

## Arithmetic Pipeline

## Basic Concept

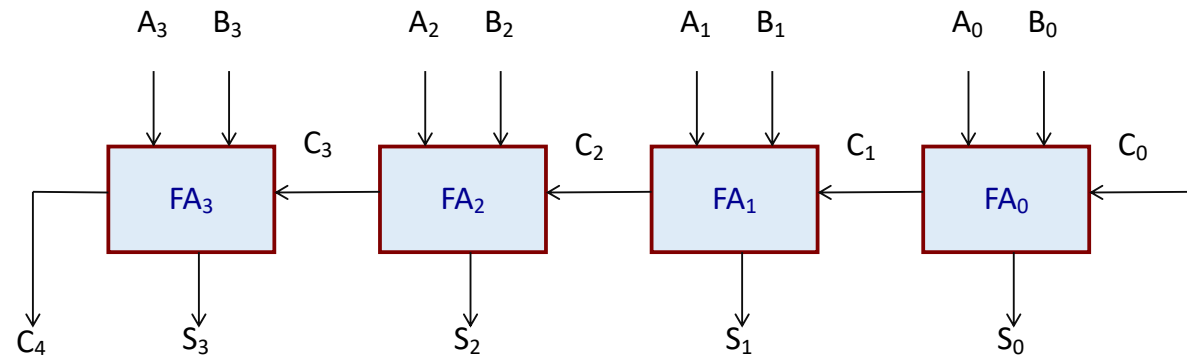
- Various arithmetic operations require a set of simpler computations to be carried out in sequence.
  - Can be split into stages with buffers in between, and run in a pipeline.
- Useful when several similar calculations are required to be carried out in sequence.
  - **Example:** Vector operations

```
for (i=0;i<64;i++)  
    A[i] = B[i] * C[i];
```

## Fixed Point Addition Pipeline

- We have seen how a ripple-carry adder works.
  - Rippling of the carries gives it a bad worst-case performance.
- We explore whether pipelining can improve the performance.
- *Assumption*: delay of a latch is comparable to the delay of a full adder.

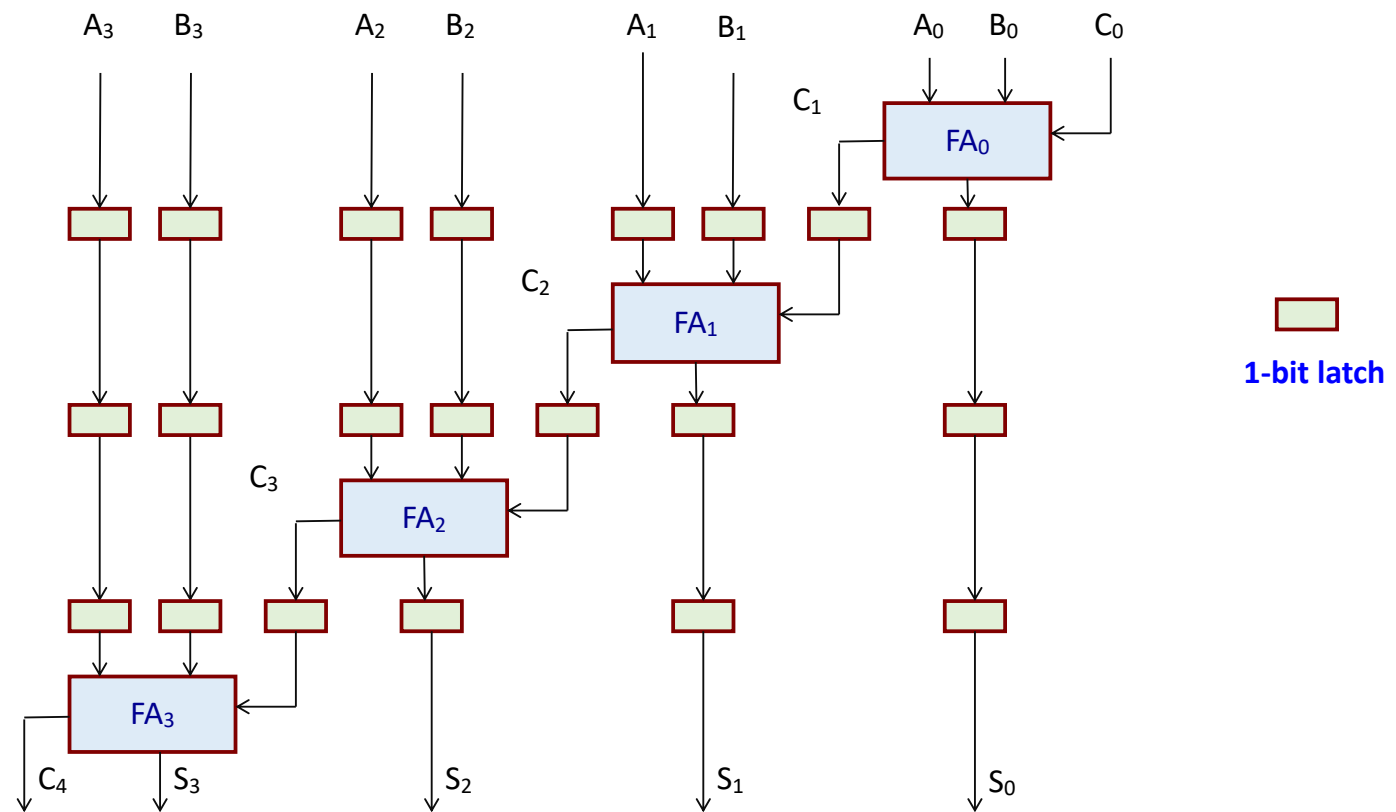
## A 4-bit Ripple Carry Adder



**Worst-case delay  $\approx 4 \times$  (carry generation time in FA)**

35

# 4-bit pipelined ripple-carry adder



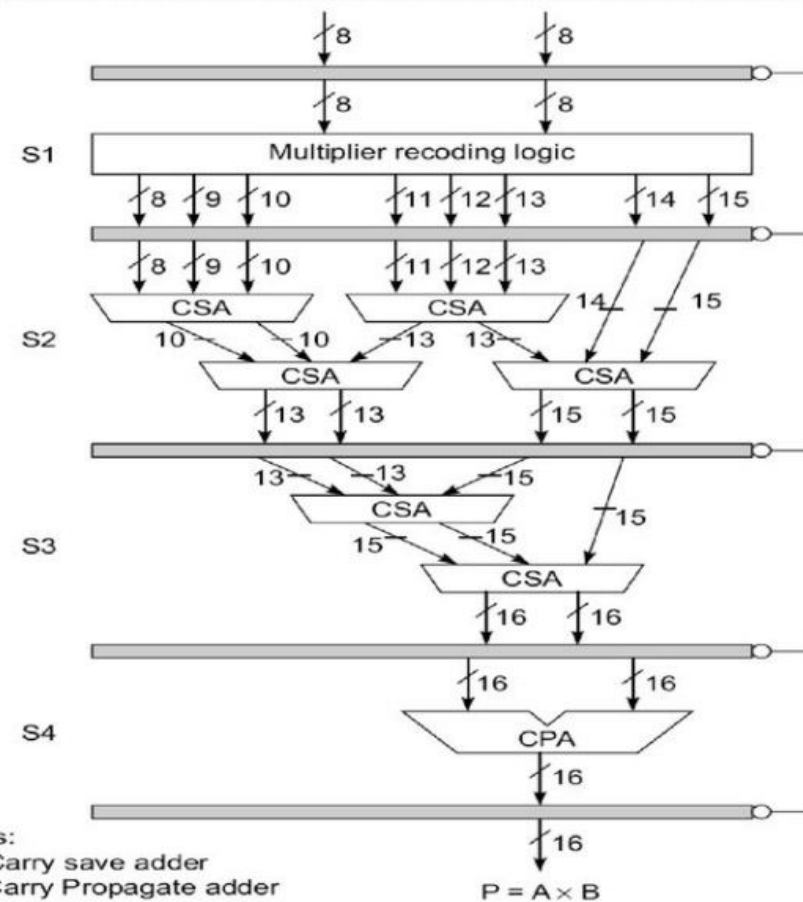
- Delay of a full adder  $= t_{FA}$
- Delay of a 1-bit latch  $= t_L$
- Clock period  $T \geq (t_{FA} + t_L)$
- After the pipeline is full, one result (sum) is generated every time  $T$ .
  - Convenient for vector addition kind of applications.

```
for (i=0; i<10000; i++)  
    a[i] = b[i] + c[i];
```

37

## Fixed Point Multiplication Pipeline

- Combinational array multiplier using carry save adders can be directly implemented as a pipeline.
- The multiplier recoding logic will contain  $8 \times 8 = 64$  AND gates to generate the partial product terms.



# Floating-Point Addition

- Floating-point addition requires the following steps:
  - a) Compare exponents and align mantissas.
  - b) Add mantissas.
  - c) Normalize result.
  - d) Adjust exponent.
- Subtraction is similar.

Example:

$$A = 0.9504 \times 10^3$$

$$B = 0.8200 \times 10^2$$

Align mantissa: 0.0820

Add mantissa:  $0.9504 + 0.0820 = 1.0324$

Normalize: 0.10324

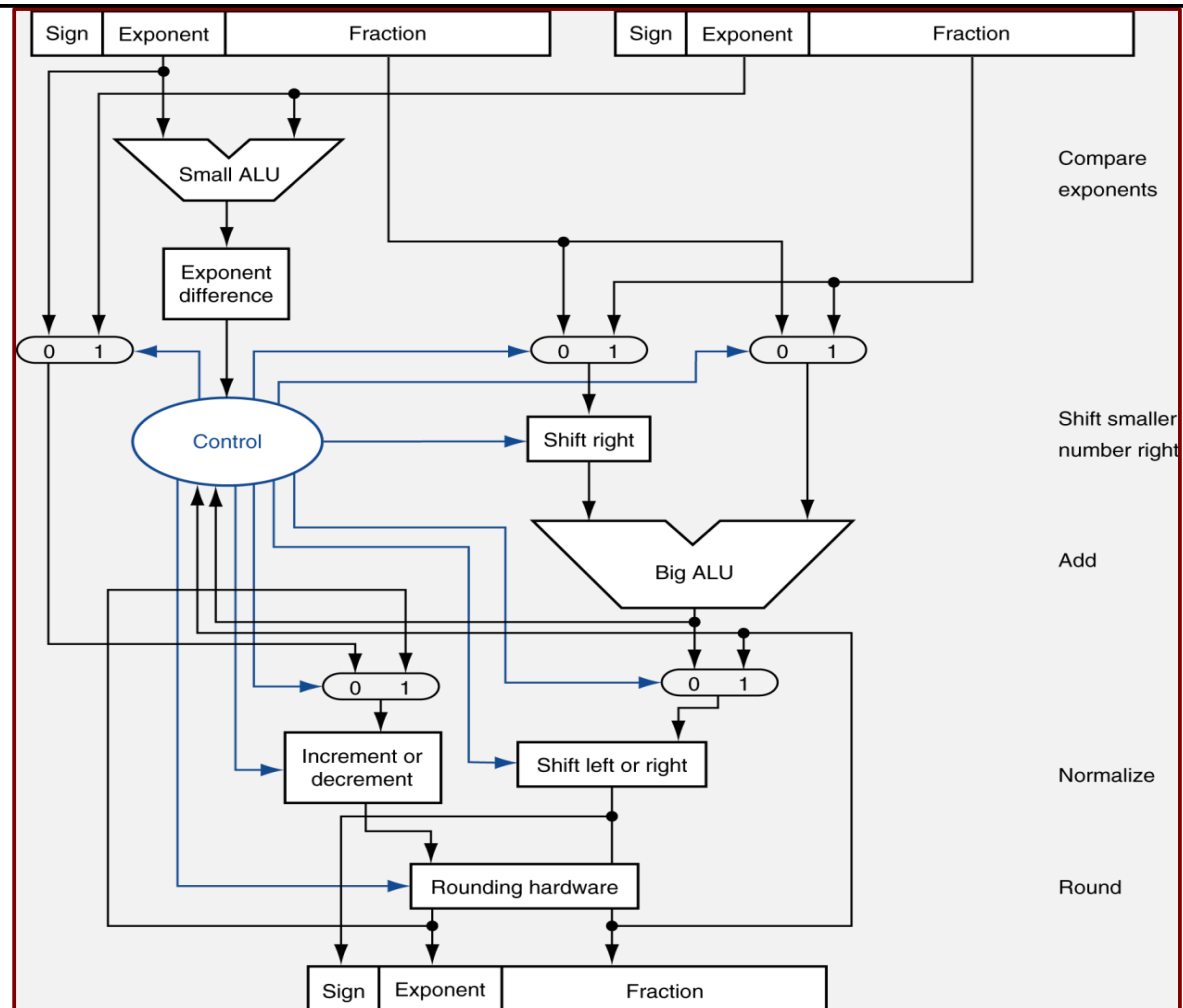
Adjust exponent:  $3 + 1 = 4$

Sum =  $0.10324 \times 10^4$

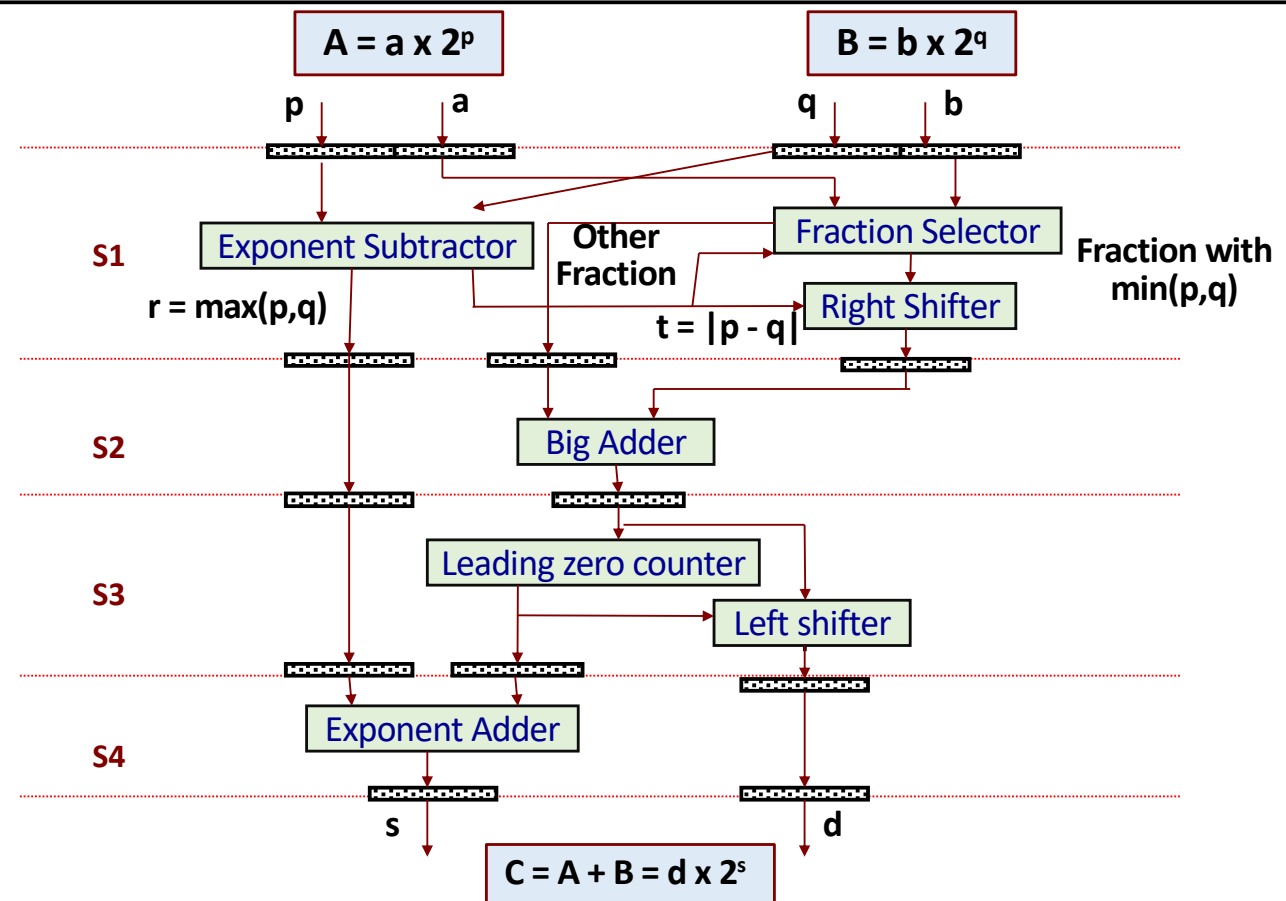


## Floating-Point Addition Hardware

- The last step of rounding is required in IEEE-754 format.



## 4-Stage Floating Point Adder



## Floating-Point Multiplication

- Floating-point multiplication requires the following steps:
  - a) Add exponents.
  - b) Multiply mantissas.
  - c) Normalize result.
- Division is similar.

A last step of rounding is required in IEEE-754 format.

Example:

$$A = 0.9504 \times 10^3$$

$$B = 0.8200 \times 10^2$$

Add exponents:  $3 + 2 = 5$

Multiply mantissa:  $0.9504 \times 0.8200 = 0.7793$

Normalize:  $0.7793$  (no change)

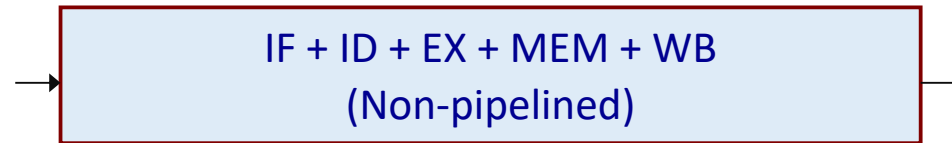
Product =  $0.7793 \times 10^5$

## **Pipelining the MIPS32 Data Path**

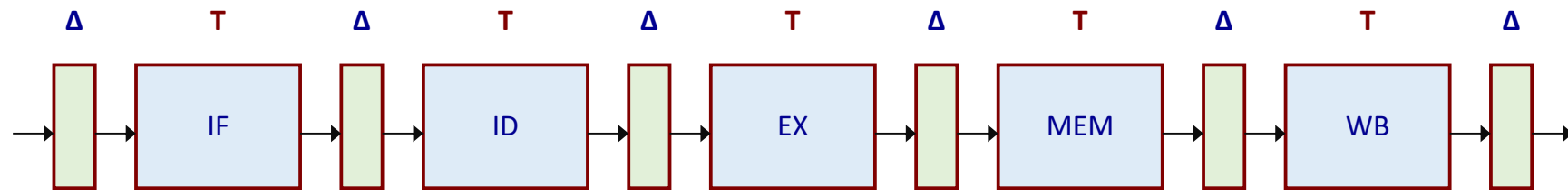
## Introduction

- Basic requirements for pipelining the MIPS32 data path:
  - We should be able to start a new instruction every clock cycle.
  - Each of the five steps mentioned before (IF, ID, EX, MEM and WB) becomes a pipeline stage.
  - Each stage must finish its execution within one clock cycle.
- Since execution of several instructions are overlapped, we must ensure that there is no conflict during the execution.
  - Simplicity of the MIPS32 instruction set makes this evaluation quite easy.
  - We shall discuss these issues in some detail.

**5T**



**Time of execute  $n$  instructions =  $5Tn$**



**Time of execute  $n$  instructions =  $(4 + n).(T + \Delta) \approx (4 + n).T$ , if  $T \gg \Delta$**

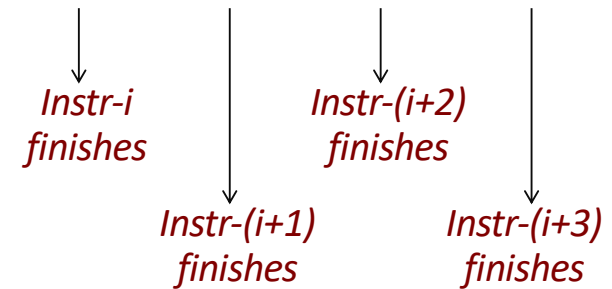
**Ideal Speedup =  $5Tn / (4 + n)T \approx 5$ , for large  $n$ .**

*In practice, due to various conflicts, speedup is much less.*

45

***Clock Cycles***

Instruction	1	2	3	4	5	6	7	8
<i><b>i</b></i>	IF	ID	EX	MEM	WB			
<i><b>i+1</b></i>		IF	ID	EX	MEM	WB		
<i><b>i+2</b></i>			IF	ID	EX	MEM	WB	
<i><b>i+3</b></i>				IF	ID	EX	MEM	WB


  
***Instr- $i$***   
*finishes*

***Instr- $(i+1)$***   
*finishes*

***Instr- $(i+2)$***   
*finishes*

***Instr- $(i+3)$***   
*finishes*

### Clock Cycles

Instruction	1	2	3	4	5	6	7	8
<i>i</i>	IF	ID	EX	MEM	WB			
<i>i+1</i>		IF	ID	EX	MEM	WB		
<i>i+2</i>			IF	ID	EX	MEM	WB	
<i>i+3</i>				IF	ID	EX	MEM	WB

Some examples of conflict:

- IF & MEM: In clock cycle 4, both instructions *i* and *i+3* access memory.
  - *Solution: use separate instructions and data cache.*
- ID & WB: In clock cycle 5, both instructions *i* and *i+3* access register bank.
  - *Solution: allow both read and write access to registers in the same clock cycle.*

47

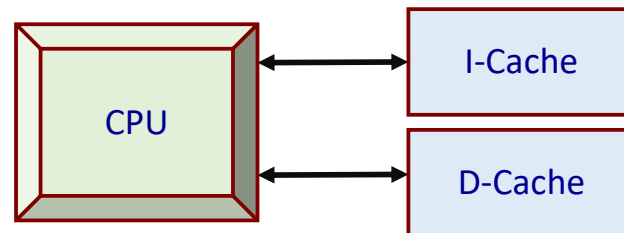


## Advantages of Pipelining

- In the non-pipelined version, the execution time of an instruction is equal to the combined delay of the five stages (say,  $5T$ ).
- In the pipelined version, once the pipeline is full, one instruction gets executed after every  $T$  time.
  - Assuming all state delays are equal (equal to  $T$ ), and neglecting latch delay.
- However, due to various conflicts between instructions (called *hazards*), we cannot achieve the ideal performance.
  - Several techniques have been proposed to improve the performance.
  - To be discussed.

## Some Observations

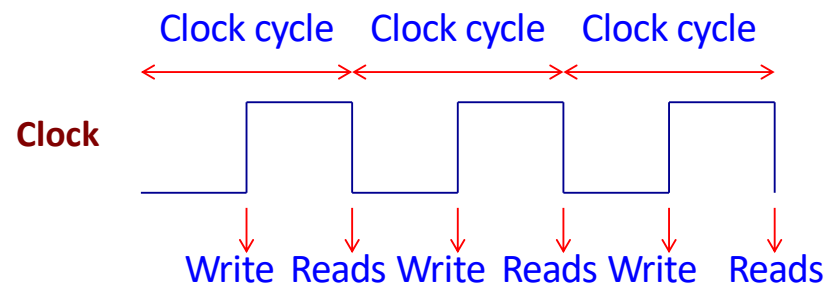
- a) To support overlapped execution, peak memory bandwidth must be increased *5 times* over that required for the non-pipelined version.
- An instruction fetch occurs every clock cycle.
  - Also there can be two memory accesses per clock cycle (one for instruction and one for data).
  - Separate instruction and data caches are typically used to support this.

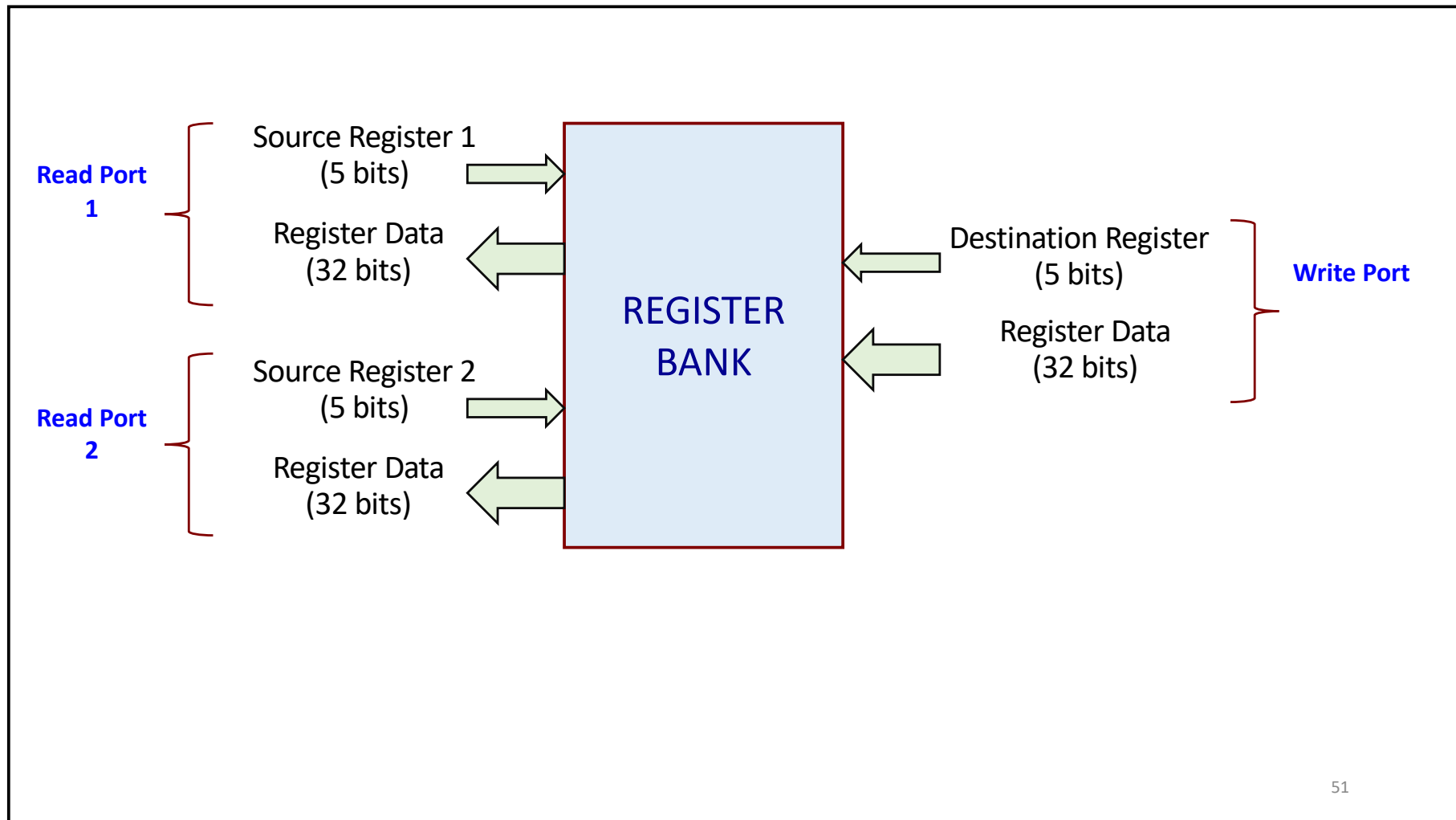


49

b) The register bank is accessed both in the stages ID and WB.

- ID requires 2 register reads, and WB requires 1 register write.
- We thus have the requirement of *2 reads and 1 write* in every clock cycle.
- Two register reads can be supported by having two register read ports.
- Simultaneous reads and write may result in clashes (e.g., same register used).
  - Solution adopted in MIPS32 pipeline is to perform the write during the *first half* of the clock cycle, and the reads during the *second half* of the clock cycle.





51

- c) Since a new instruction is fetched every clock cycle, it is required to increment the *PC* on each clock.
- PC updating has to be done *during IF stage itself*, as otherwise the next instruction cannot be fetched.
  - In the non-pipelined version discussed earlier, this was done during the MEM stage.

## Basic Performance Issues in a Pipeline



- Register stages are inserted between pipeline stages, which increases the execution time of an individual instruction.
  - Because of overlapped execution of instructions, throughput increases.
- The clock period  $T$  has to be chosen suitably based on:
  - a) Slowest stage in the pipeline.
  - b) Clock skew and jitter.
  - c) Register setup time: minimum time the register input must be held stable before the active clock edge arrives.

53

## Example 1

- Consider the 5-stage MIPS32 pipeline, with the following features:
  - Pipeline clock rate of 1GHz (i.e. 1 ns clock cycle time).
  - For a non-pipelined implementation, ALU operations and branches take 4 cycles, while memory operations take 5 cycles.
  - Relative frequencies of ALU operations, branches and memory operations are 50%, 15%, and 35% respectively.
  - In the pipelined implementation, due to clock skew and setup time, the clock cycle time increases by 0.25 ns.
  - Calculate the estimated speedup of the pipelined implementation.

- **Solution:**

a) For non-pipelined processor:

- Average instruction execution time = Clock cycle time x Average CPI  
$$= 1 \text{ ns} \times (0.50 \times 4 + 0.15 \times 4 + 0.35 \times 5) = 4.35 \text{ ns}$$

b) For pipelined processor:

- Clock cycle time =  $1 + 0.25 = 1.25 \text{ ns}$
- In the steady state, one instruction will get executed every clock cycle.
- Speedup =  $4.35 / 1.25 = 3.48$

55



## Micro-operations for Non-pipelined MIPS32

### IF

IR  $\leftarrow$  Mem [PC];  
NPC  $\leftarrow$  PC + 4;

### ID

A  $\leftarrow$  Reg [rs];  
B  $\leftarrow$  Reg [rt];  
Imm  $\leftarrow$  (IR<sub>15</sub>)<sup>16</sup> ## IR<sub>15..0</sub>  
Imm1  $\leftarrow$  IR<sub>25..0</sub> ## 00

### EX

ALUOut  $\leftarrow$  A + Imm;  
(a) Memory

ALUOut  $\leftarrow$  A func B;  
(b) R-R ALU

ALUOut  $\leftarrow$  A func Imm;  
(c) R-IMM ALU

ALUOut  $\leftarrow$  NPC + (Imm << 2);  
cond  $\leftarrow$  (A op 0);  
(d) Branch

### MEM

PC  $\leftarrow$  NPC;  
LMD  $\leftarrow$  Mem [ALUOut];  
(a) Load

PC  $\leftarrow$  NPC;  
Mem [ALUOut]  $\leftarrow$  B;  
(b) Store

if (cond) PC  $\leftarrow$  ALUOut;  
else PC  $\leftarrow$  NPC;  
(c) Branch

PC  $\leftarrow$  NPC;  
(d) Others

## WB

Reg [rd]  $\leftarrow$  ALUOut;

(a) R-R ALU

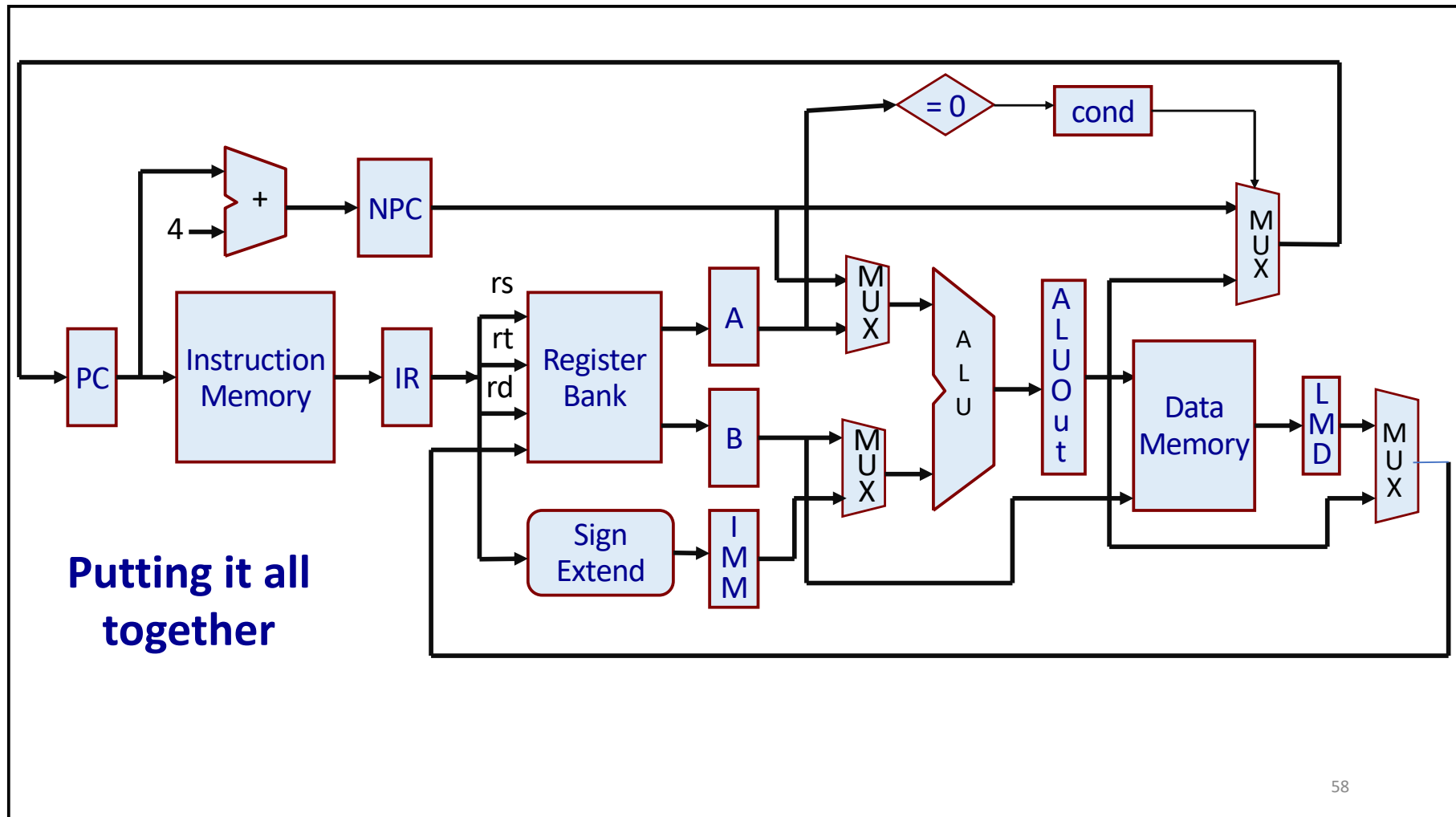
Reg [rt]  $\leftarrow$  ALUOut;

(b) R-IMM ALU

Reg [rt]  $\leftarrow$  LMD;

(c) Load

57



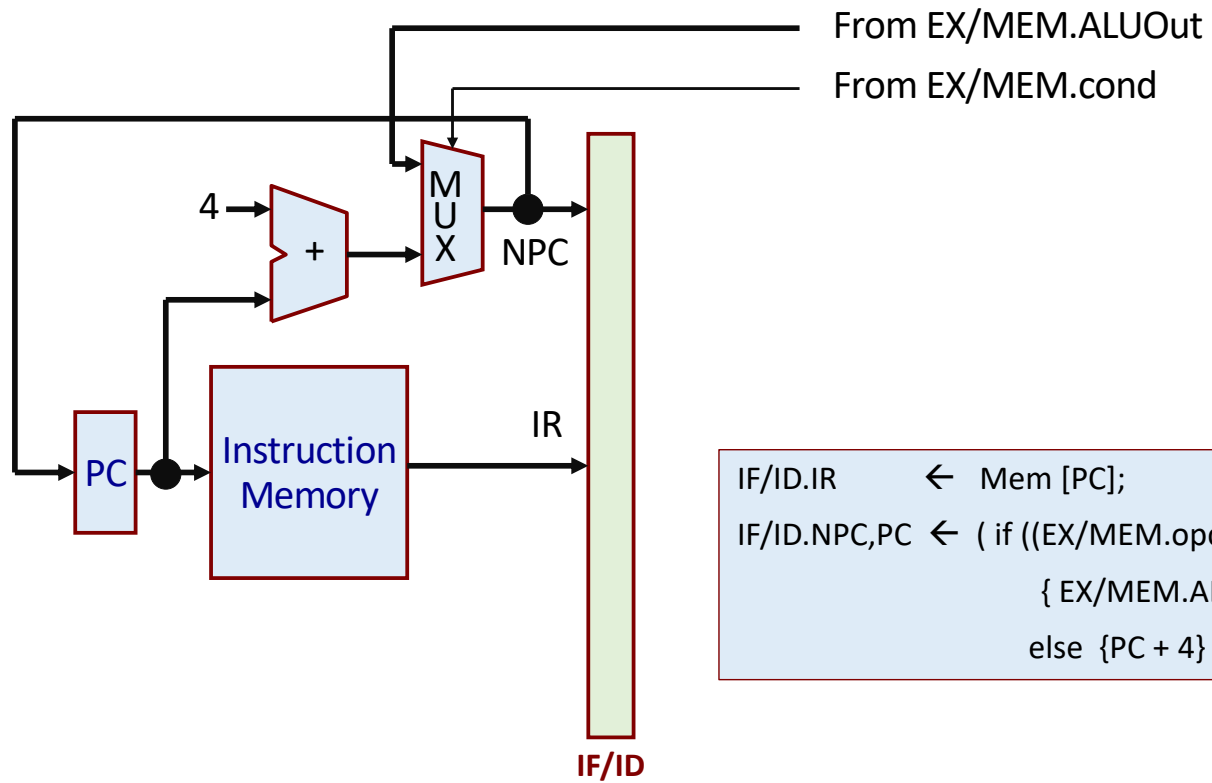
## Micro-operations for Pipelined MIPS32

- Convention used:
  - Many of the temporary registers required in the data path are included as part of the inter-stage latches.
  - IF/ID: denotes the latch stage between the IF and ID stages.
  - ID/EX: denotes the latch stage between the ID and EX stages.
  - EX/MEM: denotes the latch stage between the EX and MEM stages.
  - MEM/WB: denotes the latch stage between the MEM and WB stages.
- **Example:**
  - ID/EX.A refers to a register *A* that is implemented as part of the *ID/EX* latch stage.

59

### (a) Micro-operations for Pipeline Stage IF

```
IF/ID.IR    ← Mem [PC];  
IF/ID.NPC,PC ← ( if ((EX/MEM.opcode == branch) & EX/MEM.cond)  
                { EX/MEM.ALUOut}  
                else {PC + 4} );
```



```

IF/ID.IR    ← Mem [PC];
IF/ID.NPC,PC ← ( if ((EX/MEM.opcode == branch) & EX/MEM.cond)
                { EX/MEM.ALUOut}
                else {PC + 4} );
  
```

61

## (b) Micro-operations for Pipeline Stage ID

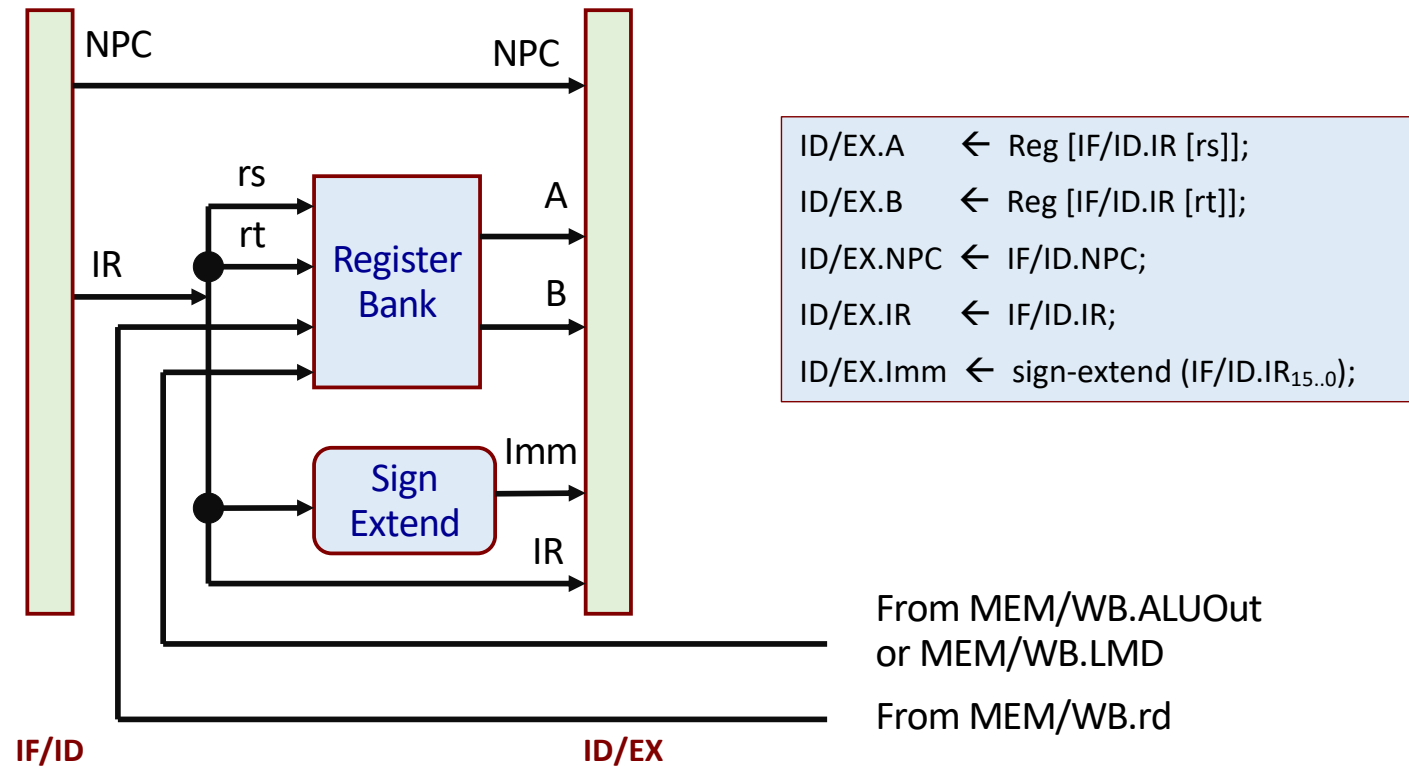
ID/EX.A  $\leftarrow$  Reg [IF/ID.IR [rs]];

ID/EX.B  $\leftarrow$  Reg [IF/ID.IR [rt]];

ID/EX.NPC  $\leftarrow$  IF/ID.NPC;

ID/EX.IR  $\leftarrow$  IF/ID.IR;

ID/EX.Imm  $\leftarrow$  sign-extend (IF/ID.IR<sub>15..0</sub>);



63



### (c) Micro-operations for Pipeline Stage EX

```
EX/MEM.IR ← ID/EX.IR;  
EX/MEM.ALUOut ← ID/EX.A func ID/EX.B;
```

## R-R ALU

```
EX/MEM.ALUOut ← ID/EX.NPC +
                (ID.EX.Imm << 2);
EX/MEM.cond ← (ID/EX.A == 0);
```

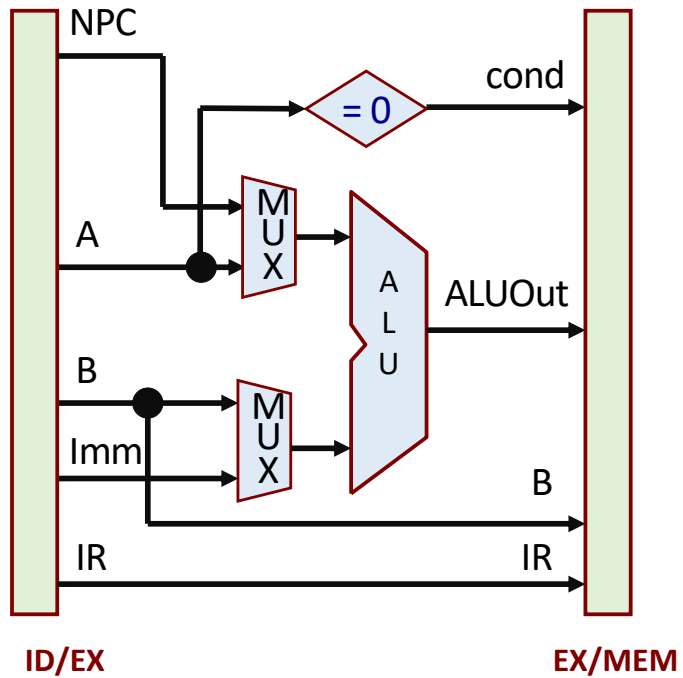
BRANCH

```
EX/MEM.IR ← ID/EX.IR;  
EX/MEM.ALUOut ← ID/EX.A func ID/EX.Imm;
```

## R-M ALU

```
EX/MEM.IR ← ID/EX.IR;  
EX/MEM.ALUOut ← ID/EX.A + ID/EX.B;  
EX/MEM.B ← ID/EX.B;
```

## LOAD / STORE



$EX/MEM.IR \leftarrow ID/EX.IR;$   
 $EX/MEM.ALUOut \leftarrow ID/EX.A \text{ func } ID/EX.B;$

$EX/MEM.IR \leftarrow ID/EX.IR;$   
 $EX/MEM.ALUOut \leftarrow ID/EX.A \text{ func } ID/EX.Imm;$

$EX/MEM.IR \leftarrow ID/EX.IR;$   
 $EX/MEM.ALUOut \leftarrow ID/EX.A + ID/EX.B;$   
 $EX/MEM.B \leftarrow ID/EX.B;$

$EX/MEM.ALUOut \leftarrow ID/EX.NPC + (ID/EX.Imm \ll 2);$   
 $EX/MEM.cond \leftarrow (ID/EX.A == 0);$

### (d) Micro-operations for Pipeline Stage MEM

MEM/WB.IR  $\leftarrow$  EX/MEM.IR;

MEM/WB.ALUOut  $\leftarrow$  EX/MEM.ALUOut;

ALU

MEM/WB.IR  $\leftarrow$  EX/MEM.IR;

MEM/WB.LMD  $\leftarrow$  Mem [EX/MEM.ALUOut];

LOAD

MEM/WB.IR  $\leftarrow$  EX/MEM.IR;

Mem [EX/MEM.ALUOut]  $\leftarrow$  EX/MEM.B;

STORE

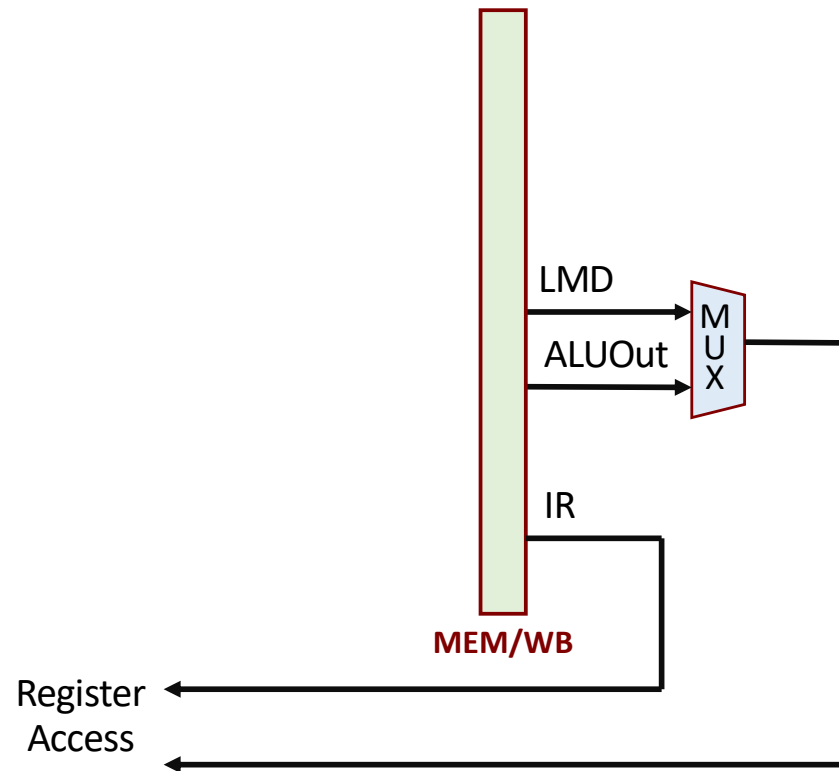


### (e) Micro-operations for Pipeline Stage WB

Reg [MEM/WB.IR [rd]]  $\leftarrow$  MEM/WB. ALUOut; **R-R ALU**

Reg [MEM/WB.IR [rt]]  $\leftarrow$  MEM/WB. ALUOut; **R-M ALU**

Reg [MEM/WB.IR [rt]]  $\leftarrow$  MEM/WB. LMD; **LOAD**



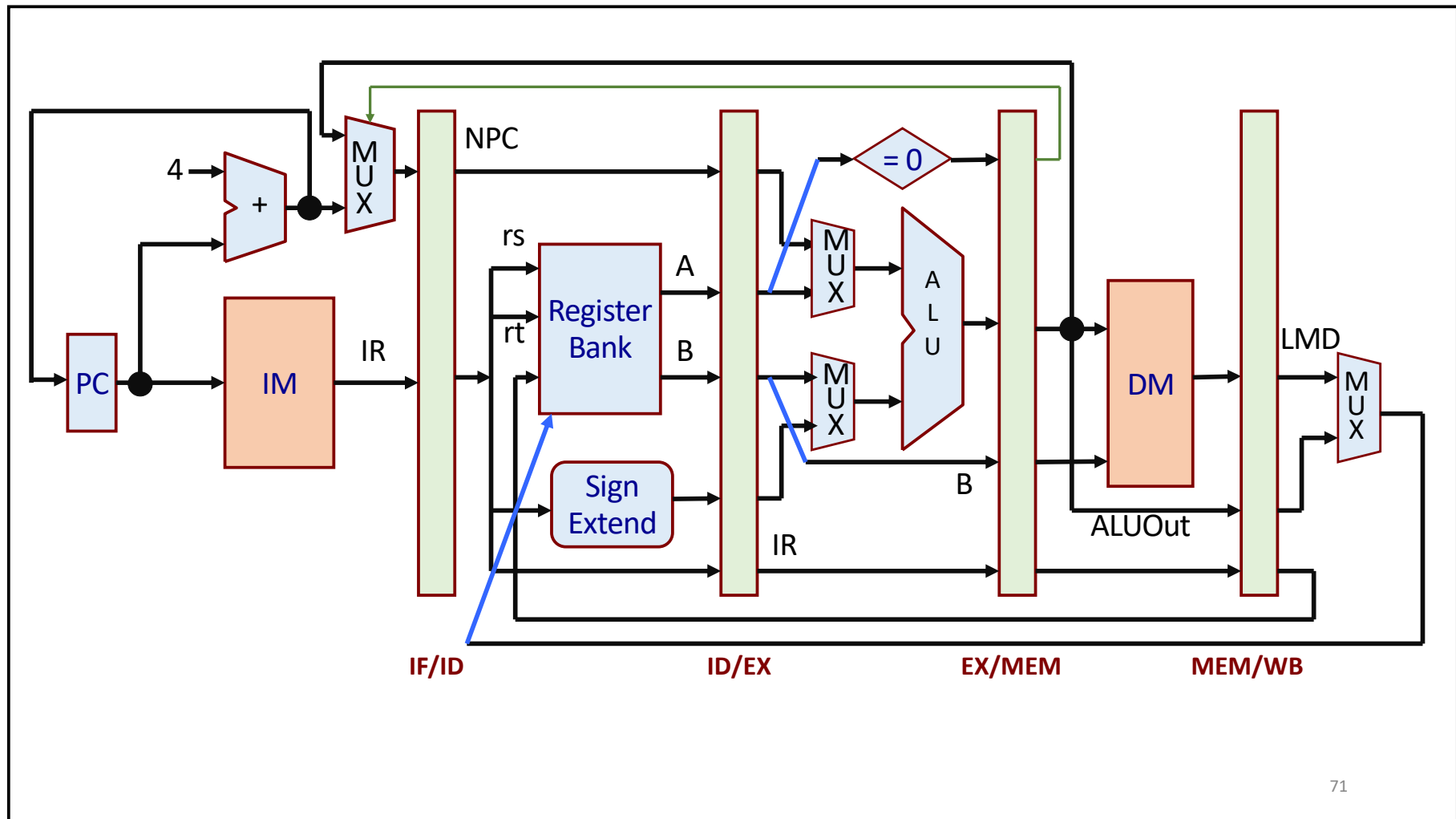
Reg [MEM/WB.IR [rd]]  $\leftarrow$  MEM/WB. ALUOut;

Reg [MEM/WB.IR [rt]]  $\leftarrow$  MEM/WB. ALUOut;

Reg [MEM/WB.IR [rt]]  $\leftarrow$  MEM/WB. LMD;

69

## **PUTTING IT ALL TOGETHER :: MIPS32 PIPELINE**



71