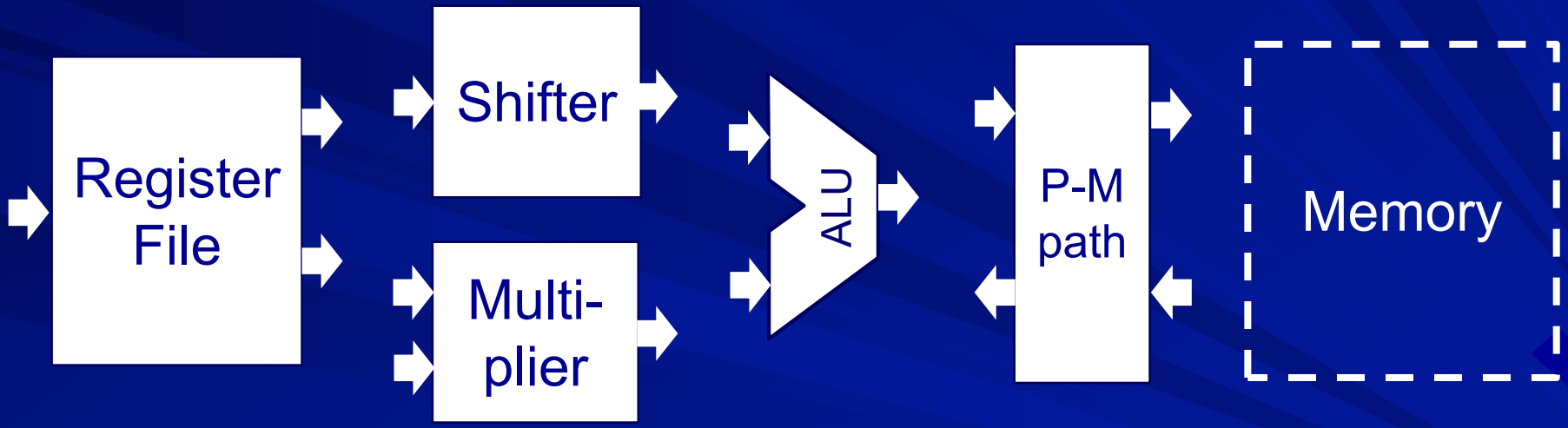
COL216
Computer Architecture

Designing a processor -
A simple approach
3rd February, 2022

Datapath major blocks

Some additional blocks are required to glue these together



Later we will see how instruction fetching, instruction decode, operand access and state updation are done

Multiply instructions

- MUL Multiply
- MLA Multiply Accumulate

32 bit result

64 bit result

- SMULL Signed Multiply Long
- SMLAL Signed Multiply Accumulate Long
- UMULL Unsigned Multiply Long
- UMLAL Unsigned Multiply Accumulate Long

4 x 4 multiplication

$a_3\ a_2\ a_1\ a_0$ x b_0

$a_3\ a_2\ a_1\ a_0$ x b_1

$a_3\ a_2\ a_1\ a_0$ x b_2

$a_3\ a_2\ a_1\ a_0$ x b_3

4x4 unsigned

0 0 0 a_3 a_2 a_1 a_0	x b_0
0 0 a_3 a_2 a_1 a_0	x b_1
0 a_3 a_2 a_1 a_0	x b_2
a_3 a_2 a_1 a_0	x b_3

4x4 signed

a_3 a_3 a_3 a_3 a_2 a_1 a_0	x b_0
a_3 a_3 a_3 a_2 a_1 a_0	x b_1
a_3 a_3 a_2 a_1 a_0	x b_2
a_3 a_2 a_1 a_0	x $-b_3$

Multiplying a number by -1

$X = x_{n-1}\ x_{n-2} \ldots\ldots\ldots\ldots\ldots x_1\ x_0$

Let x_{k-1} be the rightmost 1 $\quad (1 \le k \le n)$

Then

$$X = x_{n-1}\ x_{n-1} \ldots\ldots\ldots\ldots x_k\ 1\ 0\ .\ .\ 0\ 0$$

$$\overline{X} = \overline{x}_{n-1}\ \overline{x}_{n-2} \ldots\ldots\ldots\ldots \overline{x}_k\ 0\ 1\ .\ .\ 1\ 1$$

$$-X = \underbrace{\overline{x}_{n-1}\ \overline{x}_{n-2} \ldots\ldots\ldots\ldots \overline{x}_k}\ \underbrace{1\ 0\ .\ .\ 0\ 0}$$

left n – k bits complemented

right k bits unchanged

4x4 unsigned

0 0 0 a_3 a_2 a_1 a_0	x b_0
0 0 a_3 a_2 a_1 a_0	x b_1
0 a_3 a_2 a_1 a_0	x b_2
a_3 a_2 a_1 a_0	x b_3

4x4 signed

a_3 a_3 a_3 a_3 a_2 a_1 a_0	x b_0
a_3 a_3 a_3 a_2 a_1 a_0	x b_1
a_3 a_3 a_2 a_1 a_0	x b_2
$\overline{a_3}$ $\overline{a_2}$ a_1 a_0	x b_3

4x4 unsigned

0 0 0	$a_3\ a_2\ a_1\ a_0$	x b_0
0 0 a_3	$a_2\ a_1\ a_0$	x b_1
0 $a_3\ a_2$	$a_1\ a_0$	x b_2
$a_3\ a_2\ a_1$	a_0	x b_3

4x4 signed

$a_3\ a_3\ a_3$	$a_3\ a_2\ a_1\ a_0$	x b_0
$a_3\ a_3\ a_3$	$a_2\ a_1\ a_0$	x b_1
$a_3\ a_3\ a_2$	$a_1\ a_0$	x b_2
$\overline{a_3}\ \overline{a_2}\ a_1$	a_0	x b_3

Same for both

Multiplication in VHDL

Signed multiplication:

```
signal a_s, b_s : signed (31 downto 0);
signal p_s : signed (63 downto 0);
p_s <= a_s * b_s;
```

Unsigned multiplication:

```
signal a_u, b_u : unsigned (31 downto 0);
signal p_u : unsigned (63 downto 0);
p_u <= a_u * b_u;
```

Synthesizing multipliers

Signed multiplication:

```
signal a_s, b_s : signed (31 downto 0);
signal p_s : signed (63 downto 0);
p_s <= a_s * b_s;        -- uses 4 DSP48E1
```

18x18 multiplier

Unsigned multiplication:

```
signal a_u, b_u : unsigned (31 downto 0);
signal p_u : unsigned (63 downto 0);
p_u <= a_u * b_u;        -- uses 4 DSP48E1
```

18x18 multiplier

Combining results

```
signal op1, op2: std_logic_vector (31 downto 0);
signal result: std_logic_vector (63 downto 0);
signal p_s : signed (63 downto 0);
signal p_u : unsigned (63 downto 0);
p_s <= signed (op1) * signed (op2);
p_u <= unsigned (op1) * unsigned (op2);
result <= std_logic_vector(p_s) when instr = smull
          else std_logic_vector(p_u);
```

-- uses 7 DSP48E1, but fails during routing

Another way

- Use a signed multiplier to do unsigned multiplication
- 0-extend the operands
- Signed and unsigned interpretations of these are same now.

Another way

```
signal op1, op2: std_logic_vector (31 downto 0);
signal result: std_logic_vector (63 downto 0);
signal p_s: signed (65 downto 0);
signal x1, x2: std_logic;
x1 <= op1(31) when instr = smull else ‘0’;
x2 <= op2(31) when instr = smull else ‘0’;
p_s <= signed (x1 & op1) * signed (x2 & op2);
result <= std_logic_vector(p_s (63 downto 0));
```

-- uses 4 DSP48E1 !

Processor Design :

Going from

Instruction Set Architecture (ISA)

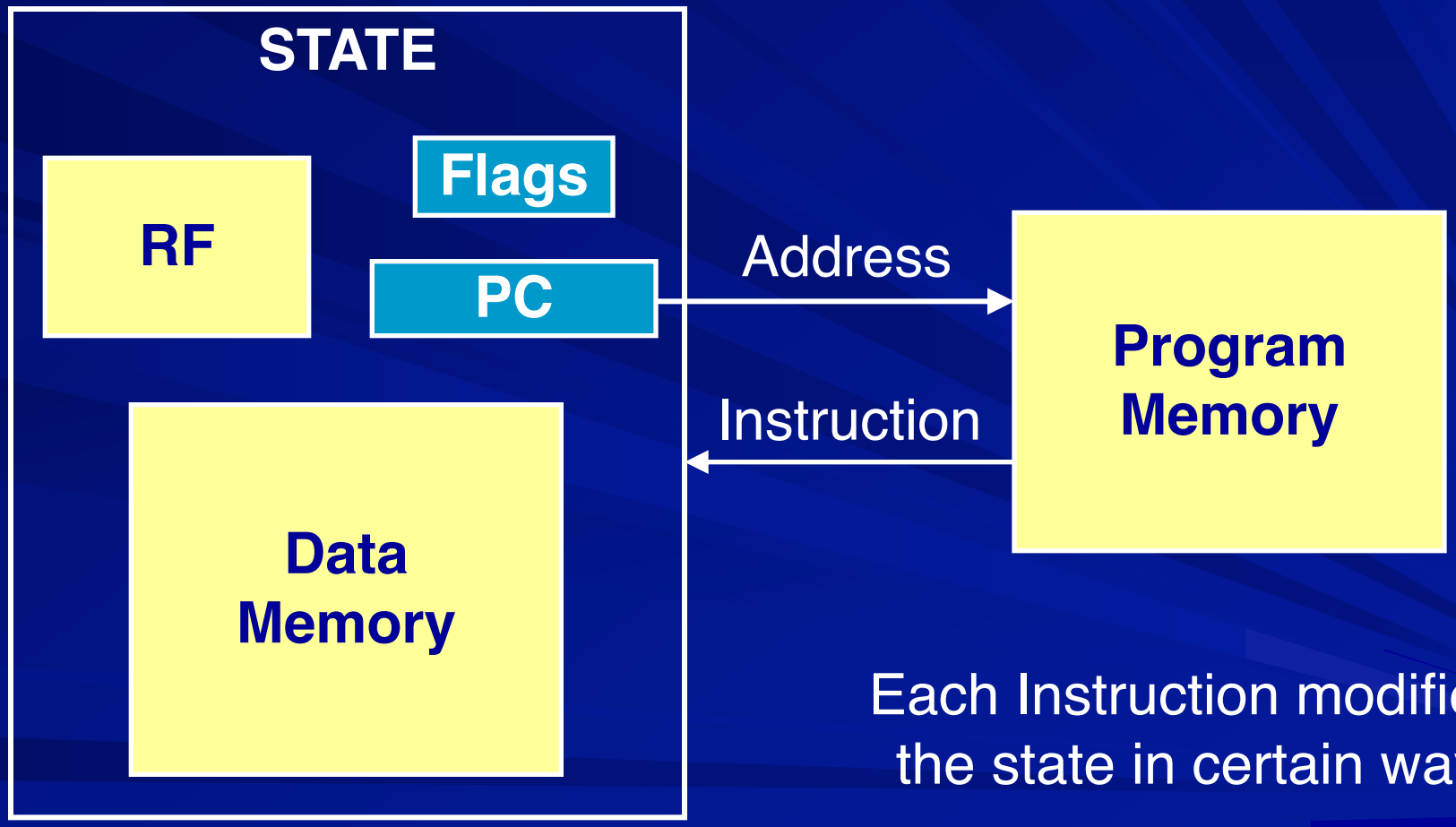
Lowest level visible to a programmer

to

Micro Architecture

High level view of hardware

The Abstract Machine



Each Instruction modifies the state in certain way

How instructions are executed?

- Look at the state (PC, Flags, RF, Memory)
- Based on current state, decide what should the next state be
- Update the state (PC, Flags, RF, Memory)

More details

- Use the program counter (PC) to supply instruction address
- Get the instruction from Memory
- Read registers
- Use the instruction to decide exactly what to do
- Update PC, registers, Flags, Memory

Some basic questions

- Hardware resources:
  - What building blocks are to be used to execute the instructions?
- Timings:
  - What is the overall approach to be followed for timing the instructions?

- The two are interlinked
- The answer depends upon the design goals

Instruction timings

- Timings within individual instructions
- Timings across instructions

Timings within

- Instruction execution involves many actions
- These actions are timed by clock cycles
- Number of cycles per instruction
  - Each instruction is done in a single cycle
  - Instructions may take multiple cycles
    - same number of cycles for all instructions
    - some may take fewer cycles some may take more

Timings across

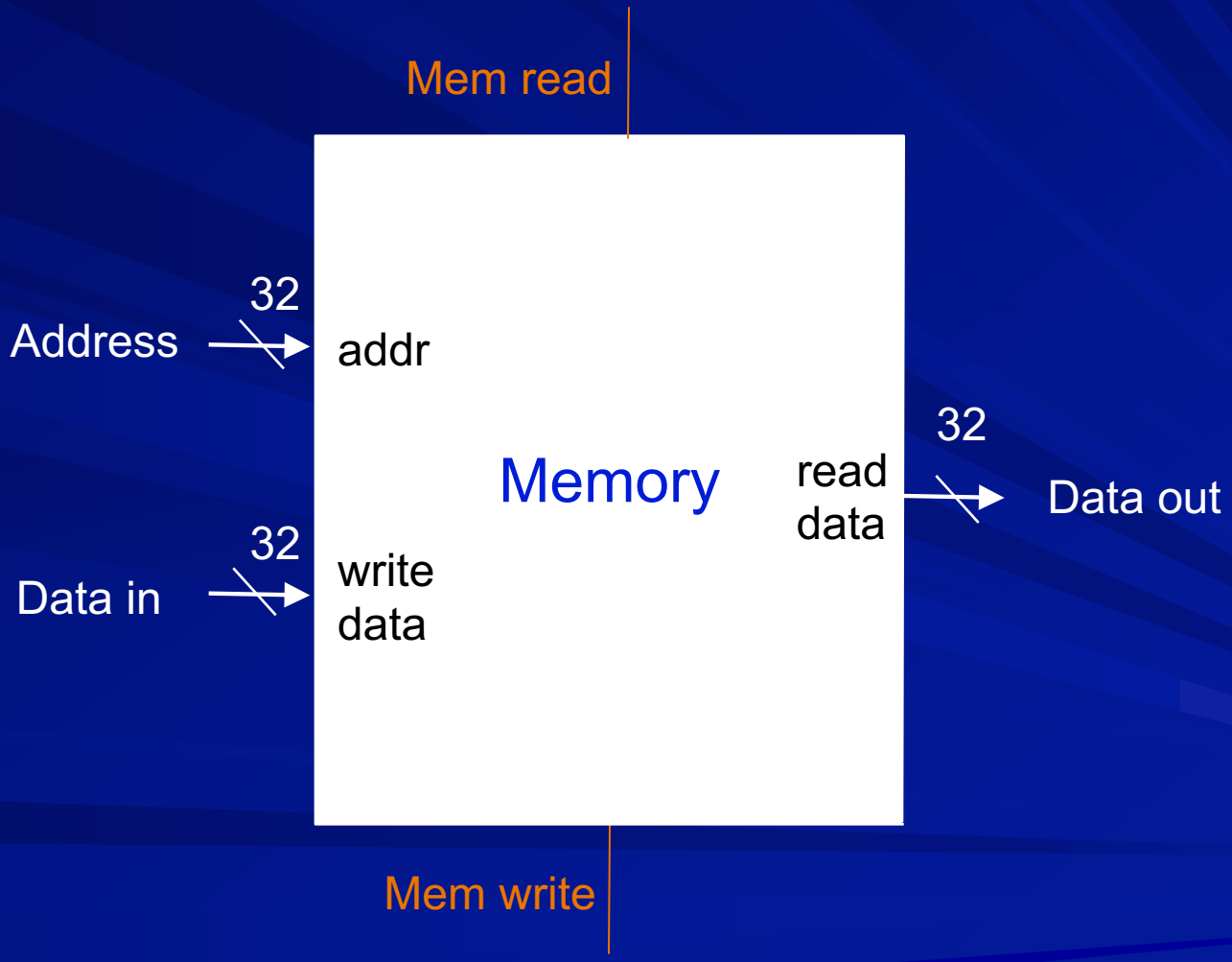
- When one instruction finishes, only then another instruction starts or execution of instructions can overlap
- Only one instruction starts at a time or multiple instructions can start concurrently
- Instructions start in strict program order or the order can be changed

Choices for first design

- Simplest design, not necessarily best in terms of speed, cost or power consumption
- Instruction subset: {add, sub, cmp, mov, ldr, str, b, beq, bne}

- Single cycle for every instruction
- No instruction overlap, no concurrency
- Execution in strict program order

Memory


Mem read
32
Address
addr
Memory
32
read
data
Data out
32
Data in
write
data
Mem write

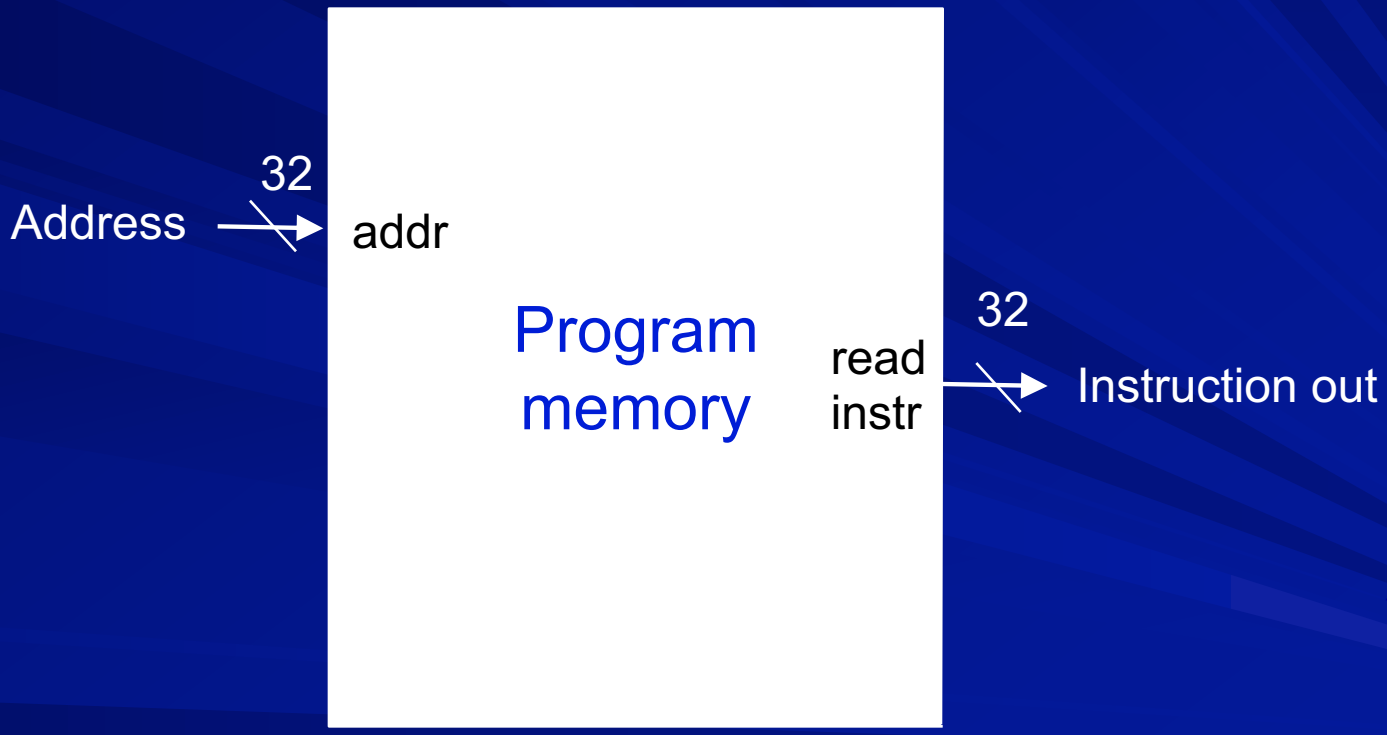
Program and data memory

- Separation of program and data memory

  allows

- accessing instruction and data within same cycle

Program memory

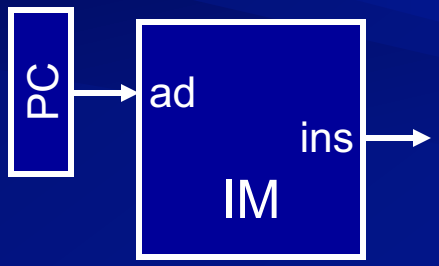


Building datapath

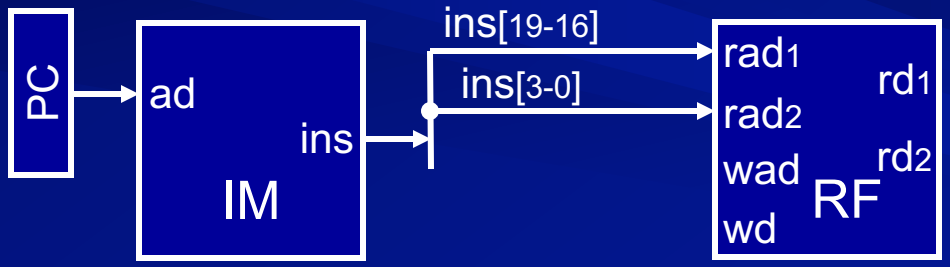
Actions for DP instructions

- fetch instruction
- access the register file
- pass operands to ALU
- pass result to register file
- increment PC

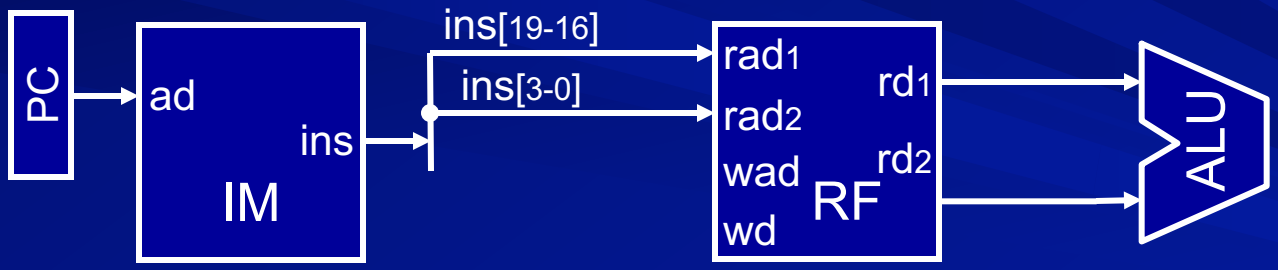
Fetching instruction



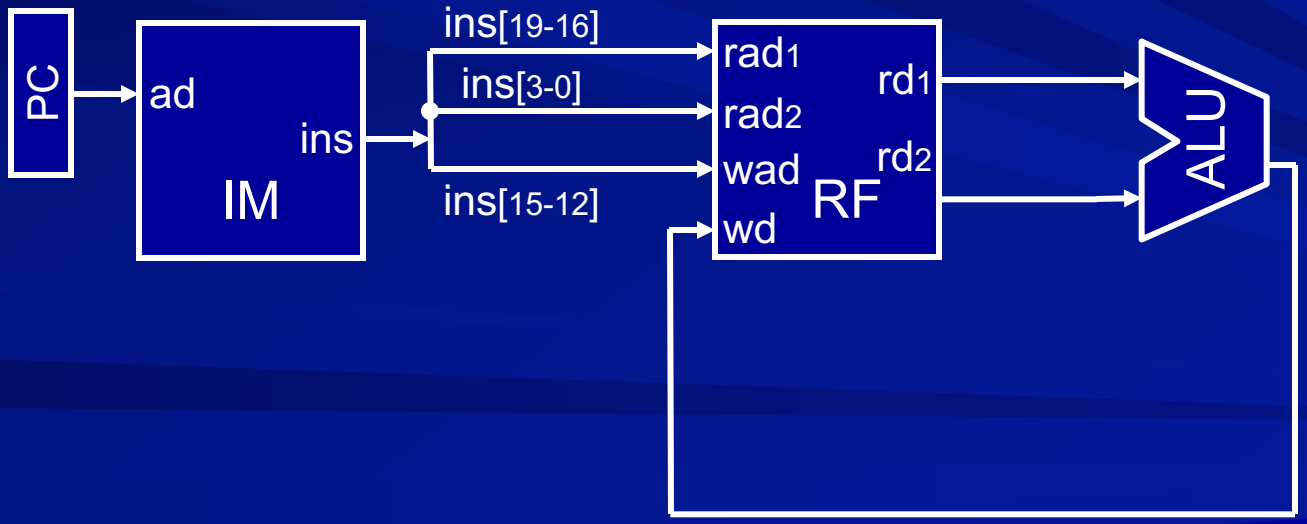
Accessing RF



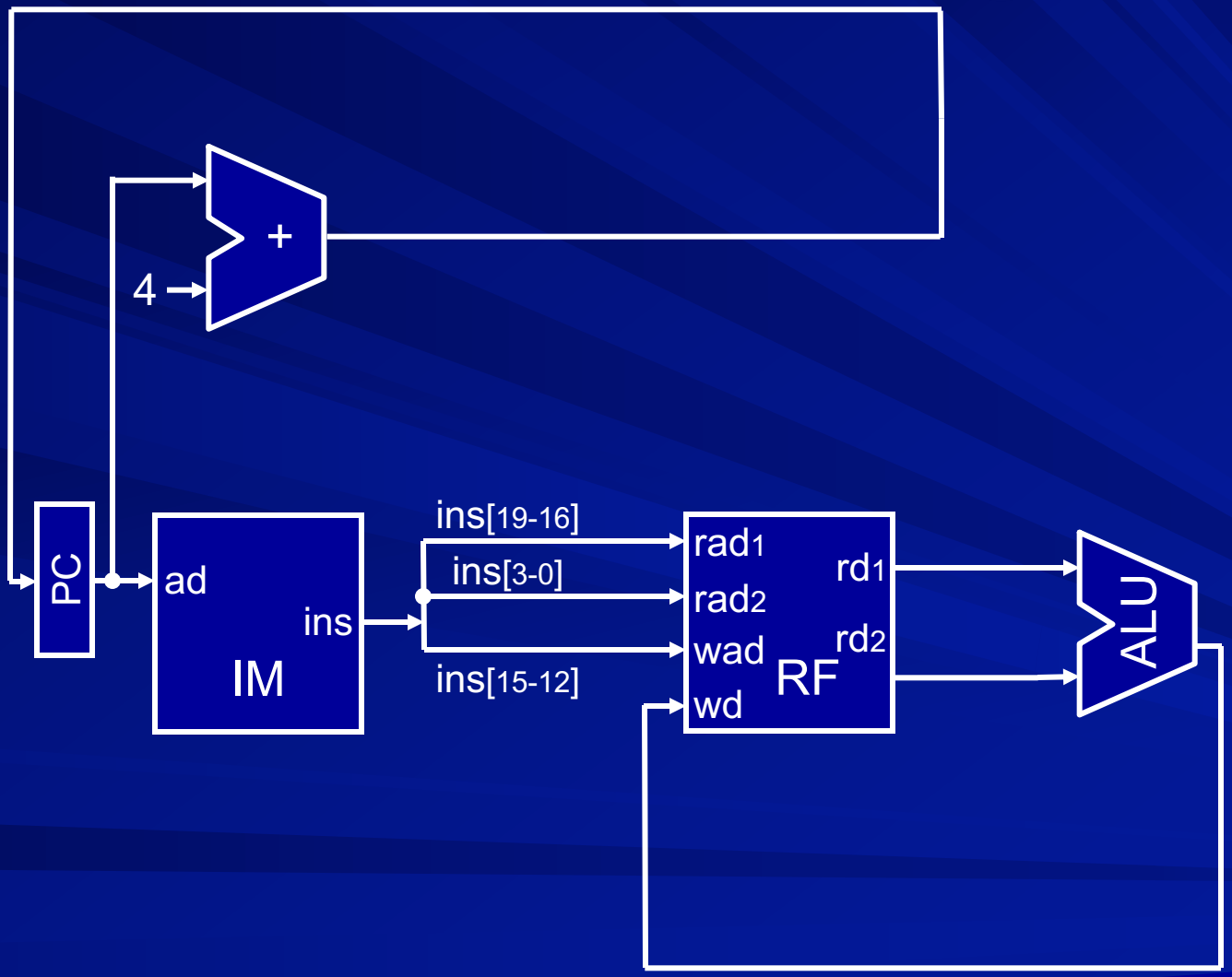
Passing operands to ALU



Passing the result to RF



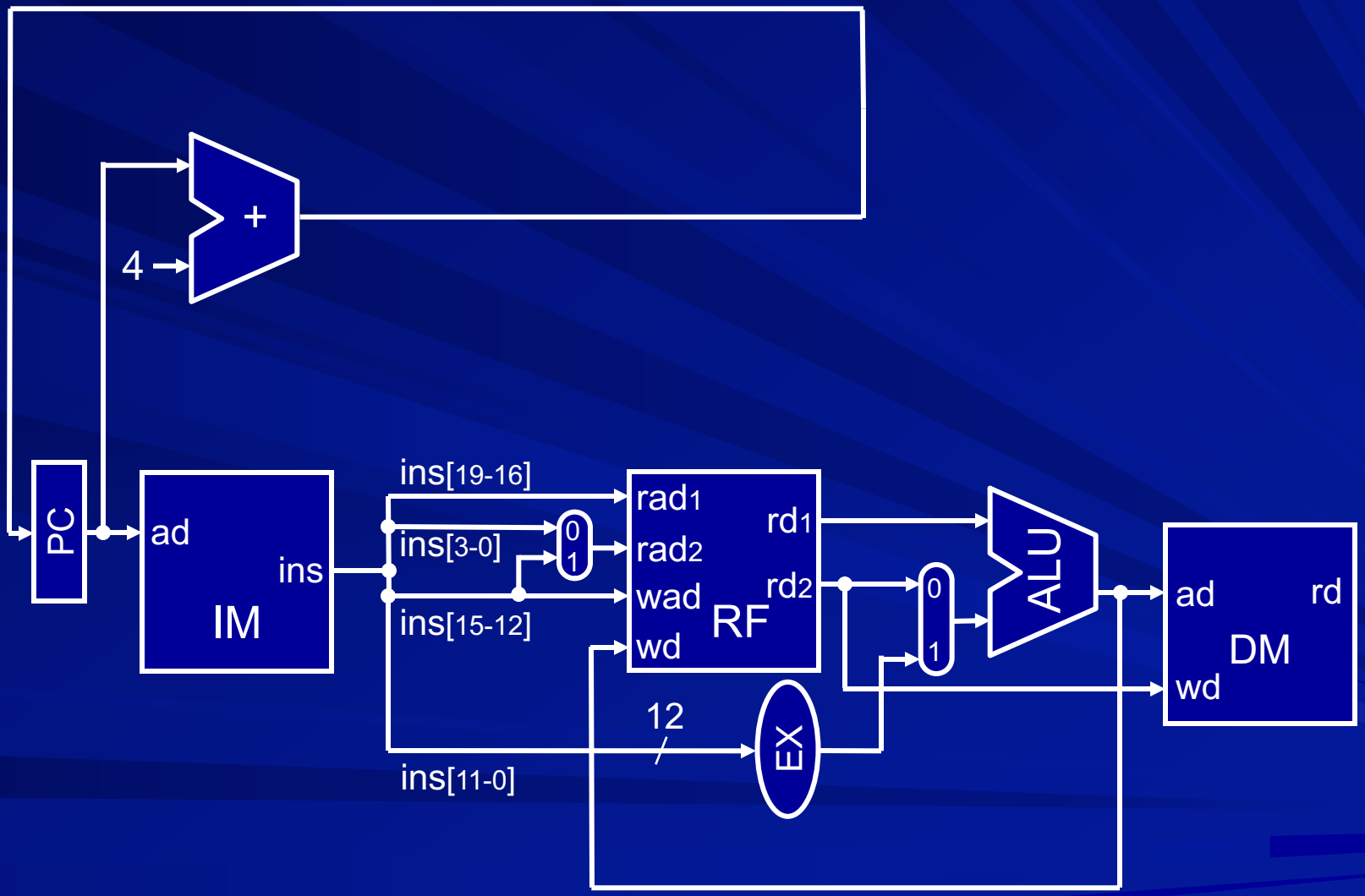
Incrementing PC



Actions for str instructions

- fetch instruction
- access the register file
- compute address in ALU
- write data into memory
- increment PC

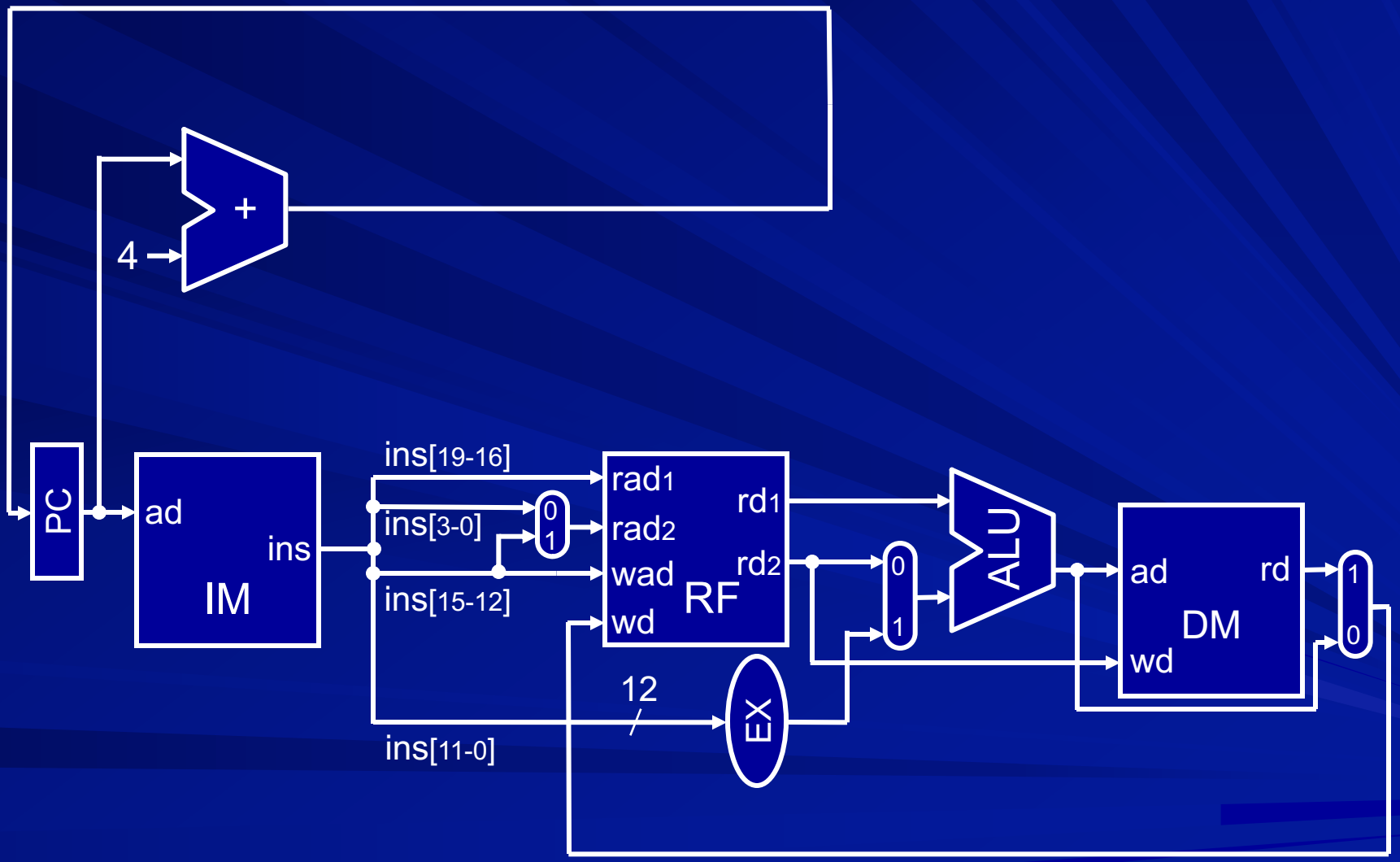
Adding “str” instruction



Actions for ldr instructions

- fetch instruction
- access the register file
- compute address in ALU
- read data from memory
- put data in register file
- increment PC

Adding “ldr” instruction

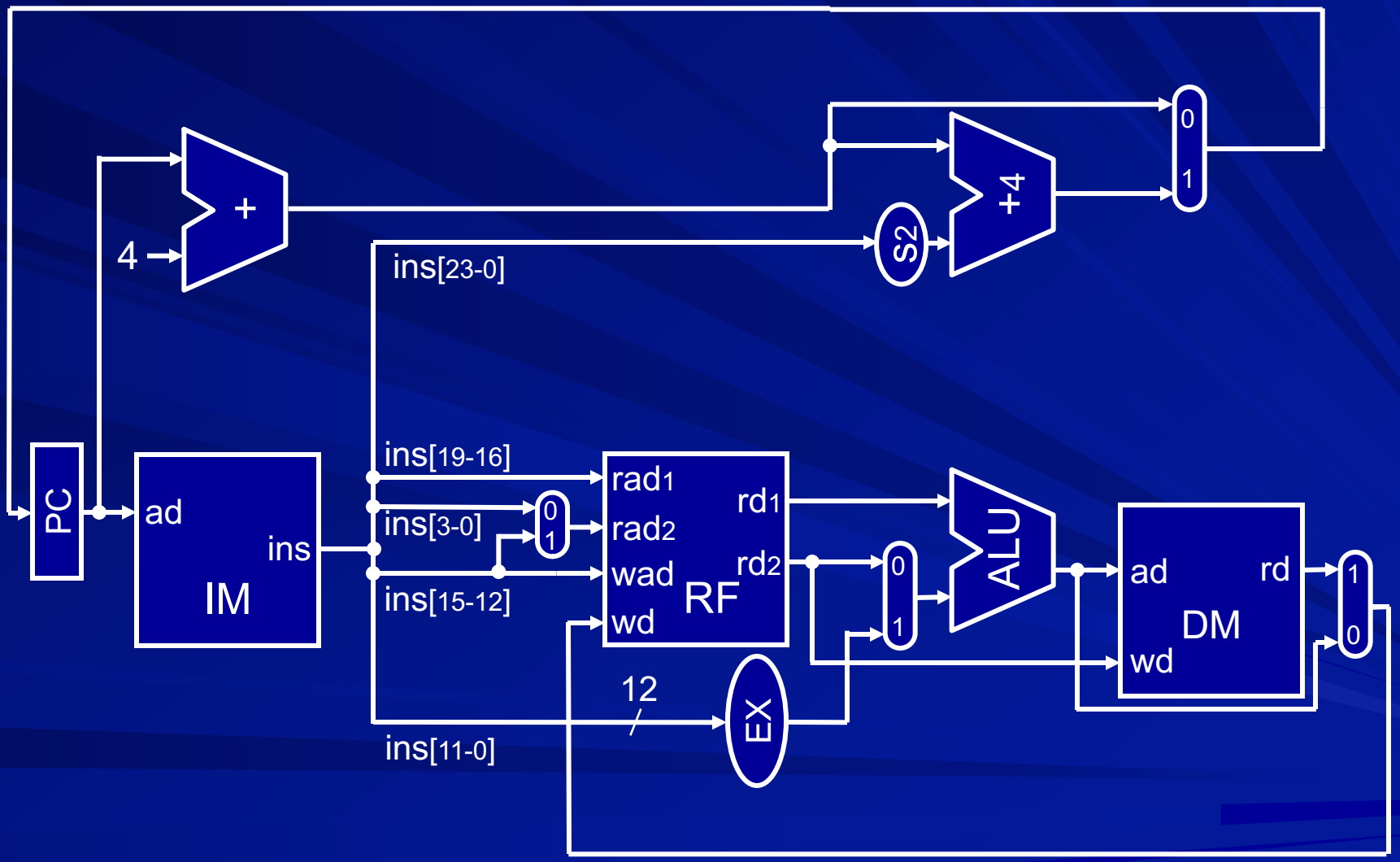


Actions for branch instruction

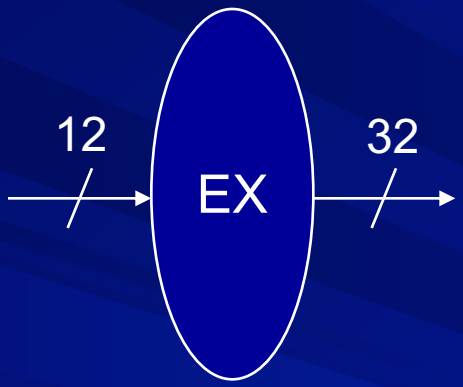
- fetch instruction

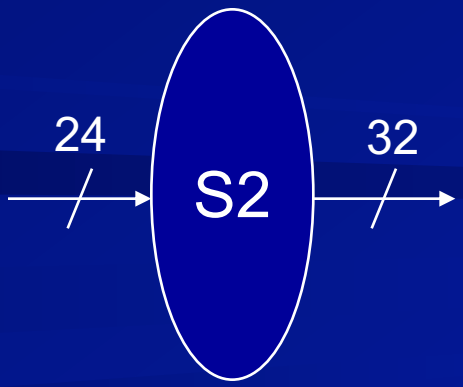
- compute target address
- transfer address to pc

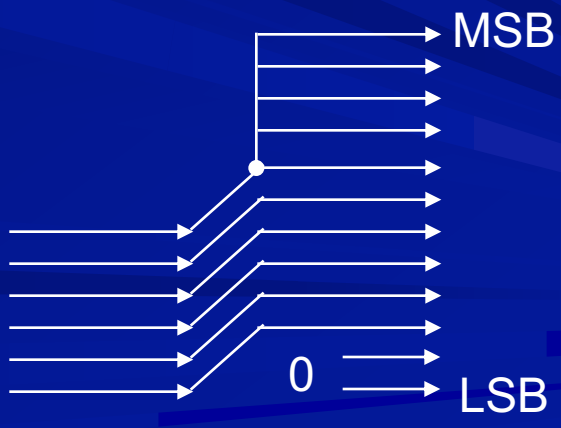
Adding “b” instruction



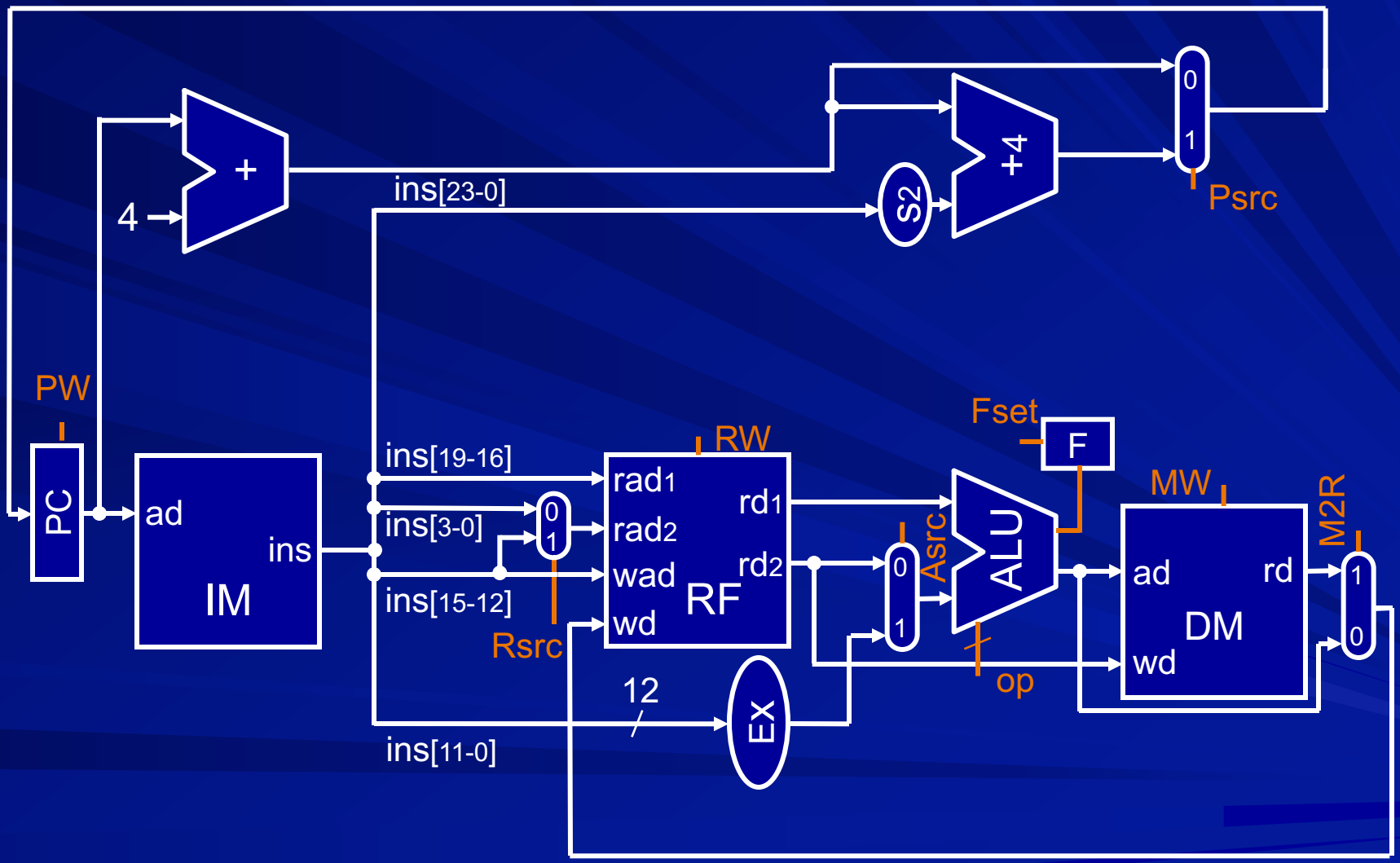
Extending offsets







Single cycle Datapath



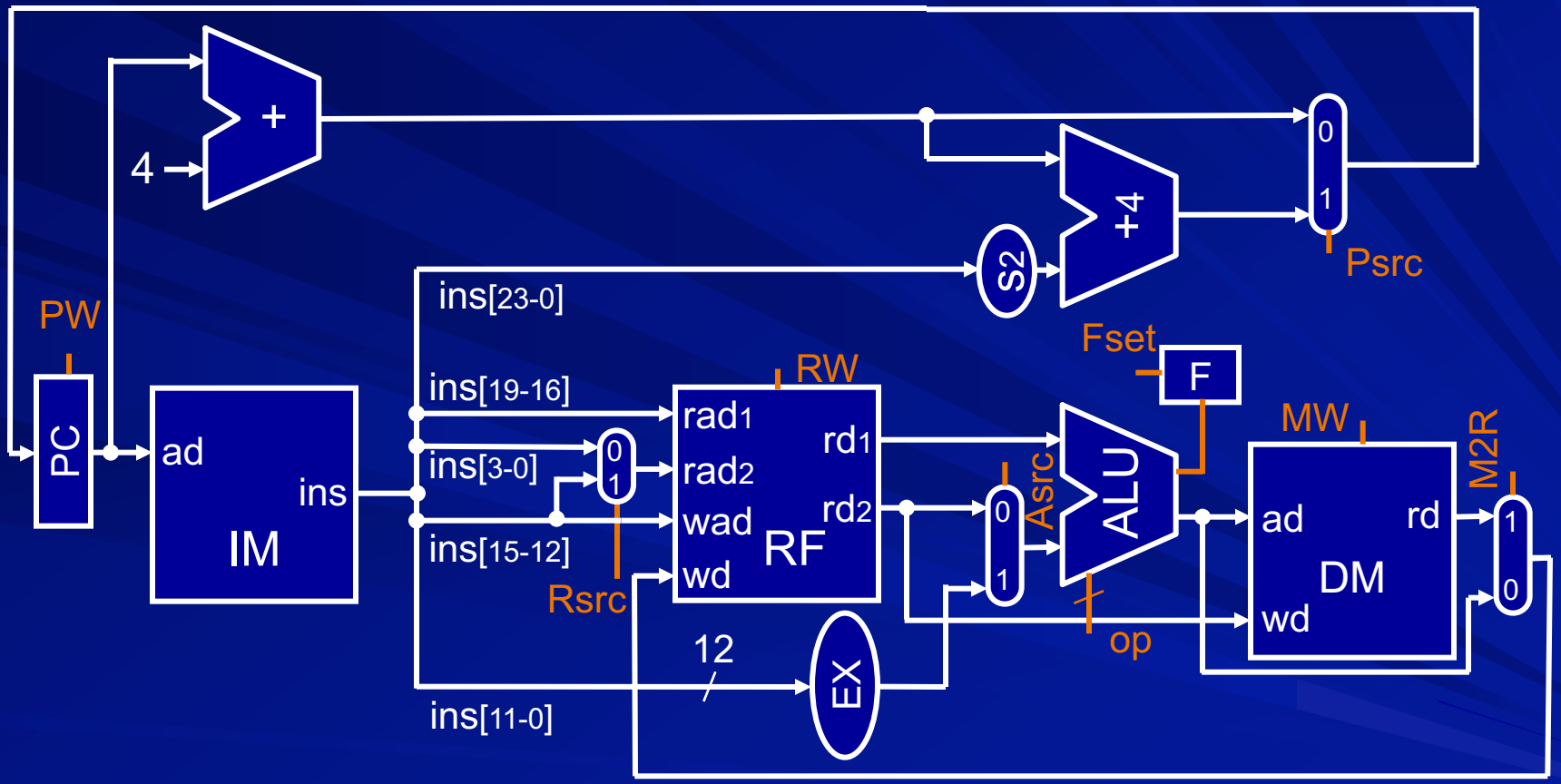
Problems with single cycle design

- Slowest instruction pulls down the clock frequency
- Resource utilization is poor
- There are some instructions which are impossible to be implemented in this manner

Analyzing performance

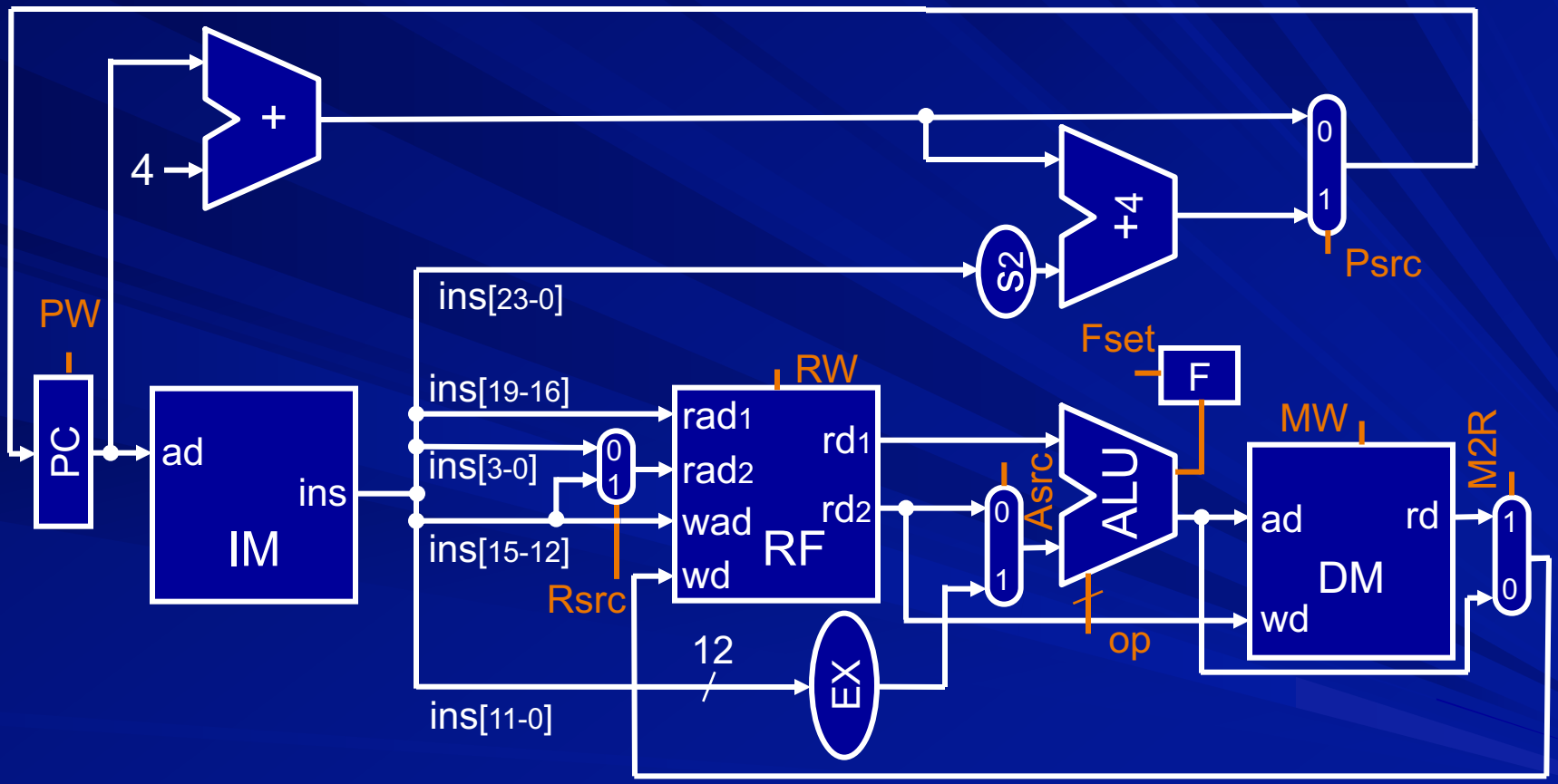
	Component delays
Register	0
Adder	t_+
ALU	t_A
Multiplexer	0
Register file	t_R
Program memory	t_I
Data memory	t_M
Bit manipulation components	0

Delay for {add, sub, …}



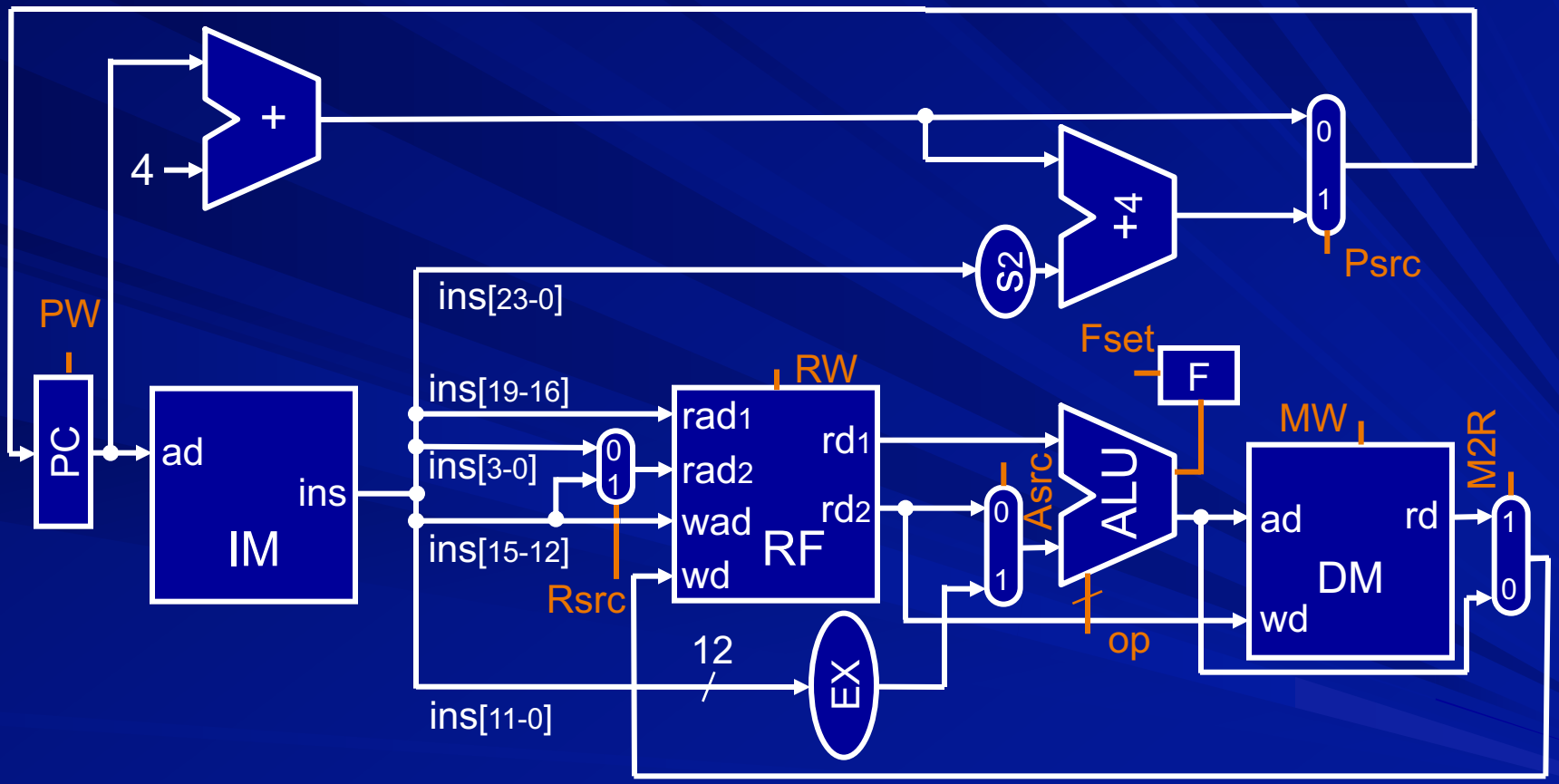
$$\max\left\{\begin{array}{c} t_{+} \\ t_I + t_R + t_A + t_R \end{array}\right.$$

Delay for {cmp, tst, …}



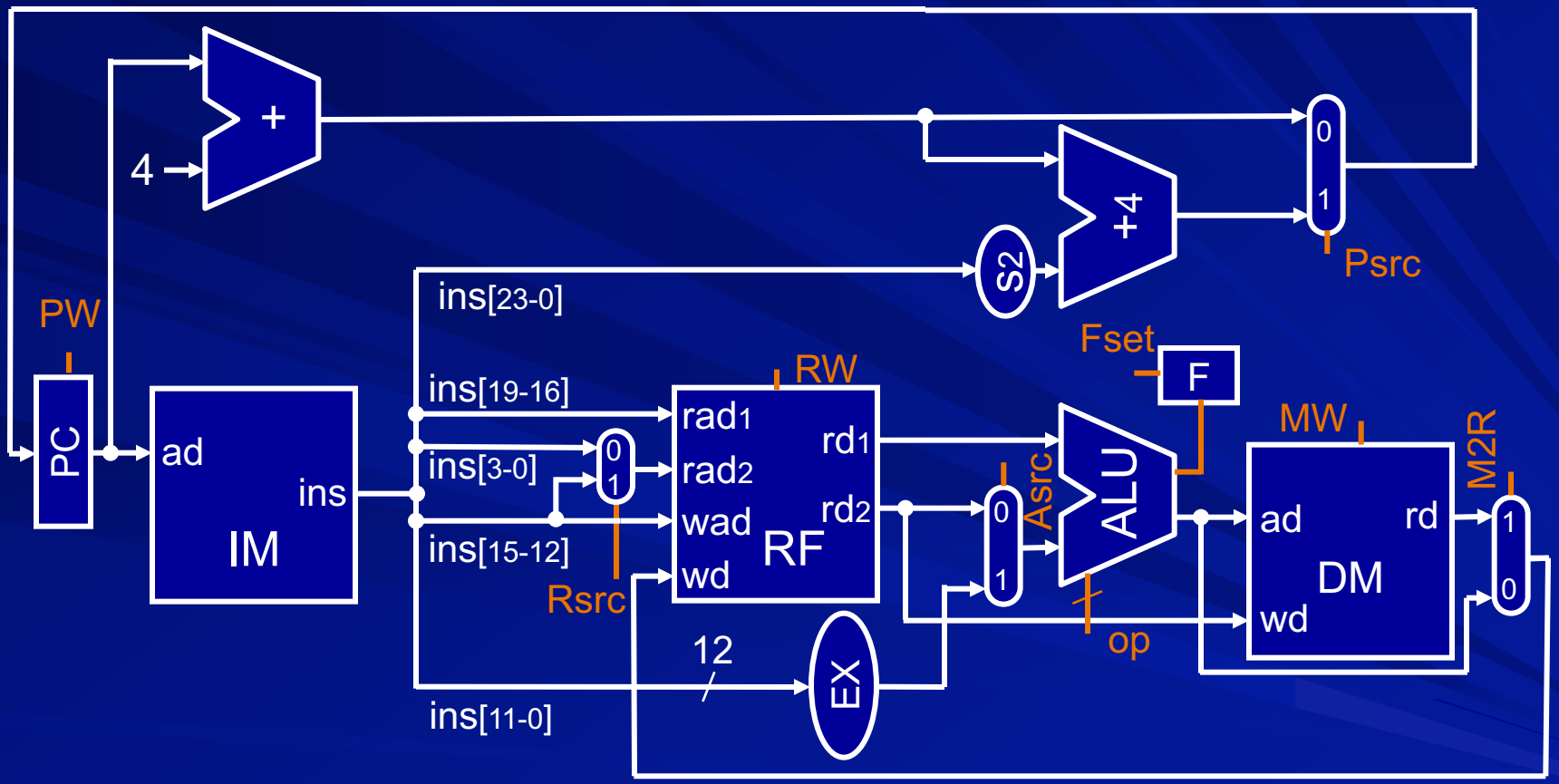
$$\max\left\{\begin{array}{c} t_{+} \\ t_I + t_R + t_A \end{array}\right.$$

Delay for {str}



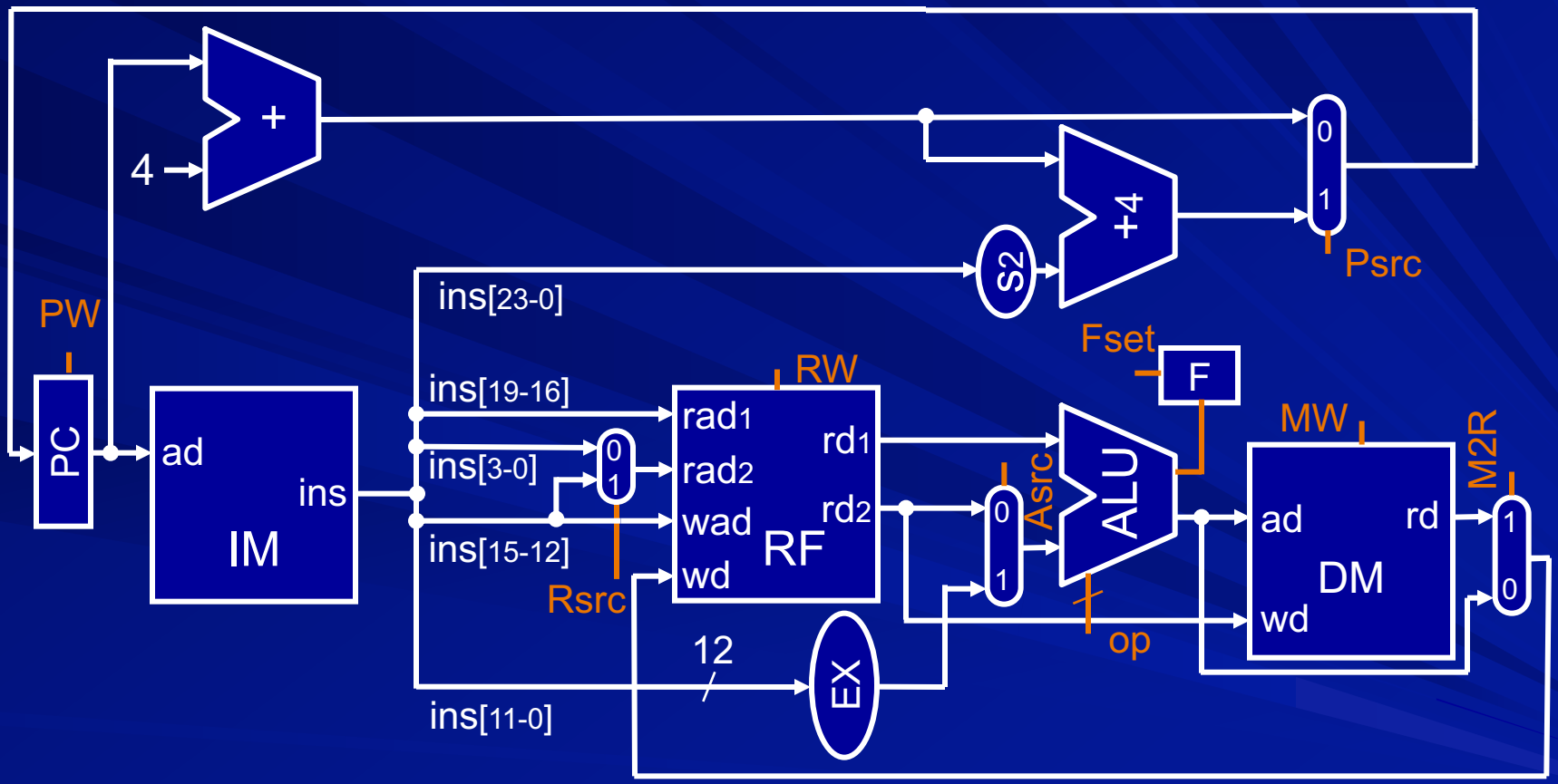
$$\max\begin{cases} t_{+} \\ t_I + t_R + t_A + t_M \end{cases}$$

Delay for {ldr}



$$\max\left\{\begin{array}{c} t_{+} \\ t_I + t_R + t_A + t_M + t_R \end{array}\right.$$

Delay for {b}



$$\max\begin{cases} t_I + t_+ \\ t_+ + t_+ \end{cases}$$

Overall clock period

$$\max \begin{cases} t_+, & t_I + t_R + t_A + t_R \\ t_+, & t_I + t_R + t_A \\ t_+, & t_I + t_R + t_A + t_M \\ t_+, & t_I + t_R + t_A + t_M + t_R \\ t_I + t_+, & t_+ + t_+ \end{cases} ?$$

Thanks