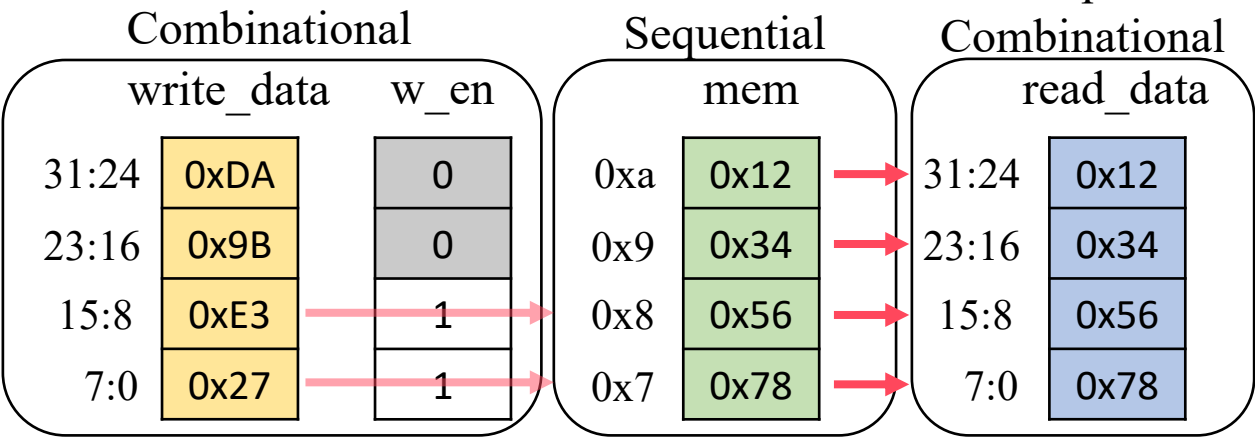
# Module – SRAM

```
module SRAM (
 input clk,
 input [3:0] w_en,
 input [15:0] address,
 input [31:0] write_data,
 output [31:0] read_data
);
 reg [7:0] mem [0:65535];
```

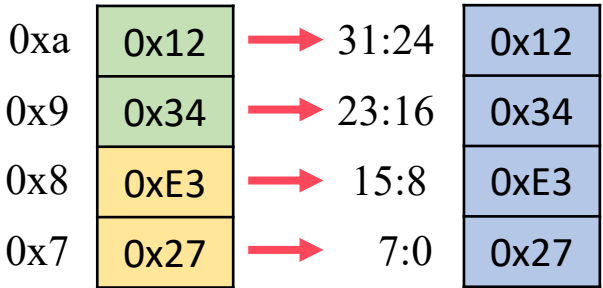
E.g.  
Suppose  
t0 = 0xDA9BE327  
t1 = 0x7  
sh t0, 0(t1)  
=> w\_en = 4'b0011



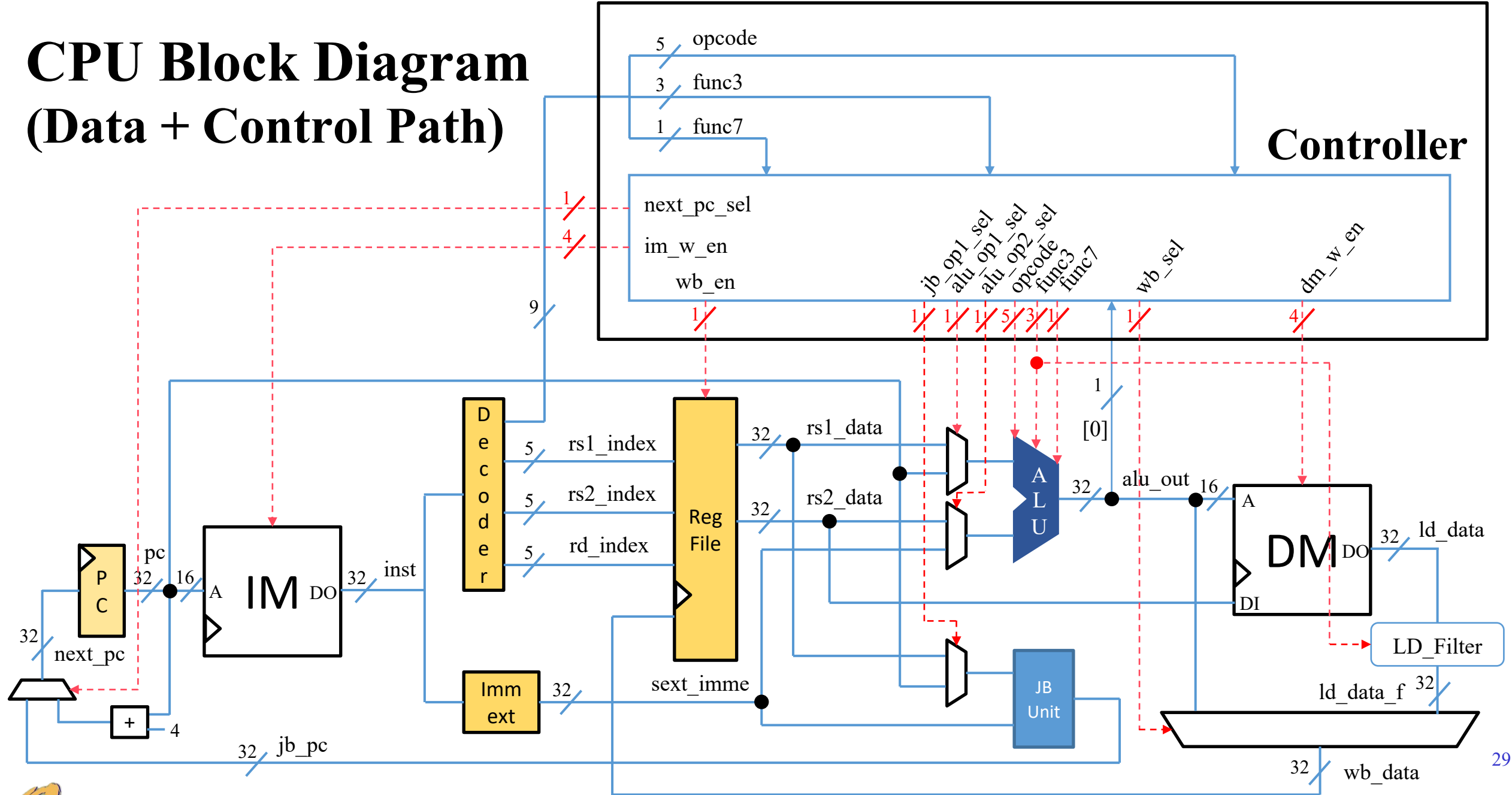
Cycle N



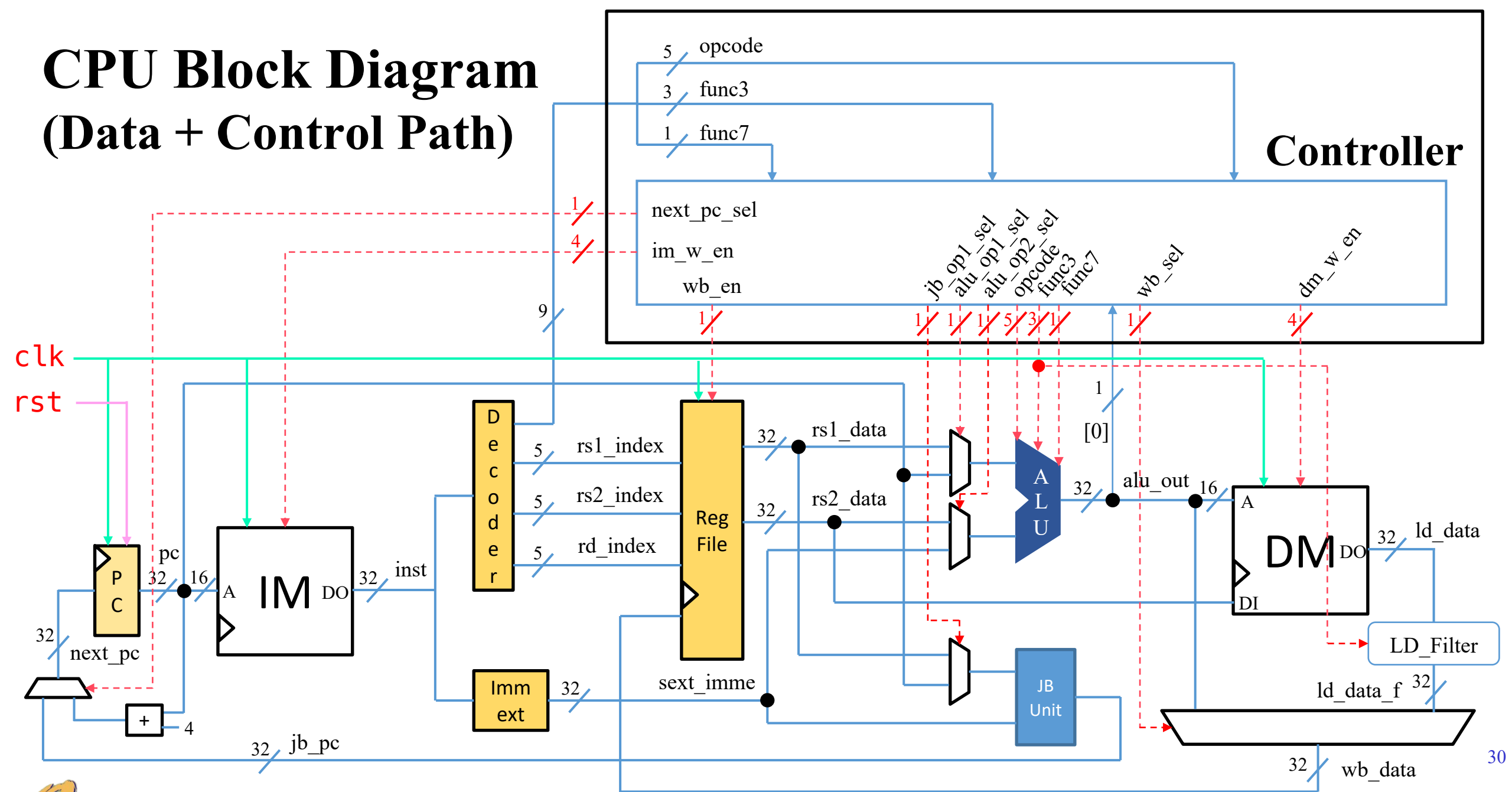
Cycle N+1



# CPU Block Diagram (Data + Control Path)



# CPU Block Diagram (Data + Control Path)

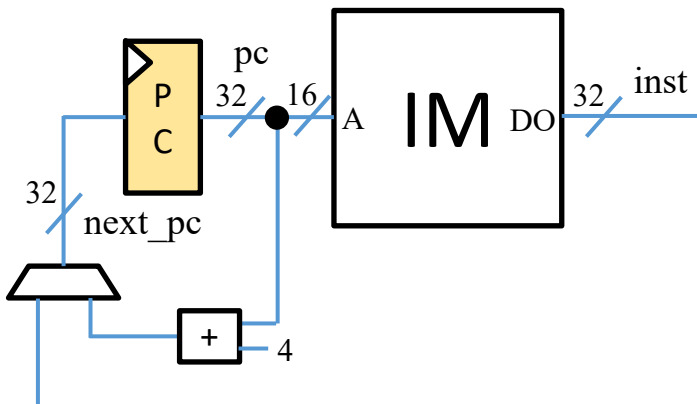


# CPU Block Diagram (Data Path)

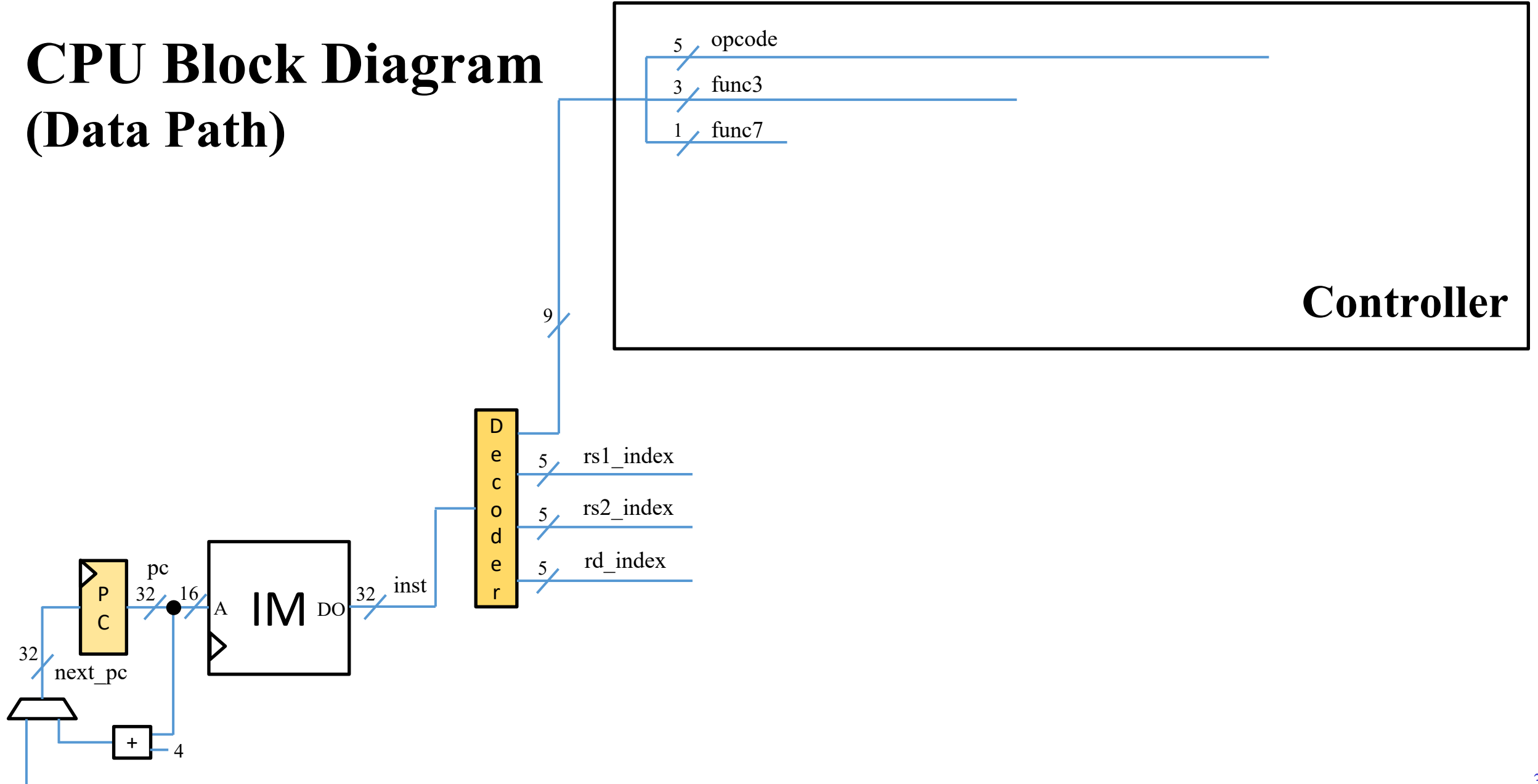
## Module – Reg\_PC

```
module Reg_PC (
 input clk,
 input rst,
 input [31:0] next_pc,
 output reg [31:0] current_pc
);
```

**current\_pc reset from 32'd0**



# CPU Block Diagram (Data Path)





# Module – Decoder

```
module Decoder (
 input [31:0] inst,
 output [4:0] dc_out_opcode,
 output [2:0] dc_out_func3,
 output dc_out_func7,
 output [4:0] dc_out_rs1_index,
 output [4:0] dc_out_rs2_index,
 output [4:0] dc_out_rd_index
);
```

**opcode** : the operation of the instruction  
**func3** : support opcode to express the operation  
**func7** : support opcode to express the operation  
**rs1** : source register 1  
**rs2** : source register 2  
**rd** : destination register

| Formats | 32 Bits (RV32I)                                 |    |    |    |    |    |    |     |    |    |    |    |     |    |    |    |    |       |    |    |                         |    |   |   |   |        |   |   |   |   |   |   |
|---------|-------------------------------------------------|----|----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|-------|----|----|-------------------------|----|---|---|---|--------|---|---|---|---|---|---|
|         | 31                                              | 30 | 29 | 28 | 27 | 26 | 25 | 24  | 23 | 22 | 21 | 20 | 19  | 18 | 17 | 16 | 15 | 14    | 13 | 12 | 11                      | 10 | 9 | 8 | 7 | 6      | 5 | 4 | 3 | 2 | 1 | 0 |
| R type  | func7                                           |    |    |    |    |    |    | rs2 |    |    |    |    | rs1 |    |    |    |    | func3 |    |    | rd                      |    |   |   |   | opcode |   |   |   |   |   |   |
| I type  | imme[11:0]                                      |    |    |    |    |    |    |     |    |    |    |    | rs1 |    |    |    |    | func3 |    |    | rd                      |    |   |   |   | opcode |   |   |   |   |   |   |
| S type  | imme[11:5]                                      |    |    |    |    |    |    | rs2 |    |    |    |    | rs1 |    |    |    |    | func3 |    |    | imme[4:0]               |    |   |   |   | opcode |   |   |   |   |   |   |
| B type  | { imme[12], imme[10:5] }                        |    |    |    |    |    |    | rs2 |    |    |    |    | rs1 |    |    |    |    | func3 |    |    | { imme[4:1], imme[11] } |    |   |   |   | opcode |   |   |   |   |   |   |
| U type  | imme[31:12]                                     |    |    |    |    |    |    |     |    |    |    |    |     |    |    |    |    |       |    |    | rd                      |    |   |   |   | opcode |   |   |   |   |   |   |
| J type  | { imme[20], imme[10:1], imme[11], imme[19:12] } |    |    |    |    |    |    |     |    |    |    |    |     |    |    |    |    |       |    |    | rd                      |    |   |   |   | opcode |   |   |   |   |   |   |



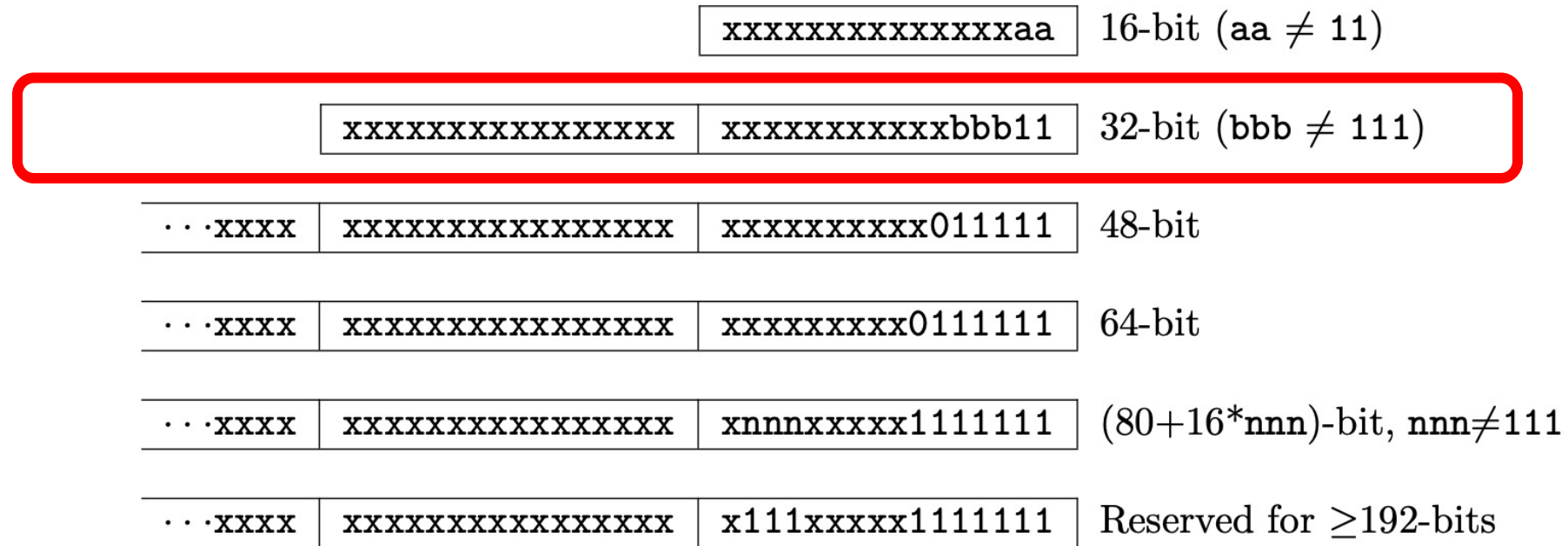
# RISC-V - Overview of RV32I Instructions

|                       |     |     |     |             |         |         |
|-----------------------|-----|-----|-----|-------------|---------|---------|
| imm[31:12]            |     |     |     | rd          | 0110111 | U lui   |
| imm[31:12]            |     |     |     | rd          | 0010111 | U auipc |
| imm[20 10:1 11 19:12] |     |     |     | rd          | 1101111 | J jal   |
| imm[11:0]             |     | rs1 | 000 | rd          | 1100111 | I jalr  |
| imm[12 10:5]          | rs2 | rs1 | 000 | imm[4:1 11] | 1100011 | B beq   |
| imm[12 10:5]          | rs2 | rs1 | 001 | imm[4:1 11] | 1100011 | B bne   |
| imm[12 10:5]          | rs2 | rs1 | 100 | imm[4:1 11] | 1100011 | B blt   |
| imm[12 10:5]          | rs2 | rs1 | 101 | imm[4:1 11] | 1100011 | B bge   |
| imm[12 10:5]          | rs2 | rs1 | 110 | imm[4:1 11] | 1100011 | B bltu  |
| imm[12 10:5]          | rs2 | rs1 | 111 | imm[4:1 11] | 1100011 | B bgeu  |
| imm[11:0]             |     | rs1 | 000 | rd          | 0000011 | I lb    |
| imm[11:0]             |     | rs1 | 001 | rd          | 0000011 | I lh    |
| imm[11:0]             |     | rs1 | 010 | rd          | 0000011 | I lw    |
| imm[11:0]             |     | rs1 | 100 | rd          | 0000011 | I lbu   |
| imm[11:0]             |     | rs1 | 101 | rd          | 0000011 | I lhu   |
| imm[11:5]             | rs2 | rs1 | 000 | imm[4:0]    | 0100011 | S sb    |
| imm[11:5]             | rs2 | rs1 | 001 | imm[4:0]    | 0100011 | S sh    |
| imm[11:5]             | rs2 | rs1 | 010 | imm[4:0]    | 0100011 | S sw    |

|           |       |     |     |    |         |         |
|-----------|-------|-----|-----|----|---------|---------|
| imm[11:0] |       | rs1 | 000 | rd | 0010011 | I addi  |
| imm[11:0] |       | rs1 | 010 | rd | 0010011 | I slti  |
| imm[11:0] |       | rs1 | 011 | rd | 0010011 | I sltiu |
| imm[11:0] |       | rs1 | 100 | rd | 0010011 | I xori  |
| imm[11:0] |       | rs1 | 110 | rd | 0010011 | I ori   |
| imm[11:0] |       | rs1 | 111 | rd | 0010011 | I andi  |
| 0000000   | shamt | rs1 | 001 | rd | 0010011 | I slli  |
| 0000000   | shamt | rs1 | 101 | rd | 0010011 | I srli  |
| 0100000   | shamt | rs1 | 101 | rd | 0010011 | I srai  |
| 0000000   | rs2   | rs1 | 000 | rd | 0110011 | R add   |
| 0100000   | rs2   | rs1 | 000 | rd | 0110011 | R sub   |
| 0000000   | rs2   | rs1 | 001 | rd | 0110011 | R sll   |
| 0000000   | rs2   | rs1 | 010 | rd | 0110011 | R slt   |
| 0000000   | rs2   | rs1 | 011 | rd | 0110011 | R sltu  |
| 0000000   | rs2   | rs1 | 100 | rd | 0110011 | R xor   |
| 0000000   | rs2   | rs1 | 101 | rd | 0110011 | R srl   |
| 0100000   | rs2   | rs1 | 101 | rd | 0110011 | R sra   |
| 0000000   | rs2   | rs1 | 110 | rd | 0110011 | R or    |
| 0000000   | rs2   | rs1 | 111 | rd | 0110011 | R and   |

# Decode – Ignore last 2 bits

- Instruction Length of Base ISA are all 32 bits
- If Including Compressed Extension ISA Instruction Length can be 16 bits or 32 bits
- Decoder will use the last n bits to determine which length it is
- **In this lab, we only consider RV32I (base ISA), so the last 2 bits are always 11**

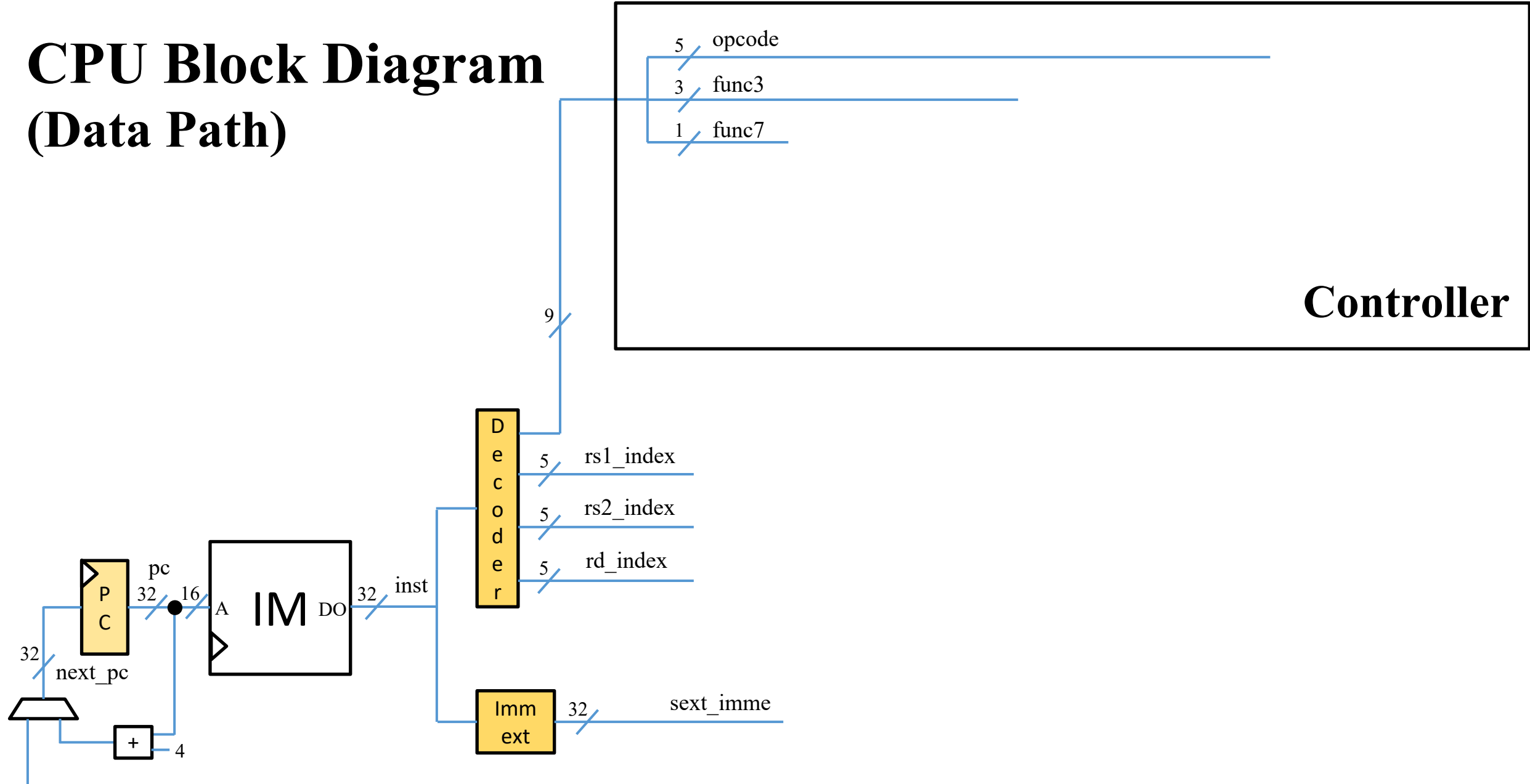


Byte Address: base+4

base+2

base

# CPU Block Diagram (Data Path)



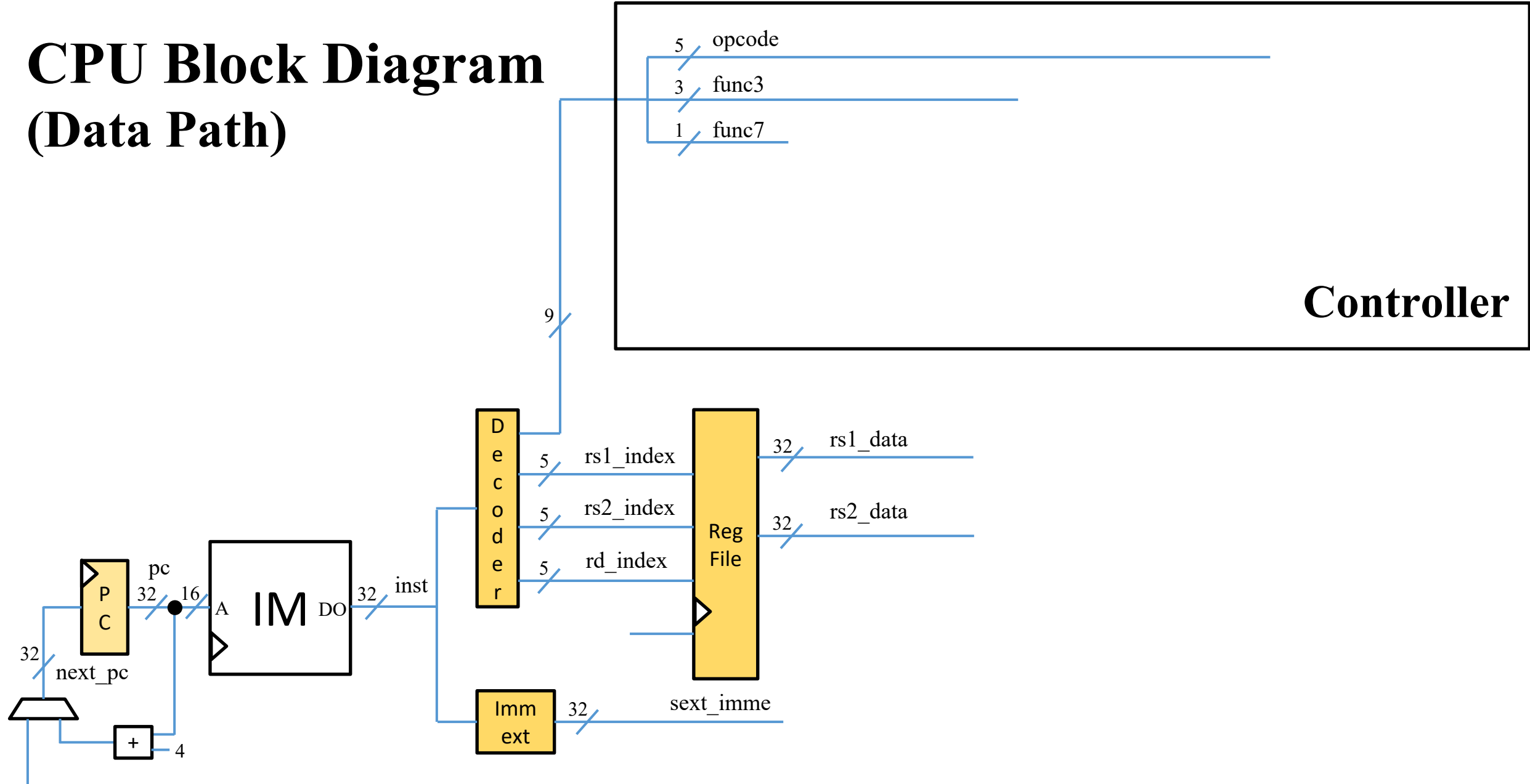
# Module – Imm\_Ext

```
module Imm_Ext (
 input [31:0] inst,
 output [31:0] imm_ext_out
);
```

| Type   | imm_ext_out                                                                                   |
|--------|-----------------------------------------------------------------------------------------------|
| R type | –                                                                                             |
| I type | {{20{ <i>inst</i> [31]}}, <i>inst</i> [31: 20]}                                               |
| S type | {{20{ <i>inst</i> [31]}}, <i>inst</i> [31: 25], <i>inst</i> [11: 7]}                          |
| B type | {{20{ <i>inst</i> [31]}}, <i>inst</i> [7], <i>inst</i> [30: 25], <i>inst</i> [11: 8], 1'b0}   |
| U type | { <i>inst</i> [31: 12], 12'b0}                                                                |
| J type | {{12{ <i>inst</i> [31]}}, <i>inst</i> [19: 12], <i>inst</i> [20], <i>inst</i> [30: 21], 1'b0} |

| Formats | 32 Bits (RV32I)                                 |    |    |    |    |    |    |     |    |    |    |    |     |    |    |    |    |       |    |    |                         |    |   |   |   |        |   |   |   |   |   |   |  |
|---------|-------------------------------------------------|----|----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|-------|----|----|-------------------------|----|---|---|---|--------|---|---|---|---|---|---|--|
|         | 31                                              | 30 | 29 | 28 | 27 | 26 | 25 | 24  | 23 | 22 | 21 | 20 | 19  | 18 | 17 | 16 | 15 | 14    | 13 | 12 | 11                      | 10 | 9 | 8 | 7 | 6      | 5 | 4 | 3 | 2 | 1 | 0 |  |
| R type  | func7                                           |    |    |    |    |    |    | rs2 |    |    |    |    | rs1 |    |    |    |    | func3 |    |    | rd                      |    |   |   |   | opcode |   |   |   |   |   |   |  |
| I type  | imme[11:0]                                      |    |    |    |    |    |    |     |    |    |    |    | rs1 |    |    |    |    | func3 |    |    | rd                      |    |   |   |   | opcode |   |   |   |   |   |   |  |
| S type  | imme[11:5]                                      |    |    |    |    |    |    | rs2 |    |    |    |    | rs1 |    |    |    |    | func3 |    |    | imme[4:0]               |    |   |   |   | opcode |   |   |   |   |   |   |  |
| B type  | { imme[12], imme[10:5] }                        |    |    |    |    |    |    | rs2 |    |    |    |    | rs1 |    |    |    |    | func3 |    |    | { imme[4:1], imme[11] } |    |   |   |   | opcode |   |   |   |   |   |   |  |
| U type  | imme[31:12]                                     |    |    |    |    |    |    |     |    |    |    |    |     |    |    |    |    |       |    |    | rd                      |    |   |   |   | opcode |   |   |   |   |   |   |  |
| J type  | { imme[20], imme[10:1], imme[11], imme[19:12] } |    |    |    |    |    |    |     |    |    |    |    |     |    |    |    |    |       |    |    | rd                      |    |   |   |   | opcode |   |   |   |   |   |   |  |

# CPU Block Diagram (Data Path)

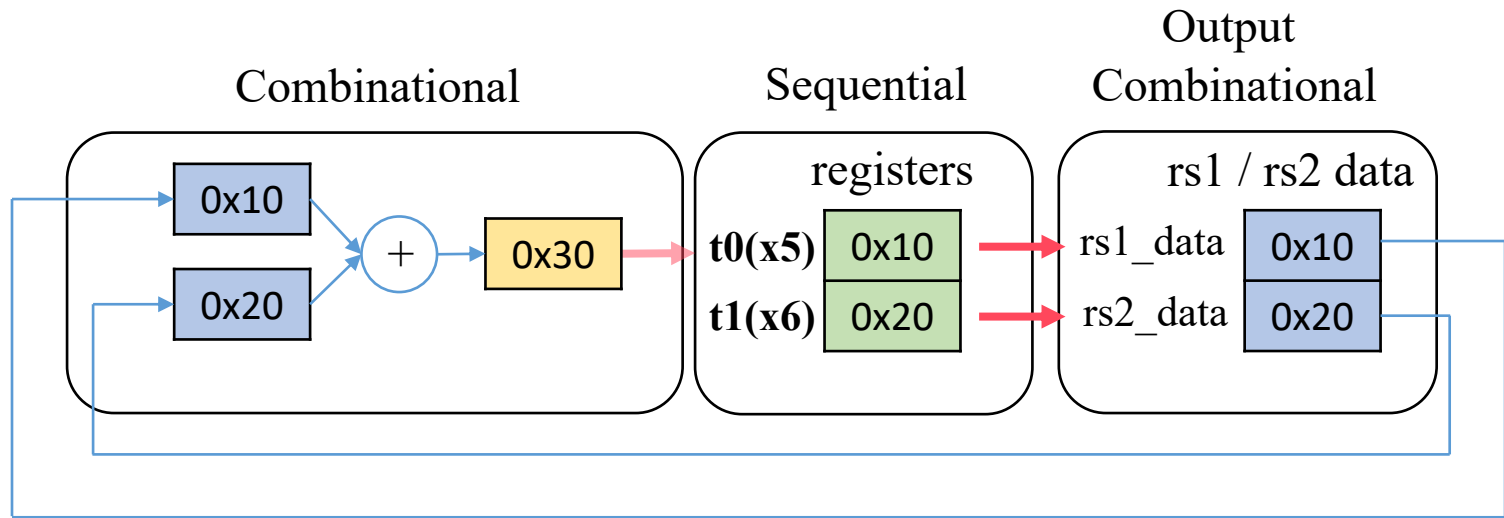


# Module – RegFile

```
module RegFile (
 input clk,
 input wb_en,
 input [31:0] wb_data,
 input [4:0] rd_index,
 input [4:0] rs1_index,
 input [4:0] rs2_index,
 output [31:0] rs1_data_out,
 output [31:0] rs2_data_out
);
 reg [31:0] registers [0:31];
```

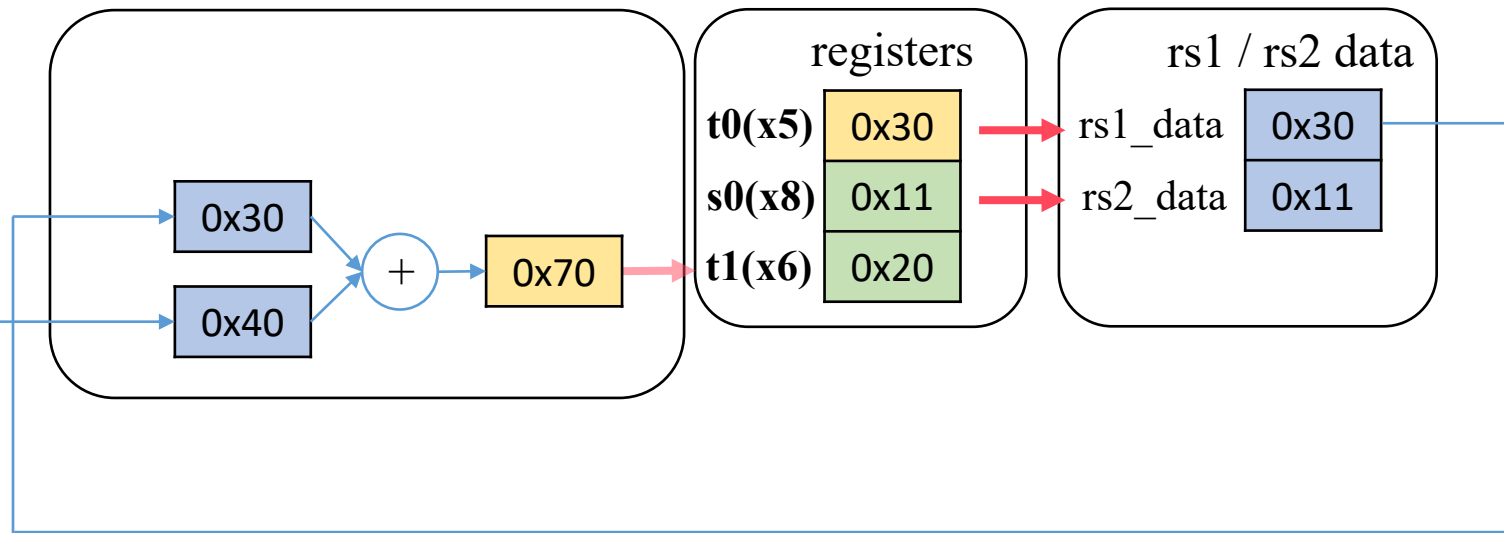
Cycle N

**Attention !!! If rd is x0, it can't write !!!**



Cycle N+1

sext(imm)

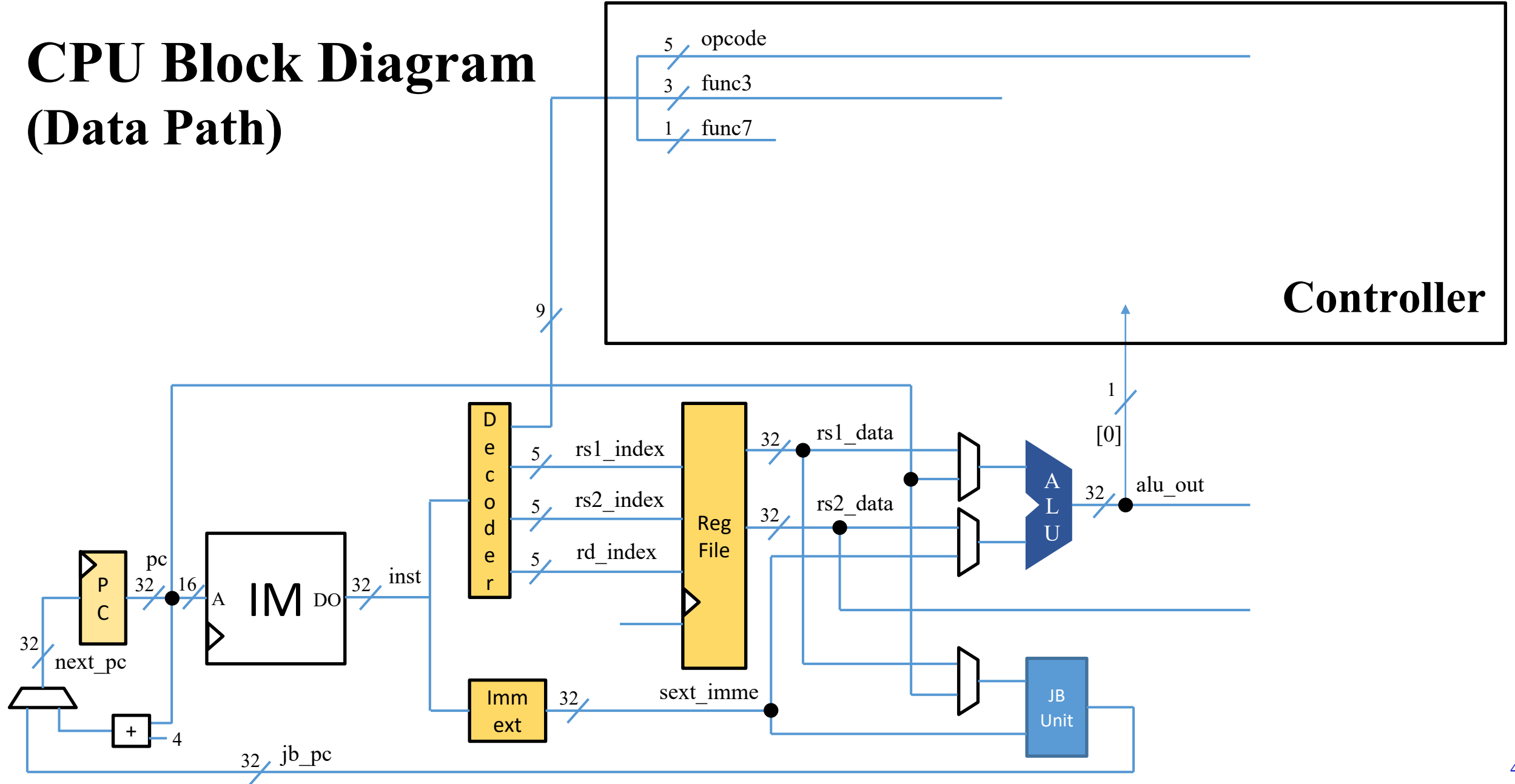


E.g.  
Suppose  
t0 = 0x10  
t1 = 0x20  
s0 = 0x11  
add t0, t0, t1 => wb\_en = 1'b1  
addi t1, t0, 40 => wb\_en = 1'b1

addi t1, t0, 40 : 0000001**01000** 00101 000 00110 0010011



# CPU Block Diagram (Data Path)





# Instruction classification of RV32I depends on functionality

- Computation Instruction

- Register – Register
- Register – Immediate
- Long Immediate

Arithmetic      Set      Shift      Logical

(add, **sub**, **slt**, **sltu**, sll, srl, sra, **xor**, or, and)  
(addi, **slti**, **sltiu**, slli, srli, srai, **xori**, ori, andi)  
(lui, auipc)

- Load & Store Instruction

- Load
- Store

(lb, lh, lw, lbu, lhu)  
(sb, sh, sw)

- Control Transfer Instruction

- Unconditional Jump
- Conditional Branch

(jal, jalr)  
(beq, bne, blt, bge, bltu, bgeu)



# Instruction classification of RV32I depends on functionality

|                                     | ALU operation                     | alu_op1 | alu_op2 | JB_Unit operation      | jb_unit_op1 | jb_unit_op2 |
|-------------------------------------|-----------------------------------|---------|---------|------------------------|-------------|-------------|
| <b>Computation Instruction</b>      |                                   |         |         |                        |             |             |
| Register - Register                 | rs1 interacts with rs2            | rs1     | rs2     | –                      | –           | –           |
| Register - Immediate                | rs1 interacts with imm            | rs1     | imm     | –                      | –           | –           |
| LUI                                 | imm                               | –       | imm     | –                      | –           | –           |
| AUIPC                               | pc + imm                          | pc      | imm     | –                      | –           | –           |
| <b>Load &amp; Store Instruction</b> |                                   |         |         |                        |             |             |
| Load                                | rs1 + imm<br>=> calculate address | rs1     | imm     | –                      | –           | –           |
| Store                               | rs1 + imm<br>=> calculate address | rs1     | imm     | –                      | –           | –           |
| <b>Control Transfer Instruction</b> |                                   |         |         |                        |             |             |
| JAL                                 | pc + 4                            | pc      | –       | pc + imm               | pc          | imm         |
| JALR                                | pc + 4                            | pc      | –       | (rs1 + imm) & (~32'd1) | rs1         | imm         |
| Branch                              | rs1 compares with rs2             | rs1     | rs2     | pc + imm               | pc          | imm         |

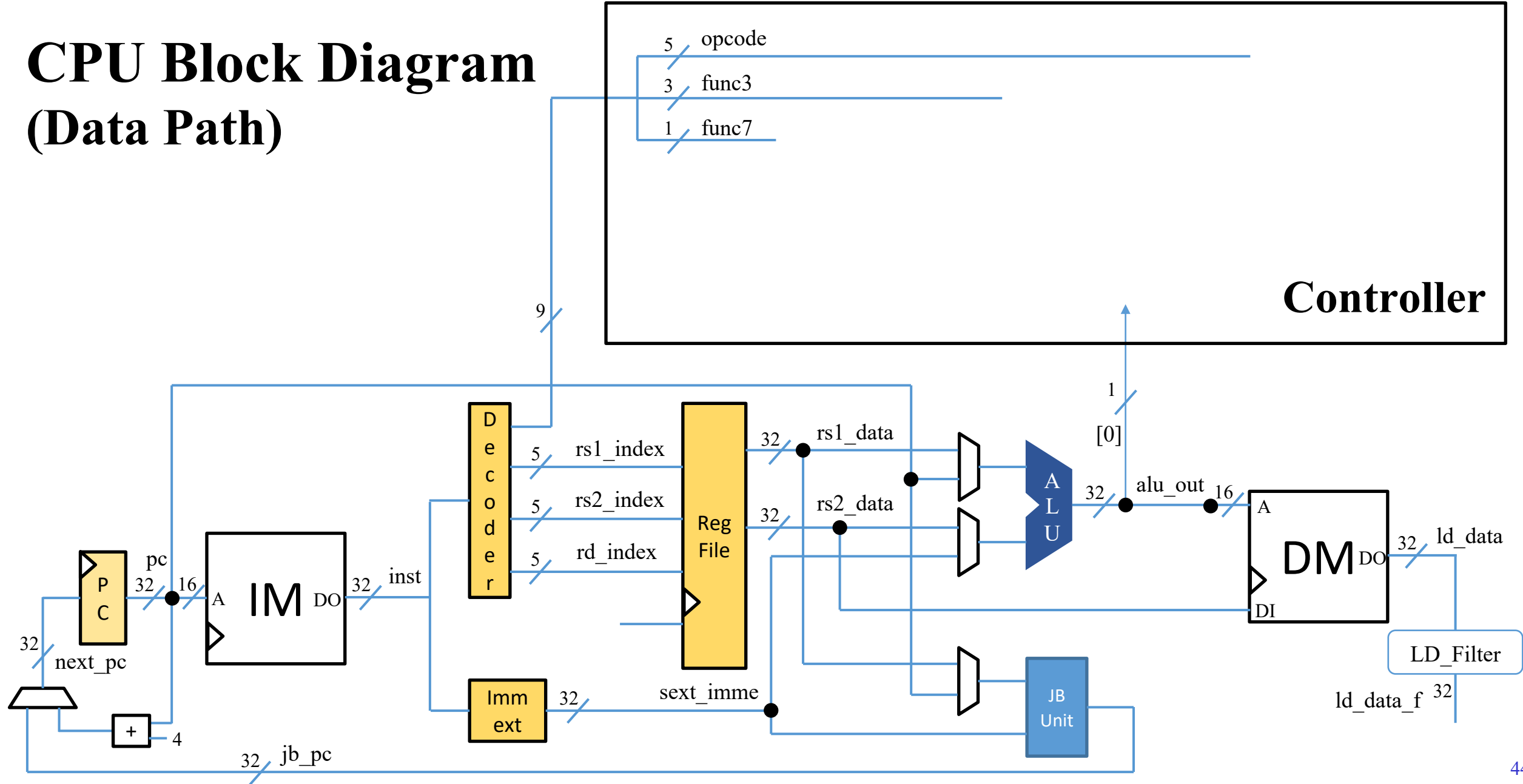
# Module – ALU / JB\_Unit

```
module ALU (
 input [4:0] opcode,
 input [2:0] func3,
 input func7,
 input [31:0] operand1,
 input [31:0] operand2,
 output [31:0] alu_out
);
```

```
module JB_Unit (
 input [31:0] operand1,
 input [31:0] operand2,
 output [31:0] jb_out
);
```

|                                     | ALU operation                          | JB_Unit operation             |
|-------------------------------------|----------------------------------------|-------------------------------|
| <b>Computation Instruction</b>      |                                        |                               |
| <b>Register - Register</b>          | $op1 \xleftrightarrow{(f3,f7)} op2$    | –                             |
| <b>Register - Immediate</b>         | $op1 \xleftrightarrow{(f3,f7)} op2$    | –                             |
| <b>LUI</b>                          | $op2$                                  | –                             |
| <b>AUIPC</b>                        | $op1 + op2$                            | –                             |
| <b>Load &amp; Store Instruction</b> |                                        |                               |
| <b>Load</b>                         | $op1 + op2 \Rightarrow \text{address}$ | –                             |
| <b>Store</b>                        | $op1 + op2 \Rightarrow \text{address}$ | –                             |
| <b>Control Transfer Instruction</b> |                                        |                               |
| <b>JAL</b>                          | $op1 + 4 \Rightarrow \text{pc} + 4$    | $op1 + op2$                   |
| <b>JALR</b>                         | $op1 + 4 \Rightarrow \text{pc} + 4$    | $(op1 + op2) \& (\sim 32'd1)$ |
| <b>Branch</b>                       | $op1 \xleftrightarrow{(f3)} op2$       | $op1 + op2$                   |

# CPU Block Diagram (Data Path)



# Module – LD\_Filter

```

module LD_Filter (
 input [2:0] func3,
 input [31:0] ld_data,
 output [31:0] ld_data_f
);

```

func3

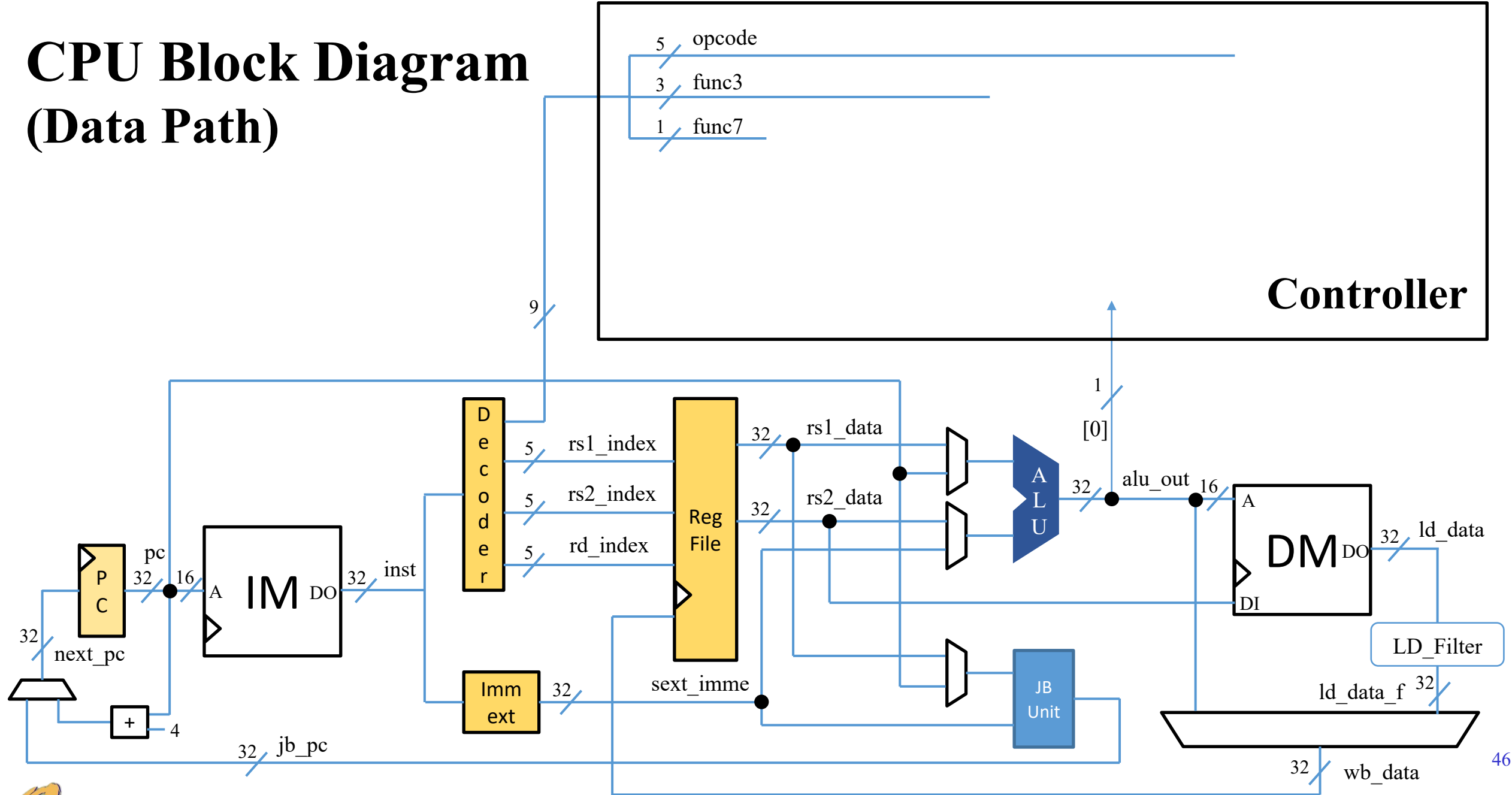
|           |     |     |    |         |       |
|-----------|-----|-----|----|---------|-------|
| imm[11:0] | rs1 | 000 | rd | 0000011 | I lb  |
| imm[11:0] | rs1 | 001 | rd | 0000011 | I lh  |
| imm[11:0] | rs1 | 010 | rd | 0000011 | I lw  |
| imm[11:0] | rs1 | 100 | rd | 0000011 | I lbu |
| imm[11:0] | rs1 | 101 | rd | 0000011 | I lhu |

E.g.

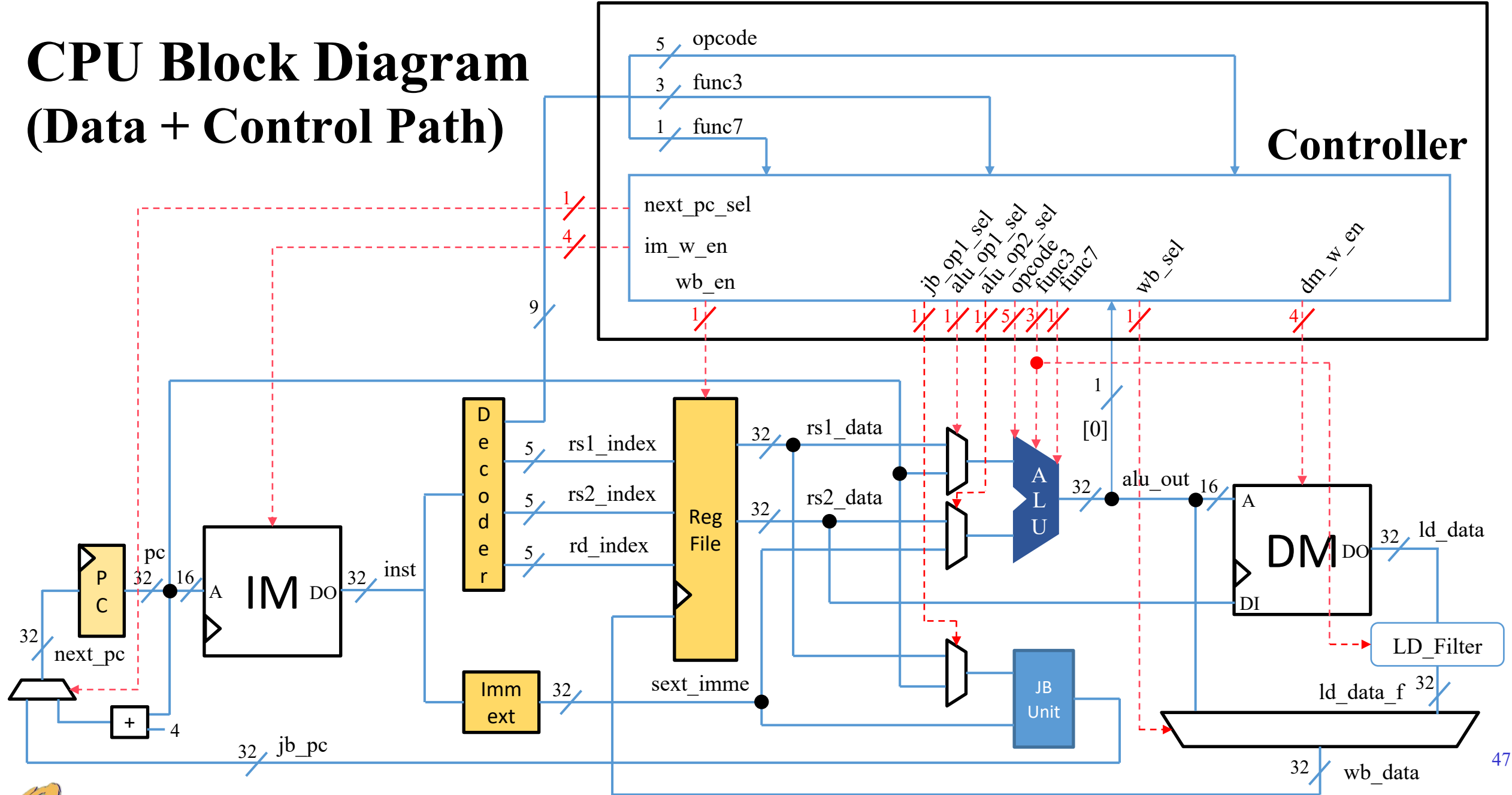
|         | 32 Bits |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |          |   |   |   |   |   |   |   |
|---------|---------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|----------|---|---|---|---|---|---|---|
|         | 31      | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7        | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ld_data | 1       | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1 | 0 | 1        | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| lb      | 1       | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1 | 1 | <u>1</u> | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| lbu     | 0       | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 1        | 0 | 1 | 0 | 1 | 0 | 1 | 0 |



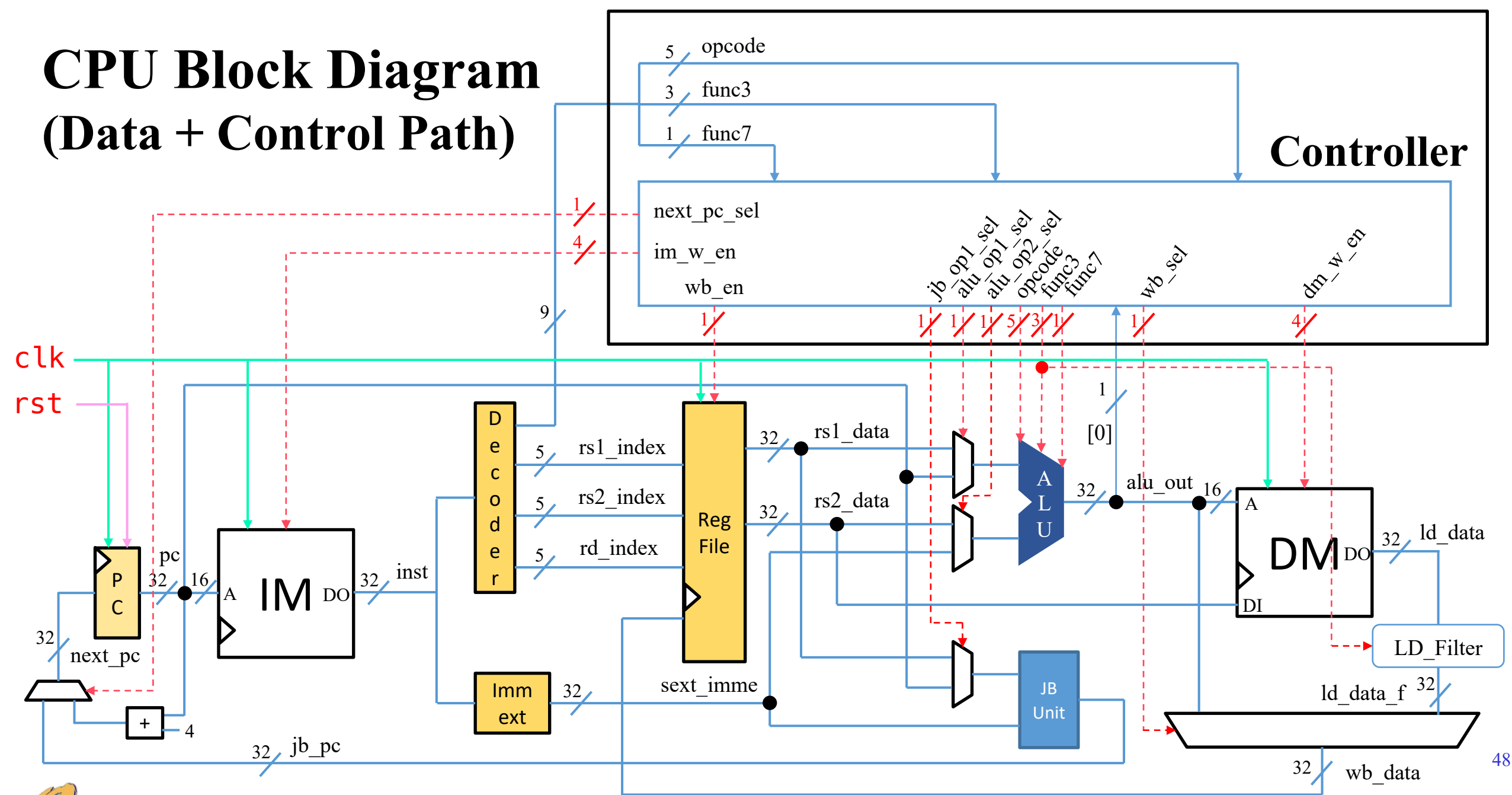
# CPU Block Diagram (Data Path)



# CPU Block Diagram (Data + Control Path)



# CPU Block Diagram (Data + Control Path)



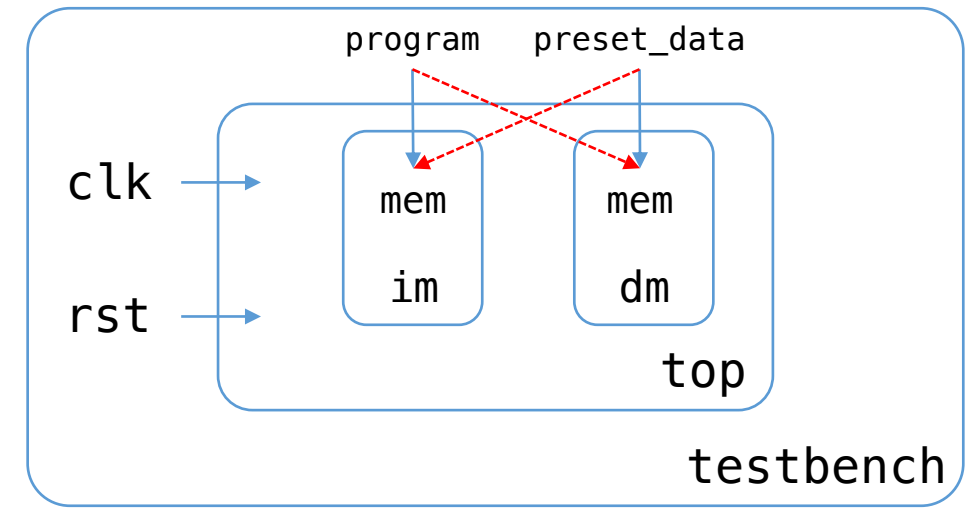


# Module List

- Top.v
  - ALU.v
  - Decoder.v
  - Imme\_Ext.v
  - JB\_Unit.v
  - LD\_Filter.v
  - RegFile.v
  - Adder.v
  - Reg\_PC.v
  - Mux.v
  - SRAM.v
  - Controller.v

```
module Top (
 input clk,
 input rst
);
 SRAM im(
 ...
);
 SRAM dm(
 ...
);
 ...
endmodule
```

current\_pc reset from 32'd0



- You can use reg on output when you need it
- Try to separate different modules in different files
- Finally, instance all of the modules and connect them in the Top module.

# Format of the Lab Report (PDF !!!)

---

- Cover (There is a default format of the report on the Moodle. )
- Content of the report
  1. Draw an Architecture Diagram
    - Please use [draw.io](https://draw.io) or powerpoint or any other painting software
    - Don't use paper & pen
    - Draw the Architecture Diagram of your CPU by yourself  
(You can draw a same diagram provided by TA if you use the same architecture as TA.)
  2. Introduce each module (function / corner case / and so on...)
  3. Screenshot the successful result

# File structure for submission

