

RSIC-V Datapath and Control Pipeline - I

Chixiao Chen

Overview

- Basic digital logic knowledge
- Datapath/Control for Single Cycle Processor
- Pipeline – Part I

Announcement

- Homework 1 is on line at

https://cihlab.github.io/course/ca_hw01.pdf

- April 11th class will be canceled.

- All slides is available on

https://cihlab.github.io/course/ai_19.html



Review: RISC-V Instruction Format

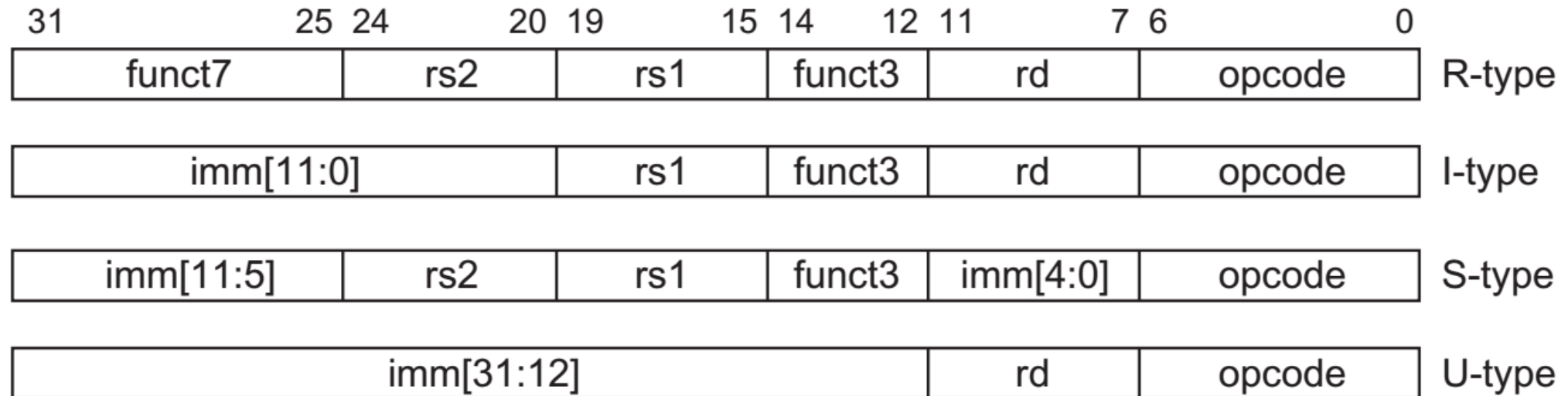


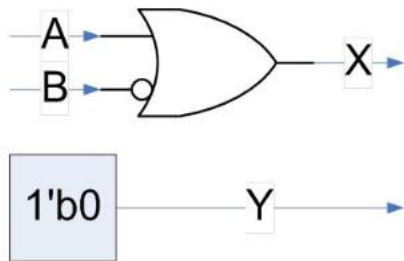
Figure A.23 The RISC-V instruction layout. There are two variations on these formats, called the SB and UJ formats; they deal with a slightly different treatment for immediate fields.

Why S-type? Is it different from I-type?

Basic Digital Logic Knowledge

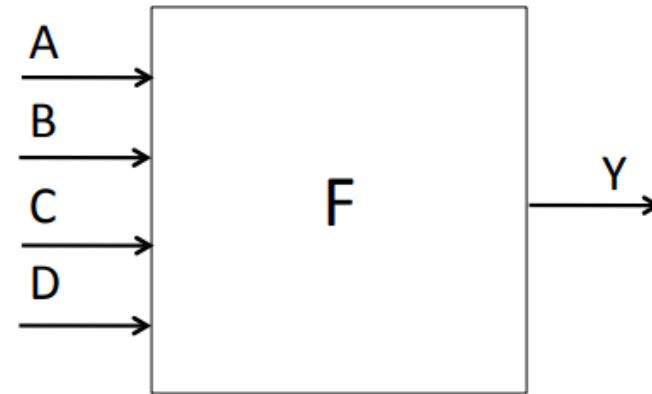
Combinational Logic

- Use Truth Table
- Verilog make is more easily



```
assign X =  
  A | ~B;
```

```
assign Y = 1'b0;
```



Exhaustive list of the output value
generated for each combination of inputs

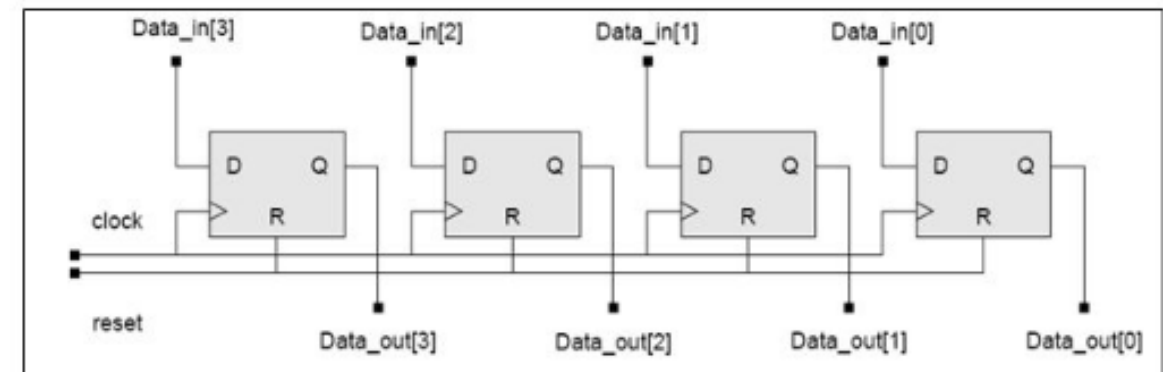
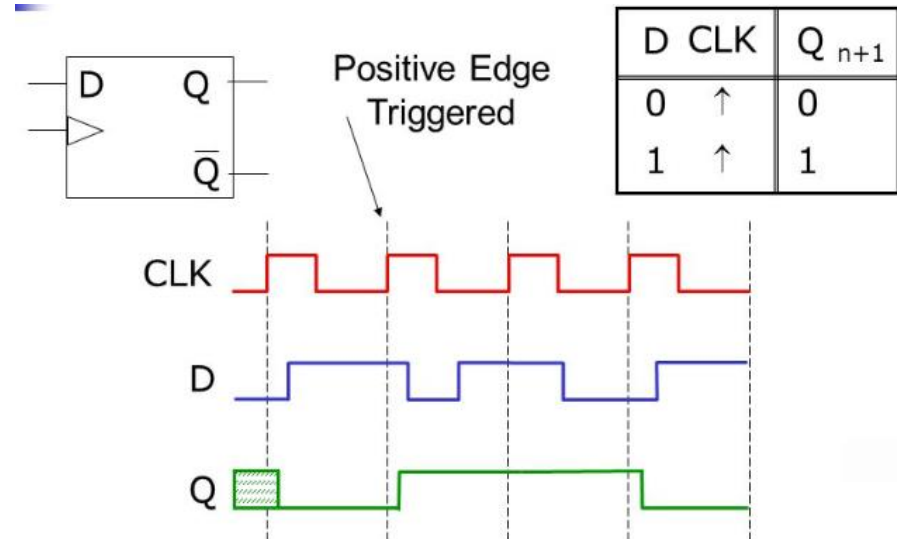
How many logic functions can be defined
with N inputs?

a	b	c	d	y
0	0	0	0	F(0,0,0,0)
0	0	0	1	F(0,0,0,1)
0	0	1	0	F(0,0,1,0)
0	0	1	1	F(0,0,1,1)
0	1	0	0	F(0,1,0,0)
0	1	0	1	F(0,1,0,1)
0	1	1	0	F(0,1,1,0)
1	1	1	1	F(0,1,1,1)
1	0	0	0	F(1,0,0,0)
1	0	0	1	F(1,0,0,1)
1	0	1	0	F(1,0,1,0)
1	0	1	1	F(1,0,1,1)
1	1	0	0	F(1,1,0,0)
1	1	0	1	F(1,1,0,1)
1	1	1	0	F(1,1,1,0)
1	1	1	1	F(1,1,1,1)

Sequential Logic and Flip-Flops

- Flip Flop (触发器) is the basic cell to store the logic with clock
- Verilog example

```
module D_reg4a (Data_in, clock, reset, Data_out);  
  input [3:0] Data_in;  
  input clock, reset;  
  output [3:0] Data_out;  
  reg [3:0] Data_out;  
  always @ (posedge reset or posedge clock)  
    if (reset == 1'b1) Data_out <= 4'b0;  
    else Data_out <= Data_in;  
endmodule
```



Finite State Machine

- Verilog Example

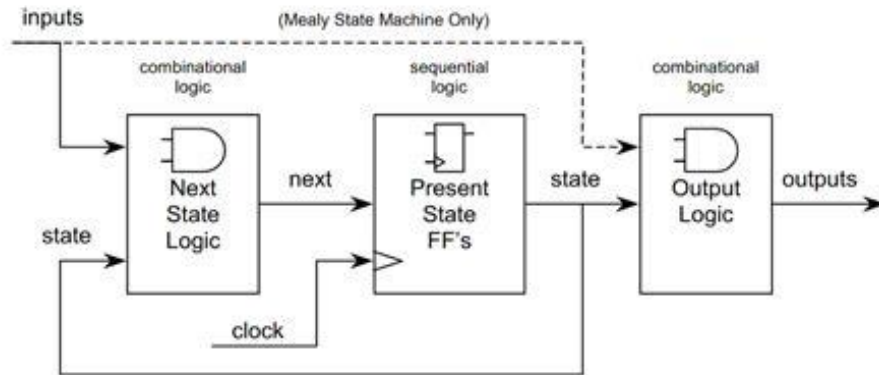


Figure 1 - FSM Block Diagram

```
timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

    input B;
    output reg X;
    input Clk, Rst;

    parameter s_Off = 0, s_On1 = 1,
              s_On2 = 2, s_On3 = 3;

    reg [1:0] State, StateNext;

    // CombLogic
    always @(State, B) begin
        case (State)
            s_Off: begin
                X = 0;
                if (B == 0)
                    StateNext = s_Off;
                else
                    StateNext = s_On1;
            end
        endcase
    end
```

```
    ...
    s_On1: begin
        X = 1;
        StateNext = s_On2;
    end
    s_On2: begin
        X = 1;
        StateNext = s_On3;
    end
    s_On3: begin
        X = 1;
        StateNext = s_Off;
    end
endcase
end

// StateReg
always @(posedge Clk) begin
    if (Rst == 1 )
        State <= s_Off;
    else
        State <= StateNext;
    end
endmodule
```


Single Cycle RV32I Processor

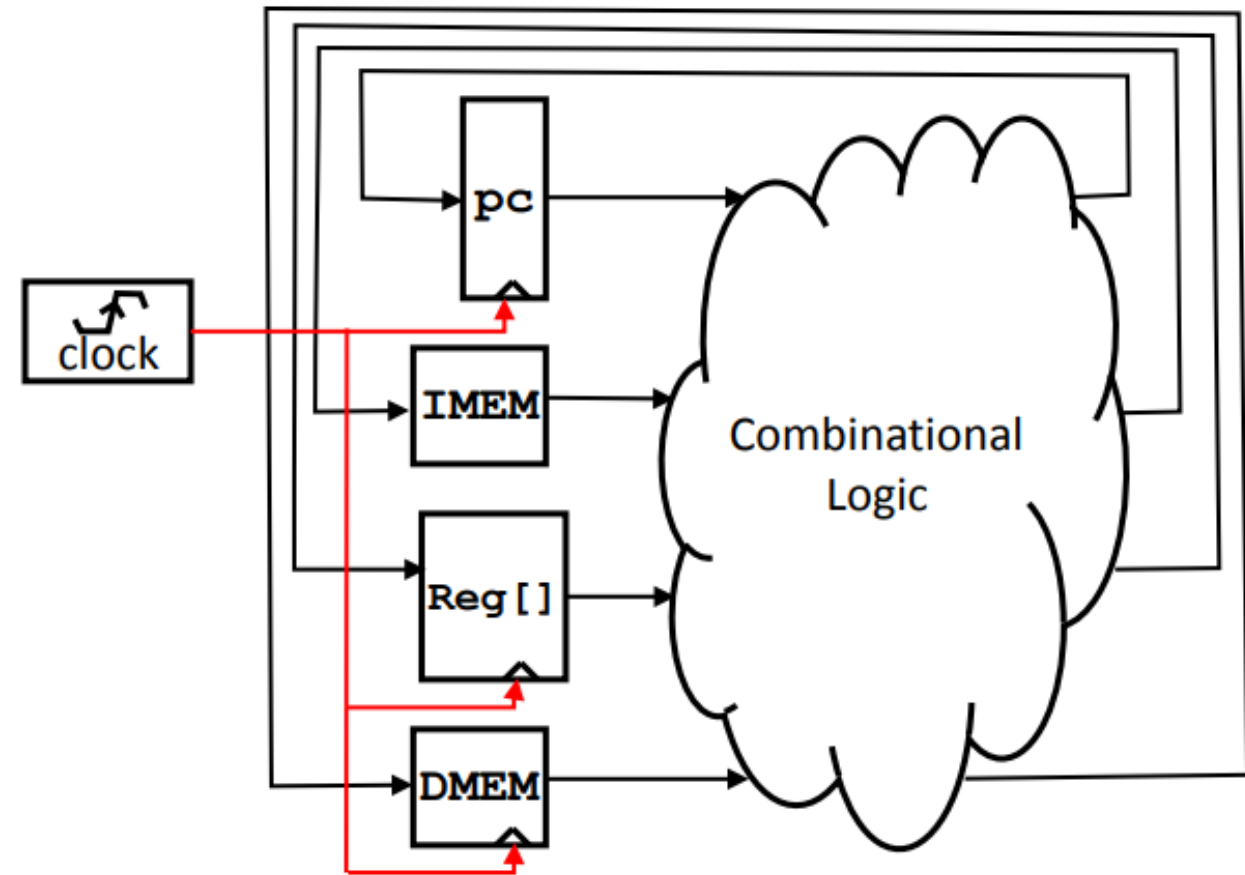
State Required by RV32I

Each instruction reads and updates this state during execution:

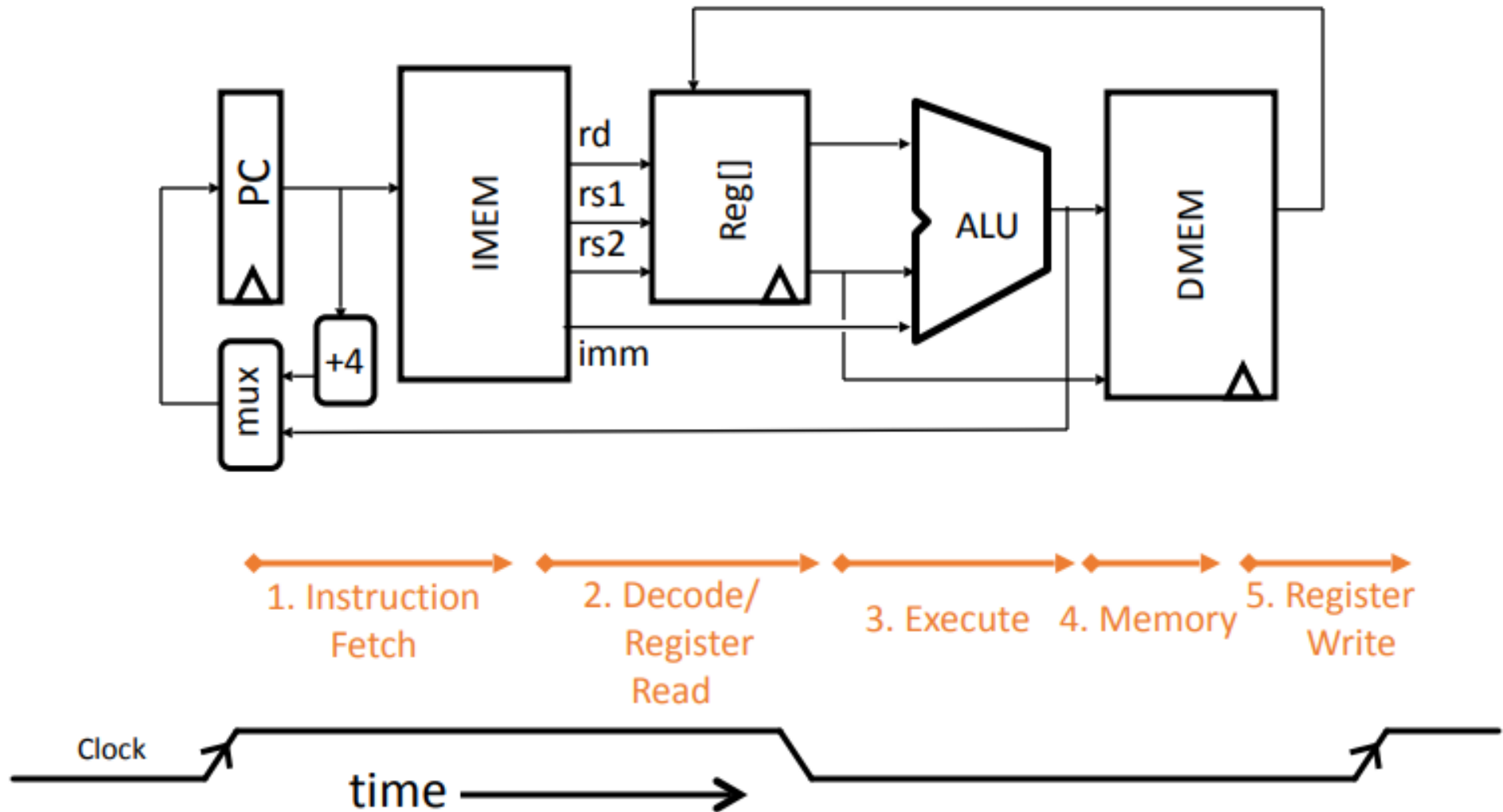
- Registers (**x0** . . **x31**)
 - Register file (or *regfile*) **Reg** holds 32 registers x 32 bits/register: **Reg**[0] . . **Reg**[31]
 - First register read specified by *rs1* field in instruction
 - Second register read specified by *rs2* field in instruction
 - Write register (destination) specified by *rd* field in instruction
 - **x0** is always 0 (writes to **Reg**[0] are ignored)
- Program Counter (**PC**)
 - Holds address of current instruction
- Memory (**MEM**)
 - Holds both instructions & data, in one 32-bit byte-addressed memory space
 - We'll use separate memories for instructions (**IMEM**) and data (**DMEM**)
 - *Later we'll replace these with instruction and data caches*
 - Instructions are read (*fetch*) from instruction memory (assume **IMEM** read-only)
 - Load/store instructions access data memory

One-Instruction-Per-Cycle RISC-V Machine

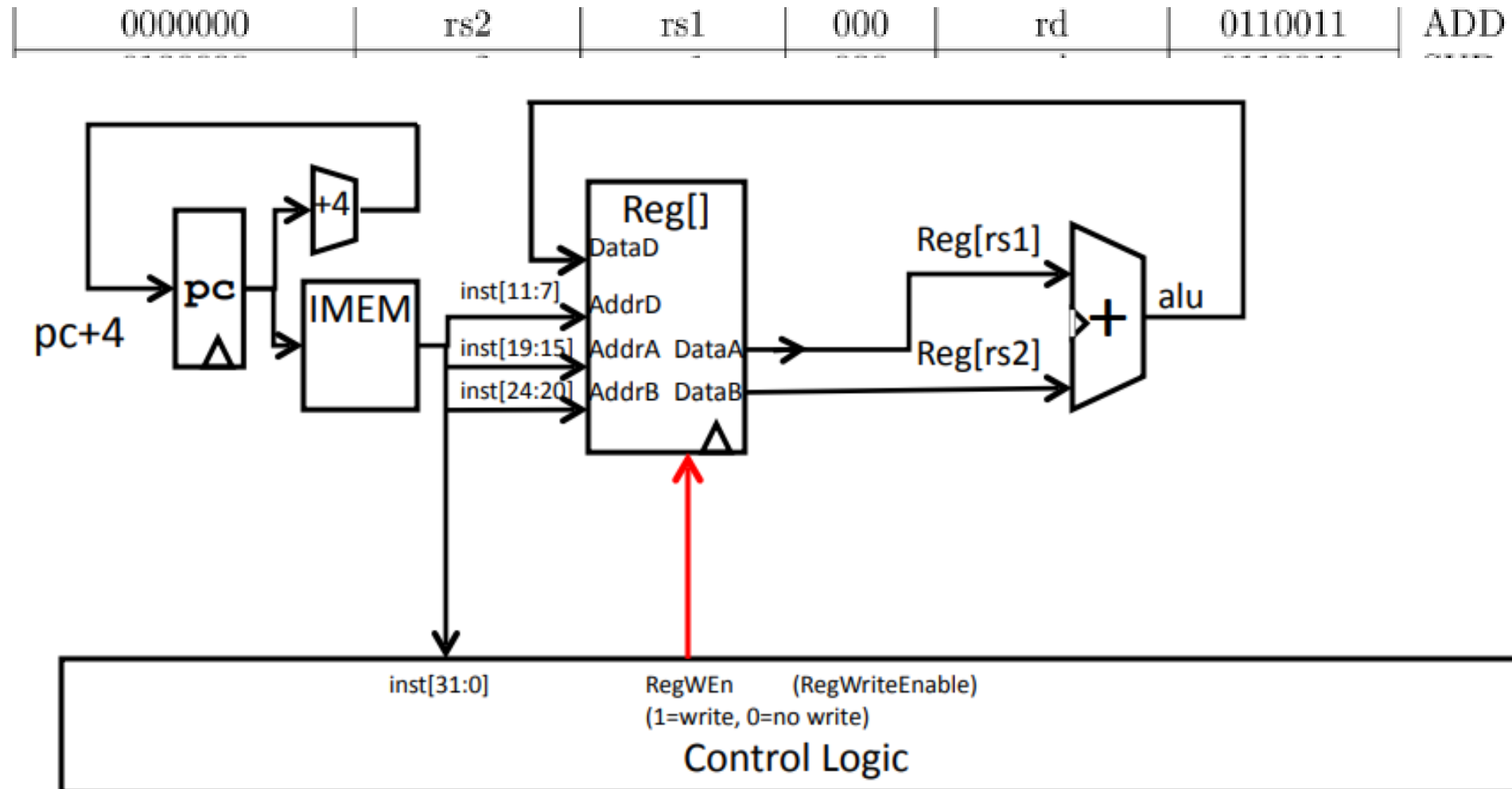
- Current state outputs drive the inputs to the combinational logic, whose outputs settle at the values of the state before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle



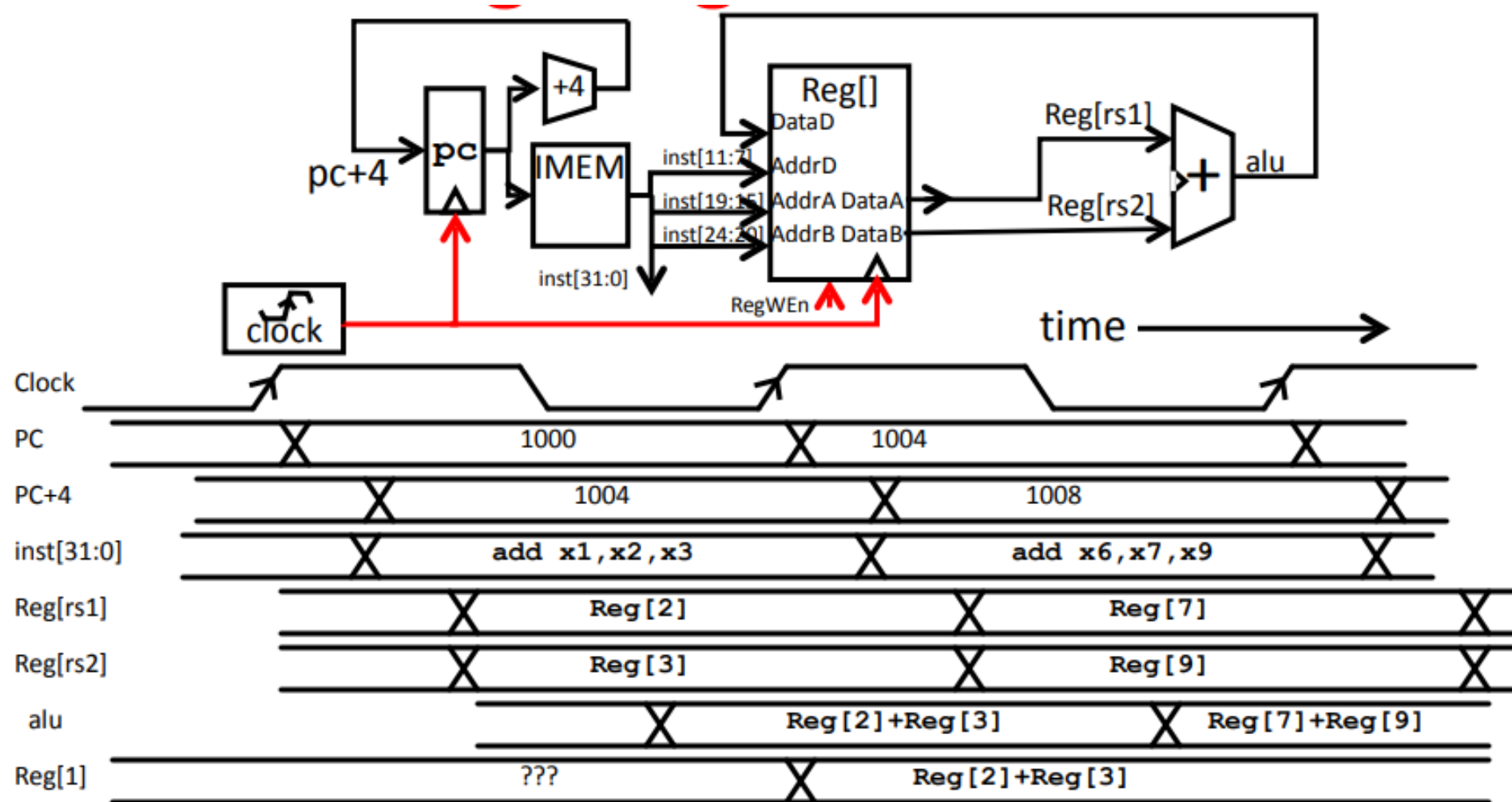
Basic Phase of Instruction Execution



Implementing the **add** instruction



Timing Diagram for add



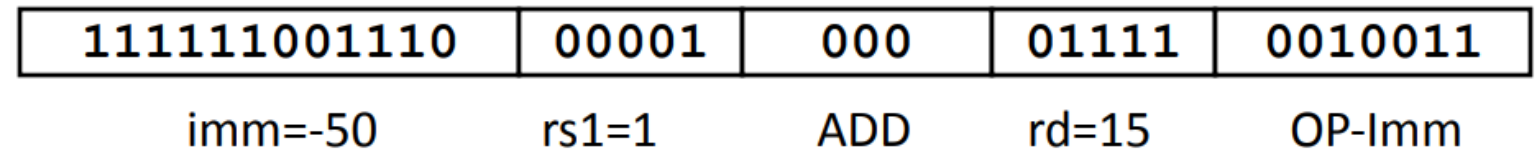
Implementing other R-format Instructions

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

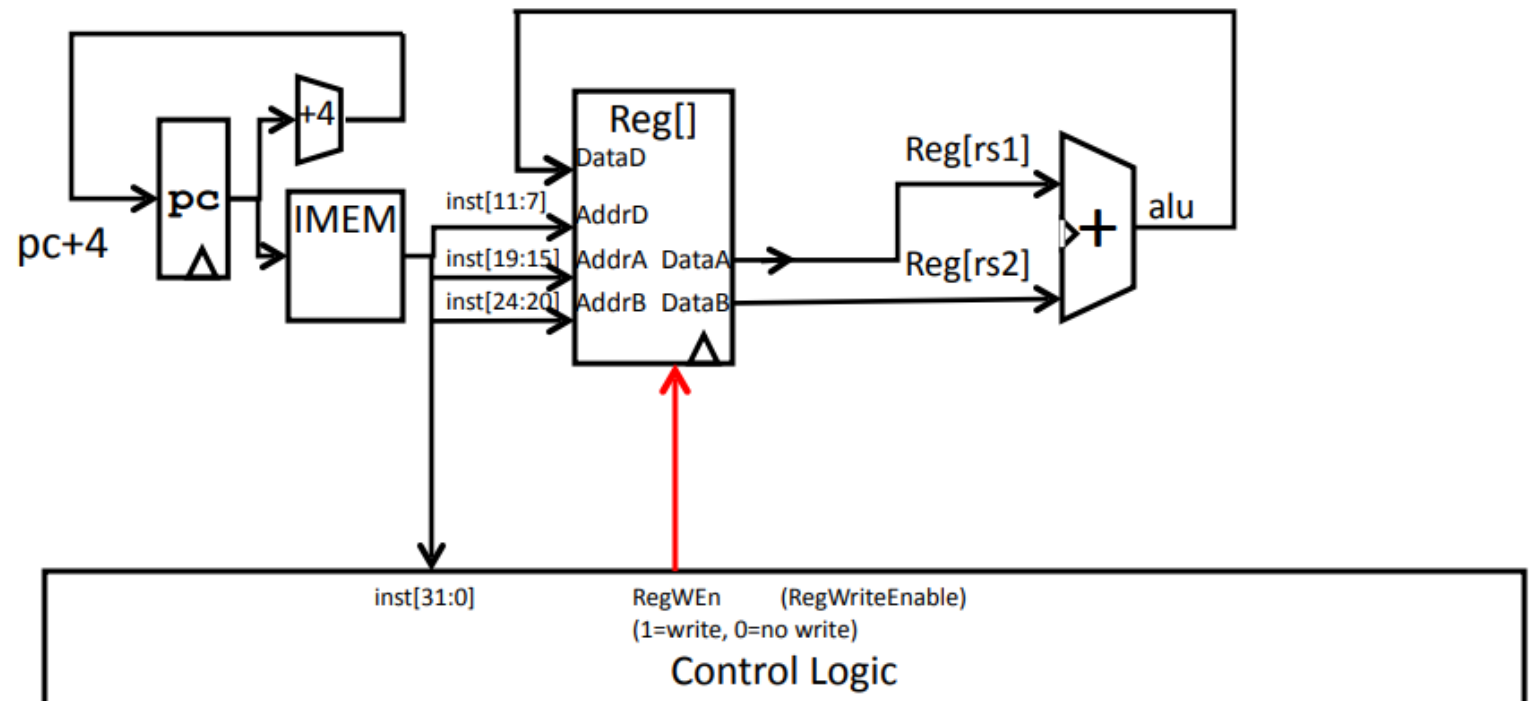
- All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

Implementing the **addi** instruction

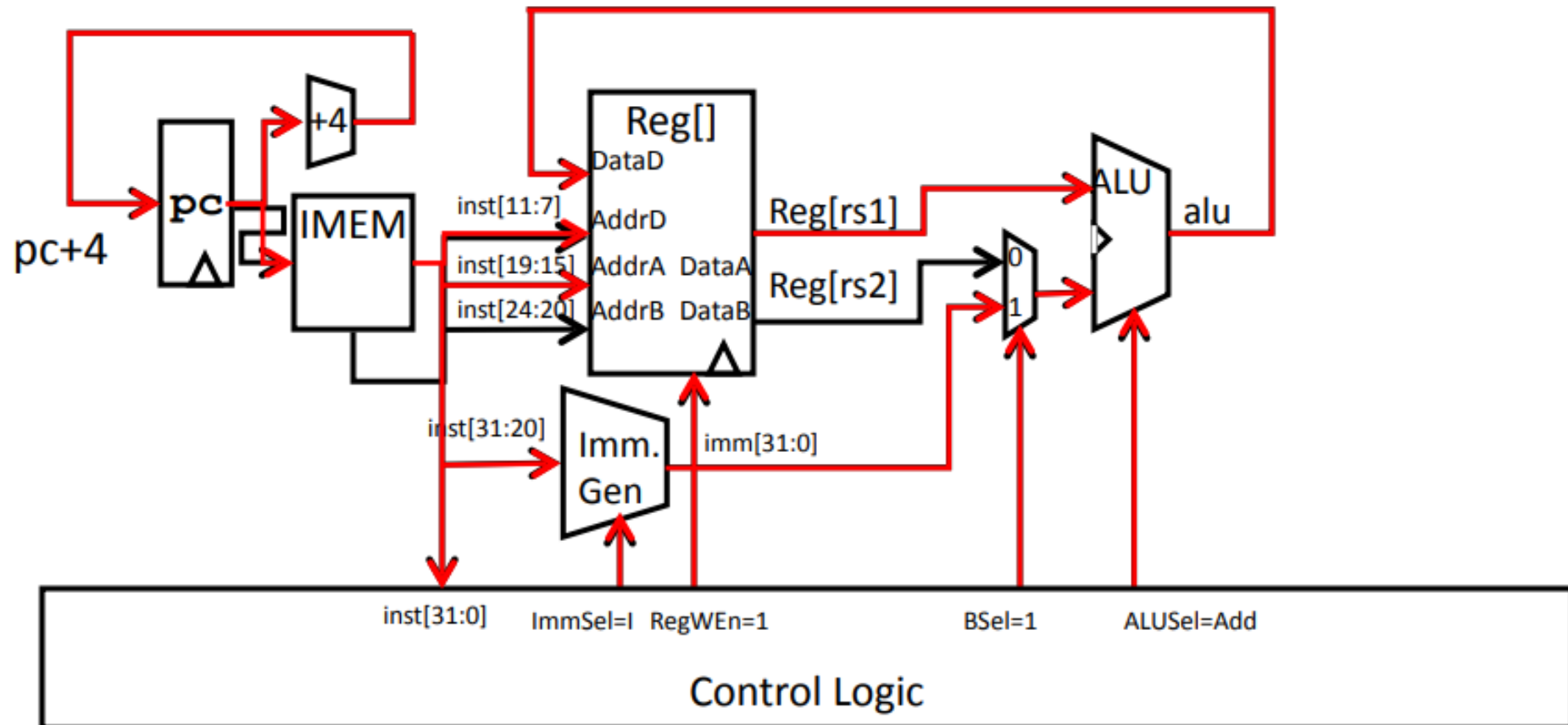
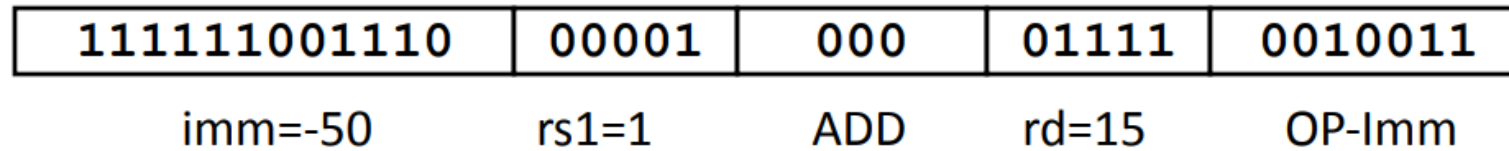
- **addi** x15,x1,-50



- Previous Architecture



Implementing the **addi** instruction



Implementing other I-format Instructions

imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

- All xxx-I instruction implemented by decoding funct3.
- **Question:** imm only have 12 bit, but input of ALU is 32bit.

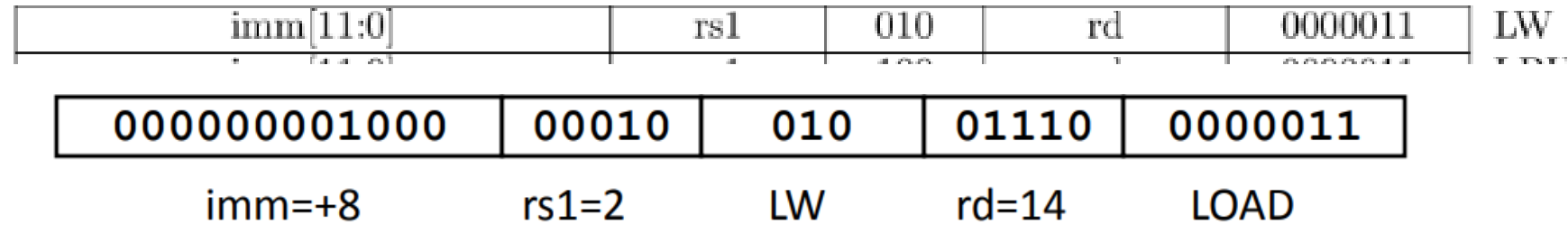
U-Type for Upper Immediate instruction

imm[31:12]	rd	0110111	LUI
imm[31:12]	rd	0010111	AUIPC
imm[31:12]	rd	1101111	UAT

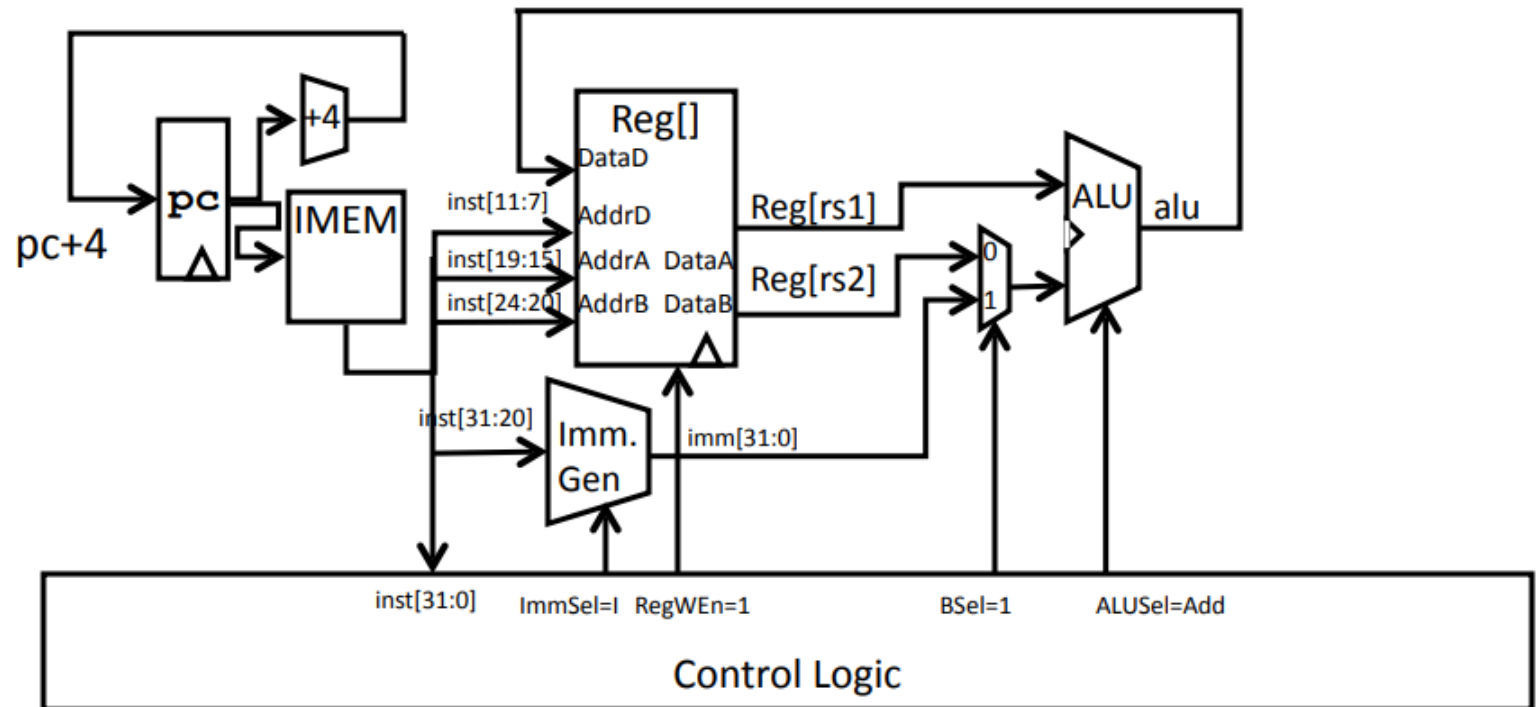
- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
 - LUI – Load Upper Immediate
 - AUIPC – Add Upper Immediate to PC

Implementing the **load(lw)** instruction

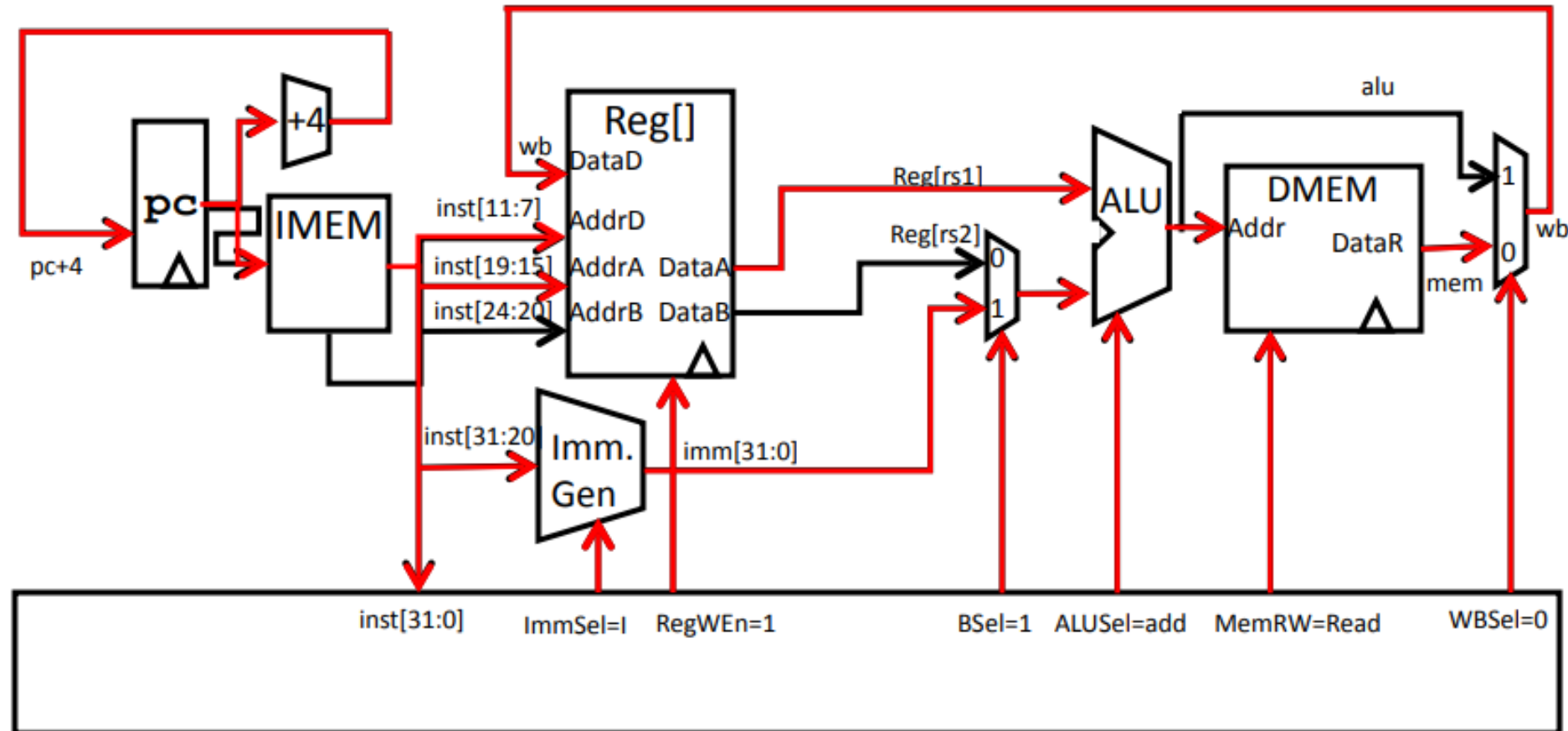
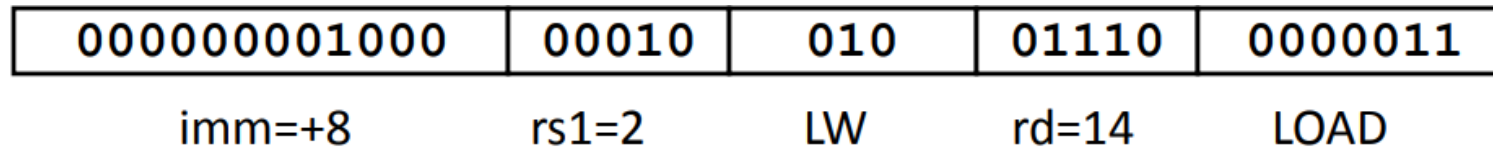
- **lw** x14, 8(x2)



- Previous Architecture



Implementing the **load(lw)** instruction



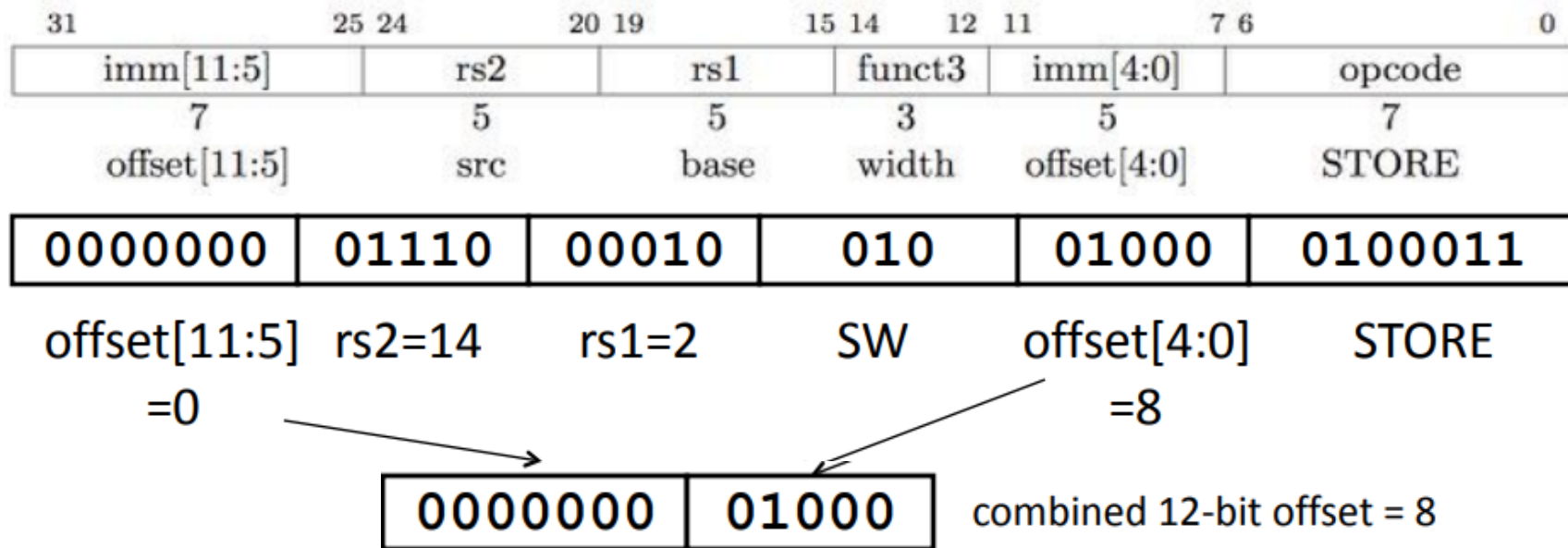
Implementing other load Instructions

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

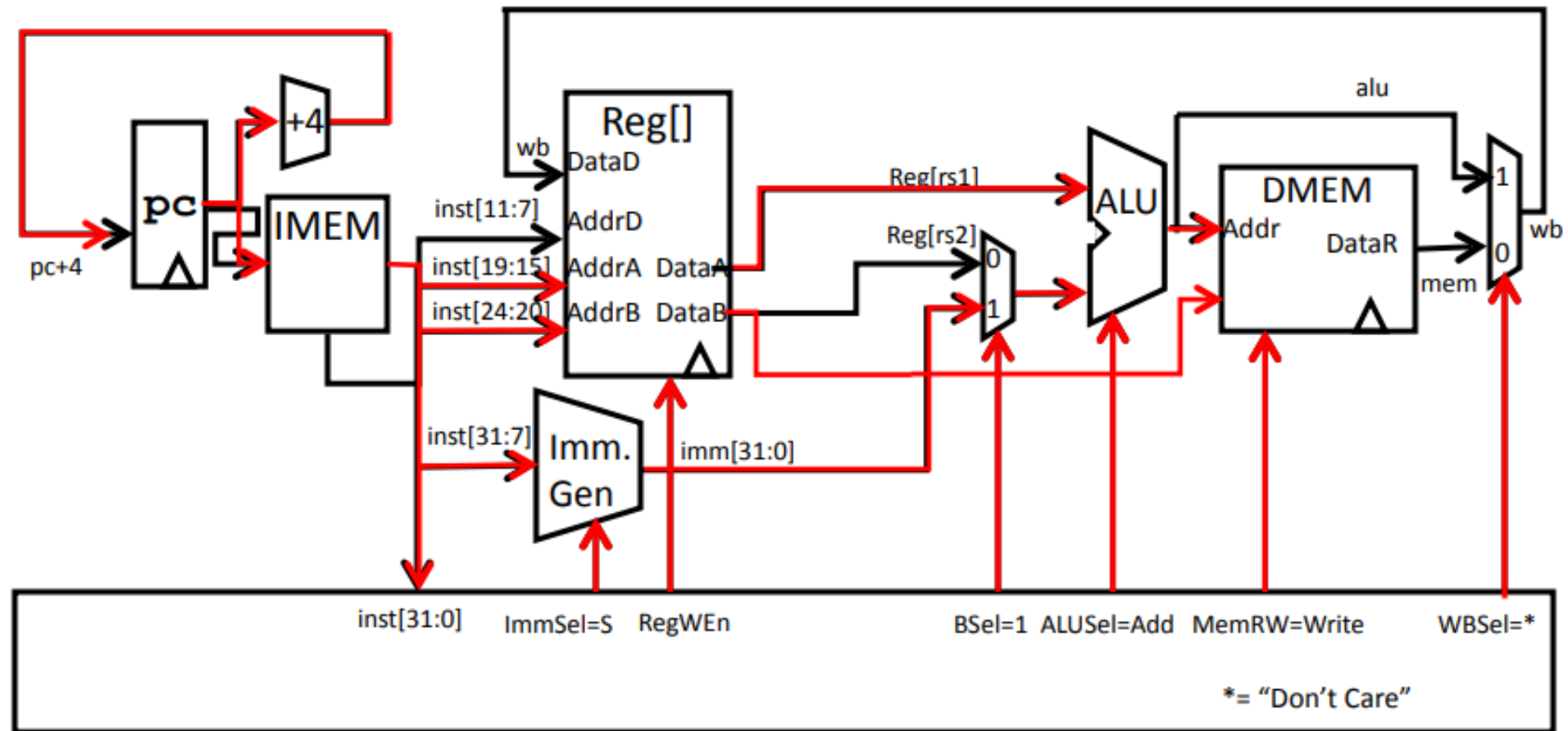
- Supporting the narrower loads requires additional circuits to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.
- Compare the load instructions with previous i-type instructions

Implementing the **store(sw)** instruction

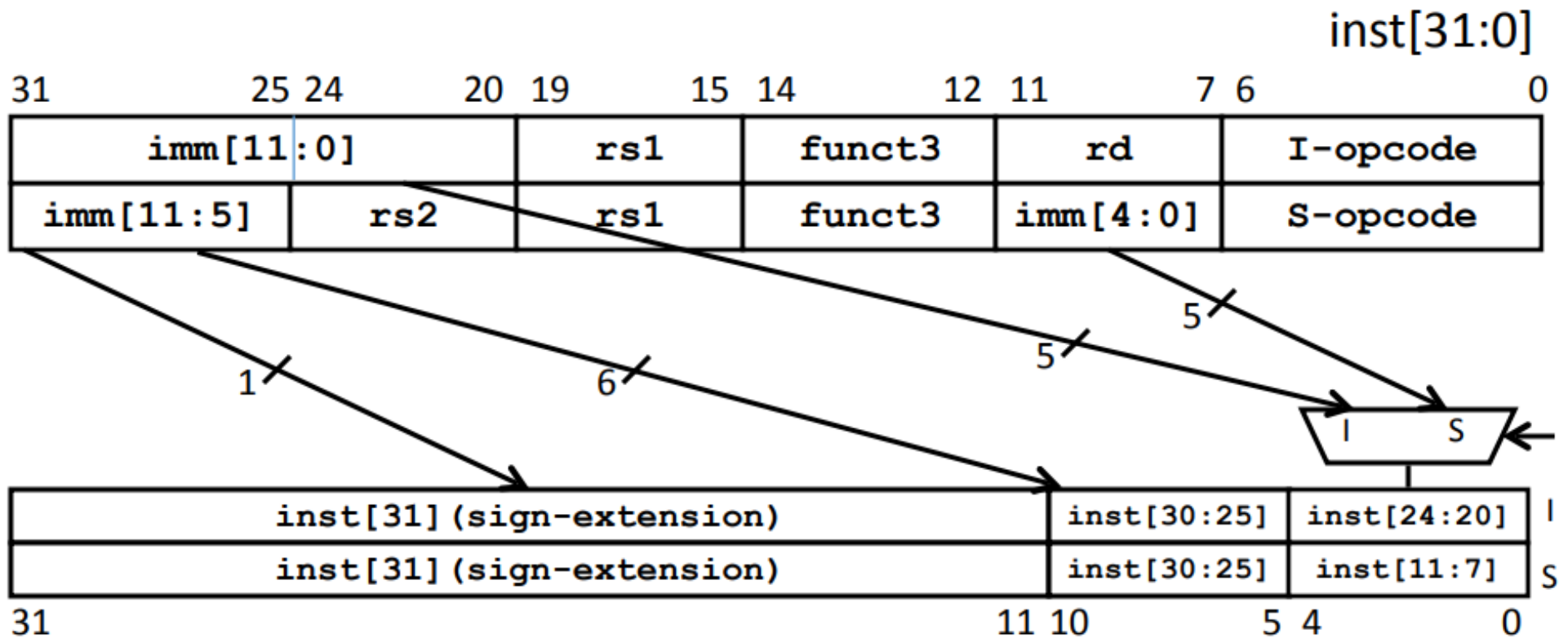
- `sw x14, 8(x2)`



Implementing the **store(sw)** instruction



I-Type, S-Type Immediate Generator



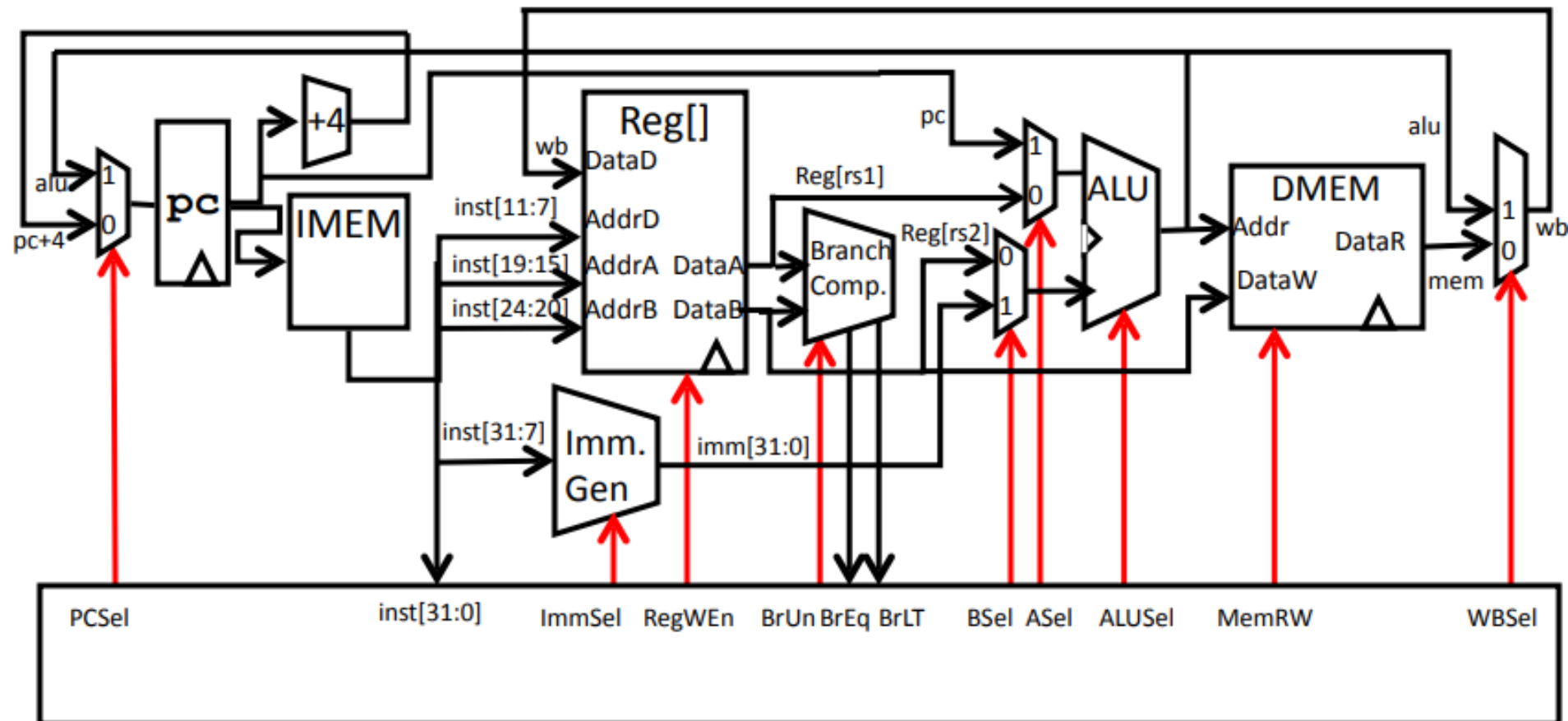
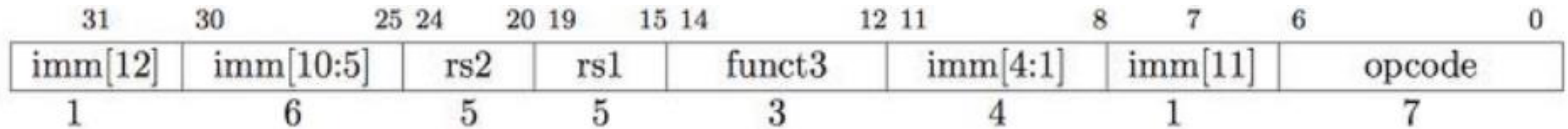
- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction

Implementing the **branch(beq)** instruction

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
--------------	-----	-----	-----	-------------	---------	-----

- B-Type is mostly same as S-Type , with two register sources (rs1/rs2) and a 12-bit immediate
 - But now immediate represents values -4096 to +4094 in 2-byte increments ,
 - The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
- i.e., $PC = PC + \text{immediate} * 2$

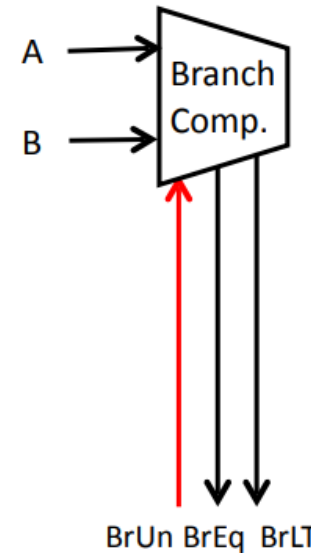
Implementing the **branch(beq)** instruction



Branch Comparator

- B-Type are decoded by func3 only

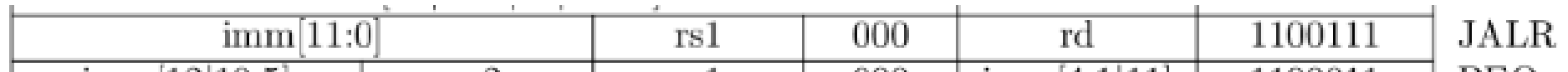
- **Question:** Why B-Type use such “weird” immediate format?



- BrEq = 1, if A=B
- BrLT = 1, if A < B
- BrUn =1 selects unsigned comparison for BrLT, 0=signed
- BGE branch: A >= B, if !(A<B)

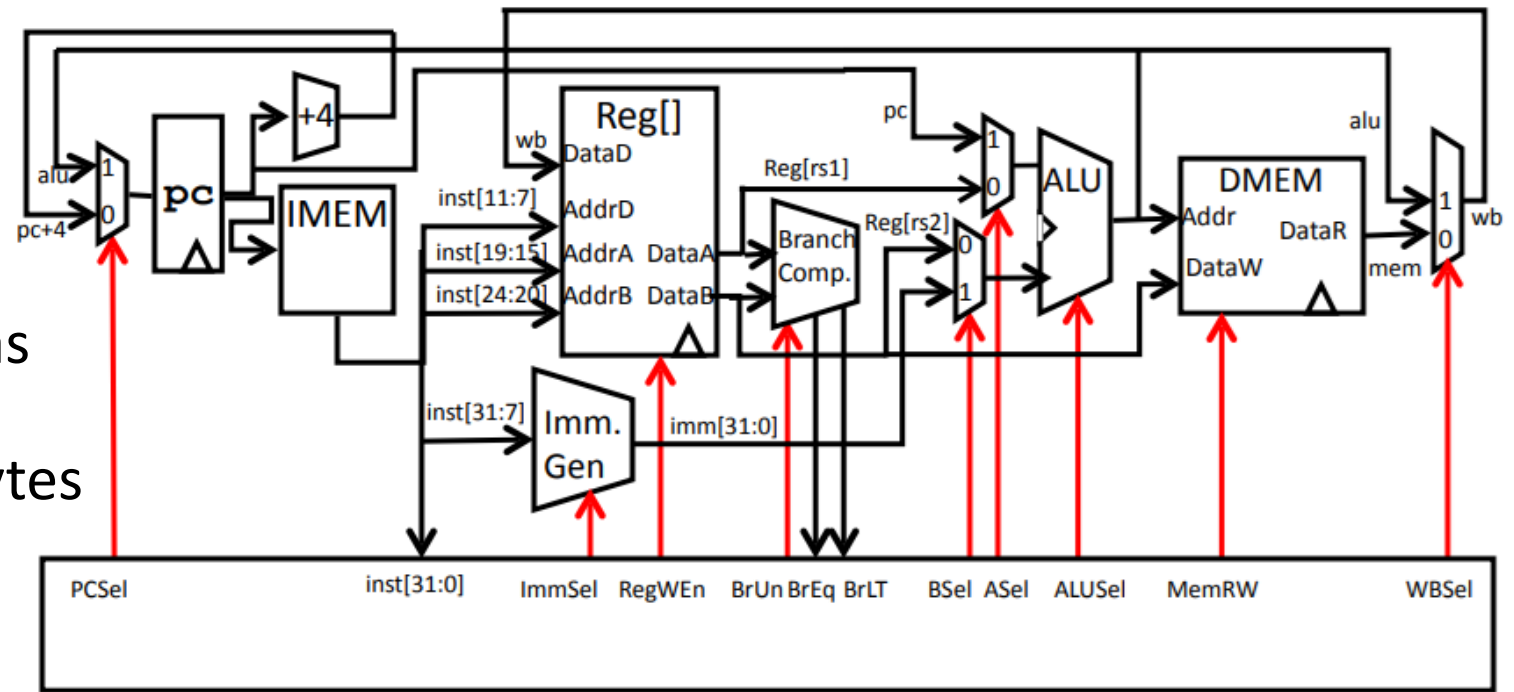
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU
imm[11:0]	rs1		000	rd	0000011	LR

Implementing the **jlr** instruction

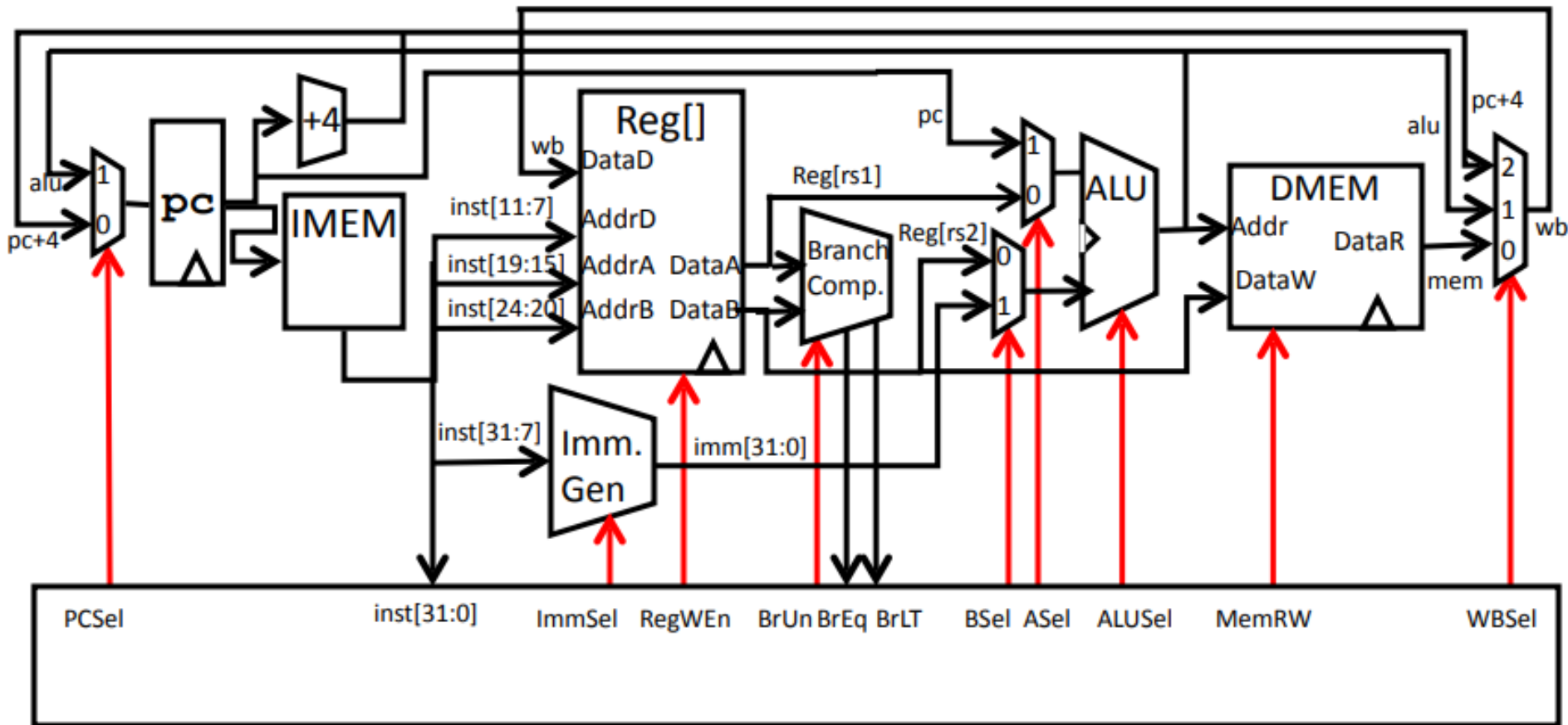


- jalr rd, rs, imme
 - Writes PC+4 to Reg[rd] (return address)
 - Sets PC = Reg[rs1] + immediate
 - Uses same immediates as arithmetic and loads
 - no multiplication by 2 bytes

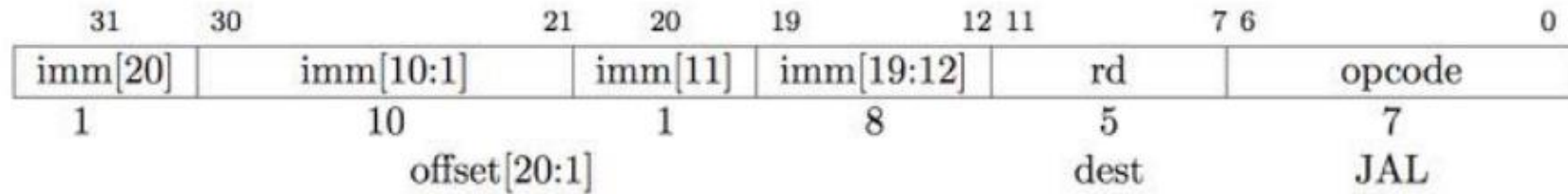
- Previous Arch



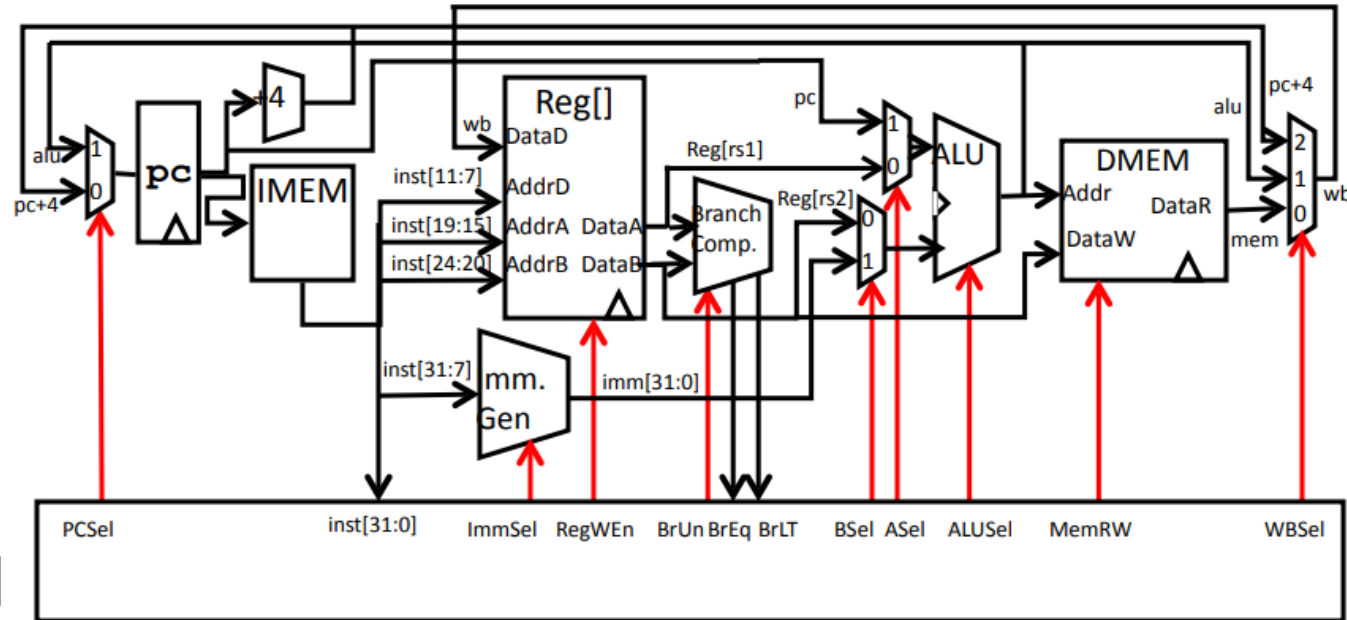
Implementing the **jalr** instruction



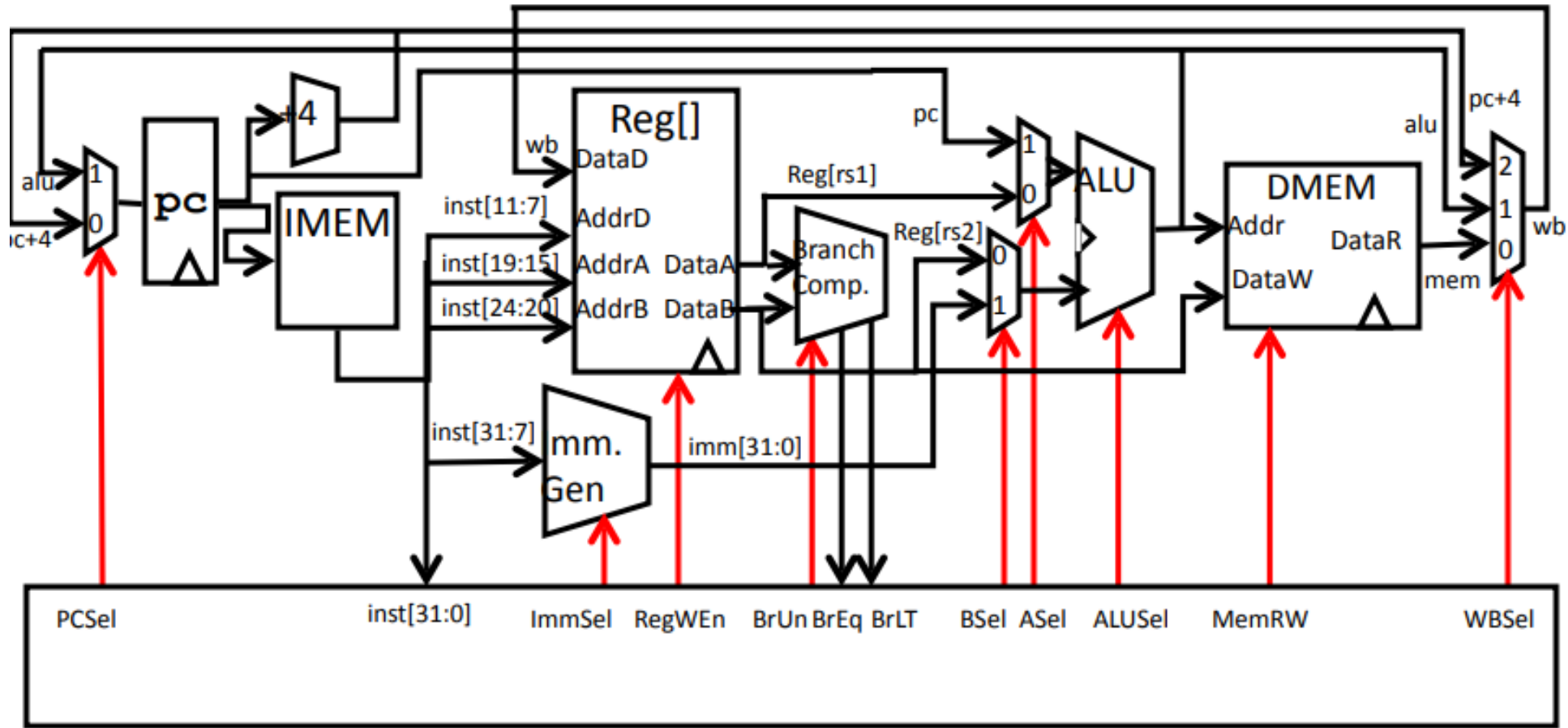
Implementing the **jal** instruction



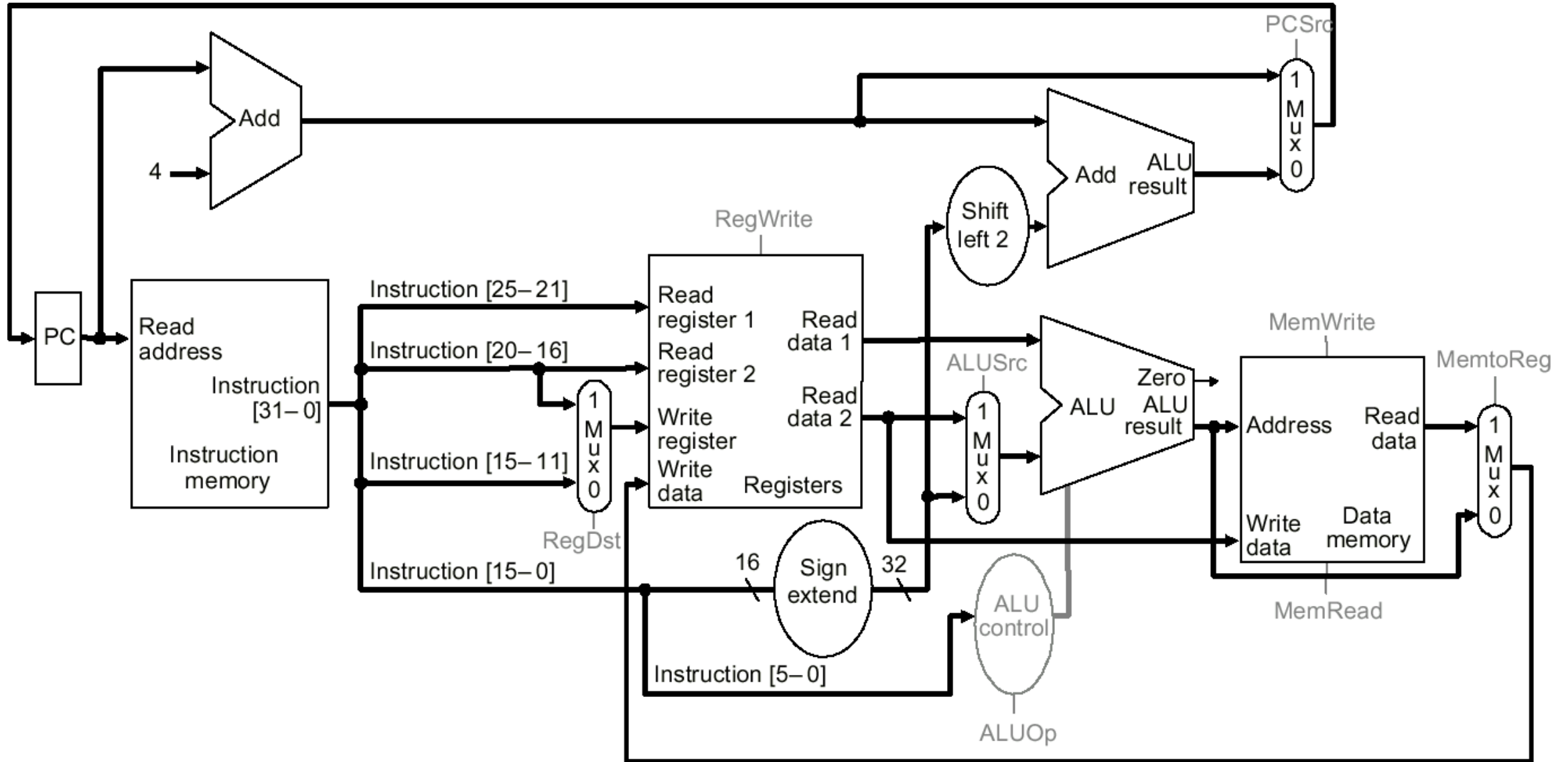
- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within ± 219 locations, 2 bytes apart
 - ± 218 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost



Overall Single Cycle RV32I Datapath



Compare with MIPS (Single Cycle)



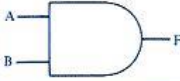
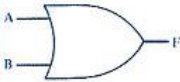
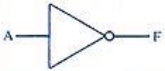
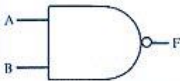

Pipeline - I

What's Pipeline ?

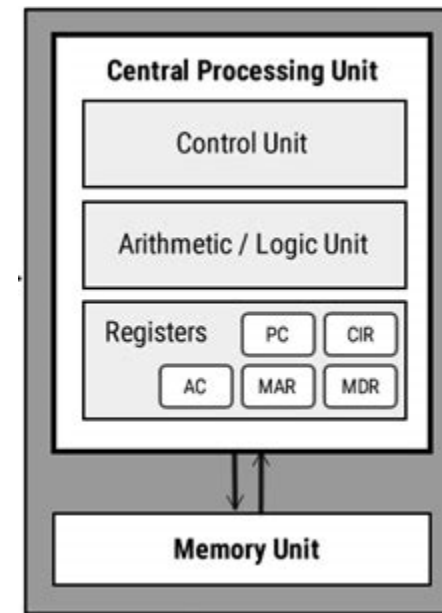
- Example: How many courses you need to take before designing a processor?



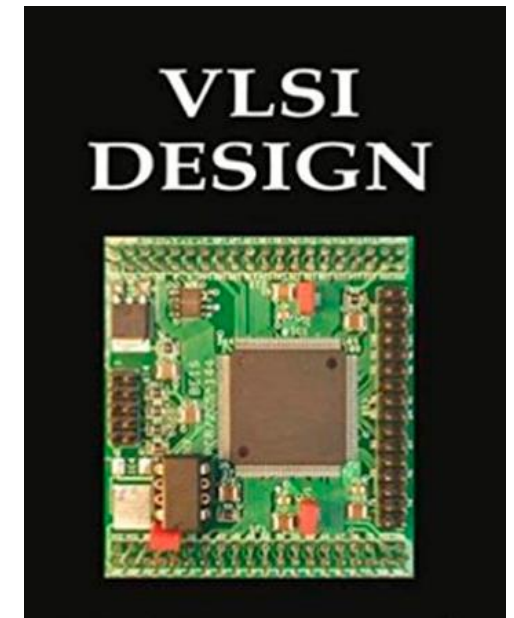
First Year: C Programming

Name	Graphic Symbol	Algebraic Function	Truth Table															
AND		$F = A \cdot B$ or $F = AB$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	F	0	0	0	0	1	0	1	0	0	1	1	1
A	B	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = A + B$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	F	0	0	0	0	1	1	1	0	1	1	1	1
A	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$F = \bar{A}$ or $F = A'$	<table><tr><th>A</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	F	0	1	1	0									
A	F																	
0	1																	
1	0																	
NAND		$F = (\overline{AB})$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	1	1	0	1	1	1	0
A	B	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (\overline{A + B})$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	0	1	0	0	1	1	0
A	B	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

Second Year: Digital Logic



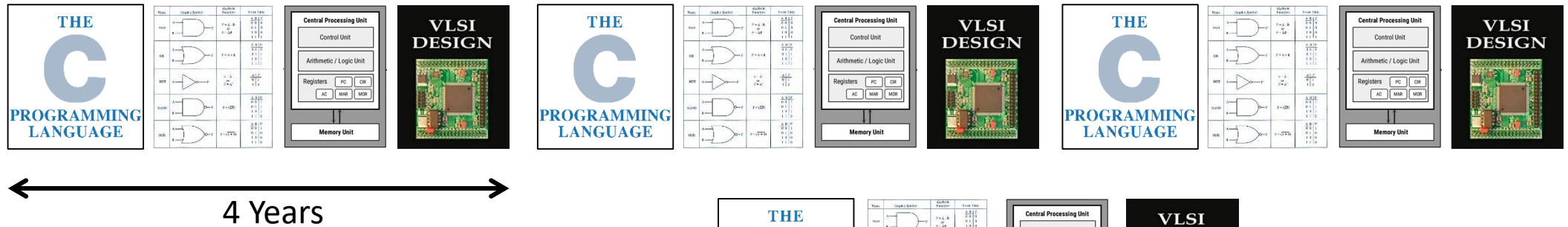
Third Year: CA



Forth Year: VLSI Design

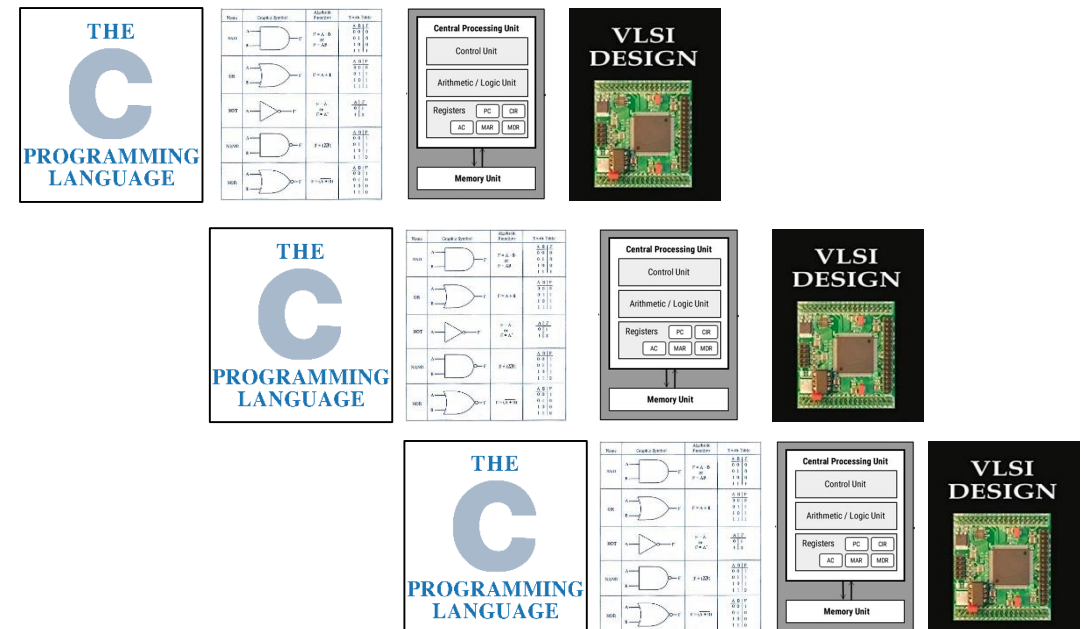
Serial Vs. Pipeline

- Option 1: Serial

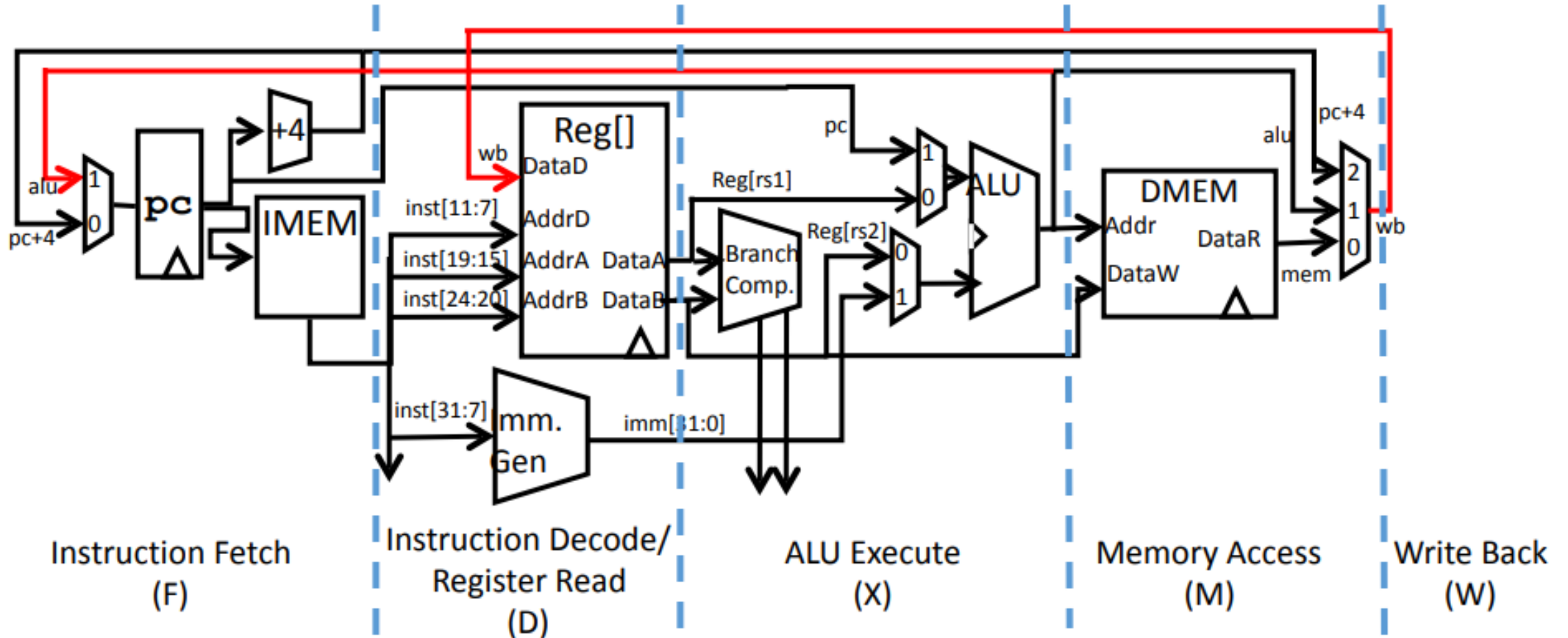


- Option 2: Pipeline

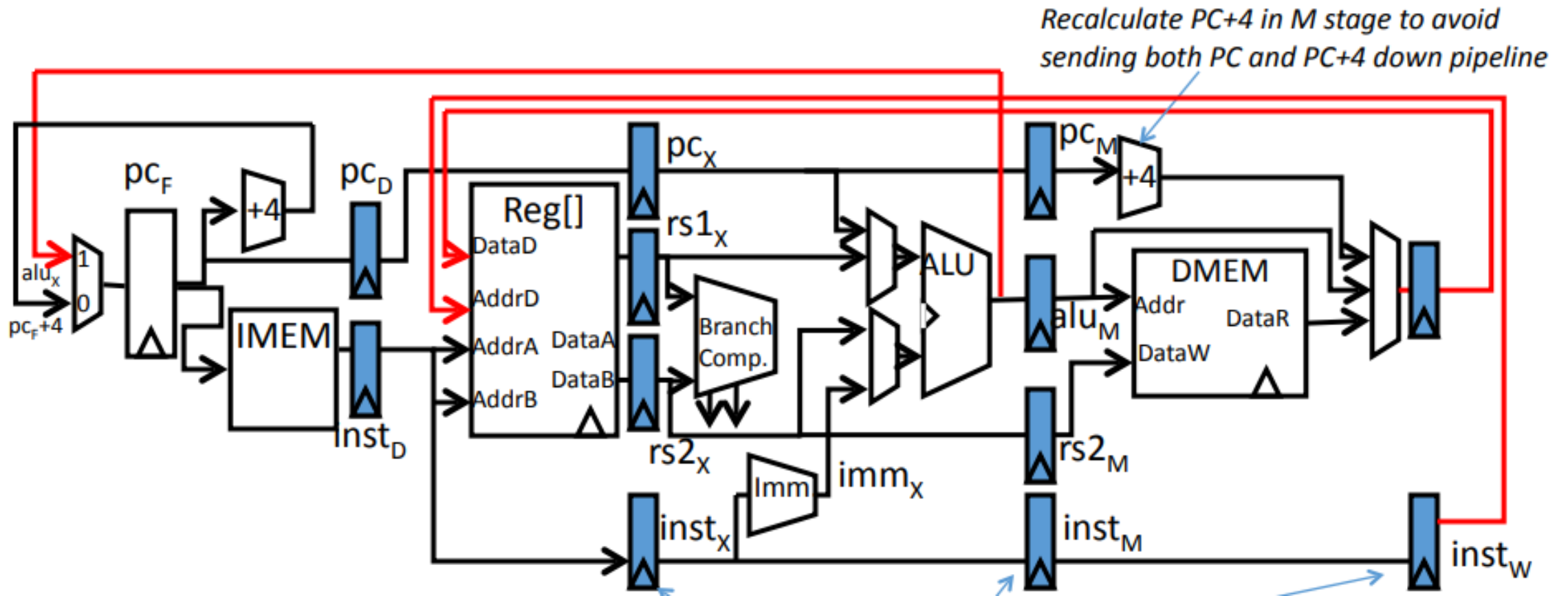
- Latency: Time from entering to graduation, Both are 4 years
- Throughput: Pipeline is 4x better than serial



Step 1: Split Stages from Single Cycle Design

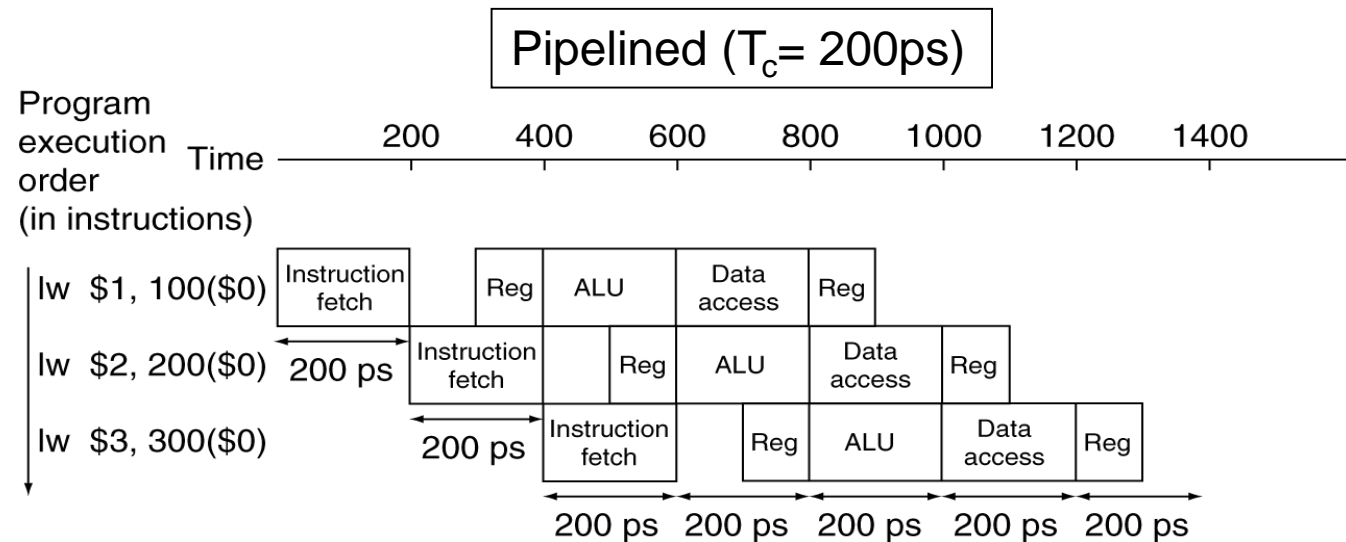
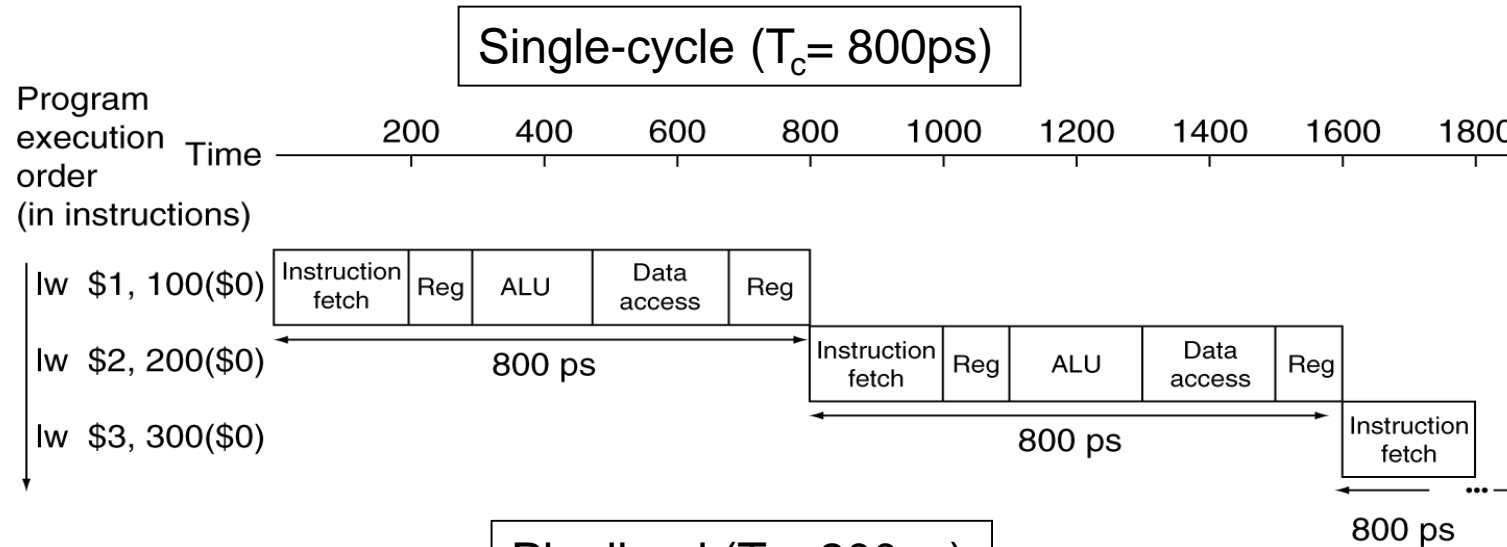


Step 1: Adding Registers



- Registers number = clock load, power efficiency ?

Pipeline Performance



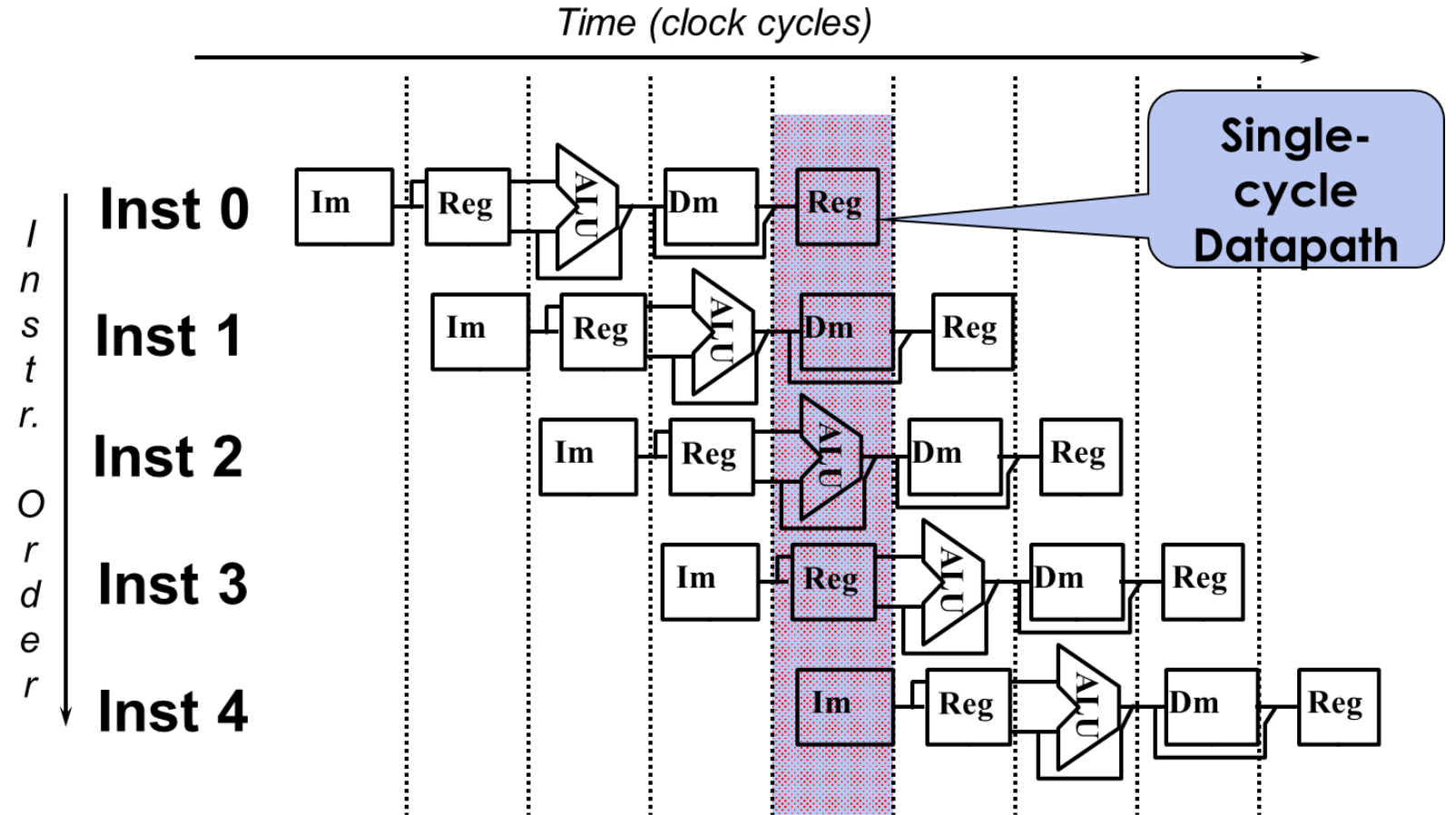
Latency is even longer in pipelined designs

The Utilization increasing

- Throughput

GOPS
TOPS

- Xxx operation
per second



Pipeline Preview

Again: do not forget your homework.

Hazards Ahead

