Lab 8
Pipeline CPU Lab

Video Link : https://youtu.be/COBnJ6c5OYw

Outline

1. Introduction
2. Critical Path
3. Hazards (Structure / Control / Data)
4. Example of CPU Micro-Architecture

Prework of implementing a Pipeline CPU

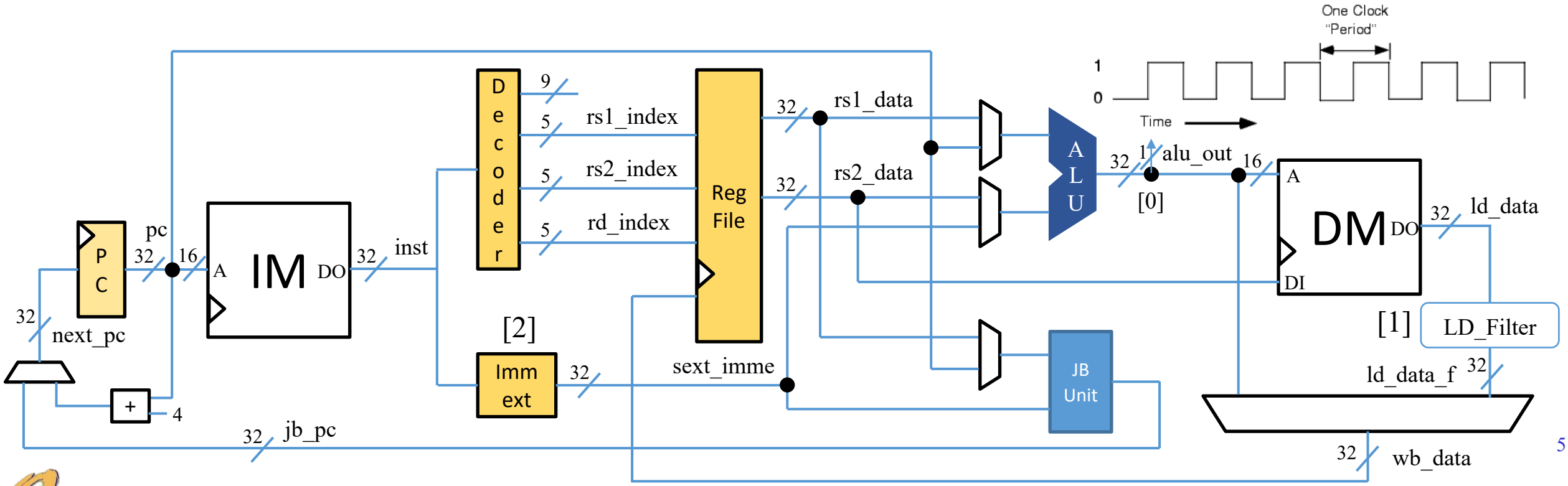
You need to **be familiar with** …

1. ISA (Lab2 ~ Lab3 – RISC-V ISA)
2. HDL (Verilog / SystemVerilog / VHDL) (Lab4 ~ Lab6 – Verilog)
3. Concept of CPU Architecture (Lab7)
4. Concept of Pipeline & Hazard (Lab8)

How to evaluate the maximum speed of a CPU

How to evaluate the maximum speed of a CPU

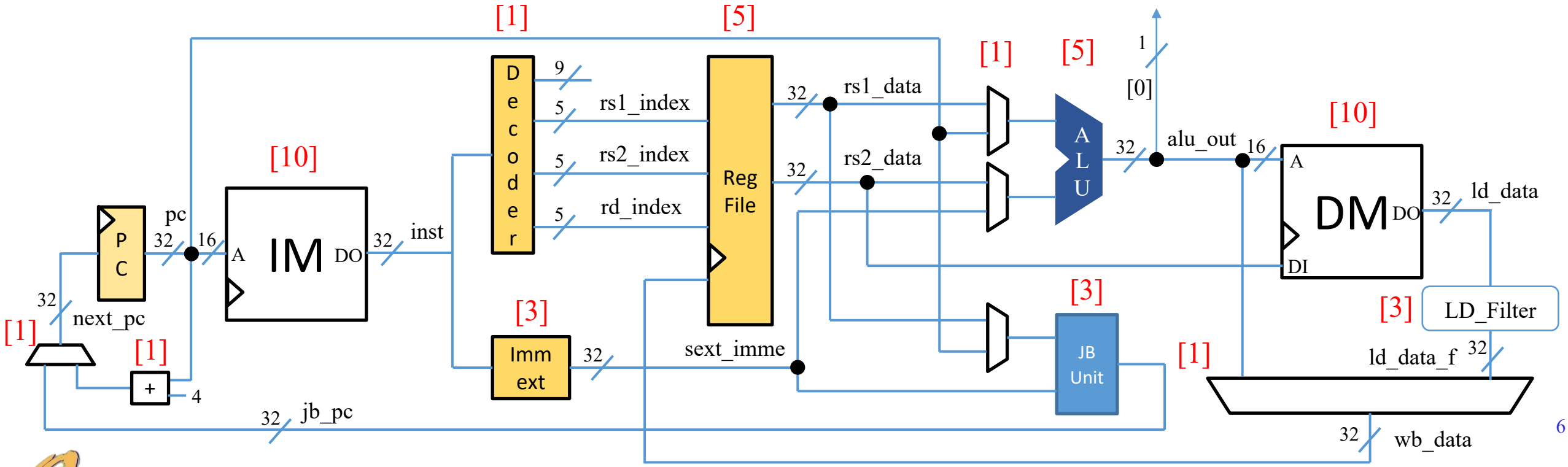
- $Speed \propto clock\ frequency = \frac{1}{period(cycle)}$ (the shorter the cycle, the faster the speed)
- After the clock cycle is determined, it is invariant and will not be adjusted with different instructions
- Single-Cycle CPU : All instructions accepted by a CPU need to be completed within 1 cycle
- We call the longest path the hardware takes from start to finish as the "Critical Path"
- The period of the critical path is the shortest clock period required for the hardware to operate as expected



Critical Path (Data Path)

Only focus on data path

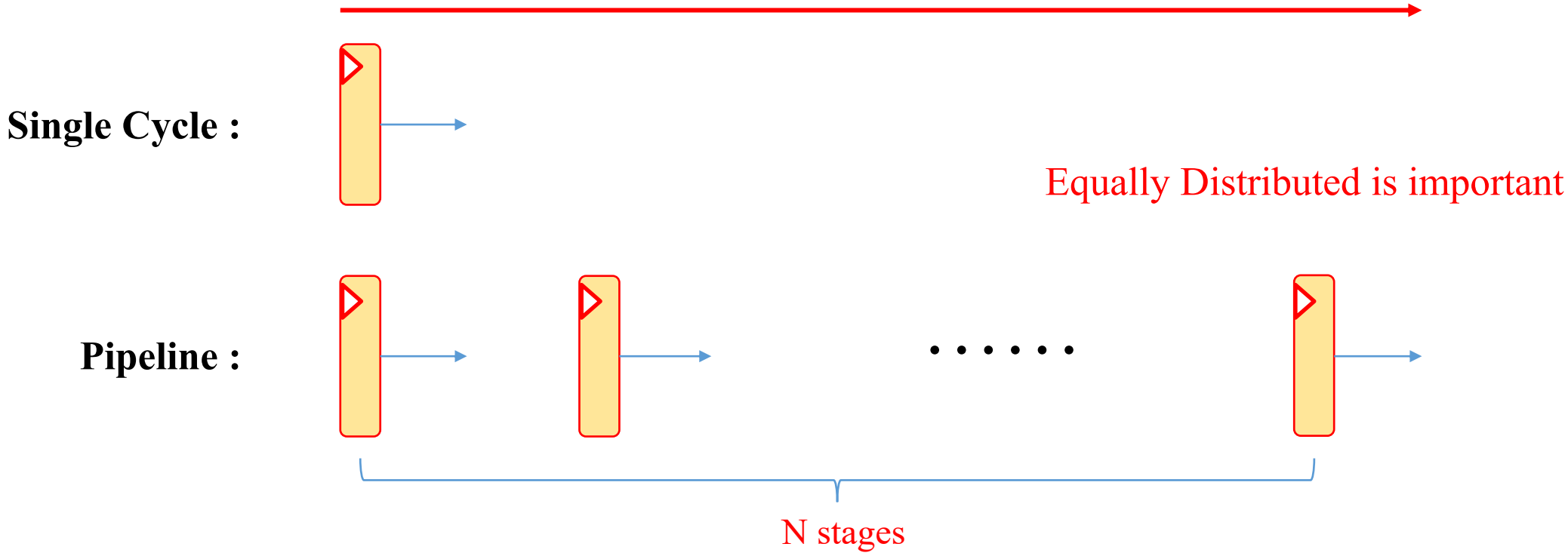
- Computational Instruction : 10 + 1 + 5 + 1 + 5 + 1 + 1(Setup time) = 24
- Load : 10 + 1 + 5 + 1 + 5 + 10 + 3 + 1 + 1(Setup time) = 37
- Store : 10 + 1 + 5 + 1 + 5 + 1(Setup time) = 23
- Jump : 10 + 1 + 5 + 1 + 5 + 1 + 1(Setup time) = 24
- Branch : 10 + 1 + 5 + 1 + 5 + 1 + 1(Setup time) = 24
- => Critical Path : 37 (Load) => clock period need to ≥ 37 => max frequency $= \frac{1}{37}$



How to accelerate ?

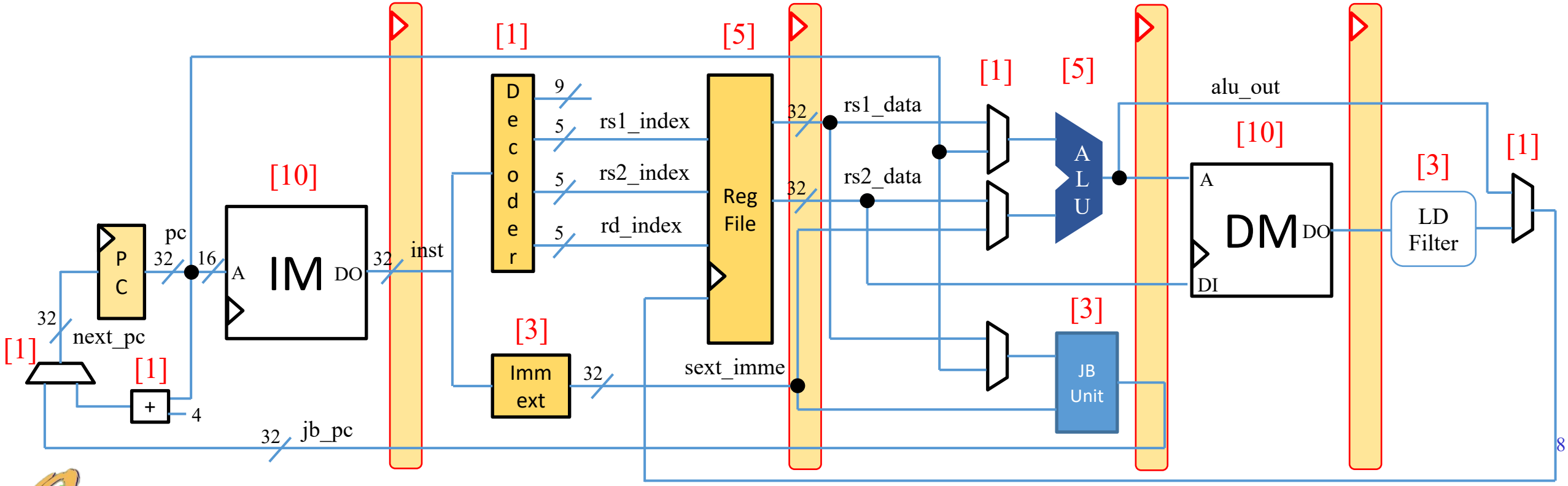
1. Split the hardware into multiple pieces
2. Do the different function simultaneously (overlap execution time => accelerate)

→ *Pipeline*



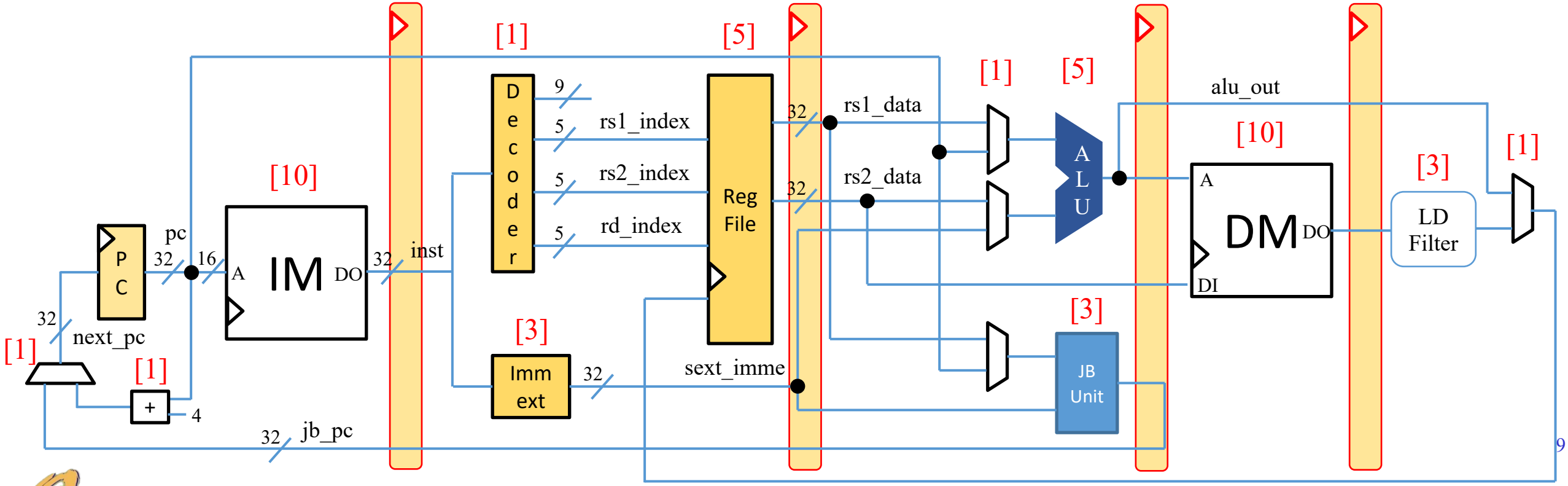
How to split our CPU ?

- Memory Access Latency is too long, memory access split into one stage
- Finally, we separate RegFile & ALU Totally 5 stages
- Critical Path : 10 + 1(Setup time) => clock period need to ≥ 11 => max frequency = $\frac{1}{11}$



How to split our CPU ?

- Instruction Latency ↑ => (11*5) per instruction
- Instruction Throughput ↑ => $\frac{Instructions\ count}{Time} = \frac{1}{11}$ (ideal)



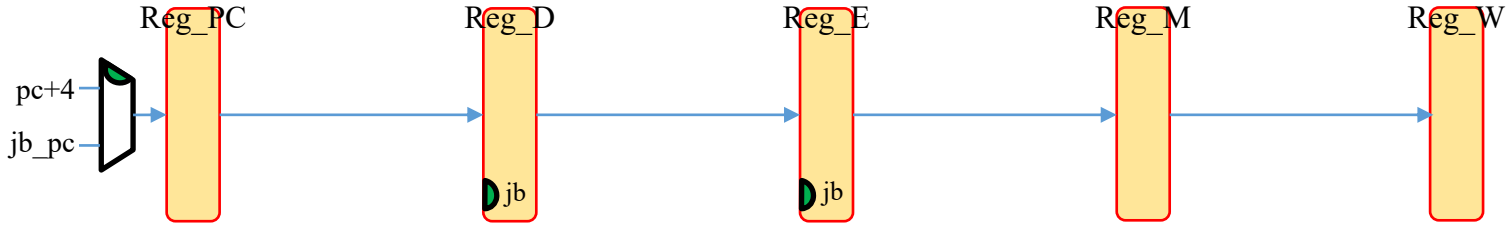
Hazard when implementing Pipeline Architecture

- **Structure Hazard**
  - Problem : Hardware resources are not enough
  - Problem in Pipeline CPU :
    - Accessing memory at the same time by fetching instructions and loading data
  - Solution :
    - We duplicate SRAM as im & dm to solve memory access problem of simultaneous instruction fetch and load data

- **Control Hazard**
  - Problem : There is an indeterminate instruction flow in the pipeline
  - Problem in Pipeline CPU :
    - The subsequent instructions have entered the pipeline before the jump or branch is determined
  - Solution :
    - We implement a flush signal in the Controller to flush the wrong instructions in pipeline registers

- **Data Hazard**
  - Problem : Unable to get the latest data for calculations
  - Problem in Pipeline CPU :
    - If the result data of an instruction needs to be writeback, subsequent instructions cannot get it until it completes
  - Solution :
    - We forwards the writeback data from MEM stage & WB stage to ID stage & EX stage

Control Hazard

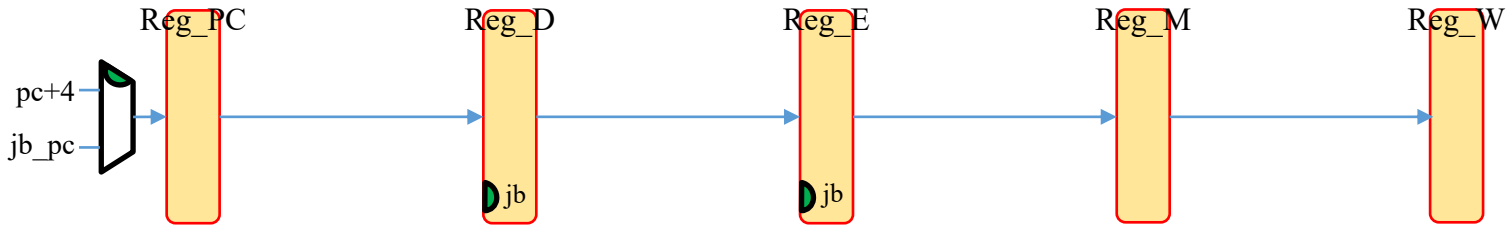


cycle	IF	ID	EX	MEM	WB
0	li t0, 1				

```
0  : li t0, 1
4  : li t1, 1
8  : beq t0, t1, equal
c  : li s0, 4
10 : jal x0, exit
equal:
14 : li t2, 3
18 : li t3, 4
1c : beq t2, t3, end
20 : li s0, 5
24 : jal x0, exit
end:
28 : li s0, 6
exit:
2c : ret
```

nop : addi x0, x0, 0

Control Hazard

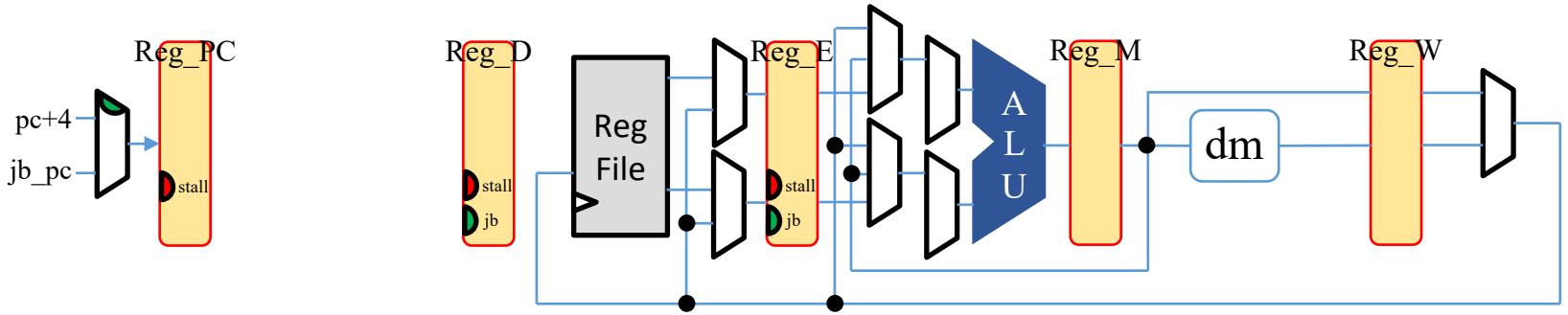



```
0  : li t0, 1
4  : li t1, 1
8  : beq t0, t1, equal
c  : li s0, 4
10 : jal x0, exit
equal:
14 : li t2, 3
18 : li t3, 4
1c : beq t2, t3, end
20 : li s0, 5
24 : jal x0, exit
end:
28 : li s0, 6
exit:
2c : ret
```

nop : addi x0, x0, 0

cycle	IF	ID	EX	MEM	WB
0	li t0, 1				
1	li t1, 1	li t0, 1			
2	beq t0, t1, equal	li t1, 1	li t0, 1		
3	li s0, 4	beq t0, t1, equal	li t1, 1	li t0, 1	
4	jal x0, exit	li s0, 4	beq t0, t1, equal	li t1, 1	li t0, 1
5	li t2, 3	nop	nop	beq t0, t1, equal	li t1, 1
6	li t3, 4	li t2, 3	nop	nop	beq t0, t1, equal
7	beq t2, t3, end	li t3, 4	li t2, 3	nop	nop
8	li s0, 5	beq t2, t3, end	li t3, 4	li t2, 3	nop
9	jal x0, exit	li s0, 5	beq t2, t3, end	li t3, 4	li t2, 3
10	li s0, 6	jal x0, exit	li s0, 5	beq t2, t3, end	li t3, 4
11	ret	li s0, 6	jal x0, exit	li s0, 5	beq t2, t3, end
12	ret	nop	nop	jal x0, exit	li s0, 5

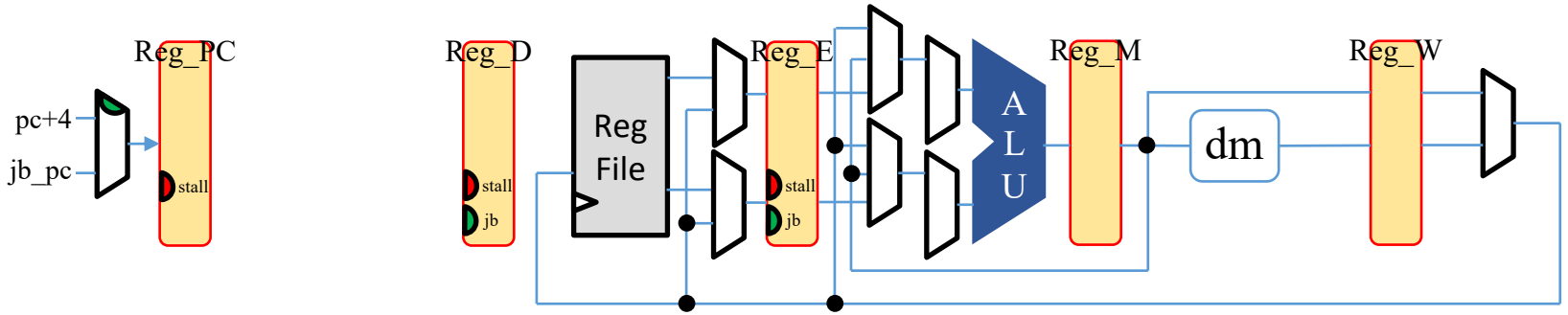
Data Hazard



cycle	IF	ID	EX	MEM	WB
0	li t0, 1				

```
0  : li t0, 1
4  : li t1, 1
8  : add t2, t0, t1
c  : li t2, 3
10 : add t3, t2, t1
14 : lw t4, 0(t3)
18 : addi t5, t4, 1
1c : ret
```

Data Hazard



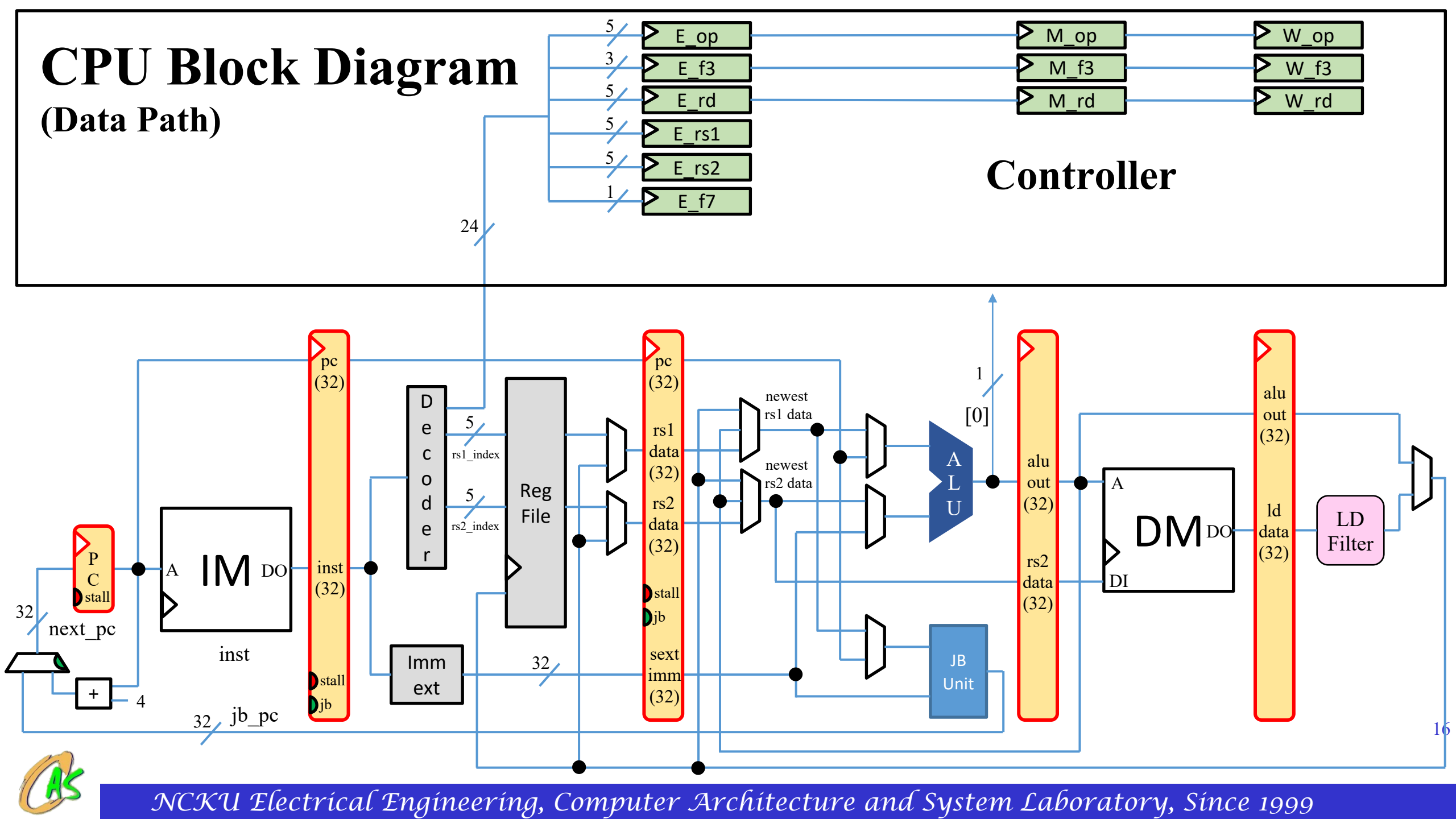

```
0  : li t0, 1
4  : li t1, 1
8  : add t2, t0, t1
c  : li t2, 3
10 : add t3, t2, t1
14 : lw t4, 0(t3)
18 : addi t5, t4, 1
1c : ret
```

cycle	IF	ID	EX	MEM	WB
0	li t0, 1				
1	li t1, 1	li t0, 1			
2	add t2, t0, t1	li t1, 1	li t0, 1		
3	li t2, 3	add t2, t0, t1	li t1, 1	li t0, 1	
4	add t3, t2, t1	li t2, 3	add t2, t0, t1	li t1, 1	li t0, 1
5	lw t4, 0(t3)	add t3, t2, t1	li t2, 3	add t2, t0, t1	li t1, 1
6	addi t5, t4, 1	lw t4, 0(t3)	add t3, t2, t1	li t2, 3	add t2, t0, t1
7	ret	addi t5, t4, 1	lw t4, 0(t3)	add t3, t2, t1	li t2, 3
8	ret	addi t5, t4, 1	nop	lw t4, 0(t3)	add t3, t2, t1
9	–	ret	addi t5, t4, 1	nop	lw t4, 0(t3)

Pipeline Registers

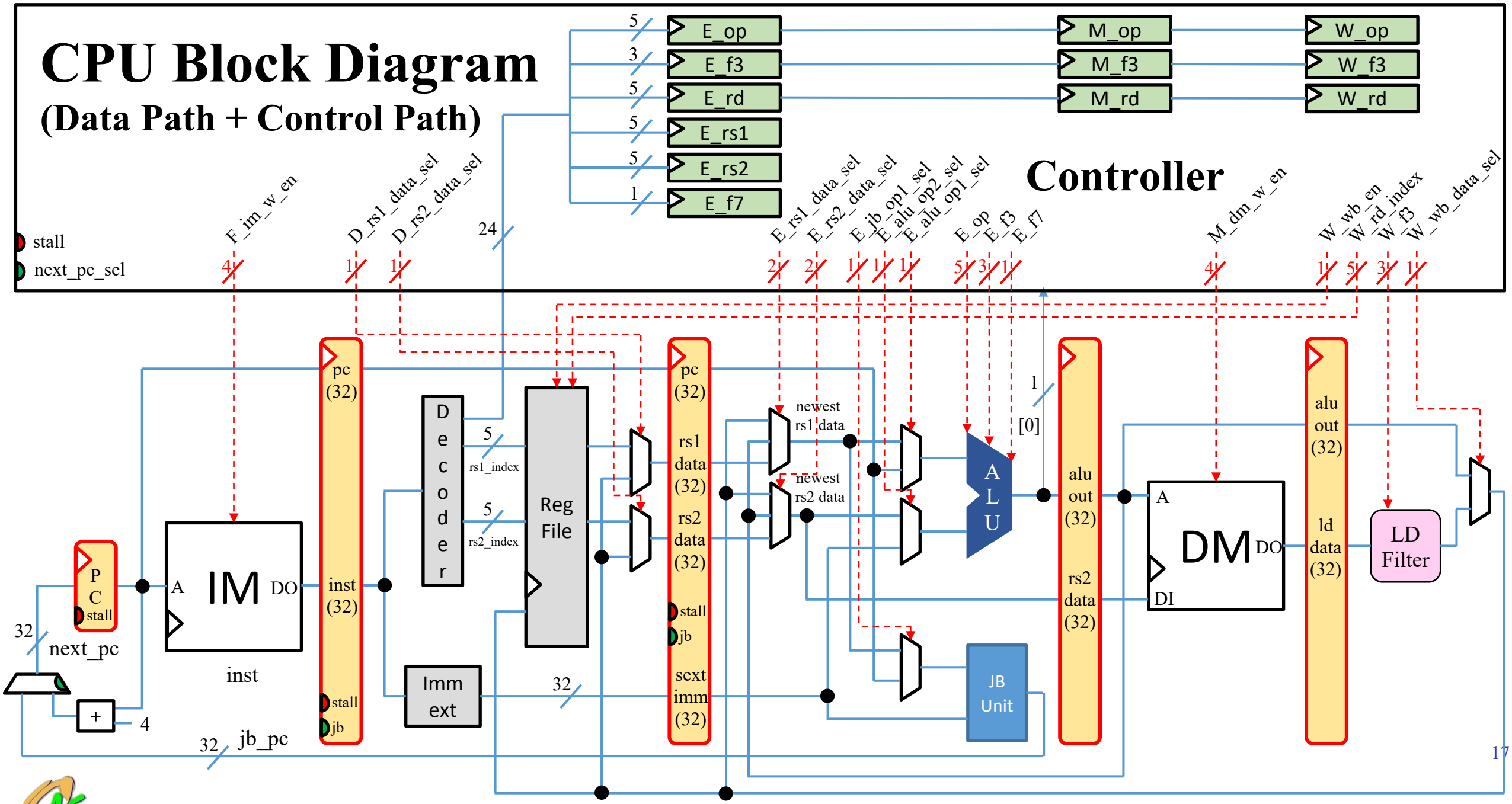
	stall	jb	others
Reg_PC	keep (output <= output)	output <= input	output <= input
Reg_D	keep (output <= output)	flush (output <= nop)	output <= input
Reg_E	bubble (output <= nop)	flush (output <= nop)	output <= input
Reg_M	output <= input	output <= input	output <= input
Reg_W	output <= input	output <= input	output <= input

CPU Block Diagram
(Data Path)



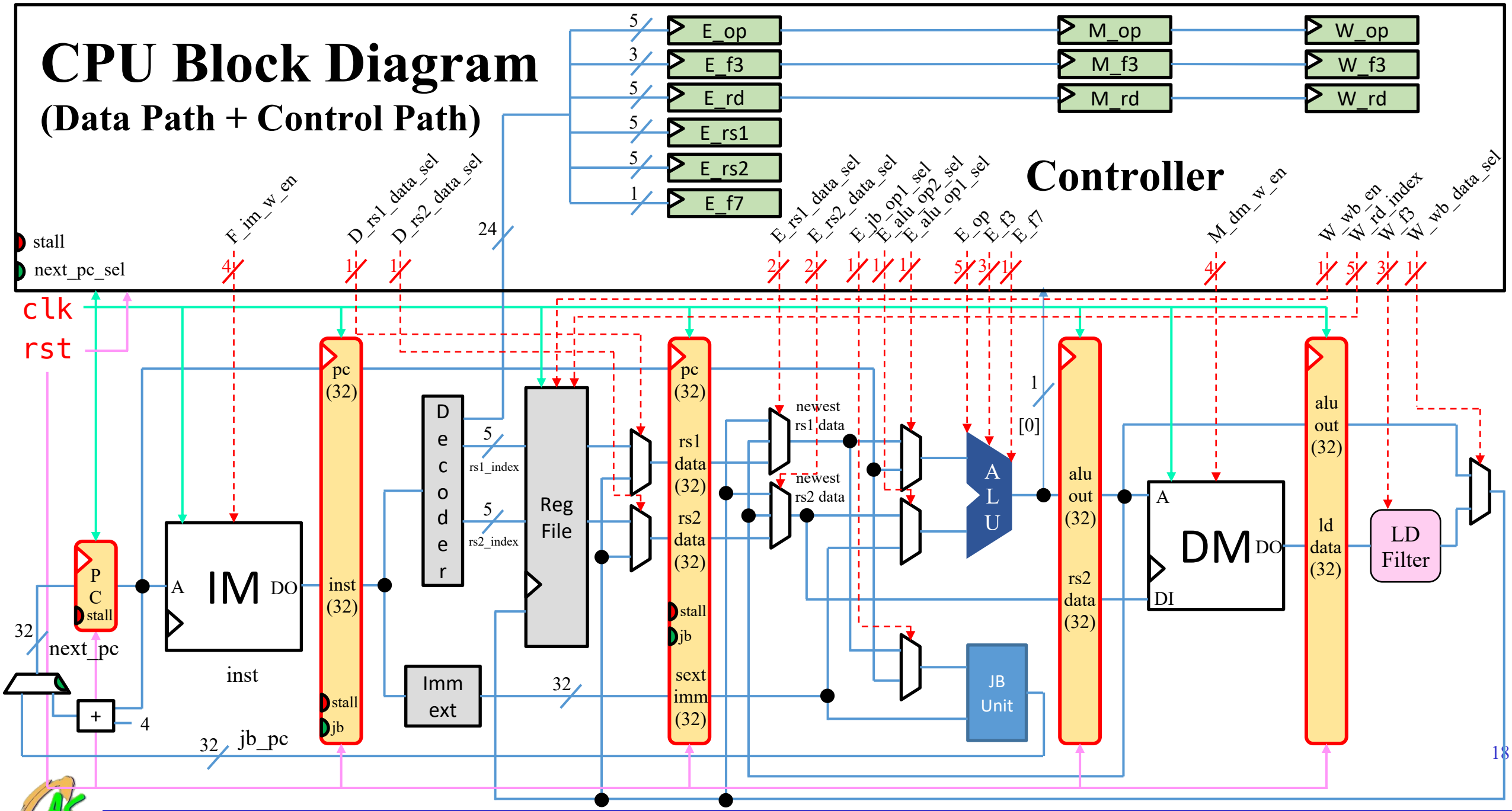
CPU Block Diagram

(Data Path + Control Path)



CPU Block Diagram

(Data Path + Control Path)



Controller

- Need add some registers to keep instruction info in each stage
- Use the instruction info kept in each stages to get the control signals in each stages

Signal	Usage
F_im_w_en (4)	Prevent writing data into im
D_rs1_data_sel (1)	Immediately Read the Writeback rs1 data
D_rs2_data_sel (1)	Immediately Read the Writeback rs2 data
E_rs1_data_sel (2)	Get the newest rs1 data
E_rs2_data_sel (2)	Get the newest rs2 data
E_alu_op1_sel (1)	Select alu op1 (rs1 / pc)
E_alu_op2_sel (1)	Select alu op2 (rs2 / imm)
E_jb_op1_sel (1)	Select jb_unit op1 (rs1 / pc)
E_op_out (5)	Output the opcode in E stage to ALU
E_f3_out (3)	Output the func3 in E stage to ALU
E_f7_out (1)	Output the func7 in E stage to ALU

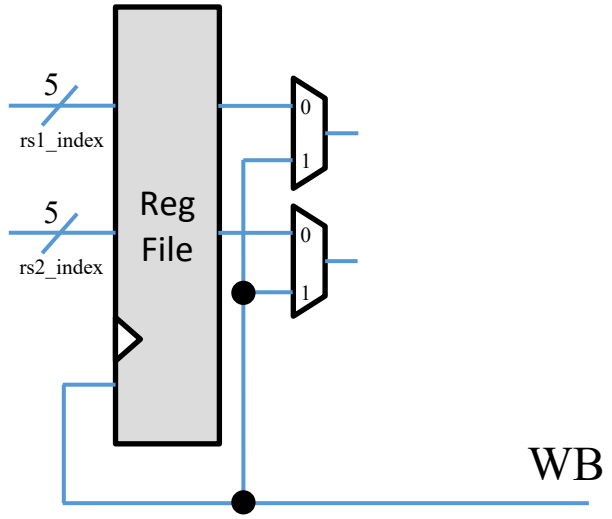
Signal	Usage
M_dm_w_en (4)	Store [byte / halfword / word] into dm
W_wb_en (1)	Whether write back data into RegFile
W_rd_index (5)	Write back register destination index
W_f3 (3)	Output the func3 in W stage to LD_Filter
W_wb_data_sel (1)	Select writeback data (ALU Result / Extended Load Data)
stall (1)	Set when there is a Load instruction in E stage and there is an instruction in D stage that uses the Load's writeback data
next_pc_sel (1)	Set when a jump or branch is established in E stage

RV32I Instruction Type Table

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						

Control Signal

(D_rs1_data_sel & D_rs2_data_sel)



D_rs1_data_sel :

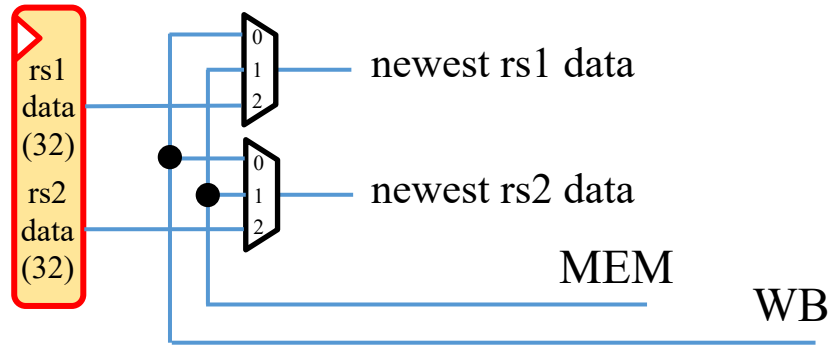
```
D_rs1_data_sel = is_D_rs1_W_rd_overlap ? 1'd1 : 1'd0;

is_D_rs1_W_rd_overlap = is_D_use_rs1 & is_W_use_rd & (dc_rs1 == W_rd) & W_rd != 0
is_D_use_rs1 = ?
is_W_use_rd = ?
```

D_rs2_data_sel :

```
D_rs2_data_sel = is_D_rs2_W_rd_overlap ? 1'd1 : 1'd0;

is_D_rs2_W_rd_overlap = is_D_use_rs2 & is_W_use_rd & (dc_rs2 == W_rd) & W_rd != 0
is_D_use_rs2 = ?
is_W_use_rd = ?
```

Control Signal

(E_rs1_data_sel & E_rs2_data_sel)



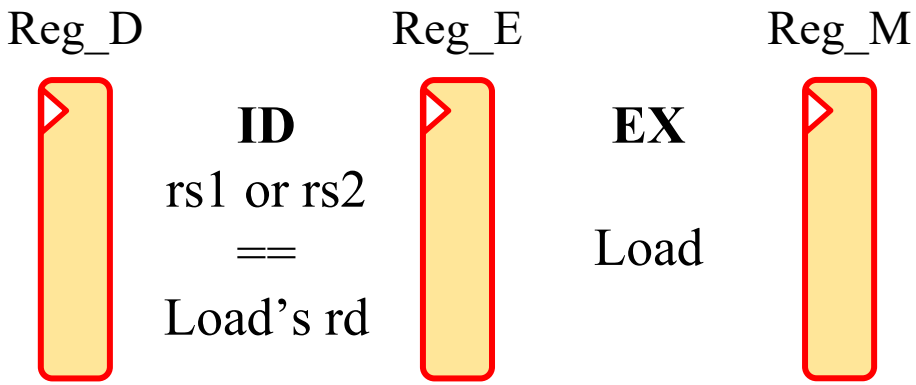
E_rs1_data_sel :

```
E_rs1_data_sel =    is_E_rs1_M_rd_overlap ? 2'd1 :
                    is_E_rs1_W_rd_overlap ? 2'd0 : 2'd2;
is_E_rs1_W_rd_overlap = is_E_use_rs1 & is_W_use_rd & (E_rs1 == W_rd) & W_rd != 0
is_E_rs1_M_rd_overlap = is_E_use_rs1 & is_M_use_rd & (E_rs1 == M_rd) & M_rd != 0
is_E_use_rs1 = ?
is_W_use_rd = ?
is_M_use_rd = ?
```

E_rs2_data_sel :

```
E_rs2_data_sel =    is_E_rs2_M_rd_overlap ? 2'd1 :
                    is_E_rs2_W_rd_overlap ? 2'd0 : 2'd2;
is_E_rs2_W_rd_overlap = is_E_use_rs2 & is_W_use_rd & (E_rs2 == W_rd) & W_rd != 0
is_E_rs2_M_rd_overlap = is_E_use_rs2 & is_M_use_rd & (E_rs2 == M_rd) & M_rd != 0
is_E_use_rs2 = ?
is_W_use_rd = ?
is_M_use_rd = ?
```

Control Signal
(stall)

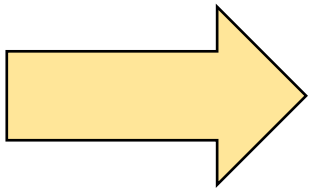


stall :

```
stall = (E == load) & is_DE_overlap

is_DE_overlap = (is_D_rs1_E_rd_overlap | is_D_rs2_E_rd_overlap)

is_D_rs1_E_rd_overlap = is_D_use_rs1 & is_E_use_rd & (dc_rs1 == E_rd) & E_rd != 0
is_D_use_rs1 = ?
is_E_use_rd = ?

is_D_rs2_E_rd_overlap = is_D_use_rs2 & is_E_use_rd & (dc_rs2 == E_rd) & E_rd != 0
is_D_use_rs2 = ?
is_E_use_rd = ?
```

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



- Top.v (modify)
  - ALU.v
  - Decoder.v
  - Imme_Ext.v
  - JB_Unit.v
  - LD_Filter.v
  - RegFile.v
  - Adder.v
  - Reg_PC.v (modify)
  - Mux2.v (2 inputs, 1 output)
  - Mux3.v (3 inputs, 1 output)
  - SRAM.v
  - Controller.v (modify)
  - Reg_D.v (new)
  - Reg_E.v (new)
  - Reg_M.v (new)
  - Reg_W.v (new)