CSC 320 Exam #2 study guide

*** Bring a calculator with you and you can use the number conversion function (Decimal to Binary).**
*** Bring the 2 MIPS instruction sheets/tables with you**

The exam has 8 questions:

#1. Be able to do 2's complement arithmetic operations (for example: a+b or a-b). Study Ch3-1 slides #5 to #9.

#2. Be able to convert a decimal number into the IEEE 754 format (in binary). The formula will be provided. Study Ch3-2 slides #17 to #19 and HW#3 question #3.

#3. Need to understand how a MIPS 32-bit ALU works: including logic gates, ALU operations, control lines and output lines. Study CH3-1 slides #15 to #33. (Note the final version of our 1-bit ALU and 32-bit ALU are shown in slides #32 and #33)

#4. Be able to do the final versions of integer "multiplication algorithm" and "division algorithm". Study your HW#3, questions #1 and #2 (and CH3-2, slides #11, #12, #15, and #16). The algorithms (flow charts) and tables will be provided.

#5. Given a MIPS instruction and the hardware; be able to find (calculate) the values for all control lines. **Study your HW#3, question #6-a) to d)**.

#6. Need to understand how the MIPS datapath and control lines work (and their relationships). Given a MIPS instruction and the general datapath diagram (see CH4-1, slides #29 to #33), be able to label the data path and control lines. **Study your HW#3, question #6-d)** and our lecture examples.

#7. Need to understand the meanings/definitions of the 5 stages of MIPS **single-cycle** instructions and their relationships with our MIPS instructions (study CH4-1, slides #21 to #25).

#8. Given MIPS instructions with basic **5-stage pipeline cycles**. Be able to fill in the pipeline diagram indicating the cycle when each stage for each instruction will be performed (as was done in class). Study Ch4-2 slides #20 to #26 and HW#3 question #7.

Chapter 4.5 - 4.6

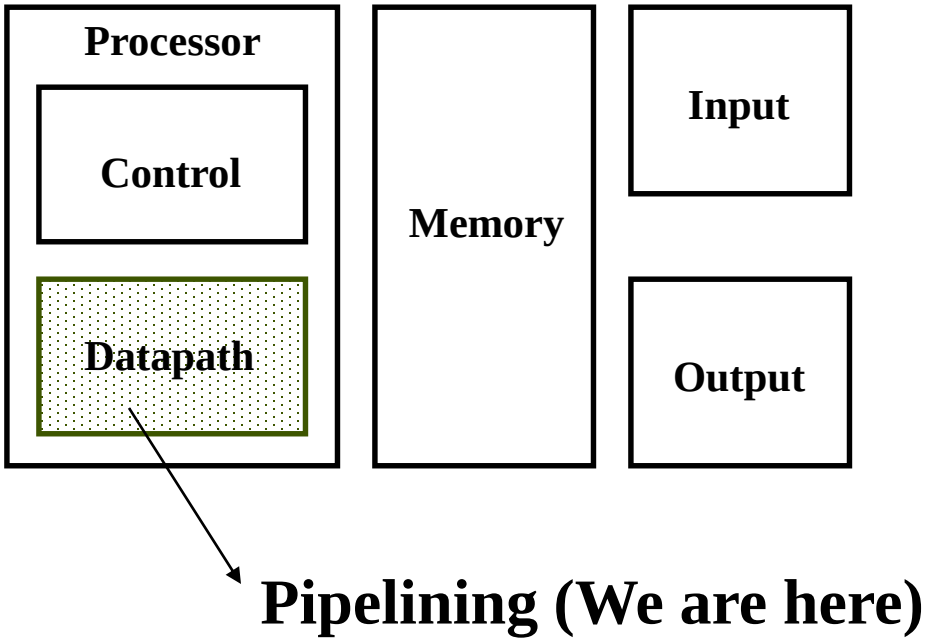
Pipelining

Section 4.5

An Overview of Pipelining

The Big Picture: Where are We Now?

- **The Five Classic Components of a Computer**



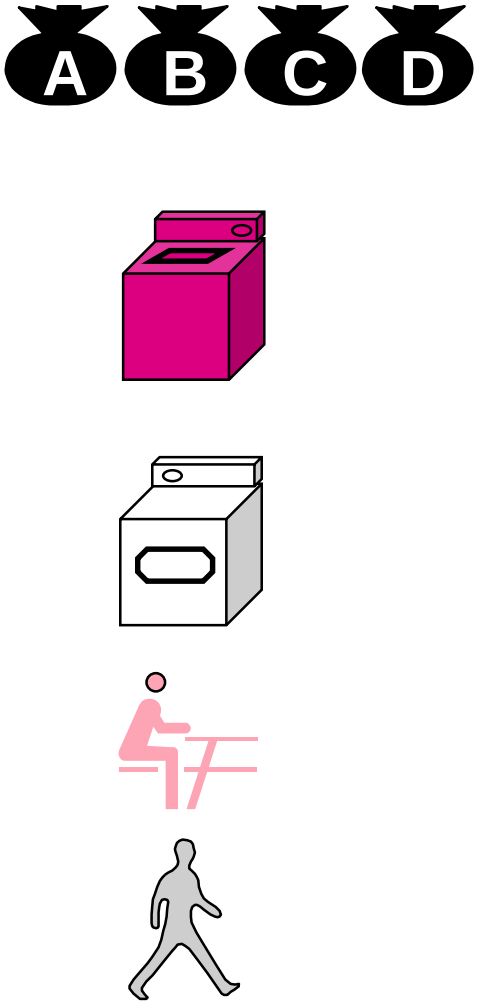
An Overview of Pipelining

- ***Pipelining*** **is an implementation technique in which multiple instructions are overlapped in execution.**

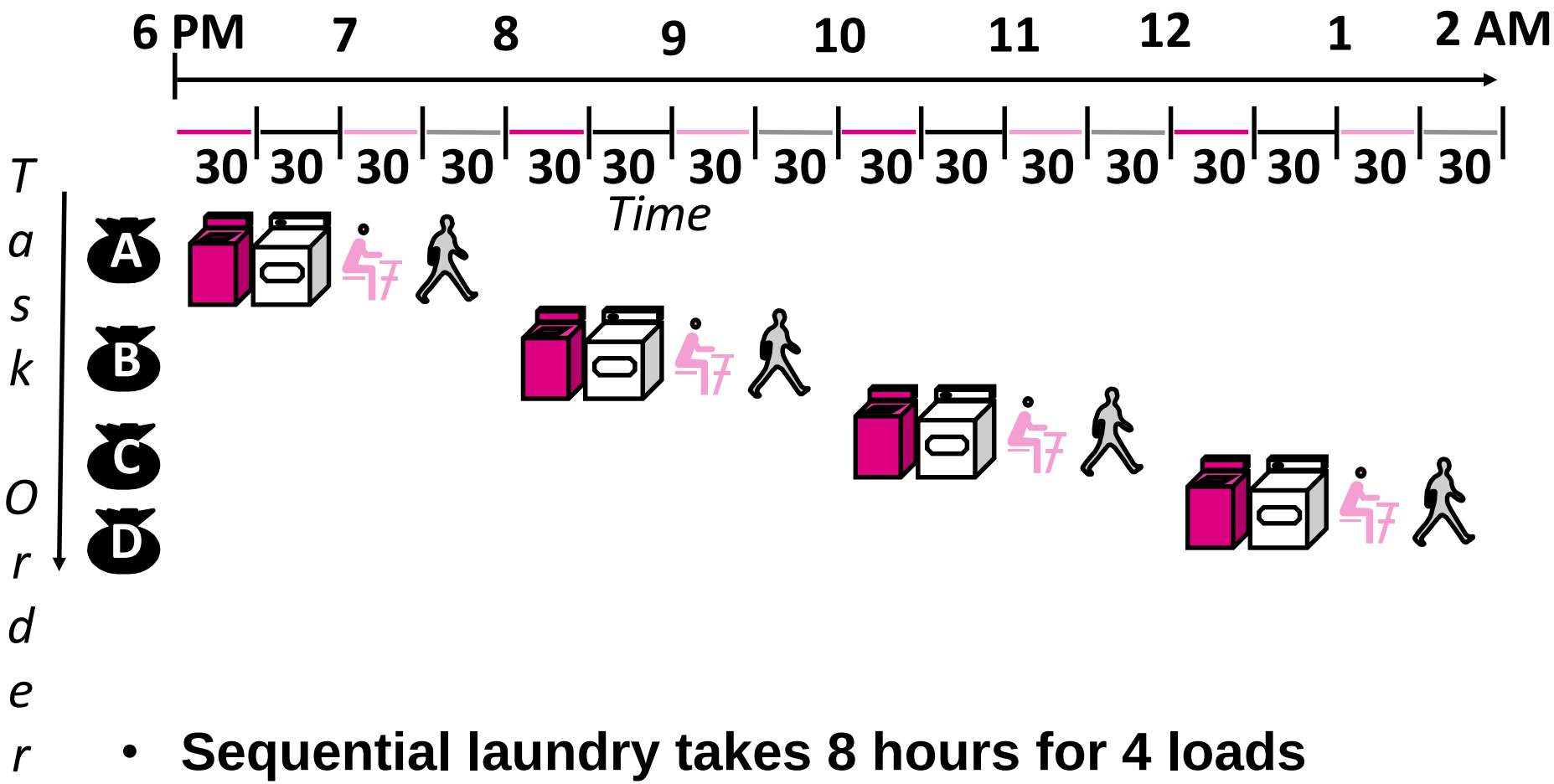
- **Today, pipelining is key to making processors fast**

Pipelining is Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers

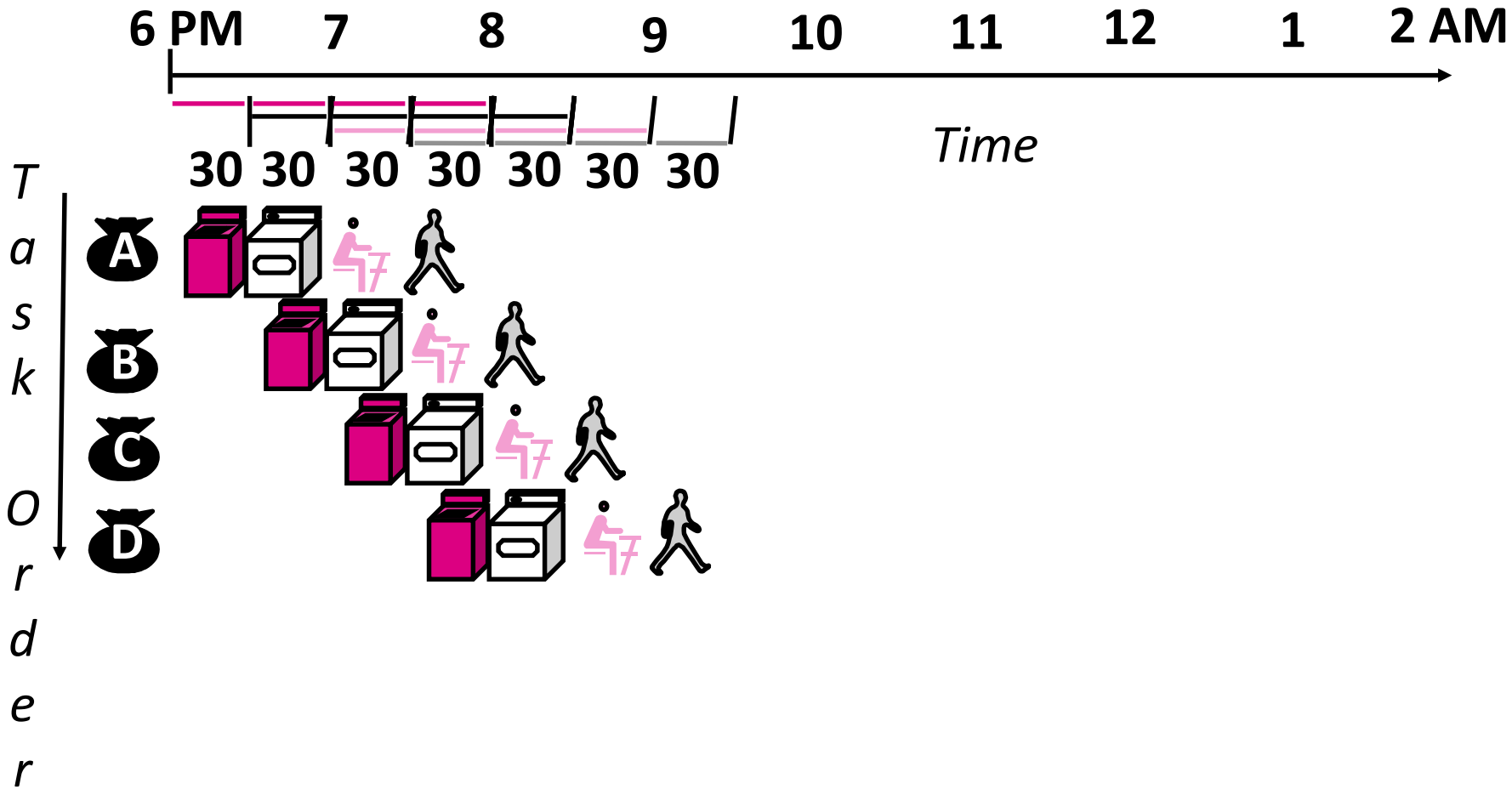


Sequential Laundry



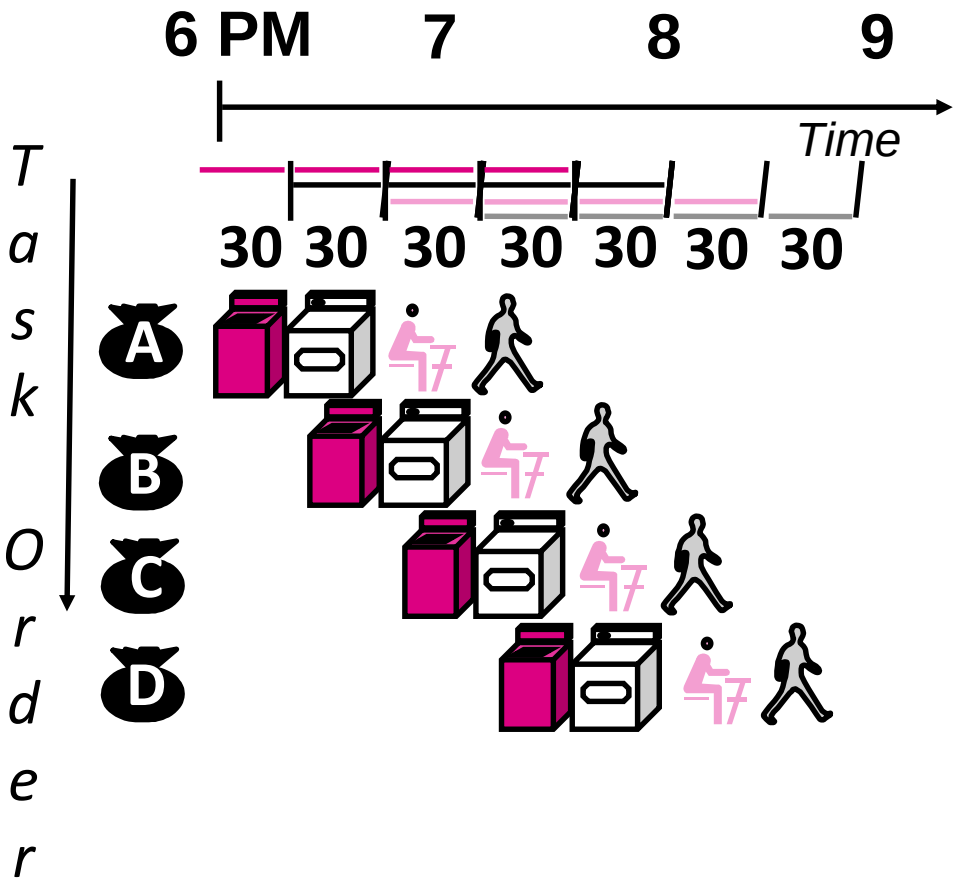
- **Sequential laundry takes 8 hours for 4 loads**
- **If they learned pipelining, how long would laundry take?**

Pipelined Laundry: Start work ASAP



- **Pipelined laundry takes 3.5 hours for 4 loads!**

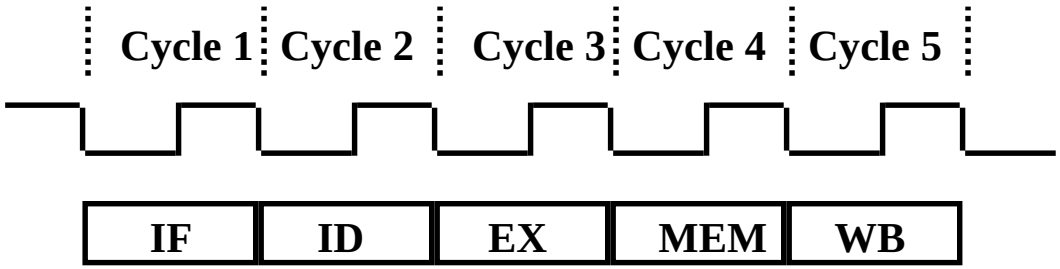
Pipelining Lessons



- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number pipe stages**
- Pipeline rate limited by **slowest** pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "**fill**" pipeline and time to "**drain**" it reduces speedup
- **Stall** for Dependences

The Five Stages of MIPS Instructions

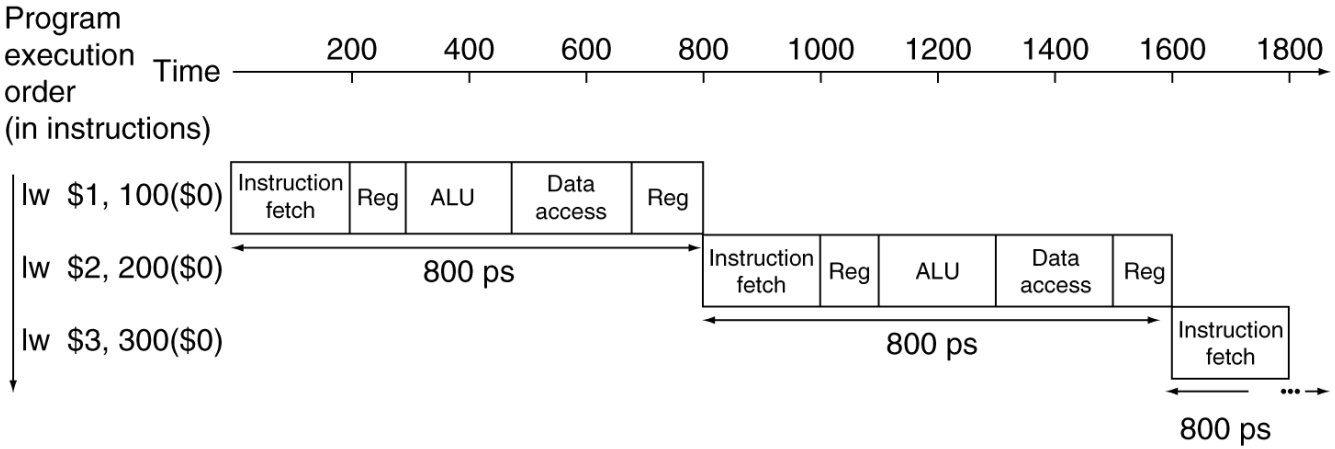
MIPS instructions classically take 5 steps :
(Each step takes 1 clock cycle)



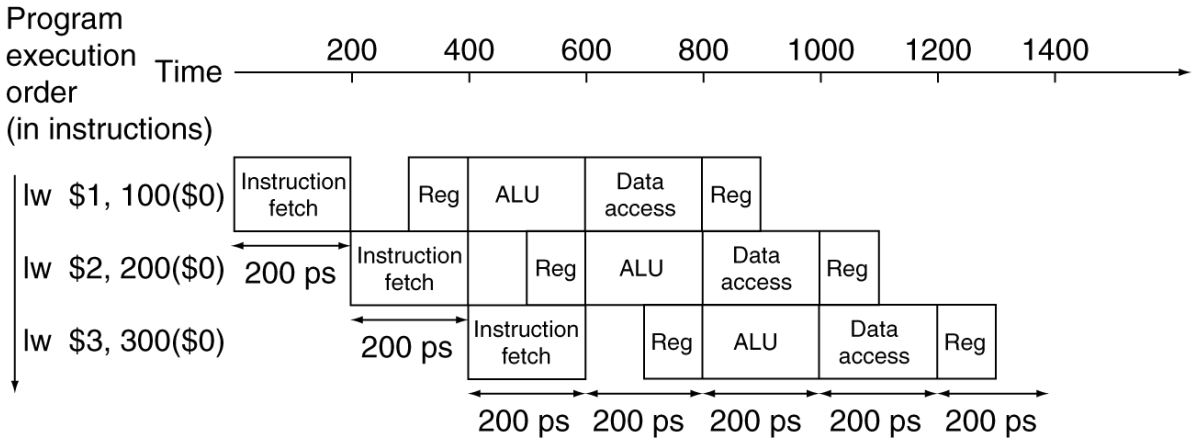
- **IF: Instruction Fetch**
  - **Fetch the instruction from the Instruction Memory**
- **ID: Registers Fetch and Instruction Decode**
- **EX: Execution or Address calculation**
- **MEM: Data Memory Assess**
- **WB: Write the data back to the register file**

Pipelining Performance

- **Pipelining overlaps the steps.**
- **Improve performance by increasing instruction throughput**



⇨ 2400ps



⇨ 1400ps

Pipelining

- **What makes it easy**
  - **all instructions are the same length**
  - **just a few instruction formats**
  - **memory operands appear only in loads and stores**

- **What makes it hard?**
  - **Hazards: There are situations in pipelining when the next instruction cannot execute in the following clock cycle.**
  - **There are 3 types of hazards:**
    1) **structural hazards**
    2) **control hazards**
    3) **data hazards**

Structural Hazards

- **structural hazards: The hardware cannot support the combination of instructions to be executed in the same cycle.**

  (suppose we had only one memory)
- **Example : IF and MEM cannot be executed in the same cycle since both need to access the data from memory.**

Cycle	1	2	3	4	5	6	7	8
inst 1	IF	ID	EX	MEM	WB			
inst 2		IF	ID	EX	MEM	WB		
inst 3			IF	ID	EX	MEM	WB	
inst 4				IF	ID	EX	MEM	WB

Structural hazards

Structural Hazards

- **Solution: Delay inst 4 until the 7th cycle.**

Cycle	1	2	3	4	5	6	7	8	9
inst 1	IF	ID	EX	MEM	WB				
inst 2		IF	ID	EX	MEM	WB			
inst 3			IF	ID	EX	MEM	WB		
inst 4				stall	stall	stall	IF	ID	EX ..

Start the 4th instruction here

Control Hazards

- A control hazard occurs because the CPU does not know soon enough:
  - whether or not the conditional branch will be taken
  - the target of the transfer of control
- Solutions:
  - Stall : we can delay the execution of the branch until after the information is known.
  - Predict : We can predict this information earlier.
  - Delay branch slot : re-order the instructions

Control Hazards (Using stall) - 1

- Normally we know if branch will be taken or not after the EX stage.
- Example:

Cycle	1	2	3	4	5	6	7	8
add $4, $5, $6	IF	ID	EX	MEM	WB			
beq $1, $2, 40		IF	ID	EX	MEM	WB		
lw $3, 300($0)			Stall	Stall	IF	ID	EX	MEM

Since we don't know the branch will be taken or not,
we delay the lw until after the EX stage (the 5th cycle).
But if we do this, there will be a Structural Hazard in cycle 5.

Control Hazards (Using stall) - 2

- Assume that we can compute the branch address and update the PC in the ID stage (see section 4.6).
- Example:

Cycle	1	2	3	4	5	6	7	8
add $4, $5, $6	IF	ID	EX	MEM	WB			
beq $1, $2, 40		IF	ID	EX	MEM	WB		
lw $3, 300($0)			Stall	IF	ID	EX	MEM	WB

Since we don't know the branch will be taken or not,
we delay the lw until after the ID stage (the 4th cycle).
But if we do this, there will be a Structural Hazard in cycle 4 (and cycle 5).

Control Hazards (Using stall)

- **Correct Solution**

Cycle	1	2	3	4	5	6	7	8	9	10
add $4, $5, $6	IF	ID	EX	MEM	WB					
beq $1, $2, 40		IF	ID	EX	MEM	WB				
lw $3, 300($0)			stall	stall	stall	IF	ID	EX	MEM	WB

Control Hazards (Using predict)

- Always predict that branch will fail.
- If the prediction is wrong, then do the pipeline stall.

Cycle	1	2	3	4	5	6	7	8
add $4, $5, $6	IF	ID	EX	MEM	WB			
beq $1, $2, 40		IF	ID	EX	MEM	WB		
lw $3, 300($0)			IF	ID	EX	MEM	WB	

If not branch

Cycle	1	2	3	4	5	6	7	8
add $4, $5, $6	IF	ID	EX	MEM	WB			
beq $1, $2, 40		IF	ID	EX	MEM	WB		
			Bubble	Bubble	Bubble	Bubble	Bubble	
40						IF	ID	EX ...

if branch

Control Hazards (Using Delay Branch slot)

- The assembler will automatically replace the branch instruction slot by an instruction that does not affect the branch.
- Note: MIPS uses delayed Branch Slot

```
add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)
```

delayed branch slot

Since the "add" instruction doesn't affect the branch, replace the delayed branch slot by the "add" instruction.

Re-order the code

Cycle	1	2	3	4	5	6	7	8
beq $1, $2, 40	IF	ID	EX	MEM	WB			
add $4, $5, $6		IF	ID	EX	MEM	WB		
lw $3, 300($0)			IF	ID	EX	MEM	WB	

Data Hazards

- A data hazard occurs because one instruction depends on the result of a previous instruction in the pipeline.
- Note: the add (or sub) doesn't write its result until the WB stage.
- Example :

Cycle	1	2	3	4	5	6	7	8	9
add $s0, $t0, $t1	IF	ID	EX	MEM	WB				
sub $t2, $s0, $t3		IF	stall	stall	stall	ID	EX	MEM	WB

↓

Need to wait until after WB since $s0 will has the latest result after WB stage

- We can sometimes resolve (or reduce) stalls for data hazards by using the *forwarding* technique.

Data Hazards (EX->EX Forwarding)

- **Forwarding: Since the result of "add" operation ($s0) can be obtained after the EX stage, we can forward the result ($s0) to the next EX to do the "sub" operation.**

Cycle	1	2	3	4	5	6
add $s0, $t0, $t1	IF	ID	EX	MEM	WB	
sub $t2, $s0, $t3		IF	ID	EX	MEM	WB

Forwarding $s0 to the next instruction

Data Hazards (MEM->EX Forwarding)

- Forwarding works very well. However, forwarding can not prevent all pipeline stalls.
- Example: Since the instruction "lw" won't load the desired data into memory until the MEM stage, we cannot forward the result ($s0) until after the MEM stage.

Cycle	1	2	3	4	5	6	7
lw $s0 20($t1)	IF	ID	EX	MEM	WB		
sub $t2, $s0, $t3		IF	ID	stall	EX	MEM	WB

Forwarding $s0 to the next instruction

Data Hazards (WB->ID Forwarding)

- **Example: Since the instruction "lw" writes data back to the register ($s0) at the WB stage, we can forward the result ($s0) to the ID stage of a later instruction (it is the same register $s0).**

Cycle	1	2	3	4	5	6	7	8
lw $s0 20($t1)	IF	ID	EX	MEM	WB			
add $t4, $s0, $t3		IF	ID	Stall	EX	MEM	WB	
sub $t2, $s0, $t3			IF	Stall	ID	EX	MEM	WB

Forwarding $s0 to the next instruction

Reordering Code - 1

- Reordering Code to Avoid Pipeline Stalls

  (Assume: B=>$t0+0, E=>$t0+4, A=>$t0+12, F=>$t0+8, C=>$t0+16)

  C/Java code for: `A = B + E; C = B + F;`

```
        lw   $t1, 0($t0)
        lw   $t2, 4($t0)
stall   add  $t3, $t1, $t2
        sw   $t3, 12($t0)
        lw   $t4, 8($t0)
stall   add  $t5, $t1, $t4
        sw   $t5, 16($t0)
```

17 cycles

Reordering Code - 2

- Reordering Code to Avoid Pipeline Stalls

  (Assume: B=>$t0+0, E=>$t0+4, A=>$t0+12, F=>$t0+8, C=>$t0+16)

  C/Java code for: `A = B + E; C = B + F;`

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

15 cycles

Reordering Code - 3

- Reordering Code to Avoid Pipeline Stalls

  (Assume: B=>$t0+0, E=>$t0+4, A=>$t0+12, F=>$t0+8, C=>$t0+16)

  C/Java code for: `A = B + E; C = B + F;`

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
add  $t5, $t1, $t4
sw   $t3, 12($t0)
sw   $t5, 16($t0)
```

12 cycles

Section 4.6

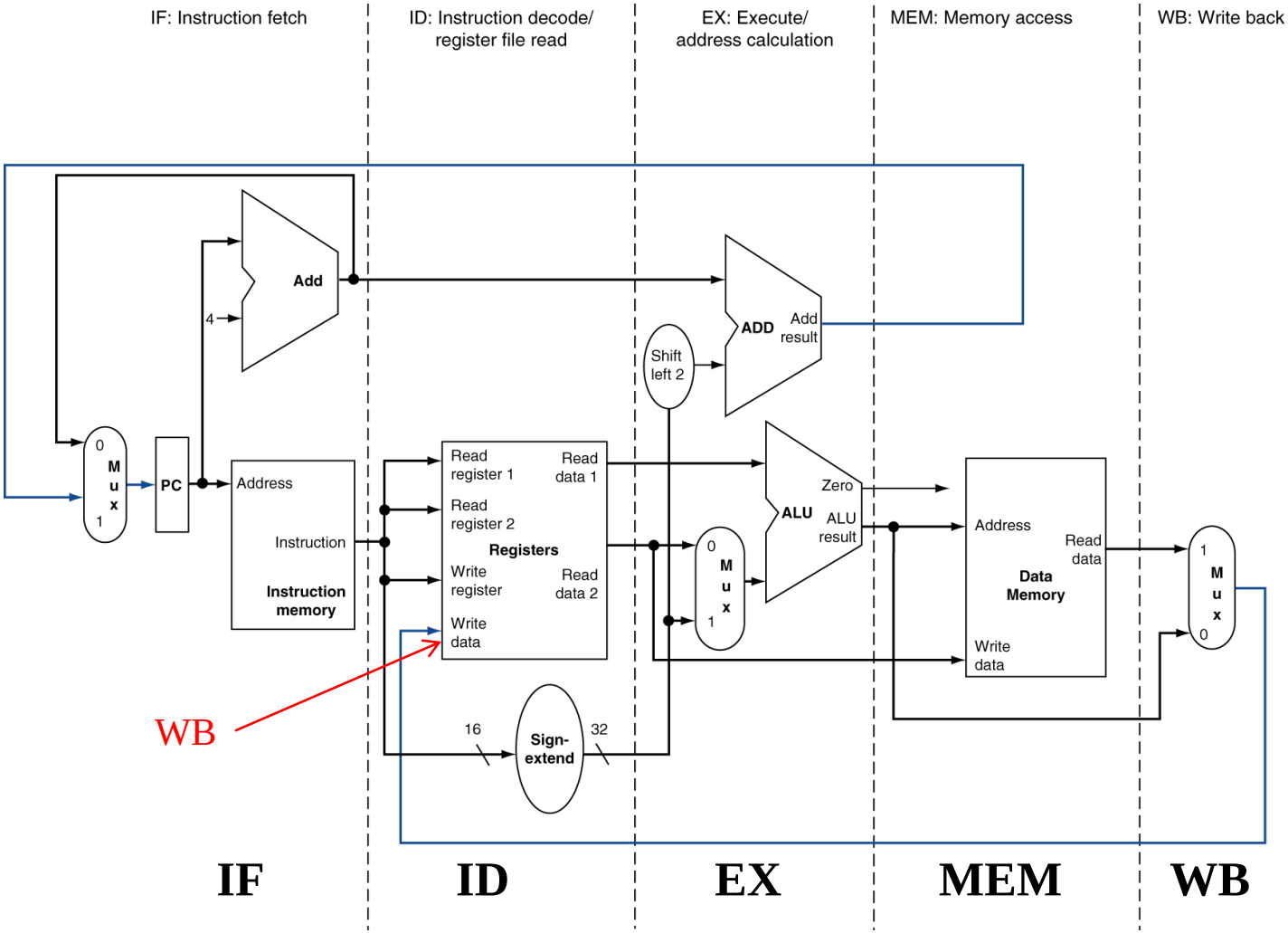
Pipelined Datapath and Control

A Pipelined Datapath

- **In this section, we will show how the single-cycle datapath can be modified into a pipelined datapath.**

Basic Idea of datapath

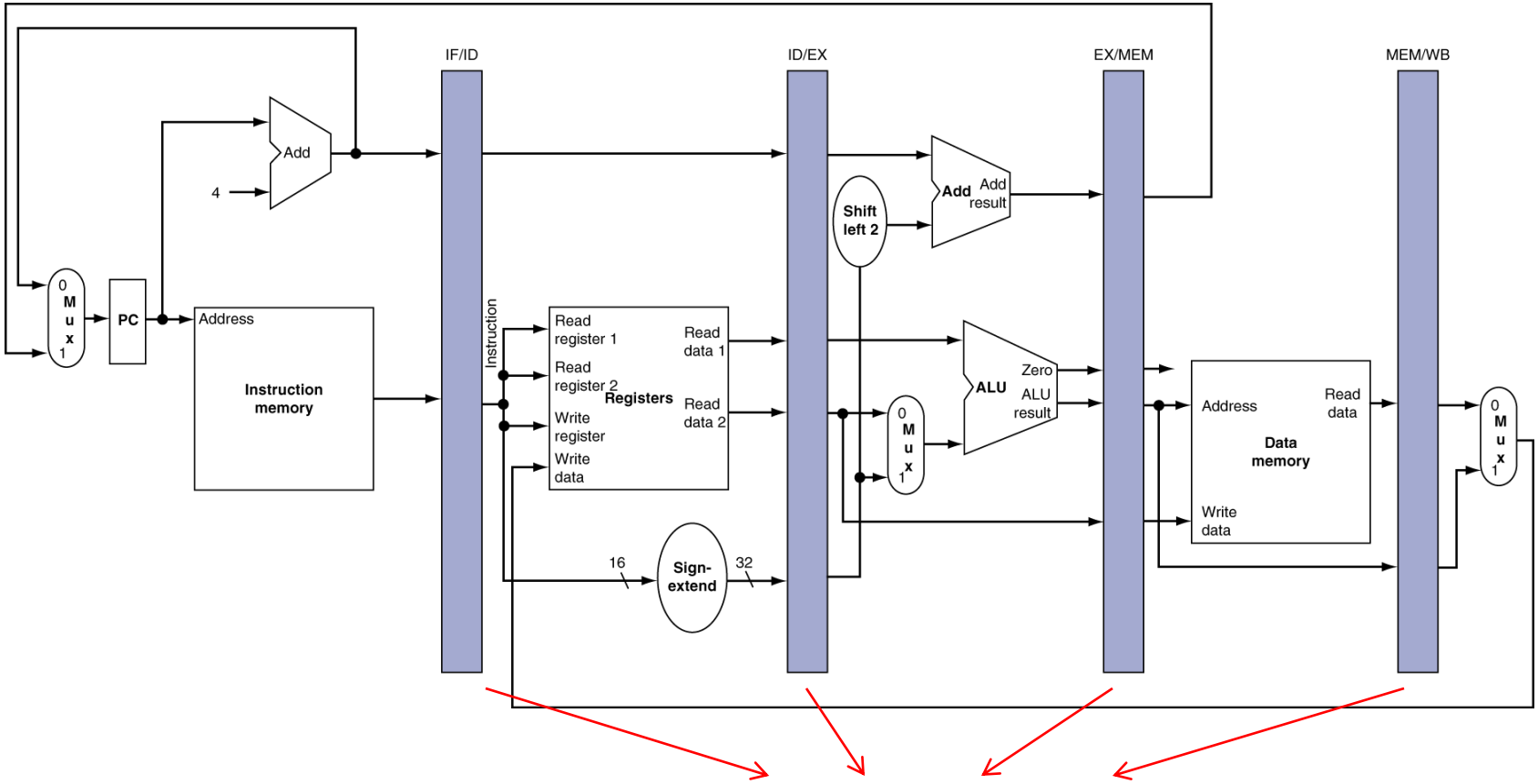
- **We can break the single-cycle datapath into the 5 pipeline stages:**



- *What do we need to add to actually split the datapath into stages? - registers*

Pipelined Datapath

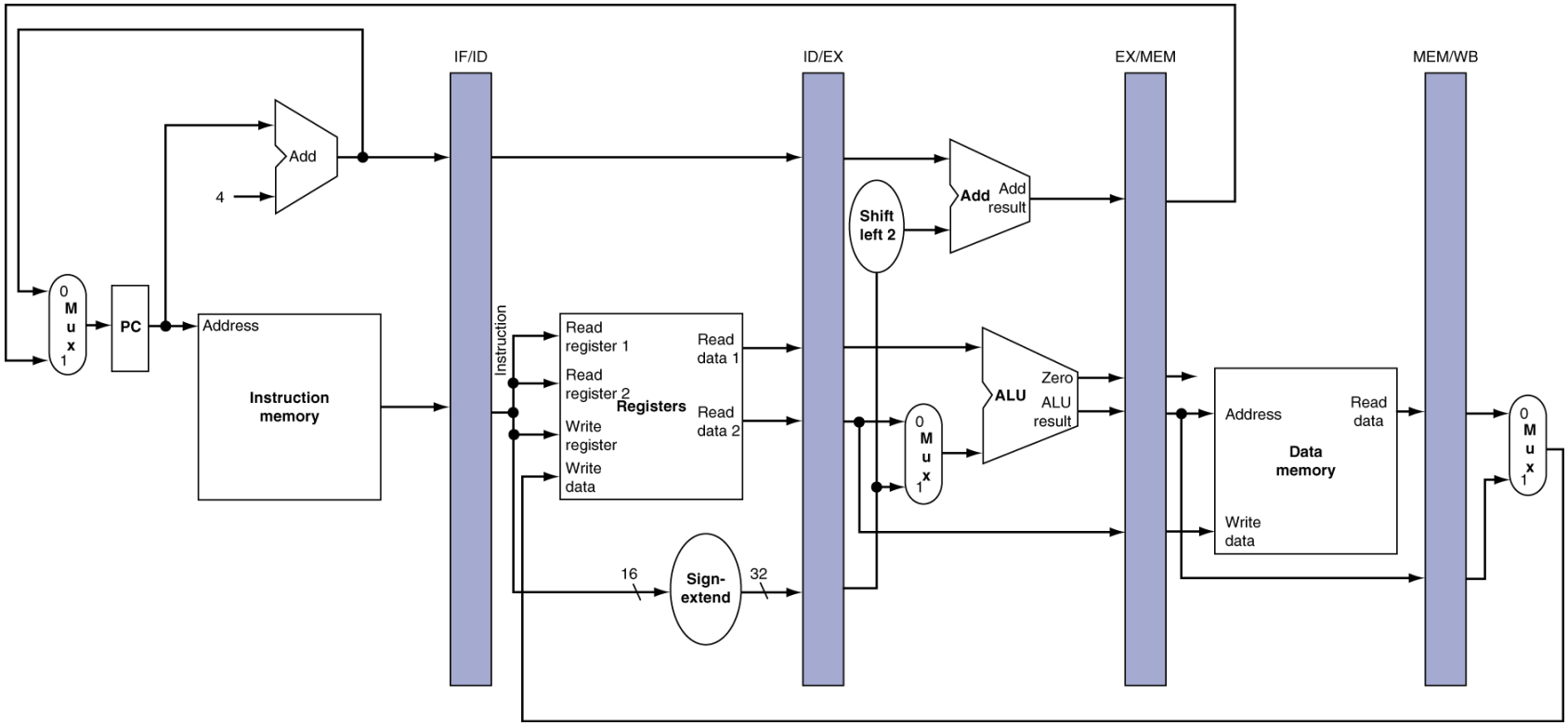
- **In order to store values between stages, we need to put registers between stages. (Note: no registers between WB and IF, since all instructions must update some data to memory, register, or PC)**



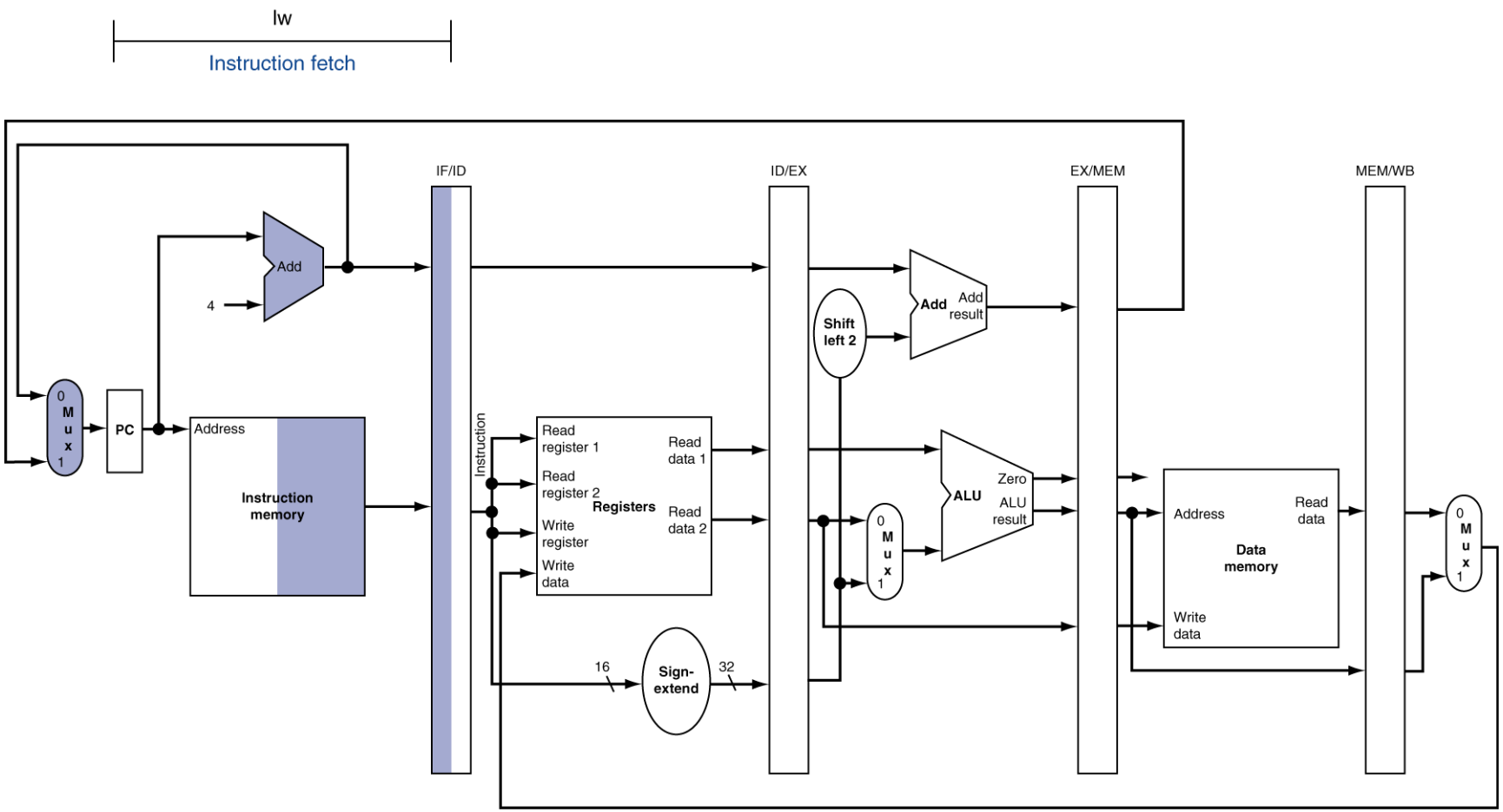
Pipeline registers

There is a problem - Where?

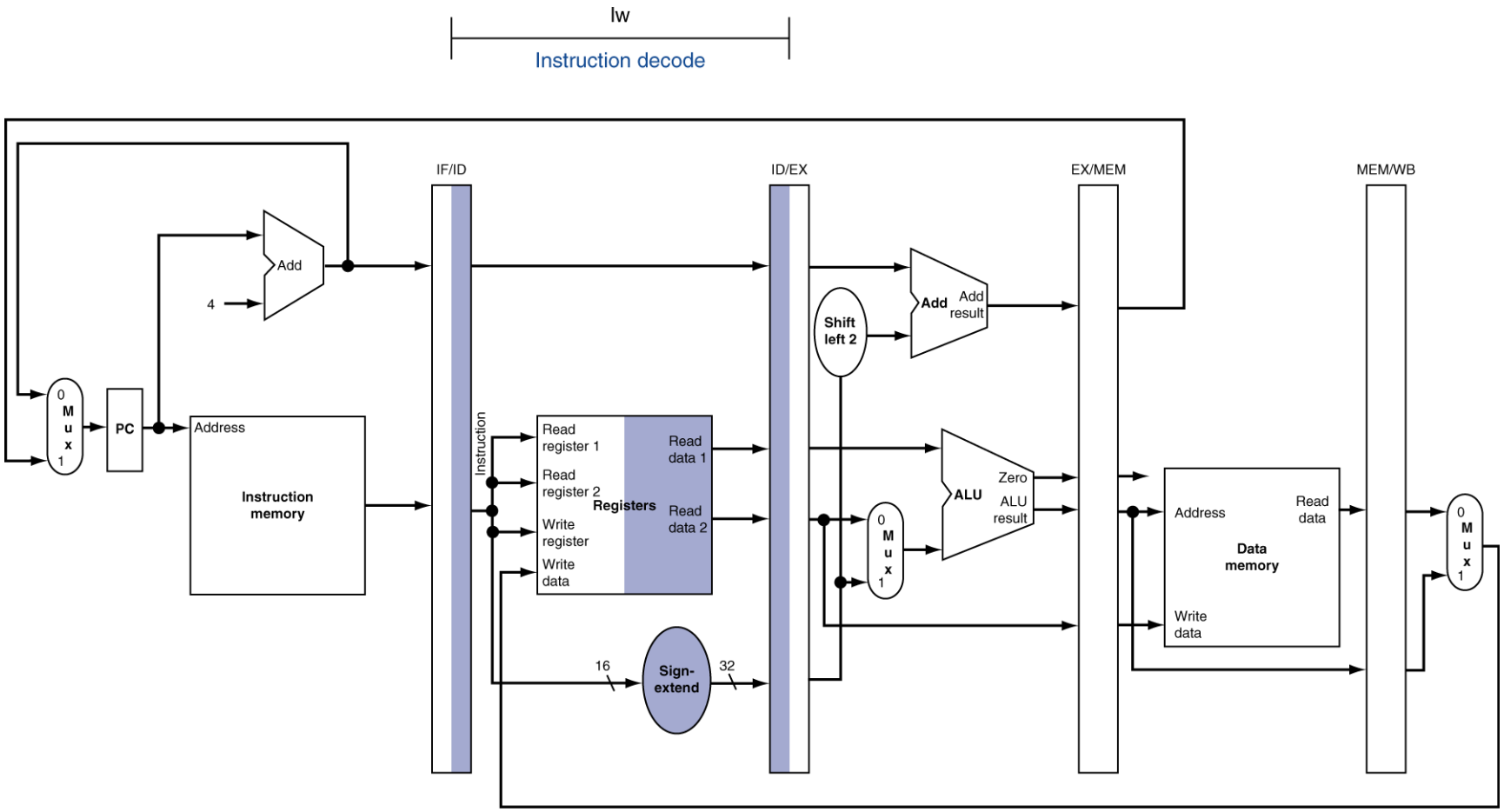
- **The "write register" is changed after the final stage of lw (load).**
- **Therefore, we need to change this data path to preserve the destination register number in the load instruction.**



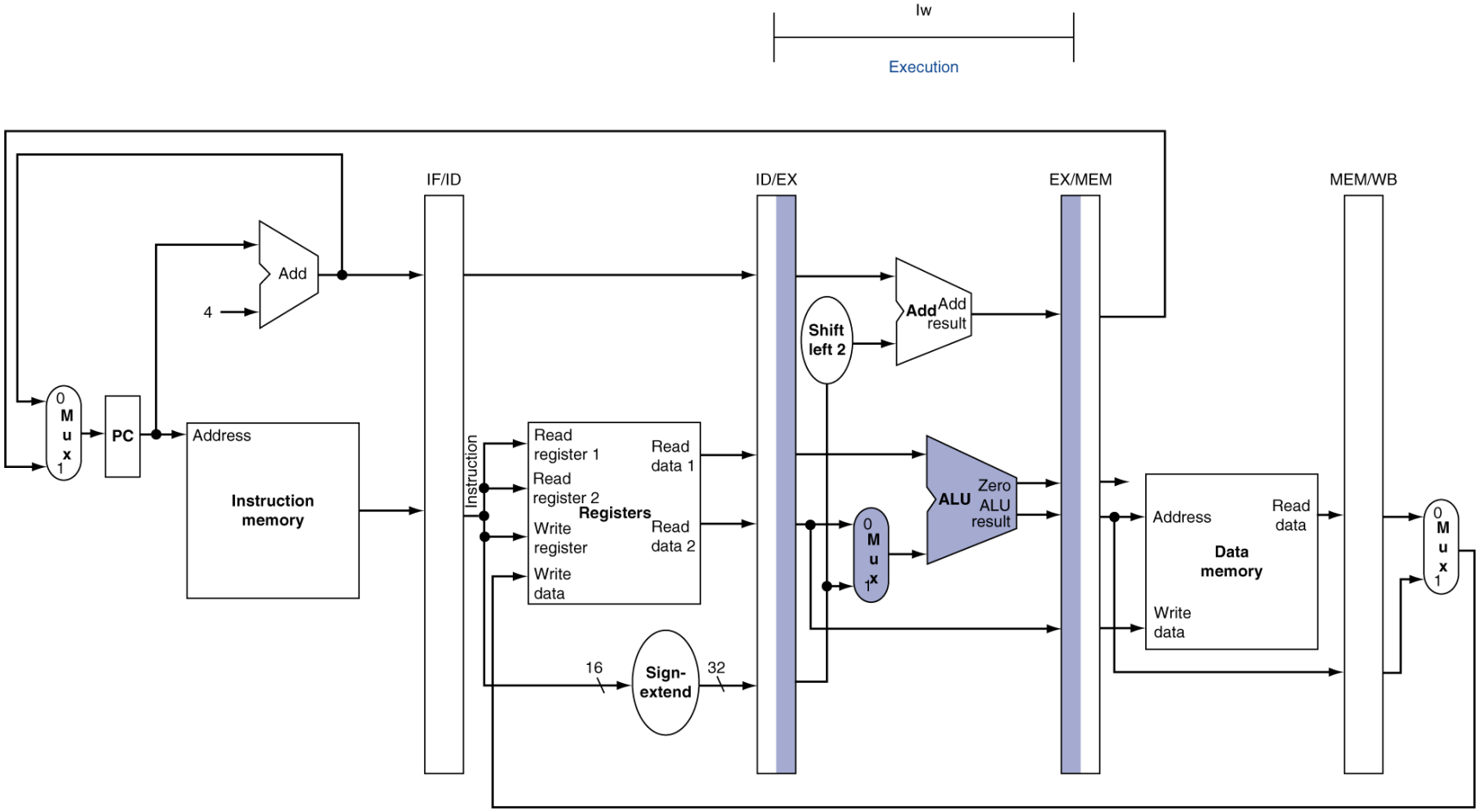
IF for Load, Store, …



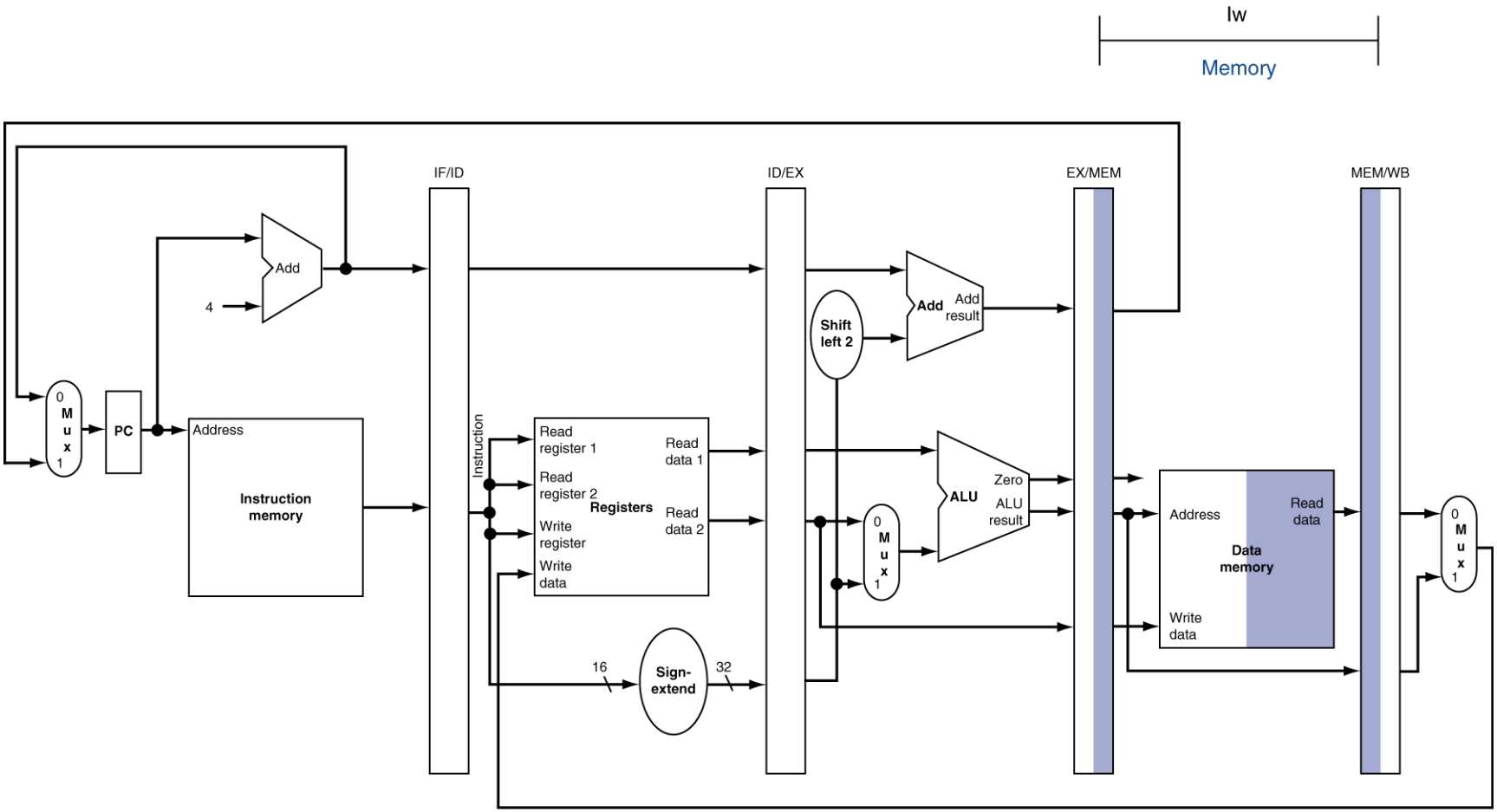
ID for Load, Store, …



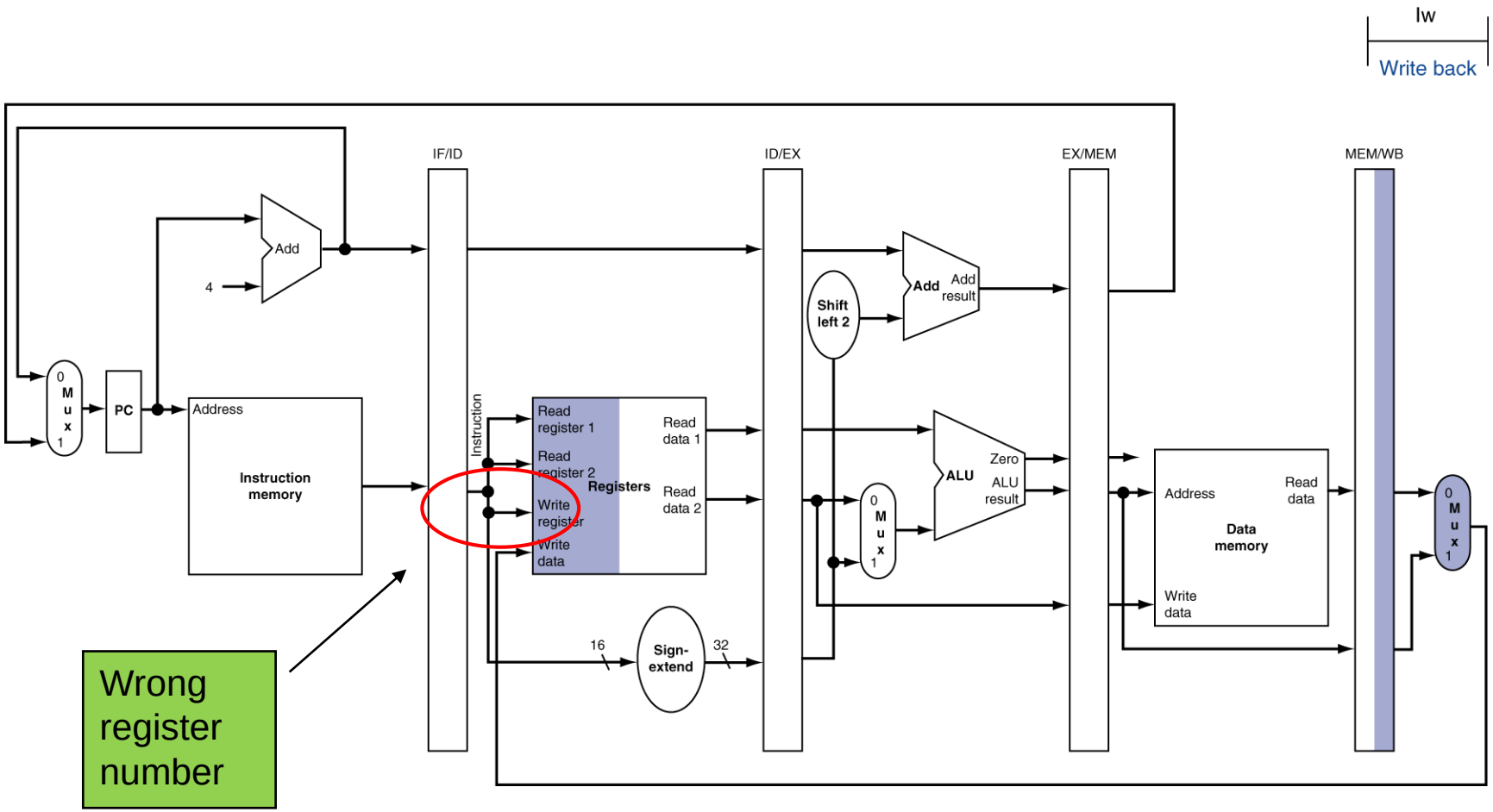
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

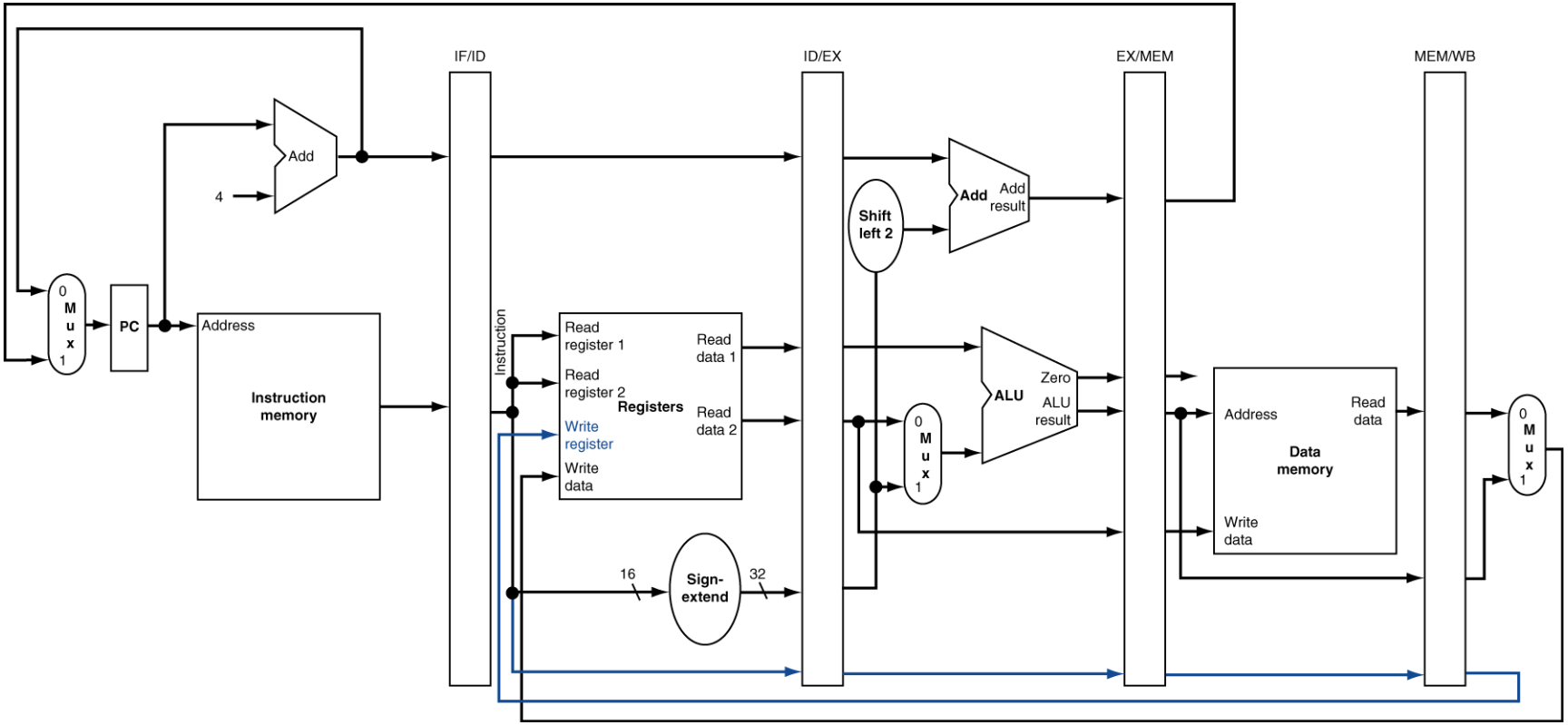
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

WB for Load

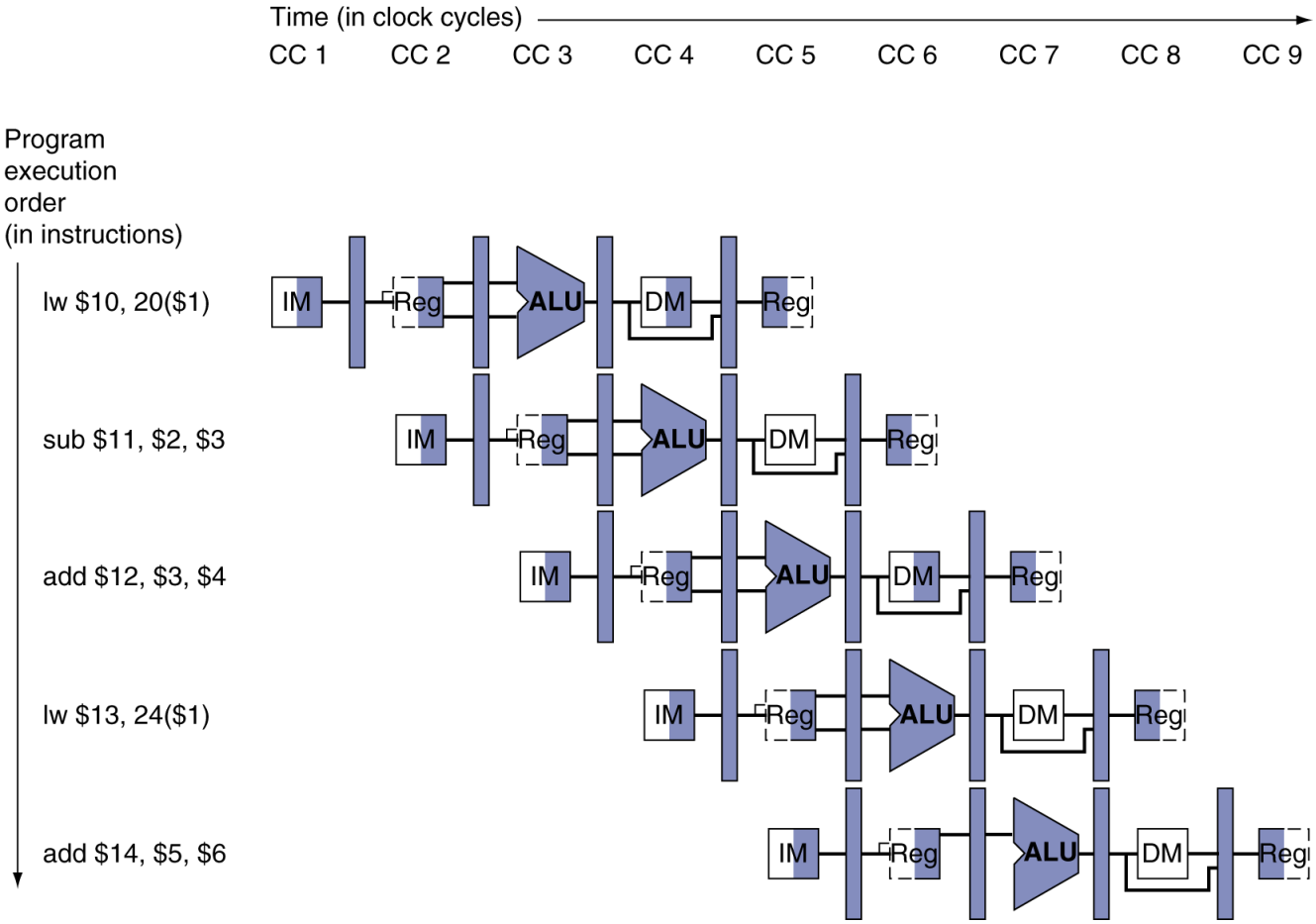

lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
Wrong register number

Corrected Datapath



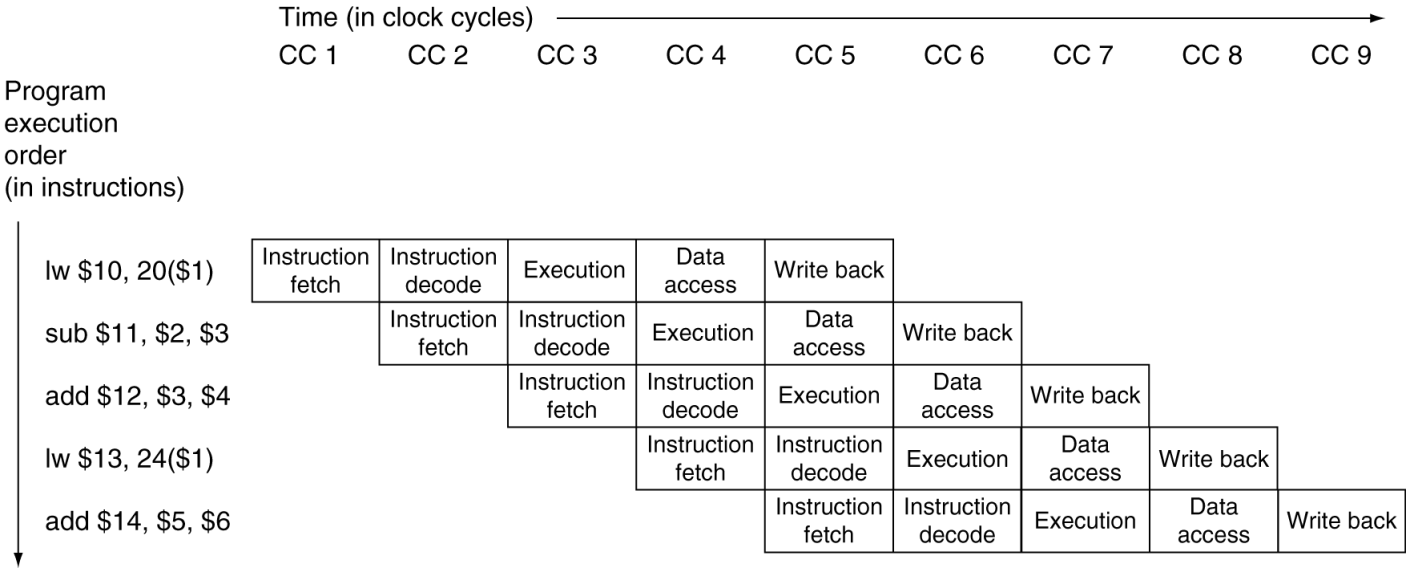
Graphically Representing Pipelines

Form showing resource usage



Graphically Representing Pipelines

- **Traditional form**



Chapter Four

The Processor: Datapath & Control

Concepts Introduced in Chapter 4

- Will go over some of the details for the data path and control of a subset of the MIPS instruction set for two different types of implementations.
  - single cycle implementation
  - multicycle/pipelined implementation

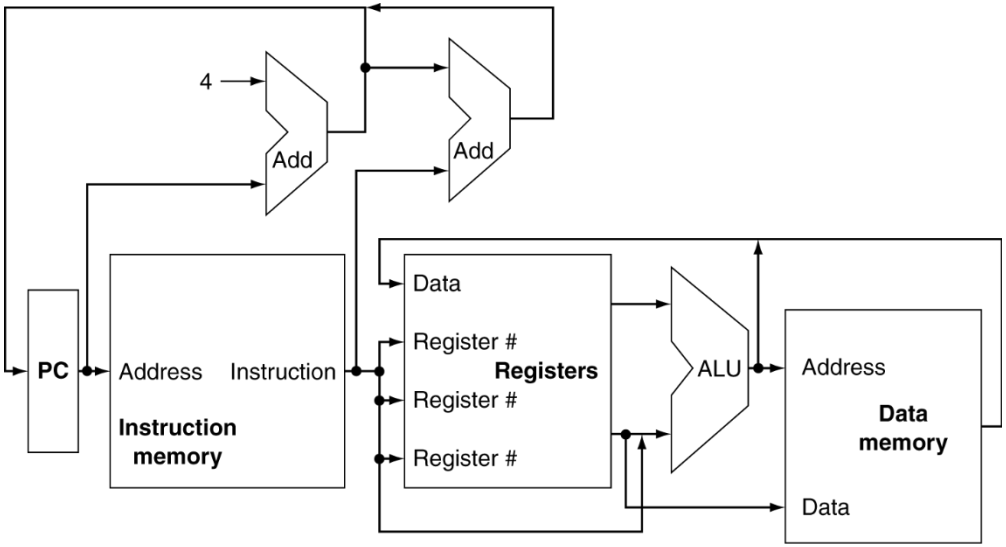
Subset of MIPS Instruction Set

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt, nor`
  - control flow instructions: `beq/bne, j`
- Generic Implementation:
  - 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
- All instructions use the ALU after reading the registers

  Ex: memory-reference, arithmetic, and control flow

More Implementation Details (Fig 4.1)

- **An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.**



- **Two types of logical units:**
  - **Combinational: outputs depend only on the current inputs**
    - ALU
  - **State element: has some internal storage (e.g. Flip-flop)**
    - registers
    - memory
    - PC

Types of Signals

- **Clock: signal to indicate <u>when</u> (rising or falling edge) an action is to be performed**
- **Control: signal to indicate <u>whether</u> an action is to be performed**
  - **Asserted means logically true (not necessary high voltage)**
  - **Deasserted means logically false (not necessary low voltage)**
- **Data: signal containing the actual data to be stored or manipulated**

Figure 4.3:

- **Combinational logic, state elements (registers), and the clock are closely related.**
- **An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values.**

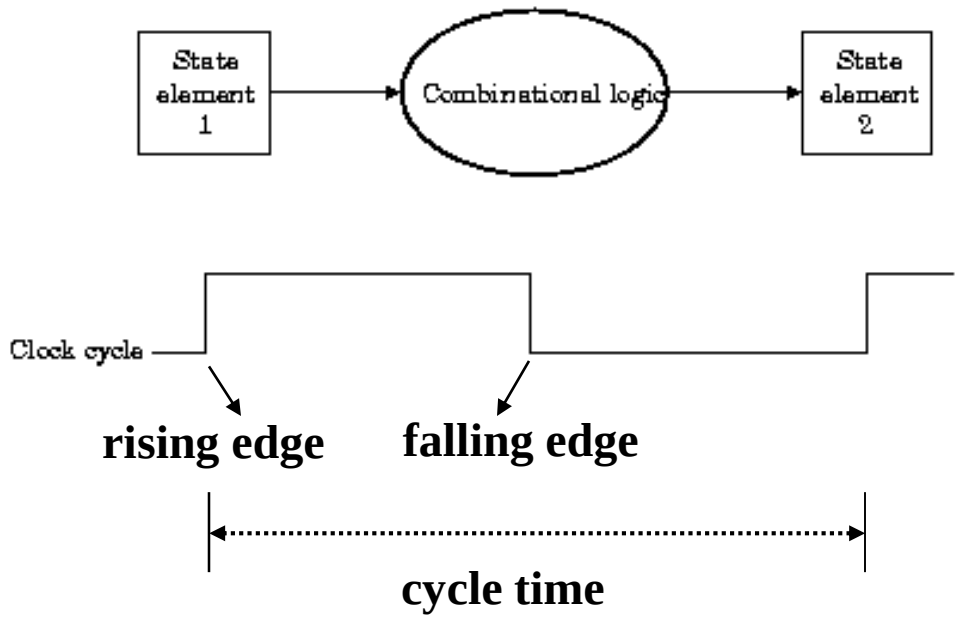
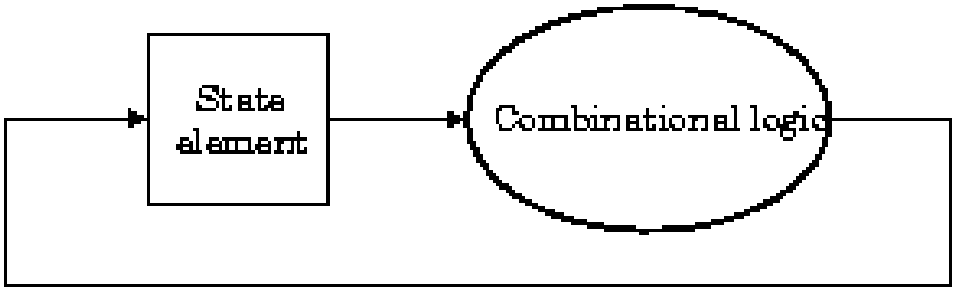


Figure 4.4:

- So a state element can be read, its value sent through combinational logic, and the new value be written back to the state element in a single clock cycle.
- Also, data can be written at the end of previous clock cycle into a state element and read at the beginning of the next cycle from the same state element.

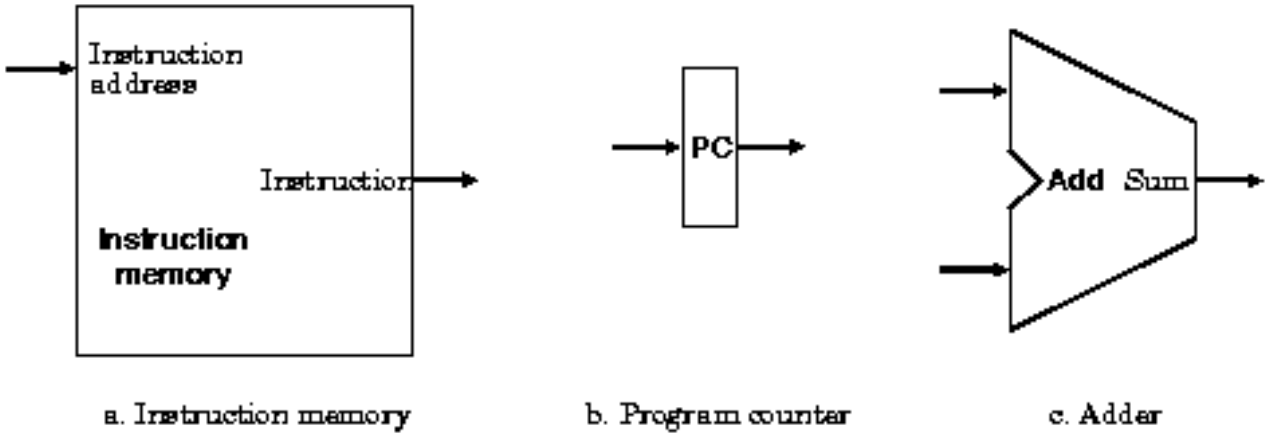


Implementing an Instruction Set

- **A processor consists of**
  - datapath: processes the data signals (read, manipulate, store)
  - control: commands the data path by using control signals
- **Types of Implementations**
  - single cycle (all instructions take the same length of time - inefficient)
  - multiple cycle
  - pipelined

Figure 4.5:

- **Two state elements (registers) are needed to store and access instructions, and an adder is needed to compute the next instruction address.**



a. Instruction memory b. Program counter c. Adder

Figure 4.6

- **A portion of the datapath used for fetching instructions and incrementing the program counter.**

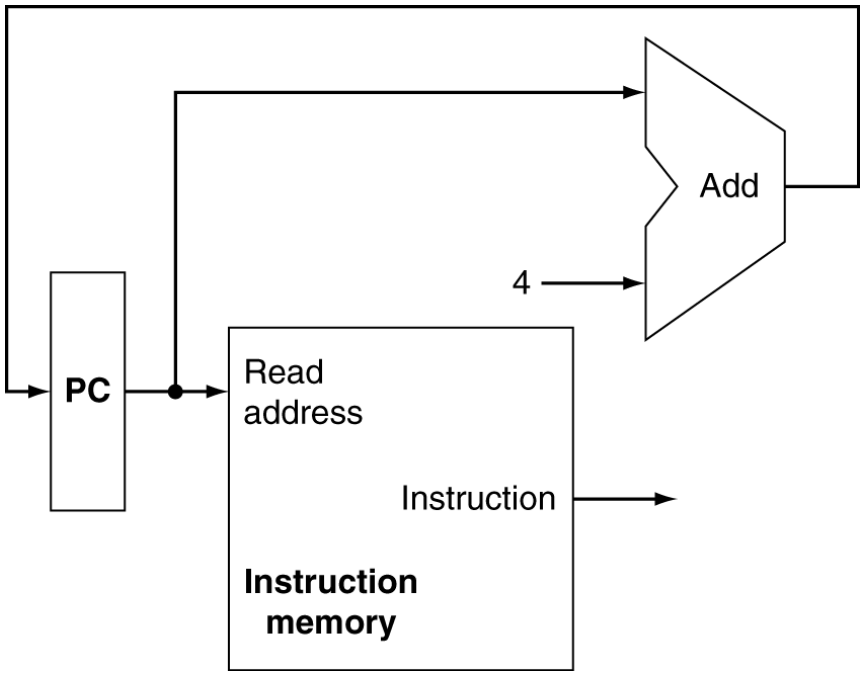
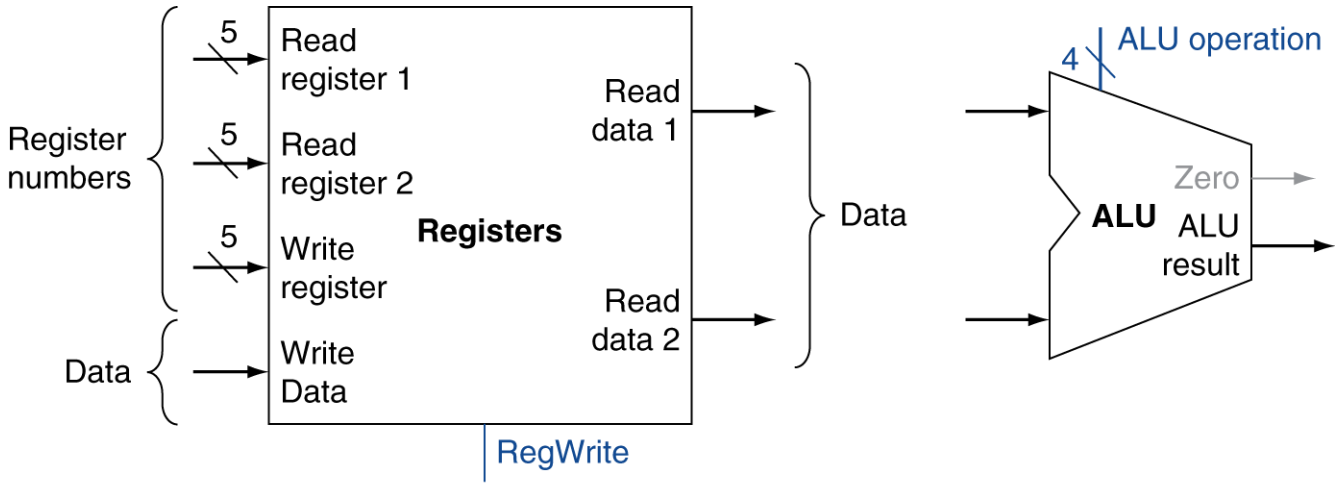


Figure 4.7:

- **The two elements needed to implement R-format ALU operations are the register file and the ALU.**



a. Registers

b. ALU

Datapath for R-type instructions

- **The datapath for R-type instructions.**

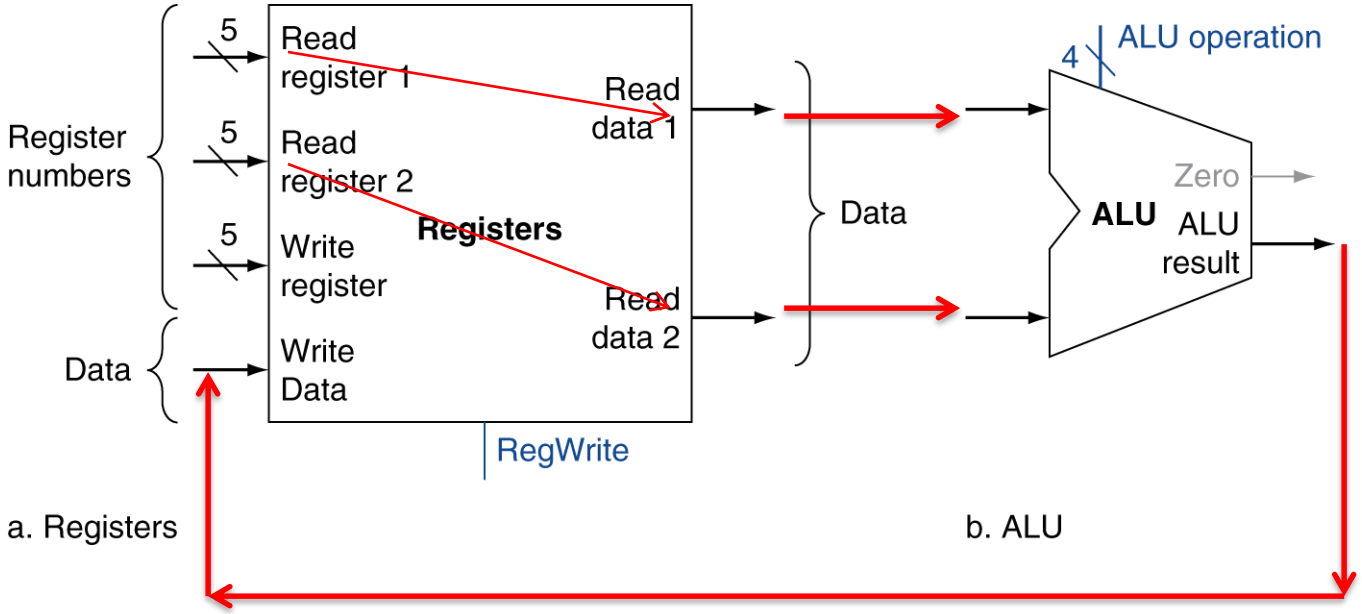
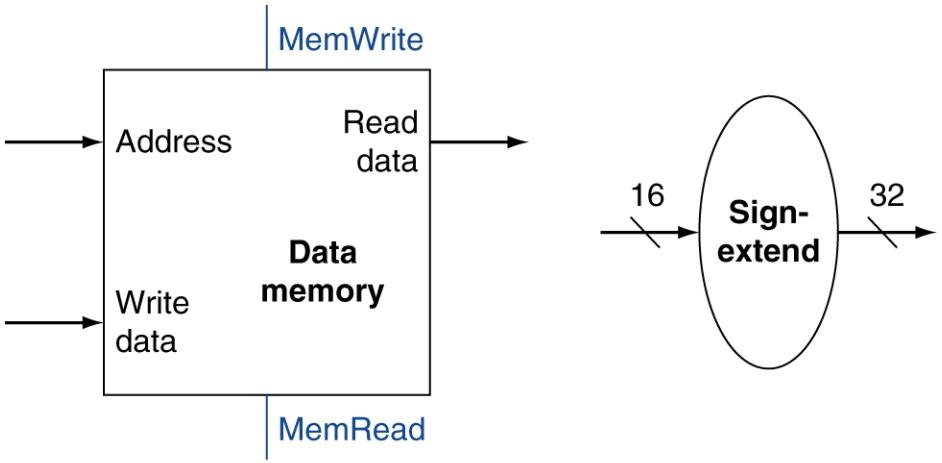


Figure 4.8:

- **The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.6, are the data memory unit and the sign-extension unit.**

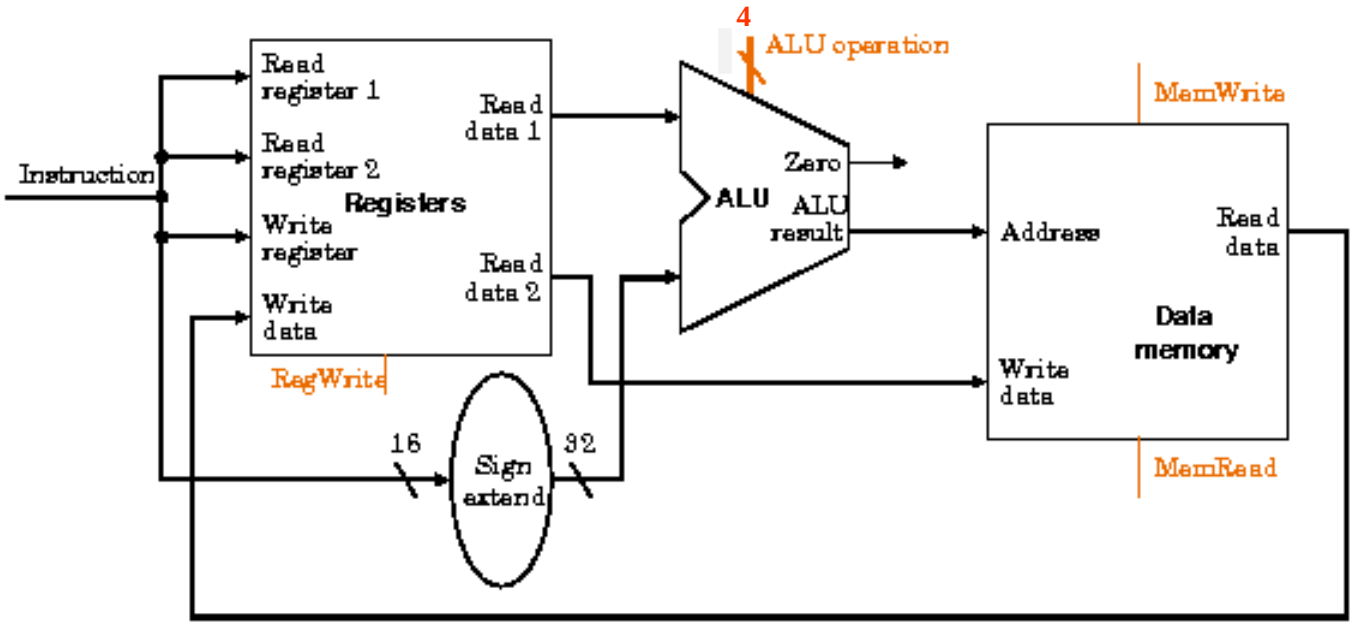


a. Data memory unit

b. Sign extension unit

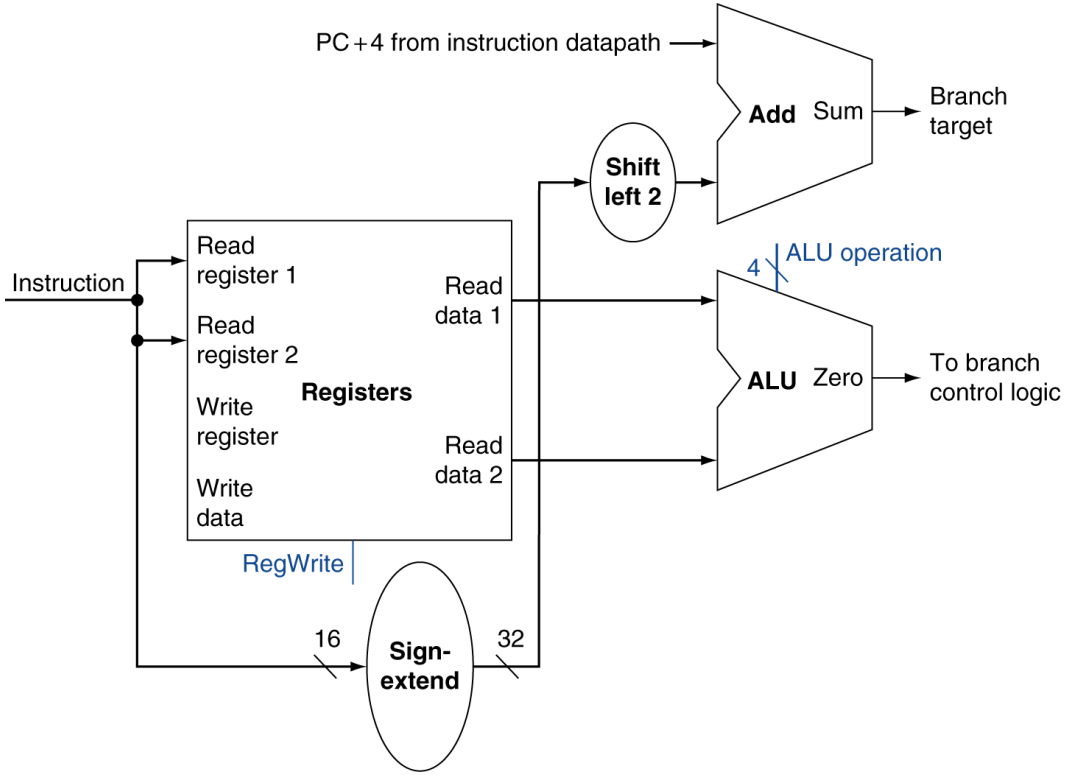
Datapath for load (lw) or store (sw)

- **The datapath for a load or store that does a register access, followed by a memory address calculation, then a read or write from memory, and a write into the register file if the instruction is a load.**



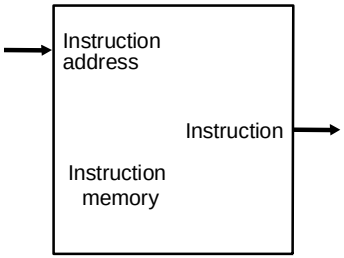
Datapath for a branch (beq) - Fig. 4.9

- **The datapath for a branch uses an ALU for evaluation of the branch condition and a separate adder for computing the branch target as the sum of the incremented PC and ...**

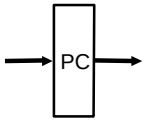


Simple Implementation

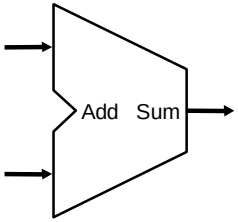
- **Include the functional units we need for each instruction**



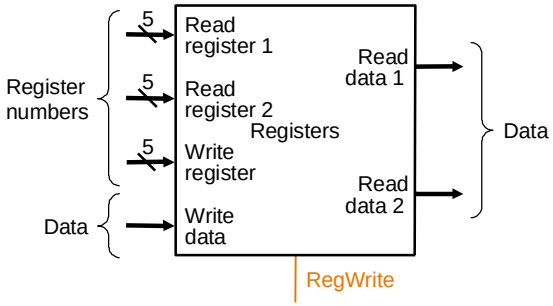
a. Instruction memory



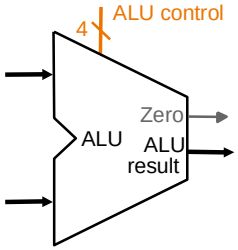
b. Program counter



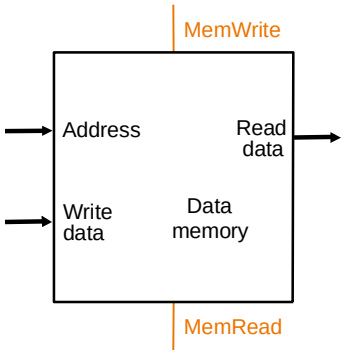
c. Adder



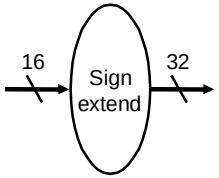
a. Registers



b. ALU



a. Data memory unit



b. Sign-extension unit

Why do we need this stuff?
Answer: to build the datapath.

Fig 4.11: Building the Full Datapath

- **Use multiplexors to stitch them together**

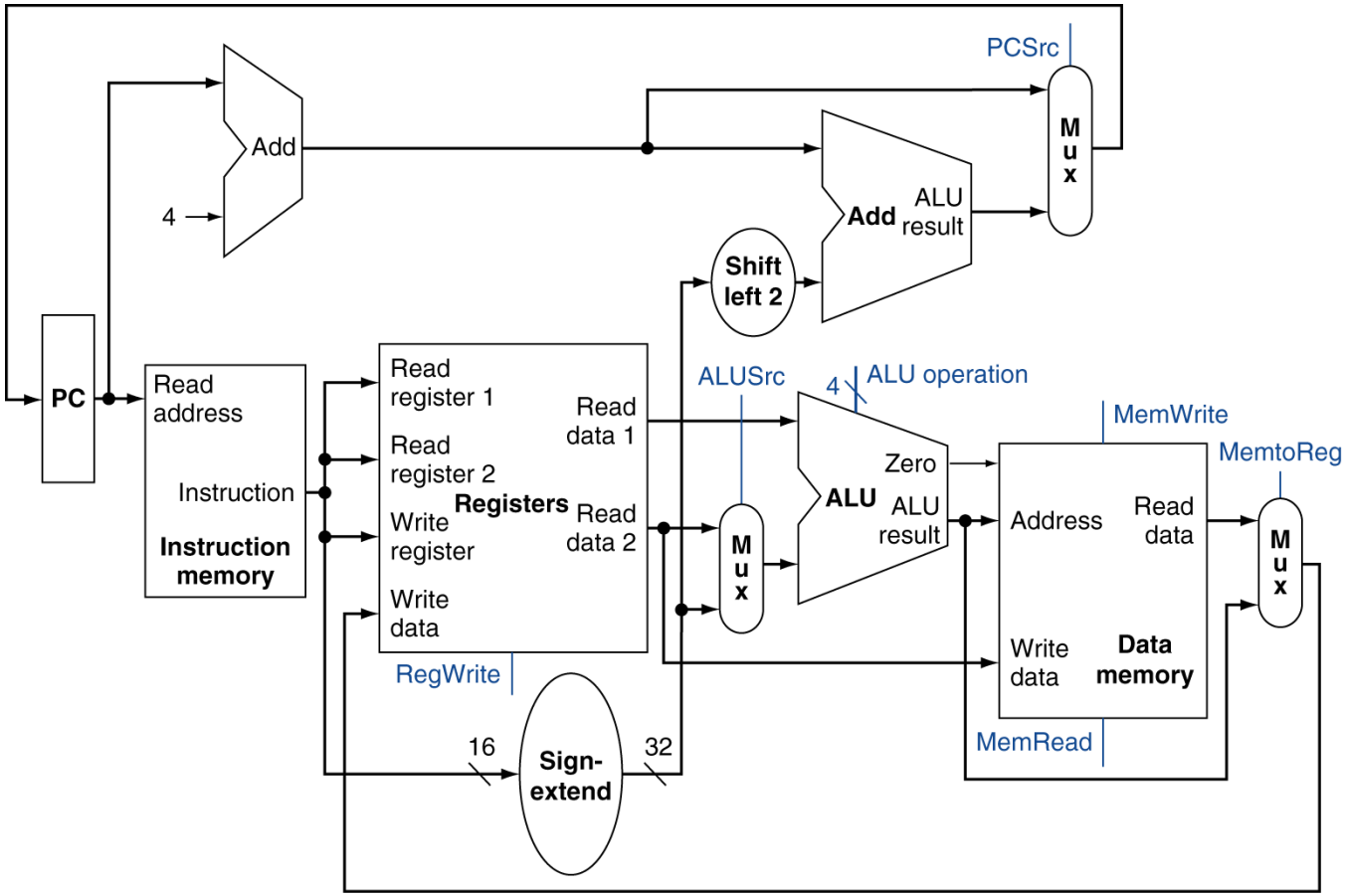


Fig 4.12:

- **How the ALU Control bits are set depending on the ALUOp control bits and the different function codes for the R-type instruction.**

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

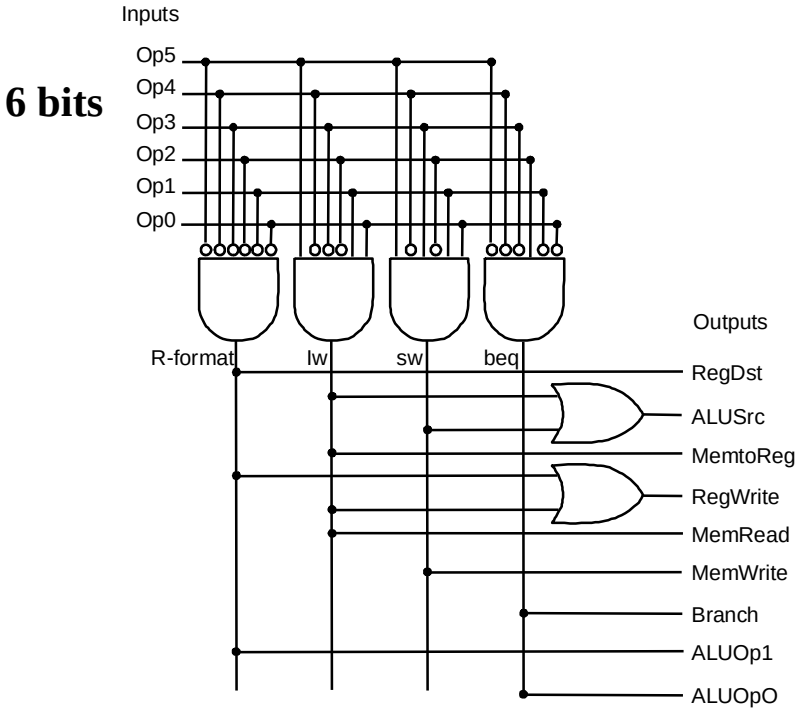
Fig 4.13:

- **The truth table for the 4 ALU Control bits (called Operation).**

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

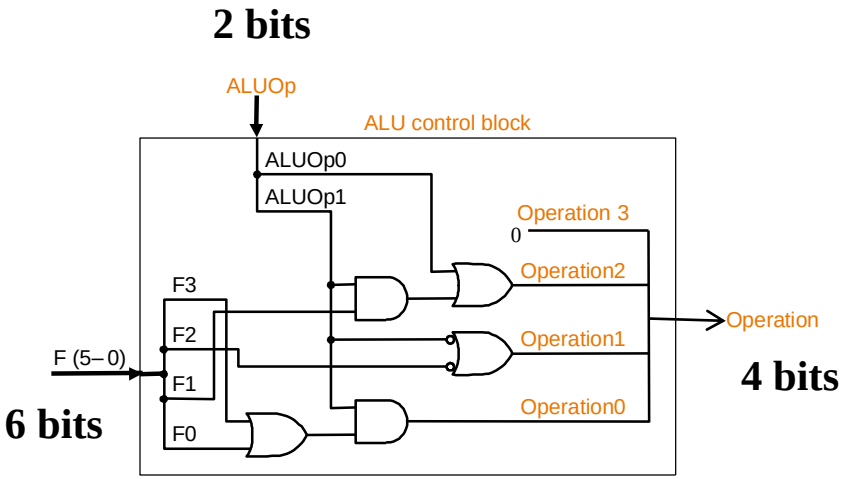
Control

- **Simple combinational logic (truth tables)**



9 bits

Control



ALU Control

Steps for an R-type Instruction

- **An instruction is fetched from instruction memory and the PC is incremented (PC points to the next instruction - 4 bytes later).**
- **Read two source register values from the register file.**
- **Perform the ALU operation on the register data operands.**
- **Write the result of the ALU operation to the register file.**

Steps for a Load Instruction (I-type)

- **An instruction is fetched from instruction memory and the PC is incremented (PC points to the next instruction - 4 bytes later).**
- **Read a source register value from the register file and sign extend the lower 16 bits of the instruction.**
- **Perform the ALU operation that computes the sum of the value in the register and the sign-extended lower 16 bits of the instruction.**
- **Access data memory at the address of the sum from the ALU**
- **Write the result of the ALU operation to the register file.**

Steps for a Branch Instruction

- **An instruction is fetched from instruction memory and the PC is incremented.**
- **Read two source register values from the register file and sign extend the lower 16 bits of the instruction.**
- **The ALU performs a subtract on the data values read from the register file. The value of PC+4 is to the sign-extended lower 16 bits of the instruction, which results in the branch target address.**
- **The ZERO result from the ALU is used to decide which adder result to store into the PC.**

Steps for a Jump Instruction

- **An instruction is fetched from instruction memory and the PC is incremented.**
- **Concatenates the upper four bits of the PC, the lower 26 bits of the instruction,and two zero bits. Assigns the result to the PC.**

The Five Stages of MIPS Instructions

- **IF: Instruction Fetch**
  - **Fetch the instruction from the Instruction Memory**
- **ID: Registers Fetch and Instruction Decode**
- **EX: ALU Execution or Address calculation**
- **MEM: Data Memory Assess**
- **WB: Write the data back to the register**

Control: R-format (Add, Sub, And, Or, Slt)

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

add $8, $17, $18 Instruction Format:

000000	10001	10010	01000	00000	100000

op	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

- ALU's operation based on instruction type and function code

Control: I-format (lw, sw, Branch)

- e.g., what should the ALU do with this instruction
- Example: lw $1, 100($2)

35	2	1	100

op	rs	rt	16 bit offset
31-26	25-21	20-16	15-0

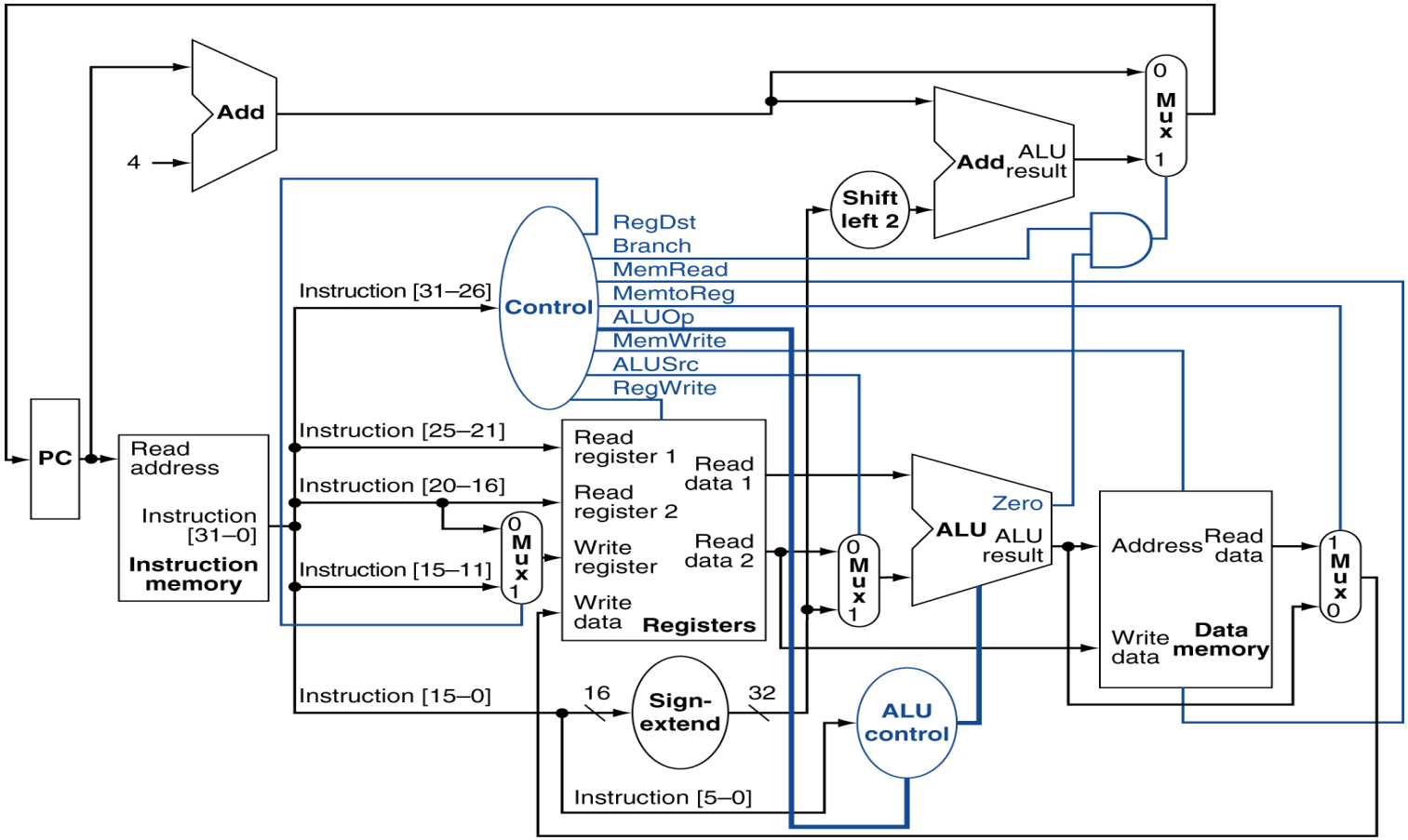
Note: lw : opcode = 35
sw : opcode = 43
branch : opcode = 4

Fig 4.16

- **The effect of each of the seven control signals.**

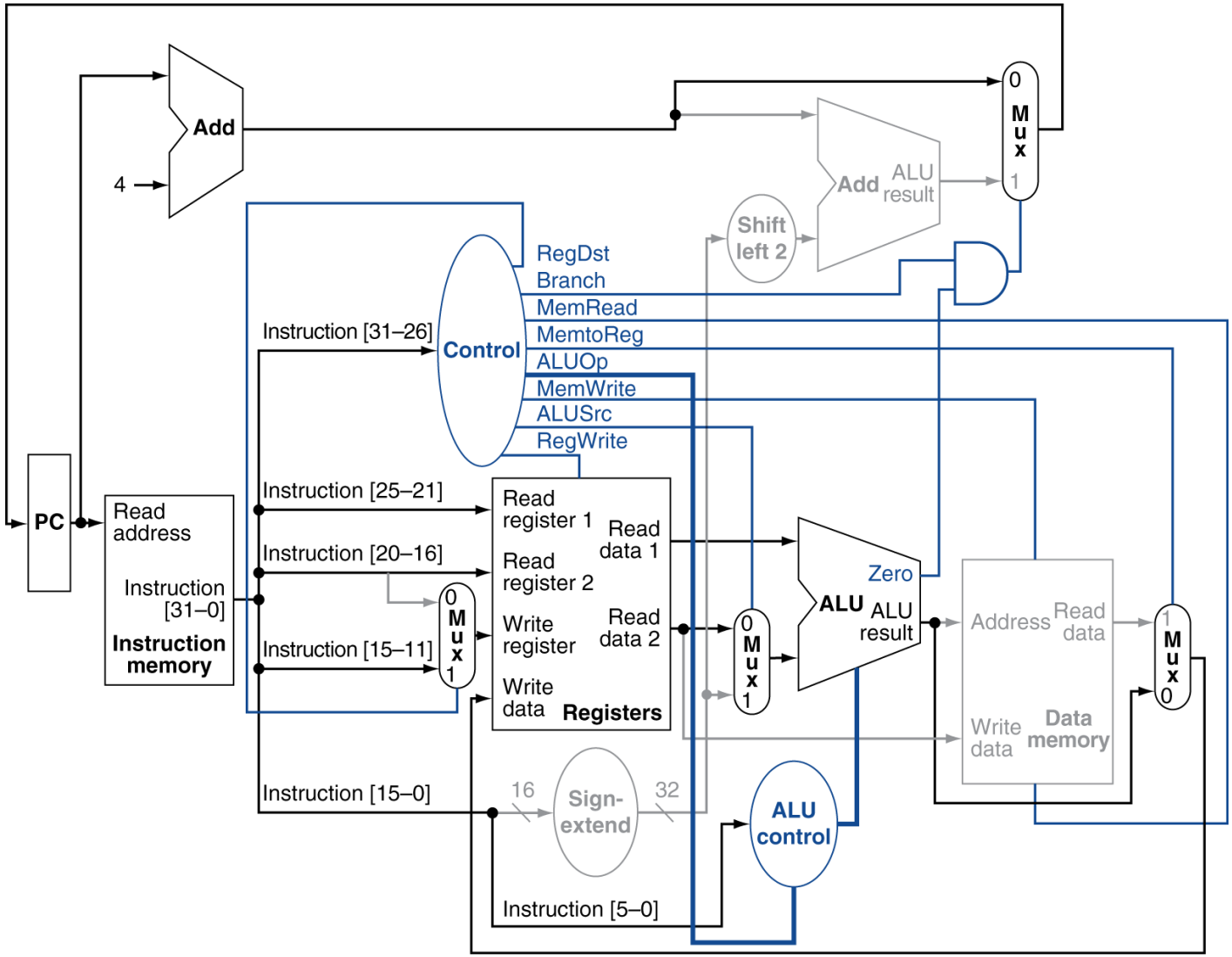
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Datapath with Control (Fig 4.17 & Fig 4.18)

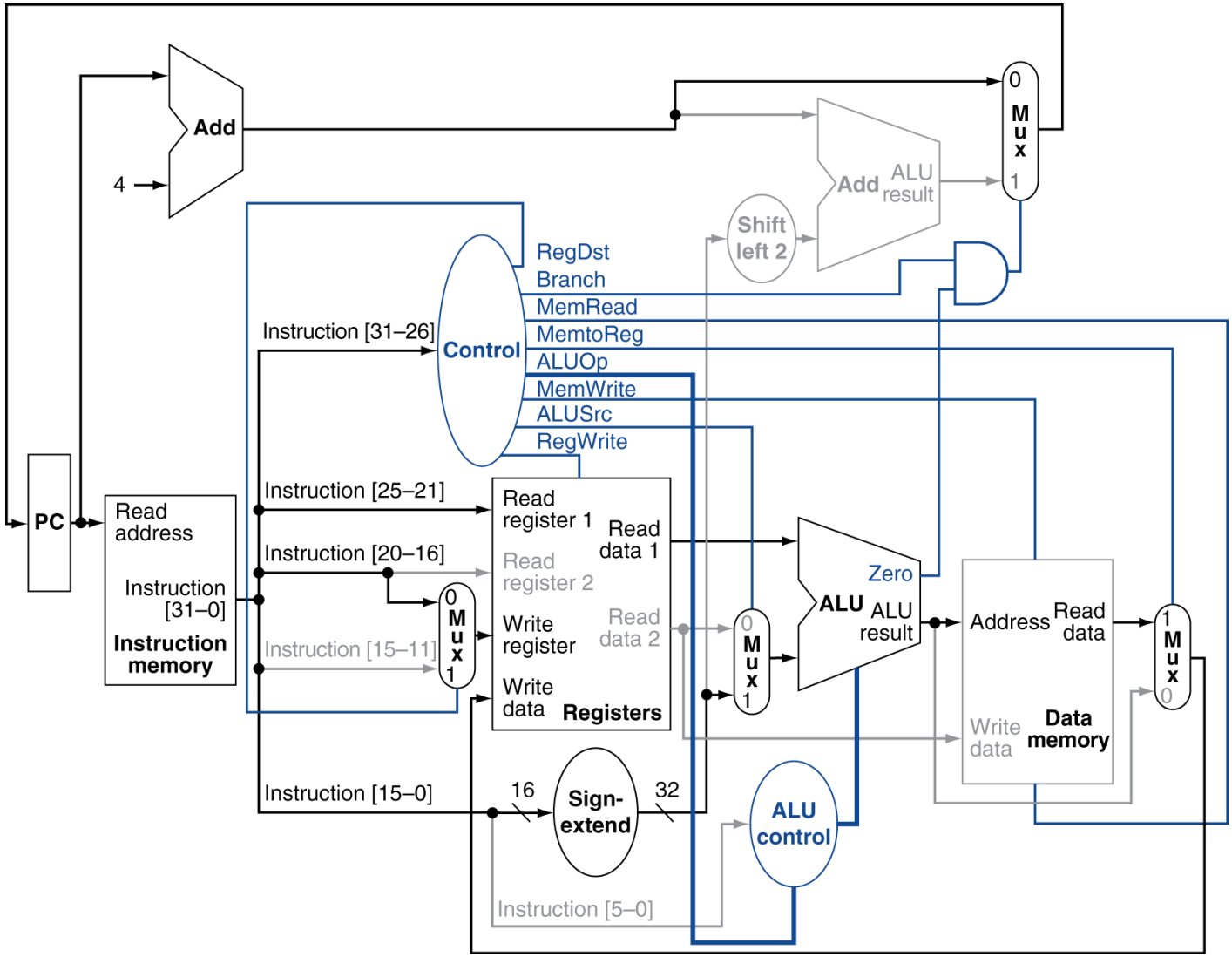


Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

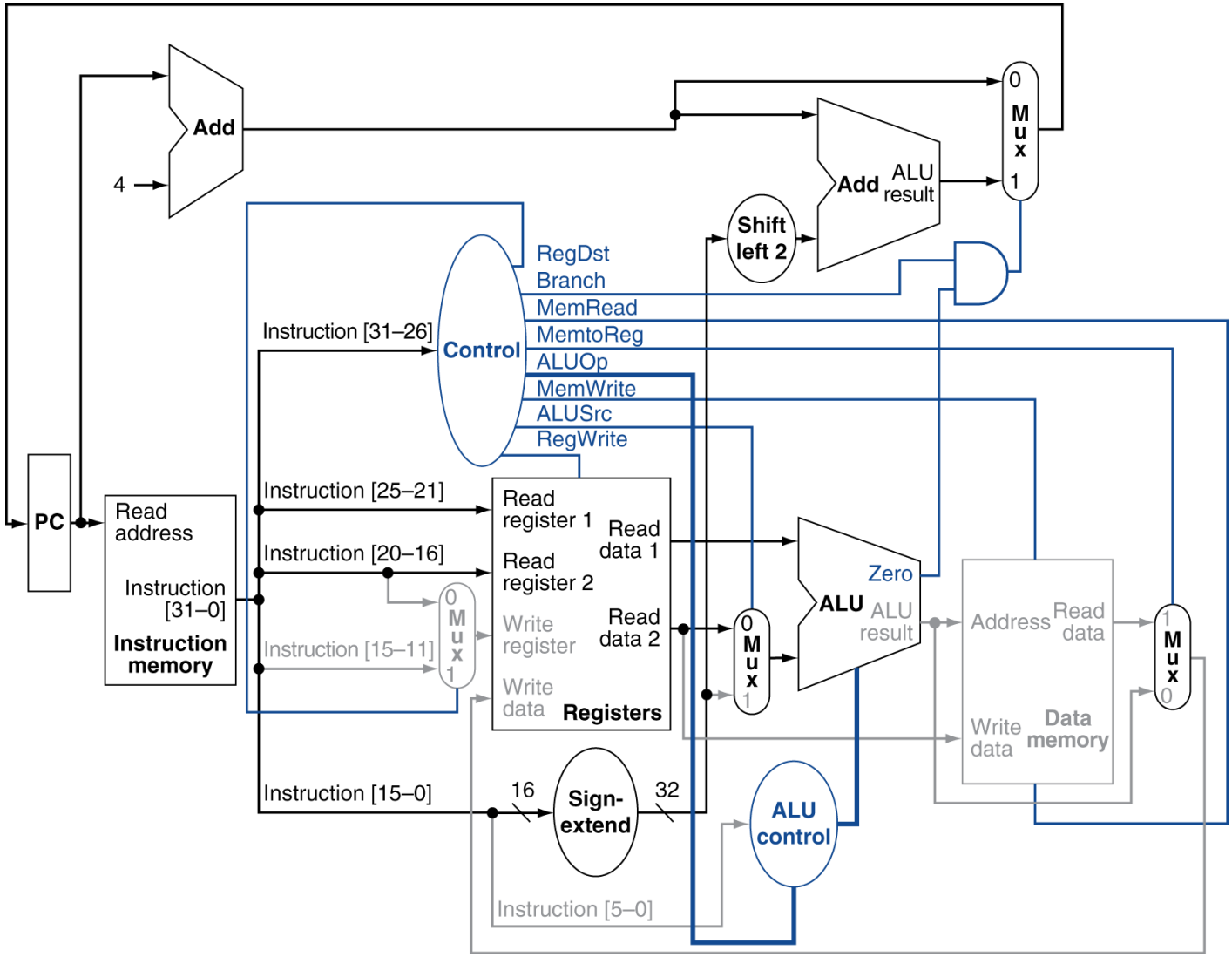
Datapath for R-Type Instruction – Fig. 4.19



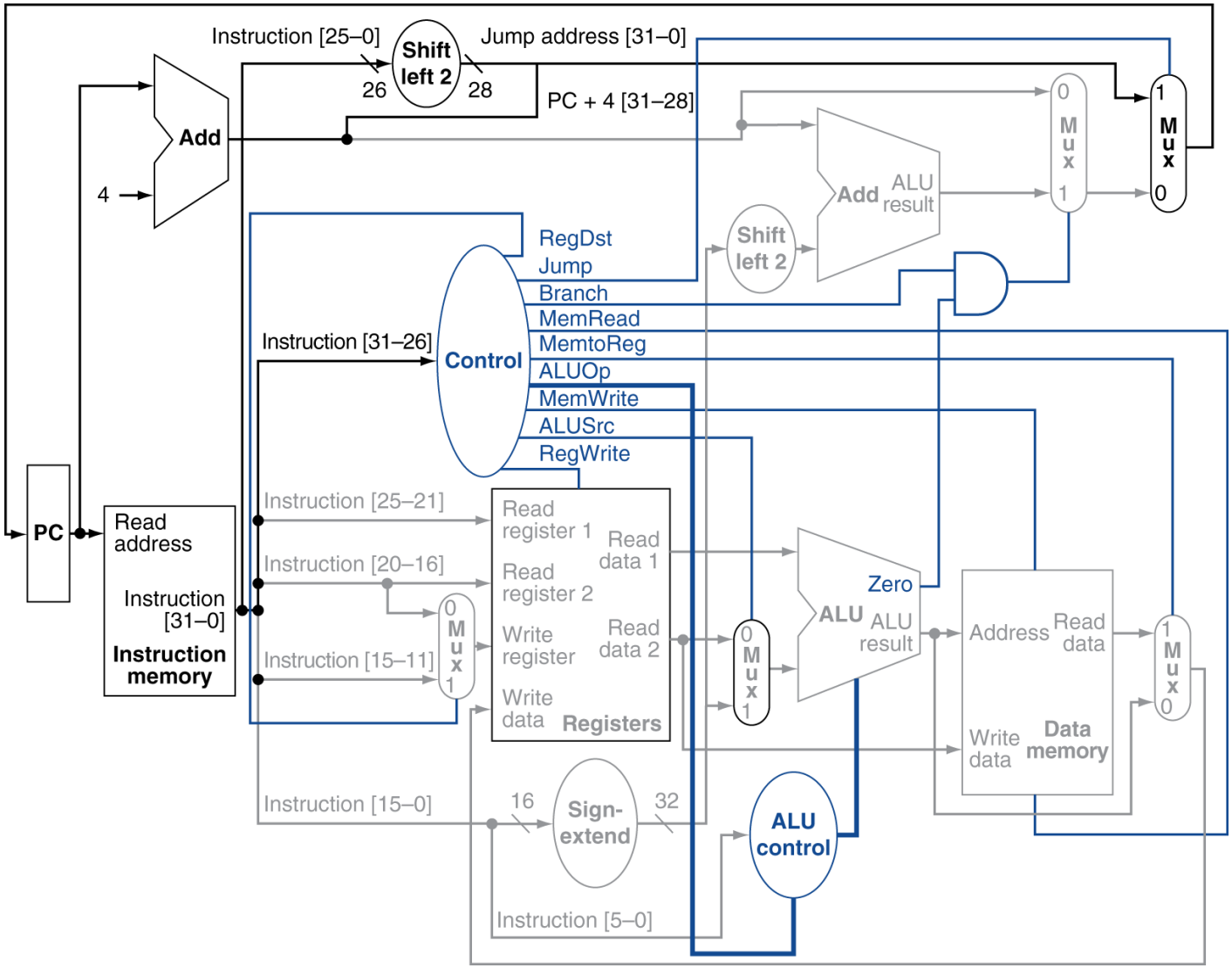
Datapath for Load Instruction – Fig. 4.20



Datapath for beq – Fig. 4.21



Datapath with jump added – Fig. 4.24



Chapter 4-1 Summary

- **If we understand the instructions…**

  We can build a simple processor!
- **If instructions take different amounts of time, multi-cycle is better**
- **Datapath implemented using:**
  - **Combinational logic for arithmetic**
  - **State holding elements to remember bits**
- **Control implemented using:**
  - **Combinational logic for single-cycle implementation**
  - **Finite state machine for multi-cycle implementation (Skipped)**

Example: let $a = 0001\ 1010\ 0011\ 0011$ $\rightarrow C_0$

$b = 1110\ 0101\ 1110\ 1011$

What is the Carry Out$_{16}$ (C_4)? $\rightarrow$ Note: $C_4 = G_3 + (P_3 \cdot G_2) + (P_3 P_2 G_1) + (P_3 P_2 P_1 G_0) + (P_3 P_2 P_1 P_0 C_0)$

Answer: let $g_i = a_i \cdot b_i$, $p_i = a_i + b_i$

So,

$$
\begin{array}{lllll}
a = & 0001 & 1010 & 0011 & 0011 \quad \rightarrow C_0 \\
b = & 1110 & 0101 & 1110 & 1011 \\
\hline
g_i = & 0000 & 0000 & 0010 & 0011 \quad \rightarrow (\text{AND}) \\
 & g_{15} & g_{11} & g_7 & g_3 g_2 g_1 g_0 \\
p_i = & 1111 & 1111 & 1111 & 1011 \quad \rightarrow (\text{OR}) \\
 & p_{15} & & & p_3 p_2 p_1 p_0
\end{array}
$$

let
$$
\left.
\begin{aligned}
P_0 &= p_3 p_2 p_1 p_0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0 \\
P_1 &= p_7 p_6 p_5 p_4 = 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\
P_2 &= p_{11} p_{10} p_9 p_8 = 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\
P_3 &= p_{15} p_{14} p_{13} p_{12} = 1 \cdot 1 \cdot 1 \cdot 1 = 1
\end{aligned}
\right\} \text{Super propagates}
$$

A use a 4-bit ALU to be a block:

Now,
$$G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 = 0$$
$$G_1 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 = 1$$
$$G_2 = g_{11} + p_{11} g_{10} + p_{11} p_{10} g_9 + p_{11} p_{10} p_9 g_8 = 0$$
$$G_3 = g_{15} + p_{15} g_{14} + p_{15} p_{14} g_{13} + p_{15} p_{14} p_{13} g_{12} = 0$$

Thus,
$$
\begin{aligned}
C_4 &= G_3 + (P_3 \cdot G_2) + (P_3 P_2 G_1) + (P_3 P_2 P_1 G_0) + (P_3 P_2 P_1 P_0 C_0) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\
&= 0 + 0 + 1 + 0 + 0 \\
&= 1
\end{aligned}
$$

See slide #5 for C_4

Chapter 3 (continue)

Arithmetic for Computers

Problem: ripple carry adder is slow

- **Is a 32-bit ALU as fast as a 1-bit ALU?**

  - no, since the sum must wait for the sequential Carry Ins/Outs from the adders

Can you see the ripple? How could you get rid of it? – use Fast-Carry

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$, $c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$, $c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$, $c_4 =$

:

:

$c_{31} = b_{30}c_{30} + a_{30}c_{30} + a_{30}b_{30}$, $c_{31} =$

Not feasible!

(See Appendix B)

Carry-lookahead adder

- An approach: Carry-lookahead adder
- From previous slide:

$$c_{i+1} = b_i c_i + a_i c_i + a_i b_i$$
$$= a_i b_i + a_i c_i + b_i c_i$$
$$= a_i b_i + (a_i + b_i) c_i$$

- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple (4 bits)?

$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 c_1$, $c_2 =$

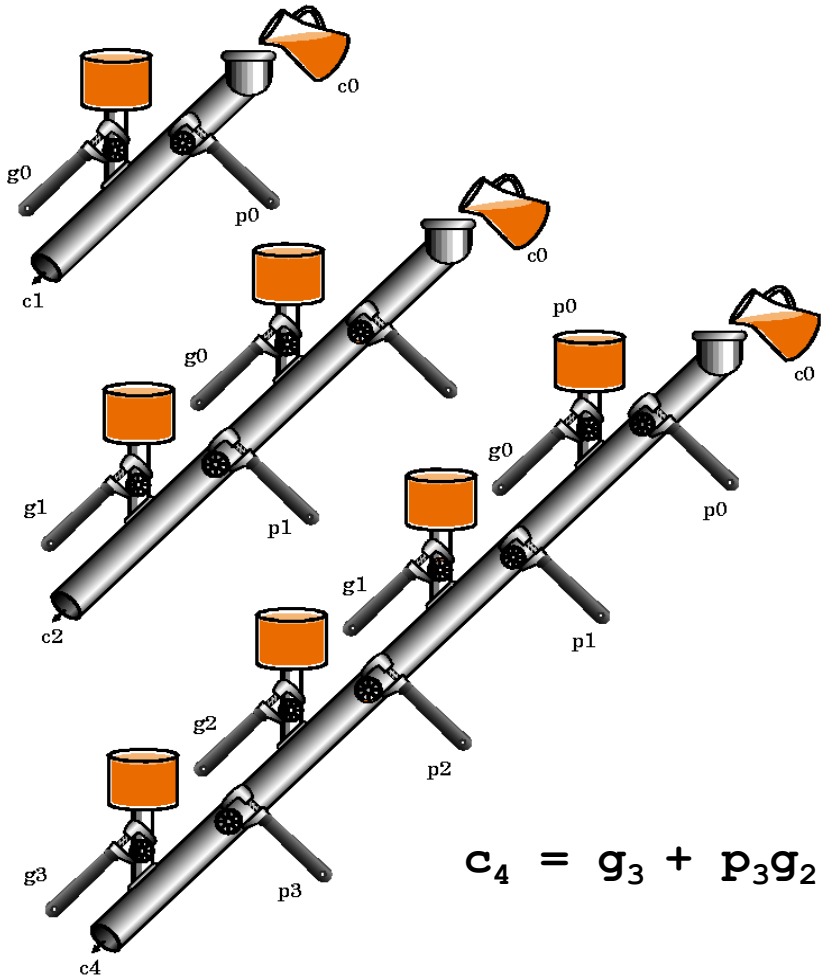
$c_3 = g_2 + p_2 c_2$, $c_3 =$

$c_4 = g_3 + p_3 c_3$, $c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$

Feasible!

Carry Lookahead

- **A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water, pipes, and valves.**



$$c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

Carry Lookahead 16-bit adder

- **Next, let's group these 4-bit ALUs to be a block:**

$$c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

Let $G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$

$P_0 = p_3p_2p_1p_0$

$C_1 = c_4$

for 4 bits (1 group)

Thus, $C_1 = G_0 + P_0c_0$

- **Note:**

$$G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$$
$$G_1 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4$$
$$G_2 = g_{11} + p_{11}g_{10} + p_{11}p_{10}g_9 + p_{11}p_{10}p_9g_8$$
$$G_3 = g_{15} + p_{15}g_{14} + p_{15}p_{14}g_{13} + p_{15}p_{14}p_{13}g_{12}$$
$$P_0 = p_3p_2p_1p_0$$
$$P_1 = p_7p_6p_5p_4$$
$$P_2 = p_{11}p_{10}p_9p_8$$
$$P_3 = p_{15}p_{14}p_{13}p_{12}$$

for 16 bits (4 groups)

Carry Lookahead 16-bit adder

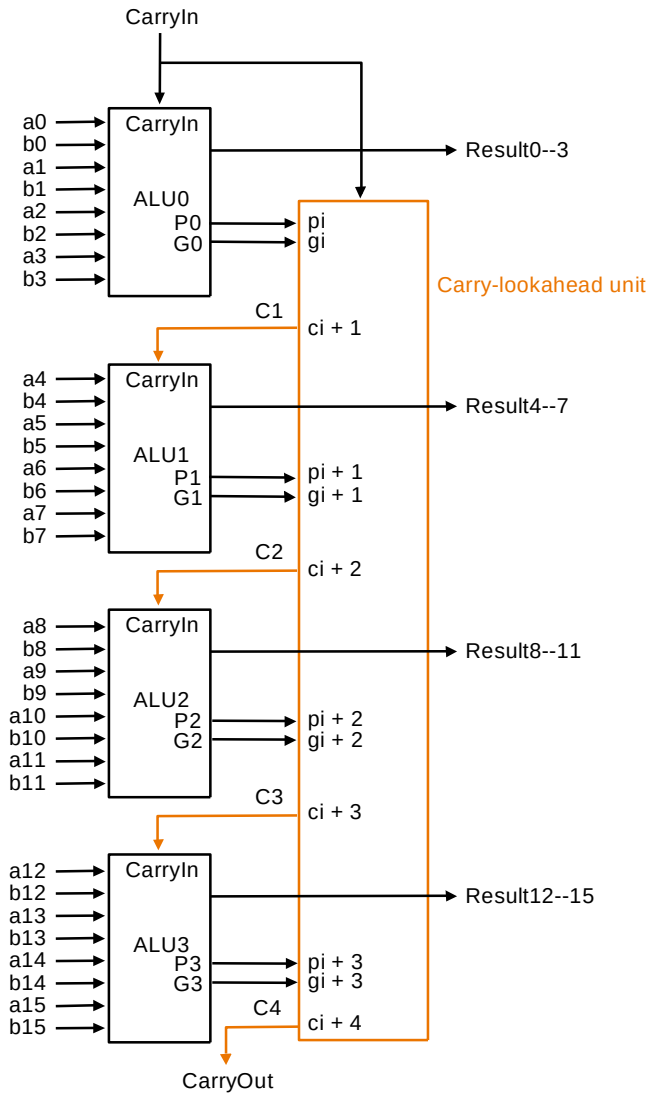
- **Therefore, for a 16-bit adder (4 groups):**

$$C_1 = G_0 + P_0 c_0$$
$$C_2 = G_1 + P_1 C_1 =$$
$$C_3 = G_2 + P_2 C_2 =$$
$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

the first CarryIn

See next slide!

16-bit Adder



Four 4-bit ALUs using carry lookahead to form a 16-bit adder.

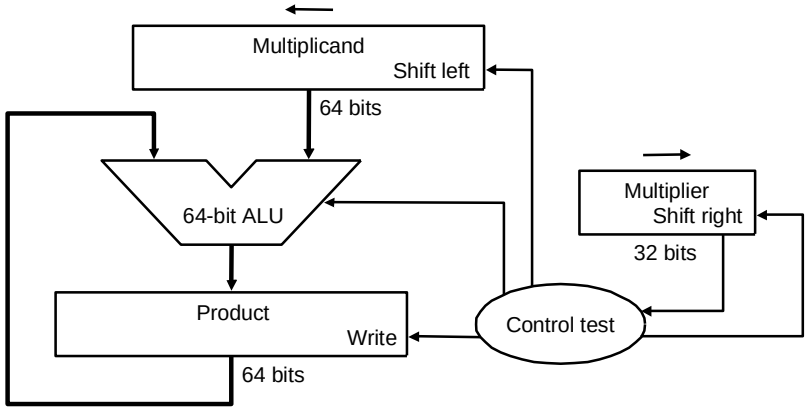
Integer Multiplication

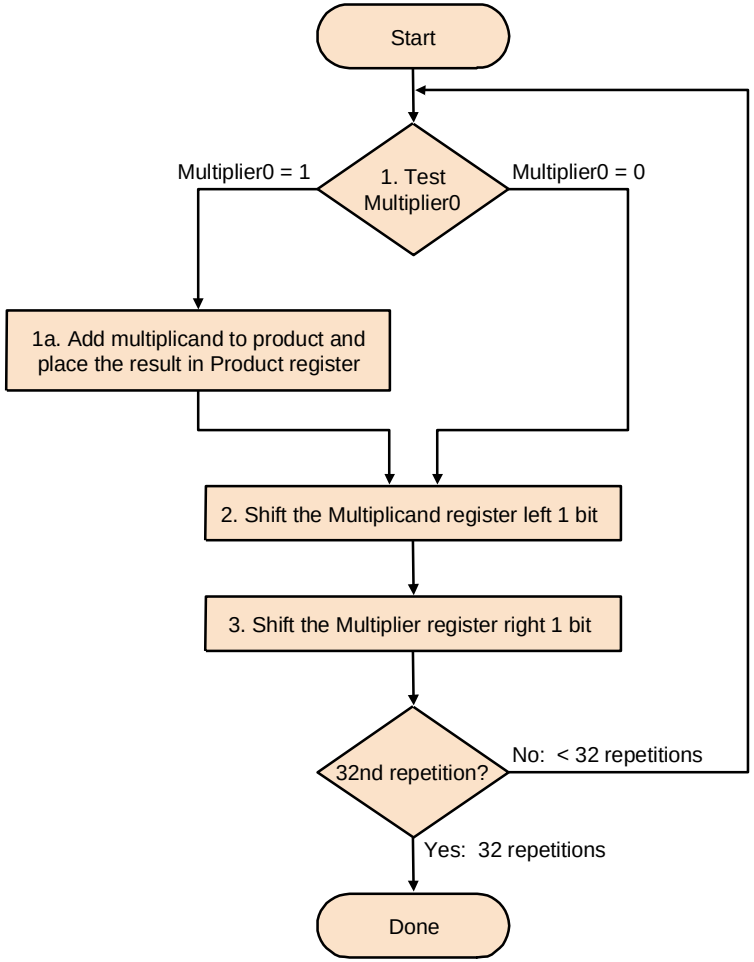
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on gradeschool algorithm

```
    1000      (multiplicand)
  x 1001      (multiplier)
   1000
  0000
 0000
1000
1001000       (product)
```

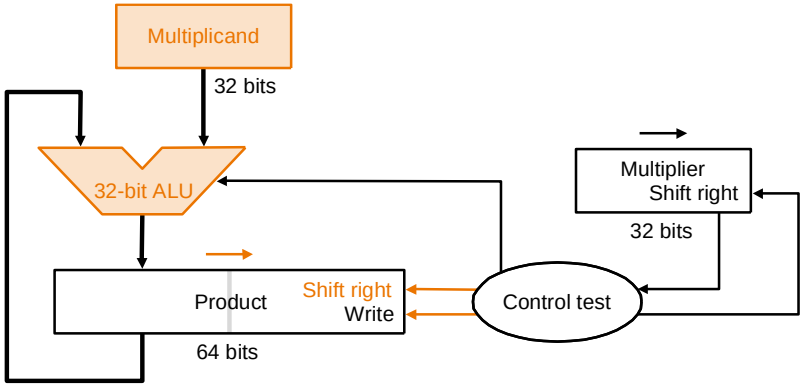
- Negative numbers: convert and multiply
- Issues :
  - Overflow can easily occur.
  - Need to deal with signed values.

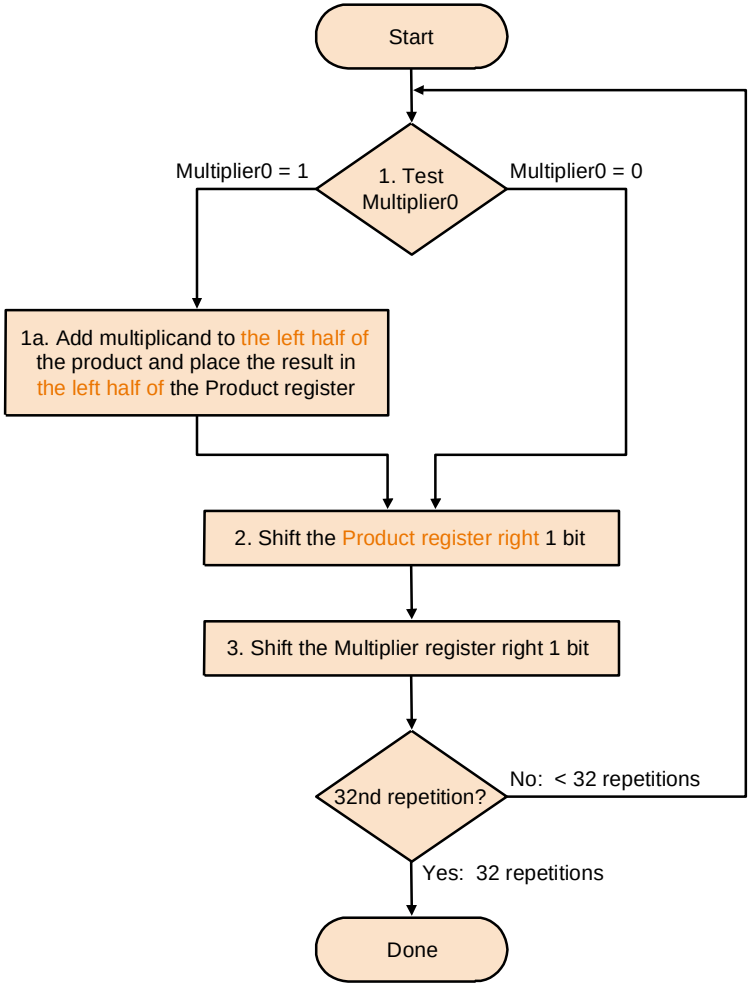
Multiplication Implementation: First Version



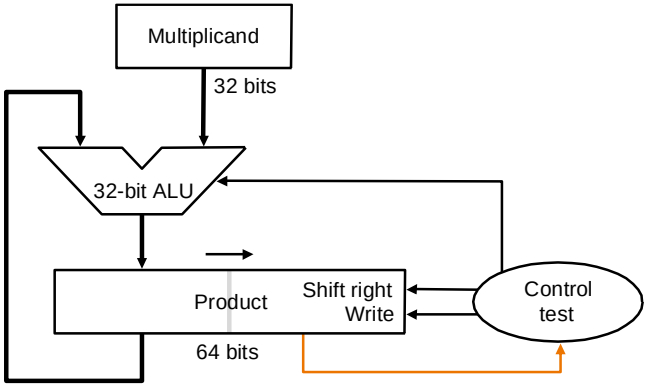


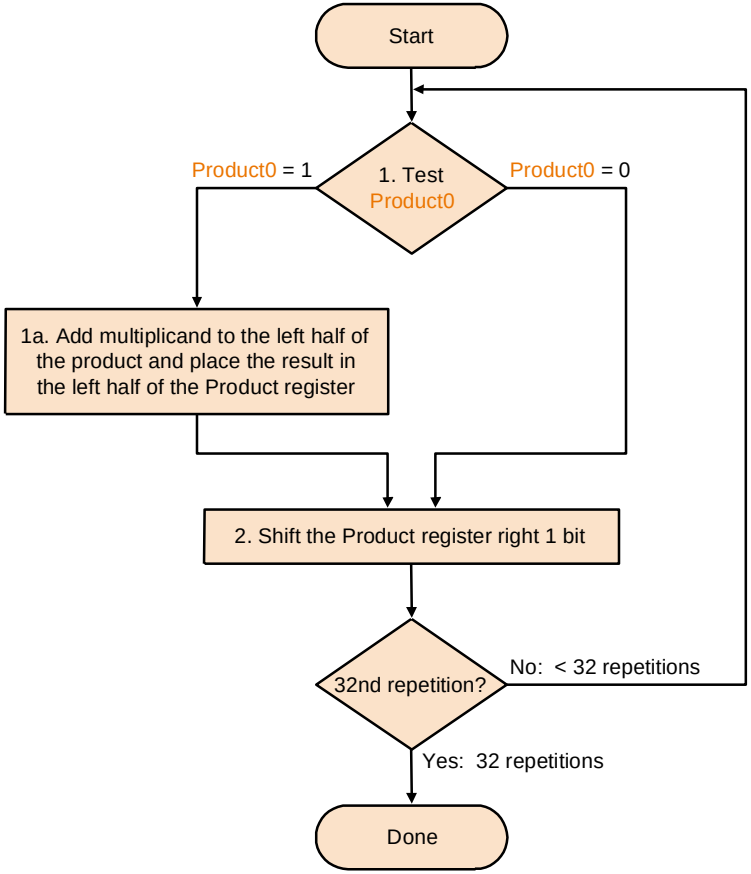
Second Version





Third Version (Final Version)





Multiply Example (Third version)

- **Multiply example using the third version of algorithm (0010X0011)**

Iteration	Step	Multiplicand	Product
0	Initial Values	0010	0000 0011
1	1a: 1 => Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 => Prod = Prod + Mcand	0010	0011 0001
	2: Shift right product	0010	0001 1000
3	1: 1 => no operation	0010	0001 1000
	2: Shift left Multiplicand	0010	0000 1100
4	1: 1 => no operation	0010	0000 1100
	2: Shift left Multiplicand	0010	0000 0110

Integer Division

- **Term:**

  - dividend : value to be divides

  - divisor : the number by which a dividend is divided

  - quotient : result

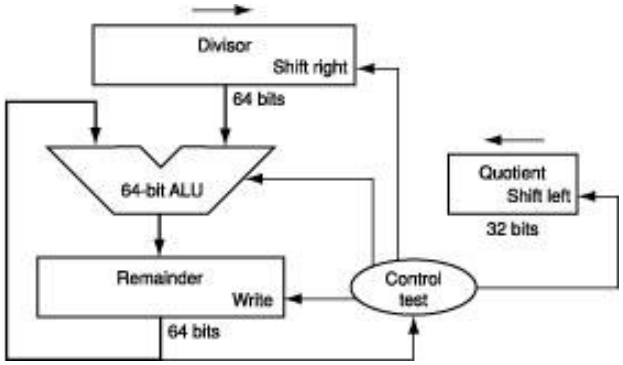
  - remainder : the final undivided part after division that is smaller than the divisor

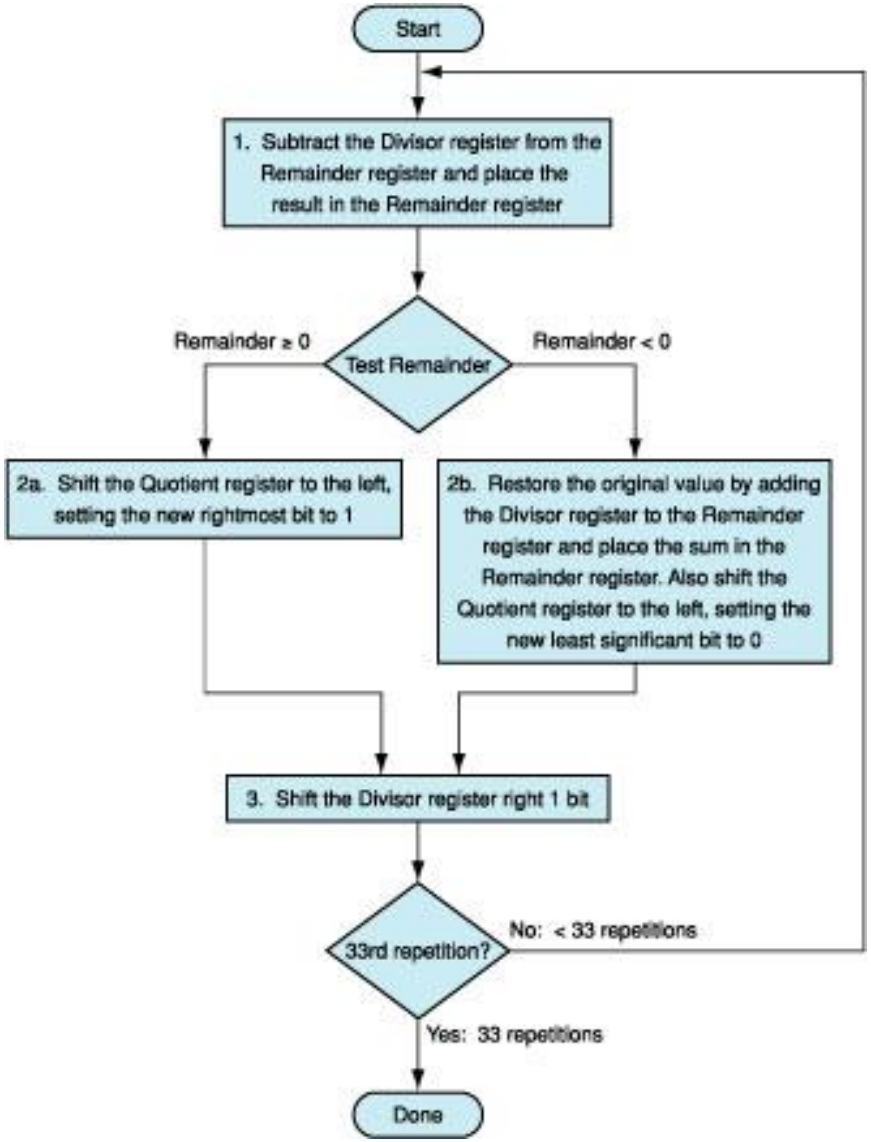
- **Issues :**

  - Divide by zero

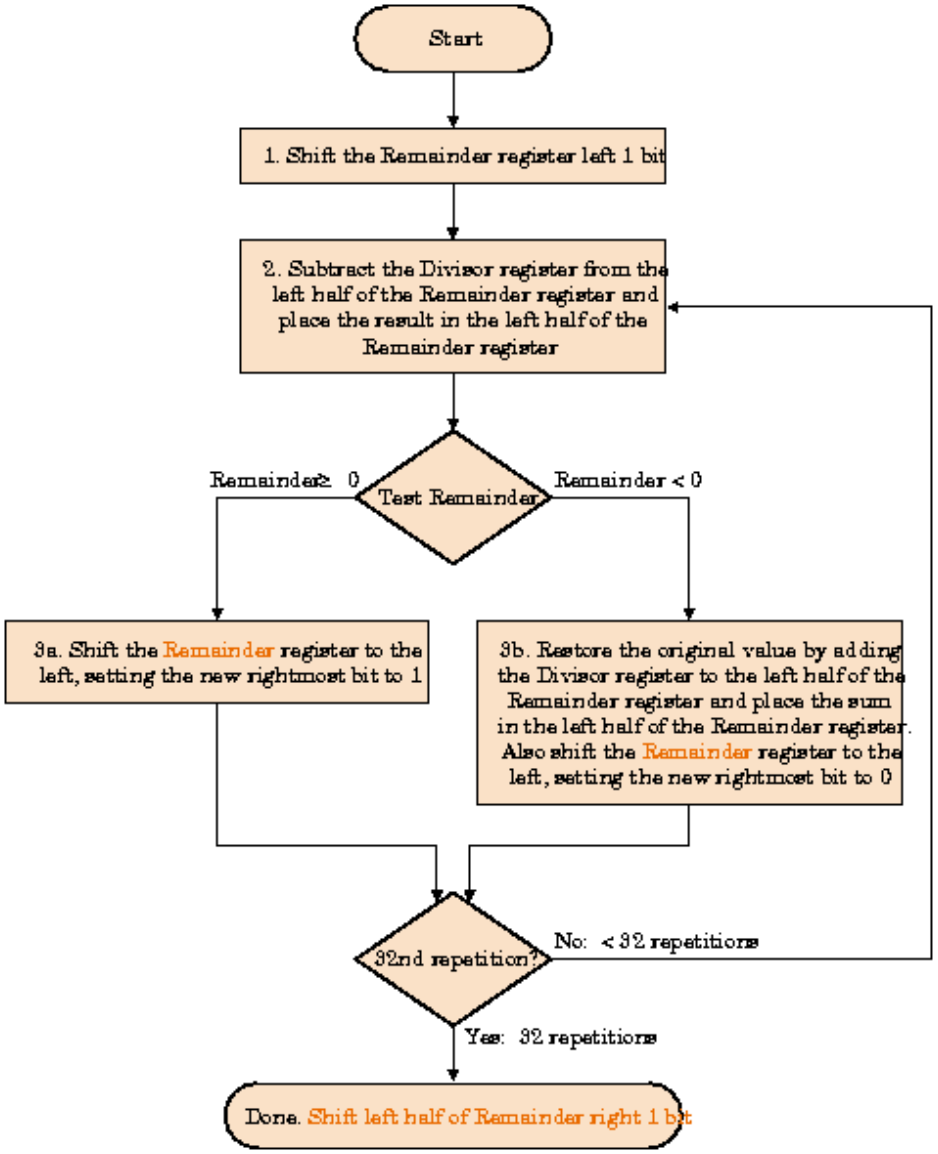
  - Need to deal with signed values

Division Algorithm (First Version)





Division Algorithm (Final Version)



Division Example (Final Version)

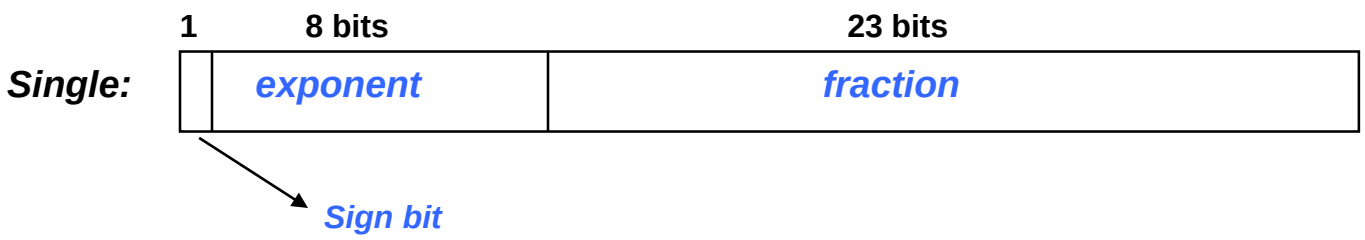
- **Division example using final algorithm.**

Iteration	Step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 1110
	3b: Rem < 0 => + Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem - Div	0010	1111 1100
	3b: Rem < 0 => + Div, sll R, R0 = 0	0010	0001 1000
3	2: Rem = Rem - Div	0010	0001 1000
	3a: Rem >= 0 => sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem - Div	0010	0001 0001
	3a: Rem >= 0 => sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

Typo: should be 1

Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., 0.000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, fraction: $(-1)^{sign} \times (1+fraction) \times 2^{exponent - bias}$
  - more bits for fraction gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit fraction
  - double precision: 11 bit exponent, 52 bit fraction



IEEE 754 floating-point standard

- **Leading "1" bit of fraction is implicit**

- **Exponent is "biased" to make sorting easier**
  - **all 0s is smallest exponent all 1s is largest**
  - **bias of 127 for single precision and 1023 for double precision**
  - **summary:** $(-1)^{sign} \times (1+fraction) \times 2^{exponent - bias}$

    127 (for 32-bit)

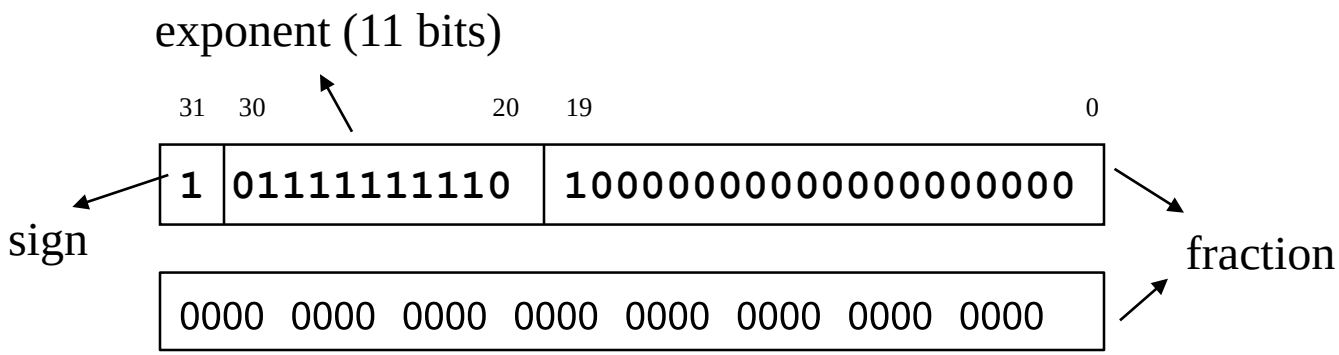
- **Example:**
  - **decimal: -.75 = -3/4 = -3/2^2 => binary: -11/ 2^2 = -.11 = -1.1 x 2^{-1}**
  - **sign=1**
    exponent = 126 = 01111110 (exp-127 = -1, therefore exp=126)
    fraction = 1
    so, IEEE single precision: `1 01111110 10000000000000000000000`

31	30 … 23	22 … 0
1	01111110	10000000000000000000000
sign	exponent (8 bits)	fraction

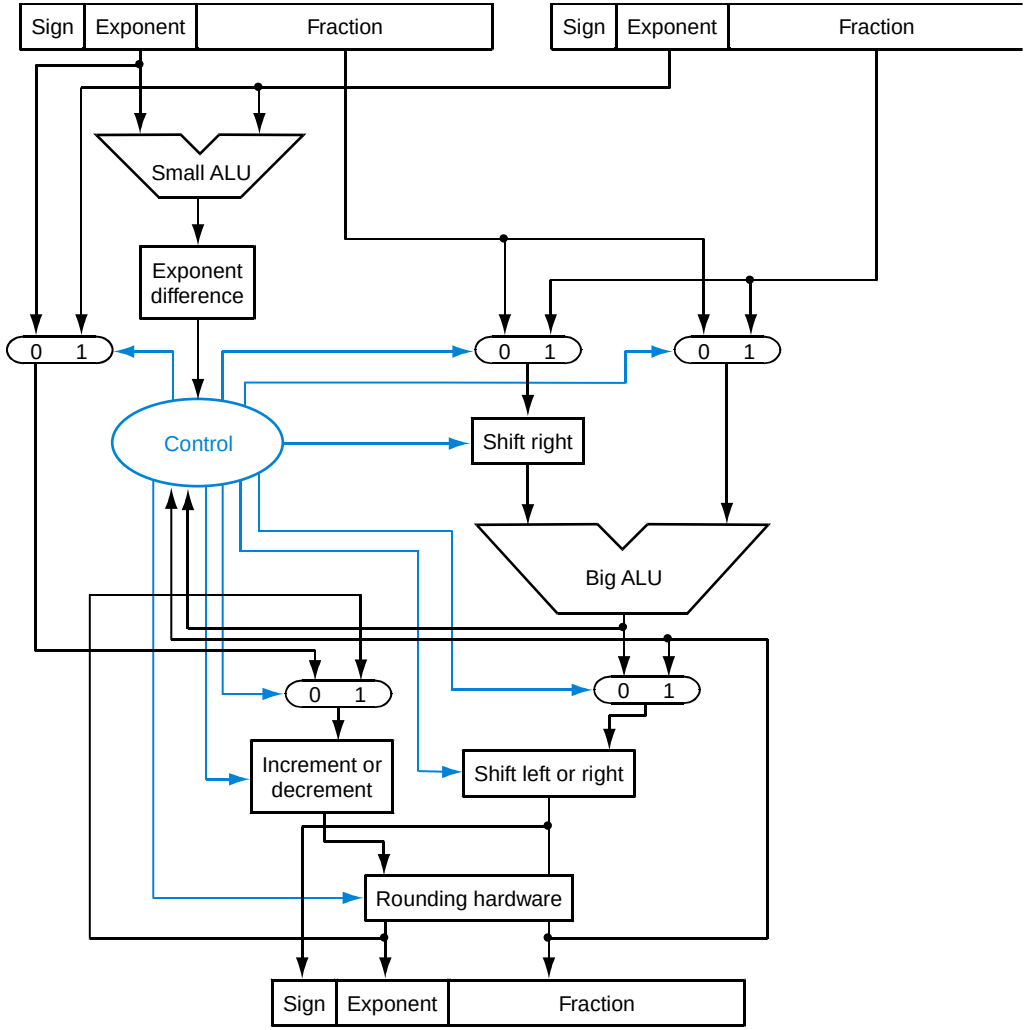
IEEE 754 floating-point standard

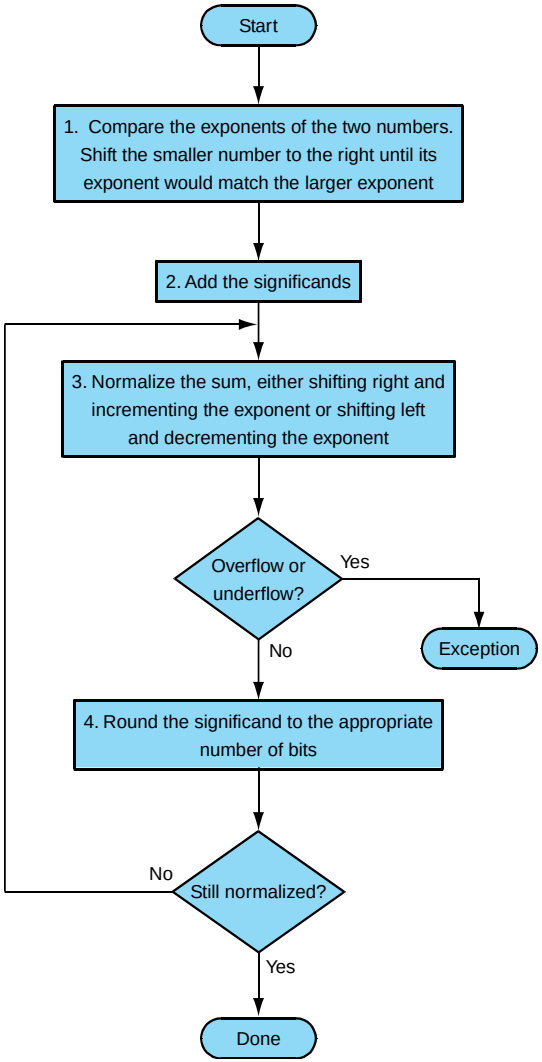
- **bias 1023 for double precision**
  - **summary:** $(-1)^{sign} \times (1+fraction) \times 2^{exponent - bias}$

    1023 (for 64-bit)
- **Example:**
  - **decimal: -.75 = -3/4 = -3/2^2 => binary: -11/ 2^2 = -.11 = -1.1 x 2^{-1}**
  - **sign=1**
    exponent = 1022 = 01111111110 (exp-1023 = -1, therefore exp=1022)
    fraction = 1
    so, IEEE double precision: `1 01111111110 10000000000000000000`
    `0 00000000000 00000000000000000000`



Floating point addition





Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Chapter 3 Summary

- **Computer arithmetic is constrained by limited precision**
- **Bit patterns have no inherent meaning but standards do exist**
  - **two's complement**
  - **IEEE 754 floating point**
- **Computer instructions determine "meaning" of the bit patterns**
- **Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).**

- **We are ready to move on (and implement the processor)**

Chapter 3

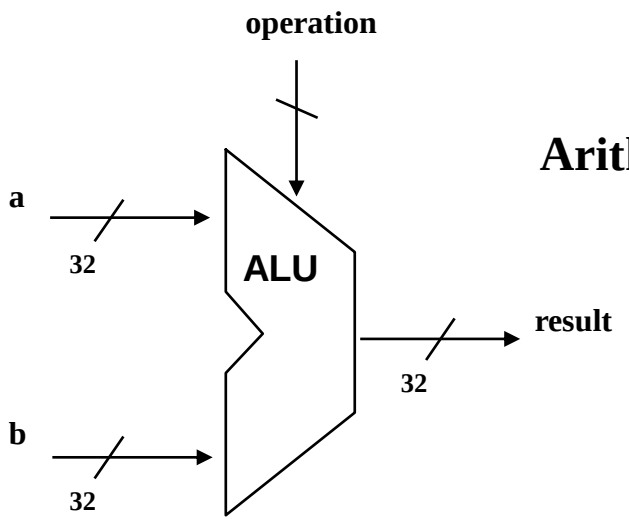
Arithmetic for Computers

Concepts Introduced in Chapter 3

- two's complement integer representation and operations on this representation
- IEEE Floating-Point Representation
- basic logic operations and hardware building blocks
- description of a simple 32-bit ALU
- floating-point representation
- multiplication and division

Arithmetic

- **Where we've been:**
  - **Abstractions:**
    Instruction Set Architecture
    Assembly Language and Machine Language
- **What's up ahead:**
  - **Implementing the Architecture**



Arithmetic Logic Unit

Binary Representations

- **32 bit *unsigned* numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
...
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Binary Representations - MIPS

- **Two's complement representation can represent both positive and negative values.**
- **32 bit *signed* numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — ***maxint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — ***minint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Two's Complement Operations

- **Negating a two's complement number:** ***<u>invert all bits and add 1</u>***
  - **remember: "negate" and "invert" are quite different!**

- **Example 1:** $3_{10} = 0000000000\ 0000000000\ 0000000000\ 11_2$

$$-3_{10} = 1111111111\ 1111111111\ 1111111111\ 00_2 + 1$$
$$= 1111111111\ 1111111111\ 1111111111\ 01_2$$

- **Example 2:** $-3_{10} = 1111111111\ 1111111111\ 1111111111\ 01_2$

$$3_{10} = 0000000000\ 0000000000\ 0000000000\ 10_2 + 1$$
$$= 0000000000\ 0000000000\ 0000000000\ 11_2$$

Examples: Addition with carries

- **1-bit addition with carry: Just like in grade school**

1	1	1	1	0	0	0	0	→ Carry in
1	1	0	0	1	1	0	0	→ A
+ 1	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1	+ 0	→ B
11	10	10	01	10	01	01	00	

carry sum

Example:

```
   01111000   (carries)
   00101101   (45)
+  00011110   (30)
   01001011   (75)
```

Examples: Addition & Subtraction

- **Addition:**

```
  00000101   (5)        00000101    (5)
+ 00000100   (4)      + 00001011   (11)
  00001001   (9)        00010000   (16)
```

- **Subtraction: Two's complement operations easy**
  - **subtraction using addition of negative numbers**

    Note: x-y = x+(-y)

2-5 = 2+(-5) = -3

```
  00000010   (2ten)
+ 11111011   (-5ten)
  11111101   (-3ten)
```

7-6 = 7+(-6) = 1

```
  00000111   (7ten)
+ 11111010   (-6ten)
1 00000001   (1ten)
```

ignore this bit

Overflow

- **Overflow: result too large for finite computer word**
  - **e.g., adding two n-bit numbers does not yield an n-bit number**
  - **e.g., adding two positive integers yield a negative integer**

```
  0111 (+7)
+ 1001 (-7)
-----------
1 0000  (0)
```

hardware overflow

```
 0111 (+7)
+0110 (+6)
----------
 1101 (-3)
```

sign overflow

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative (add)
  - or, adding two negatives gives a positive (add)
  - or, subtract a negative from a positive and get a negative (sub)
  - or, subtract a positive from a negative and get a positive (sub)

Overflow Conditions

	Operation	Operand A	Operand B	Overflow when result is
add	A+B	>=0	>=0	<0
add	A+B	<0	<0	>=0
sub	A-B	>=0	<0	<0
sub	A-B	<0	>=0	>=0

Overflow Conditions (in Truth Tables)

- **For A+B (add):**

A	B	Result (R)	Overflow?
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

A'B'R+ABR'

- **For A-B (sub):**

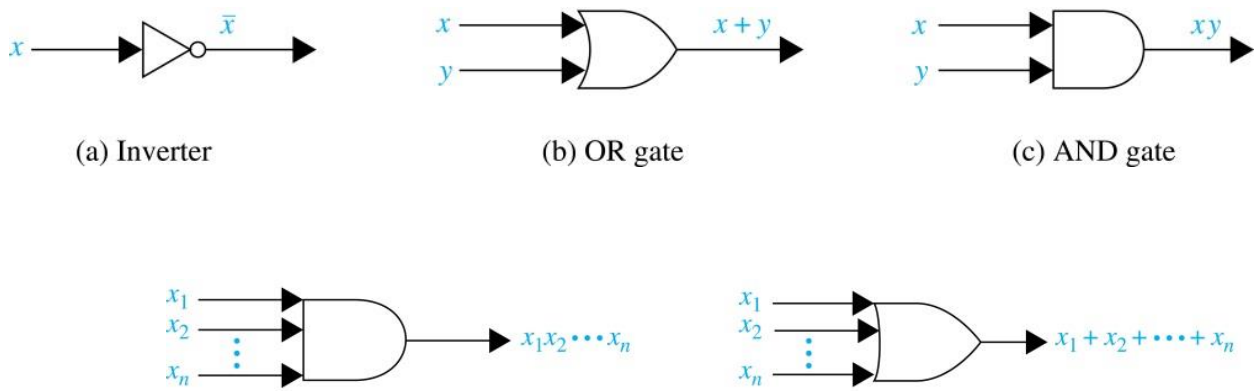
A	B	Result (R)	Overflow?
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

A'BR+AB'R'

Logic Gates

- **In digital circuits, we use**
  - **high voltage (2 volts - 5 volts) to represent 1 (true)**
  - **low voltage (0 volts - 0.8 volts) to represent 0 (false)**

- **By using high/low voltages, we are able to design digital gates:**



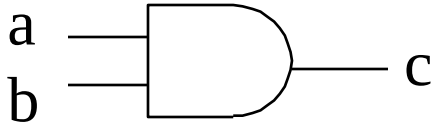
(a) Inverter (b) OR gate (c) AND gate

Boolean Algebra & Gates (See Appendix B)

- **And :**

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

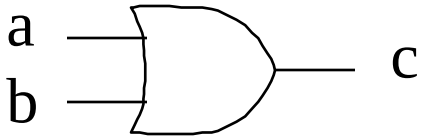
Notation: a•b (or just ab)



- **Or:**

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

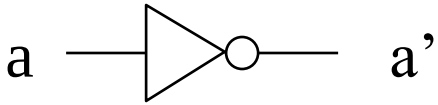
Notation: a+b



- **Inverter: (not)**

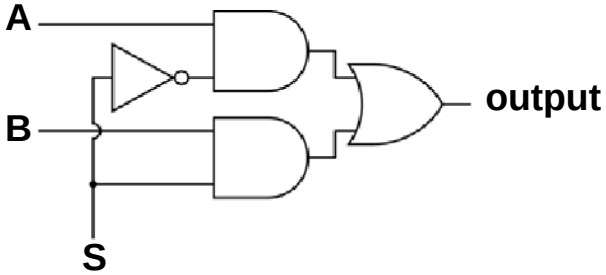
a	a'
0	1
1	0

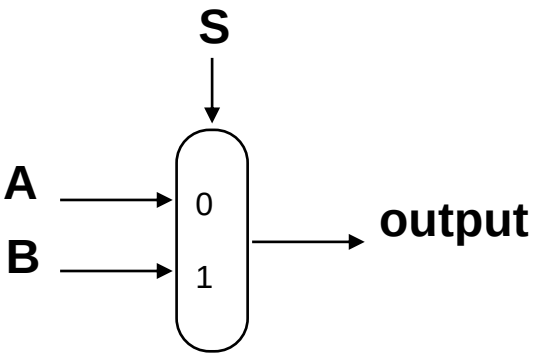
Notation: a' (or $\overline{a}$)



Review: The Multiplexer

- **Selects one of the inputs to be the output, based on a control input**



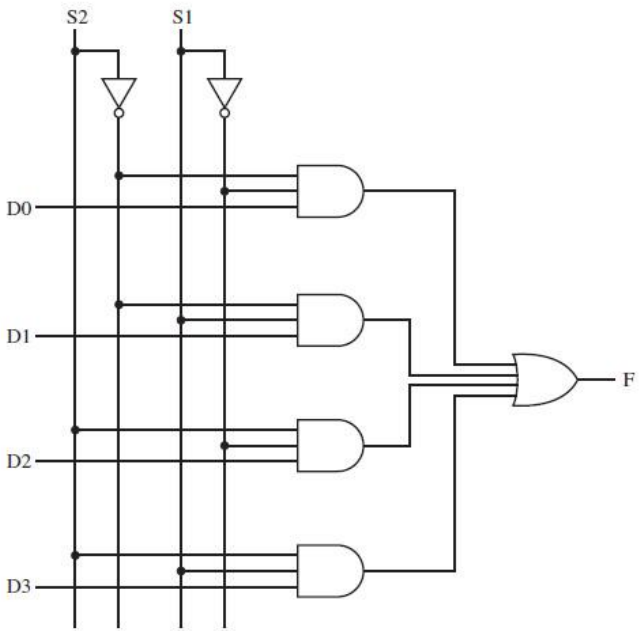


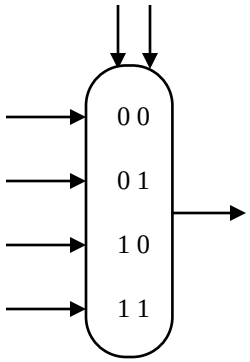
s	output
0	A
1	B

(We will build our ALU using a MUX)

Review: The Multiplexer

Multiplexer:
(4-input)

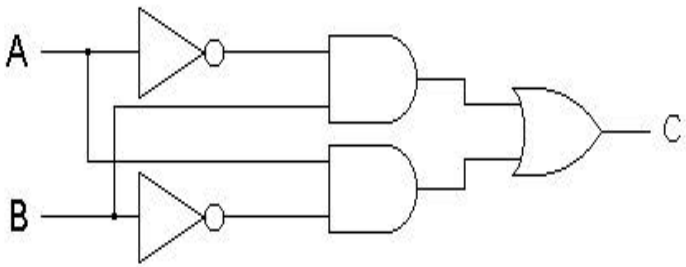


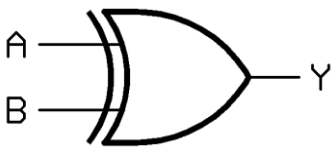


Input S2	Input S1	Output F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

Review: Boolean Algebra & Gates

XOR Gate:

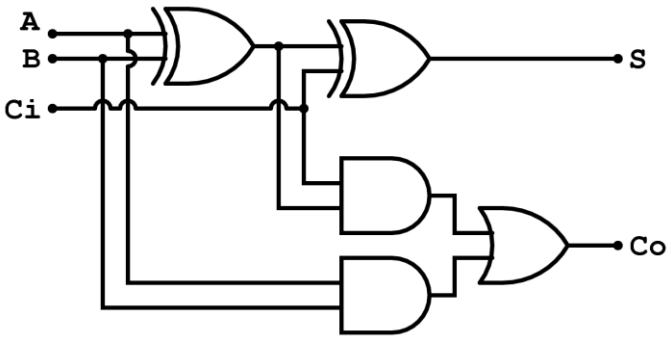


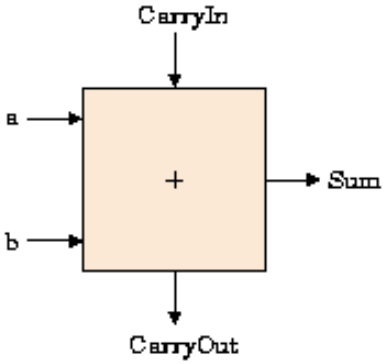


Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Review: Boolean Algebra & Gates

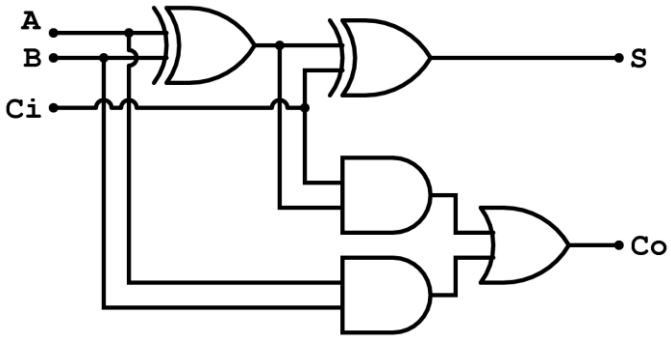
Adder:

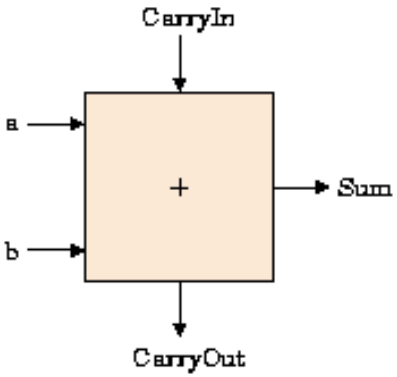




A	B	Ci (C-in)	Co (C-out)	S (Sum)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

A 1-bit adder



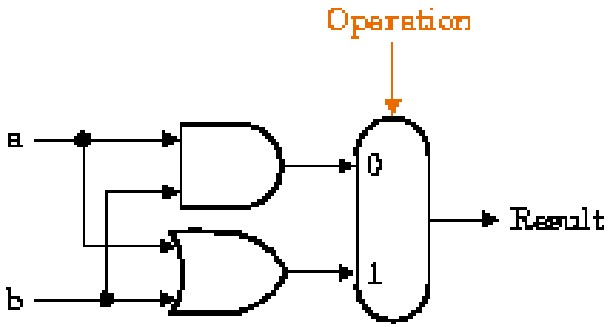


Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0+0+0=00_{two}$
0	0	1	0	1	$0+0+1=01_{two}$
0	1	0	0	1	$0+1+0=01_{two}$
0	1	1	1	0	$0+1+1=10_{two}$
1	0	0	0	1	$1+0+0=01_{two}$
1	0	1	1	0	$1+0+1=10_{two}$
1	1	0	1	0	$1+1+0=10_{two}$
1	1	1	1	1	$1+1+1=11_{two}$

$$c_{out} = a\ b + c_{in}\ (a \text{ xor } b)$$
$$= a\ b + a\ c_{in} + b\ c_{in}$$

$$\text{sum} = a \text{ xor } b \text{ xor } c_{in}$$

The 1-bit logical unit for AND and OR



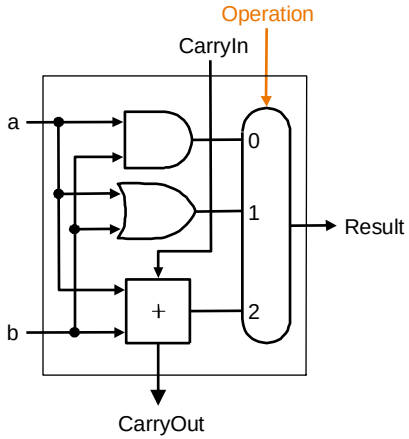
a	b	op	result
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1

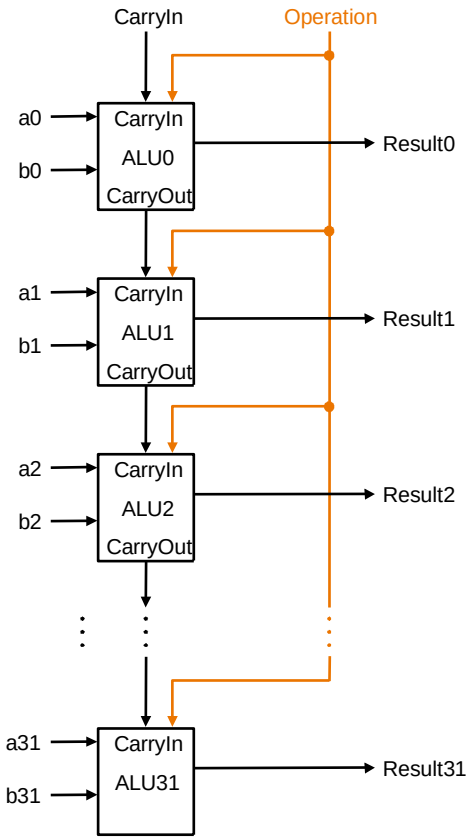
AND

a	b	op	result
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1

OR

Building a 32 bit ALU(Arithmetic Logic Unit)



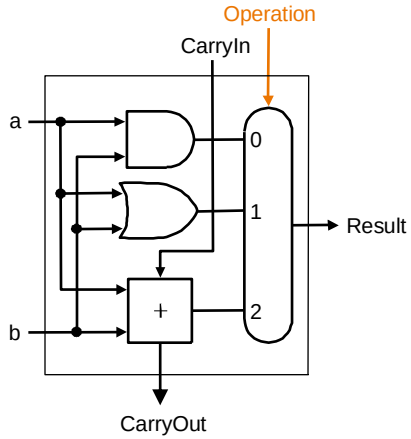


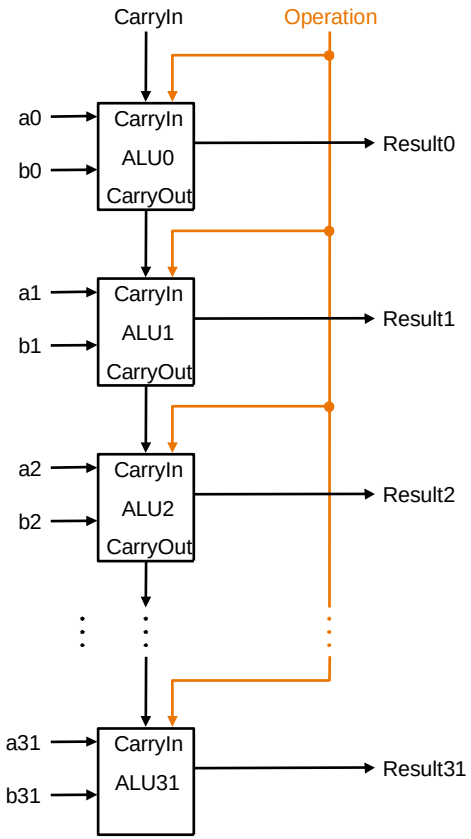
1-bit ALU **32-bit ALU**

Building a 32 bit ALU(and, or, add)

Operation = 00 (and)
01 (or)
10 (add)

CarryIn = 0





1-bit ALU **32-bit ALU**

What about subtraction (a – b) ?

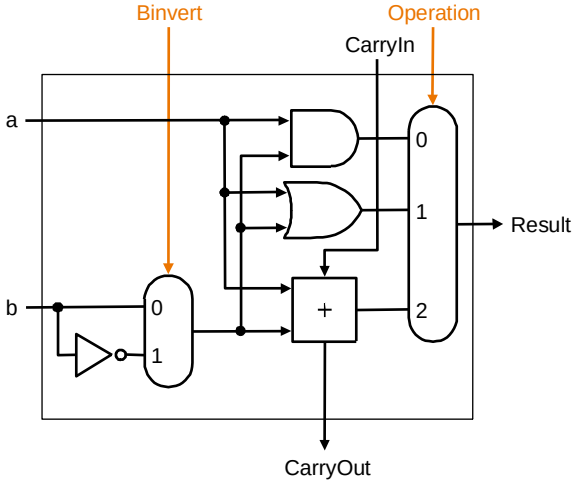
- **Two's complement approach: just negate b and add 1.**

a-b = a+(-b) = a+(2's complement of -b) = a+(invert(b)+1)

- **How do we negate?**

- **A very clever solution:**

Operation = 10
Binvert = 1
CarryIn = 1



Tailoring the ALU to the MIPS

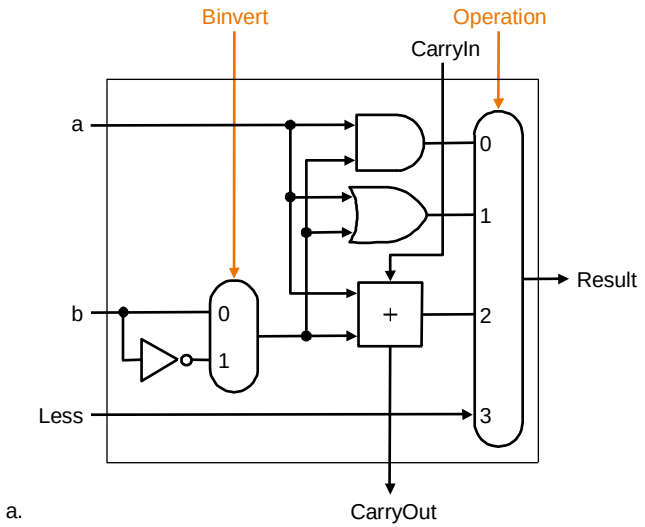
- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - slt $t0, $t1, $t2 => use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq/bne)
  - beq $t1, $t2, label => use subtraction: (a-b) = 0 implies a = b
  - bne $t1, $t2, label => use subtraction: (a-b) != 0 implies a != b

  (note: bne is the opposite of beq)

Supporting slt

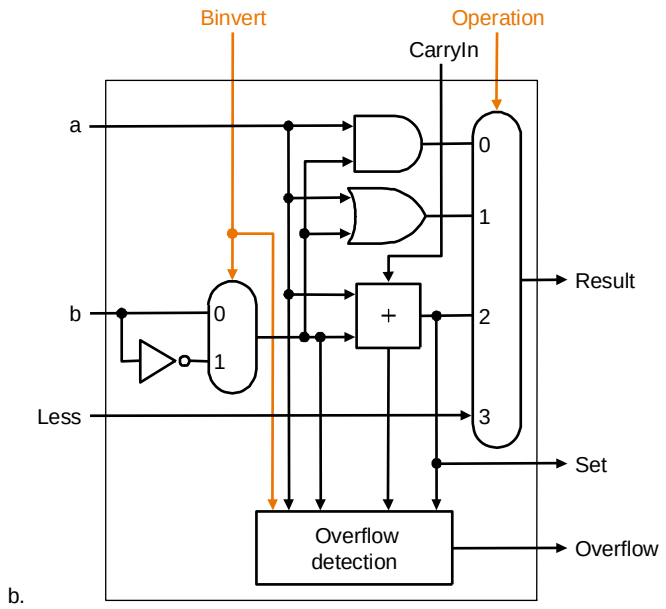
- **Can we figure out the idea?**

1-bit ALU for bit 0 to bit 30



a.

1-bit ALU for bit 31

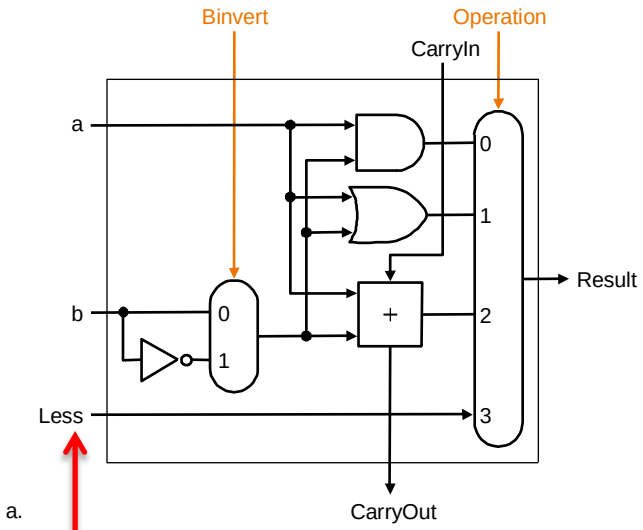


b.

Supporting slt

- Can we figure out the idea?

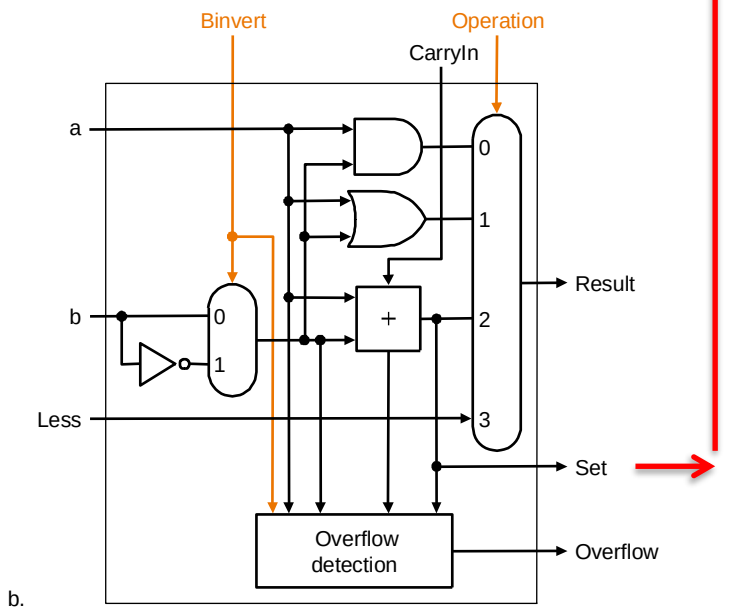
1-bit ALU for
bit 0



a.

Operation = 11
Binvert = 1
CarryIn (bit 0) = 1

1-bit ALU for
bit 31



b.

Overflow Detection

- For A+B (add): A'B'R+ABR' (see slide #12)
- For A-B (sub): A'BR+AB'R' (see slide #12)

- How do we know if it's "add" or "sub"?

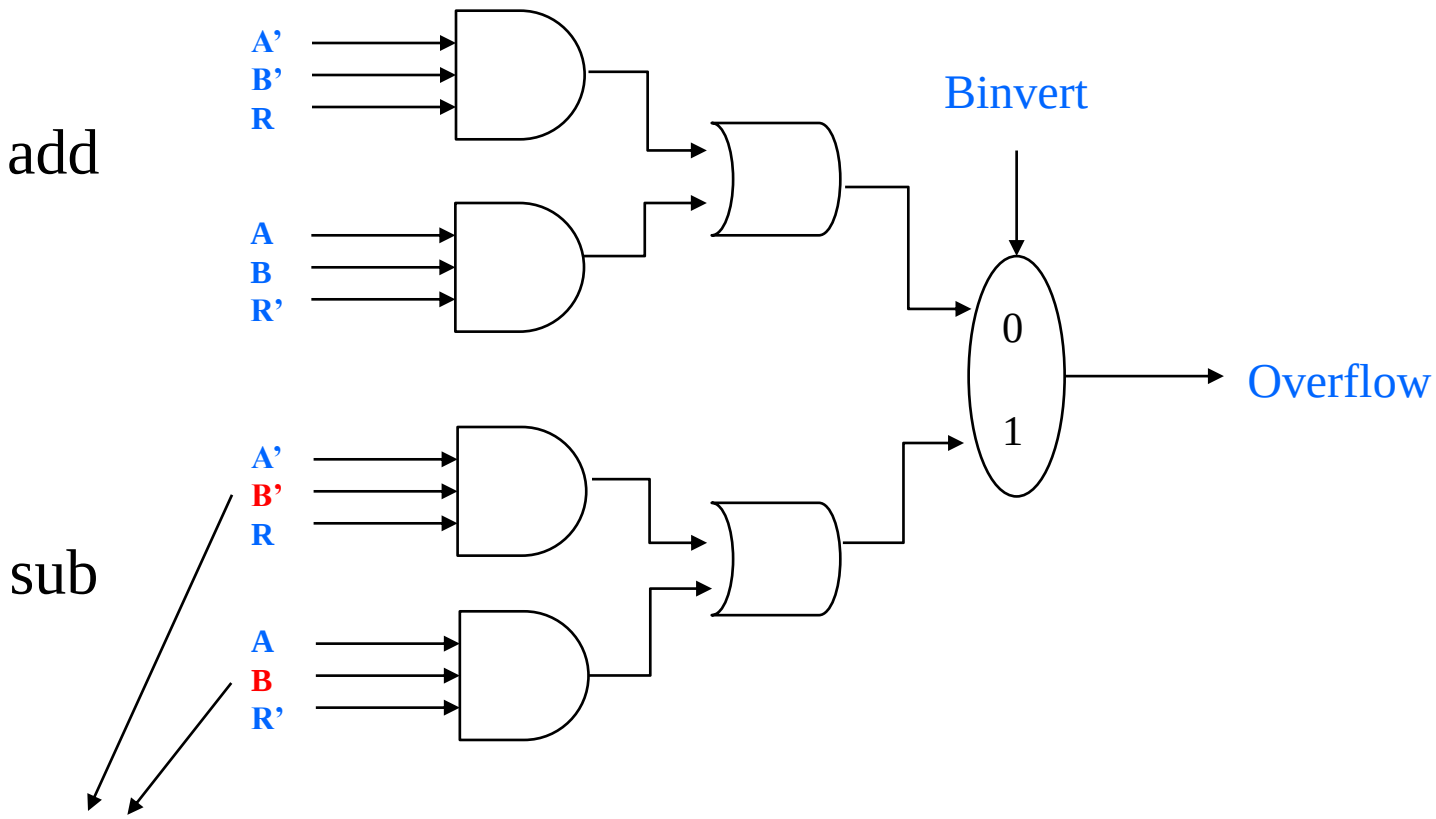
  Binvert = 0 (add)

  Binvert = 1 (sub)

- Can you draw this Overflow Detection logic circuit?

Overflow Detection

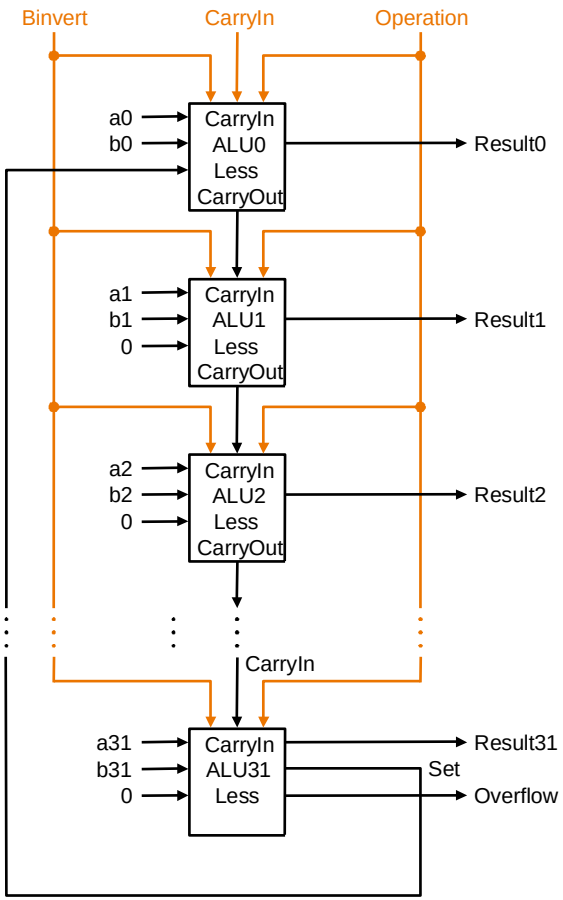
- **Overflow Detection logic circuit**



*For sub: Binvert = 1 means "not B" has already been set

32-bit MIPS ALU

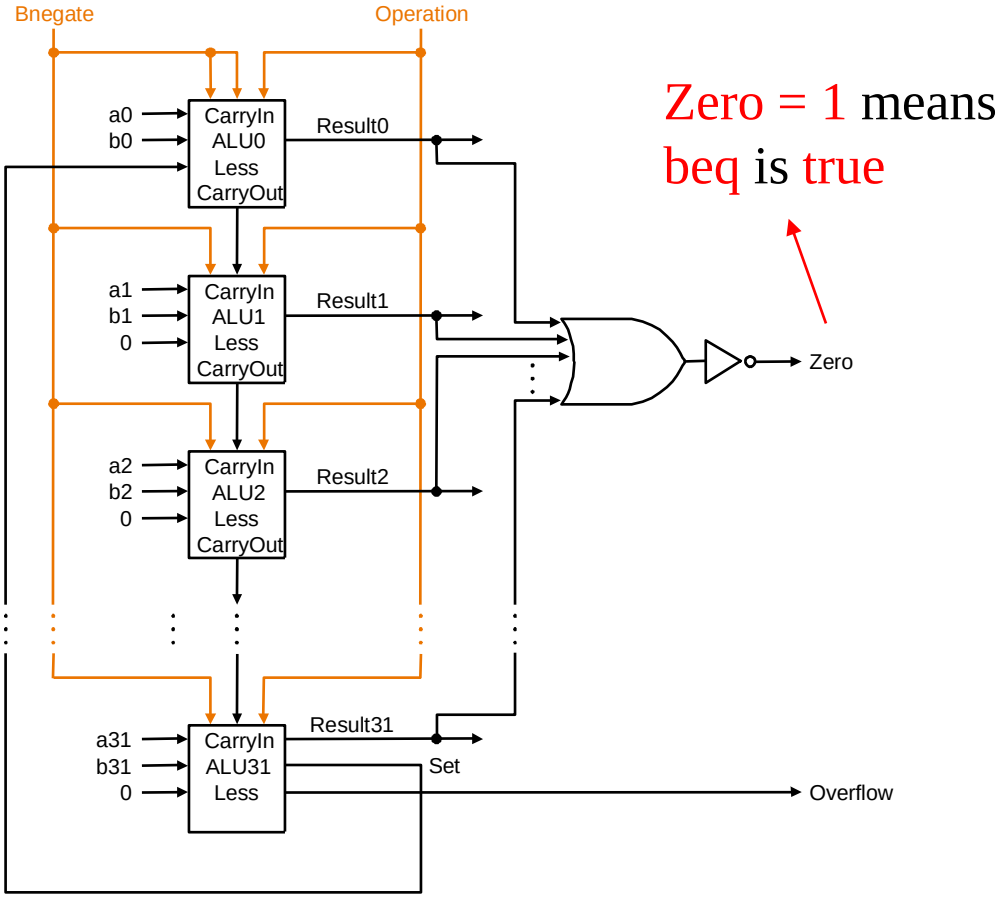
Since Binvert and CarryIn have the same signs, we can combine them together (Bnegate – see next slide).



Supporting beq/bne

***Just use sub:**

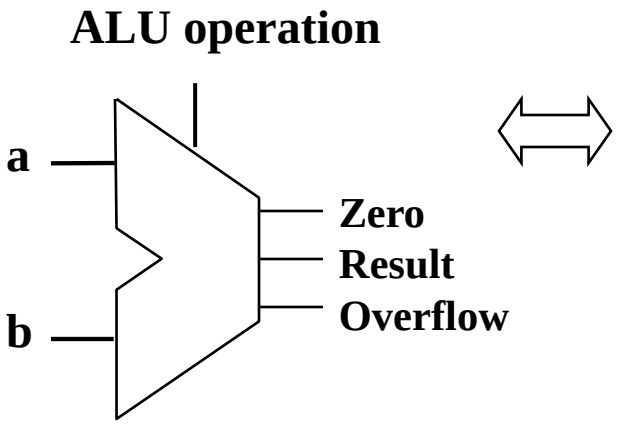
Operation = 10
Bnegate = 1

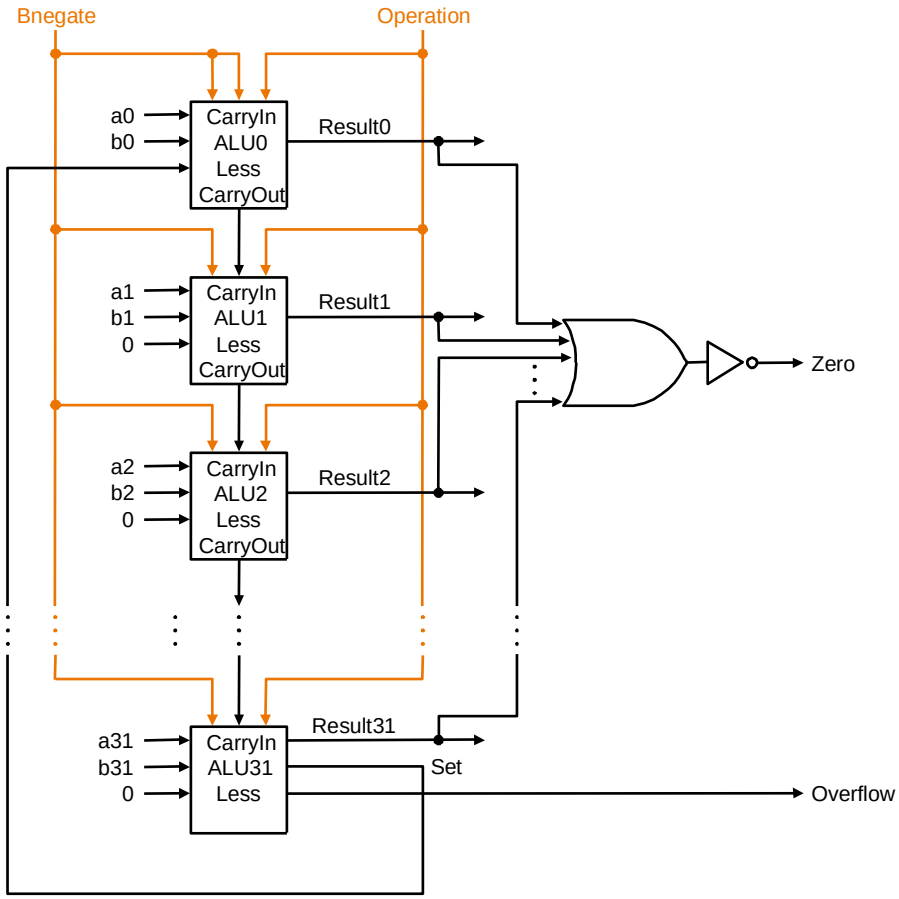


32-bit MIPS ALU

- **ALU control lines:**

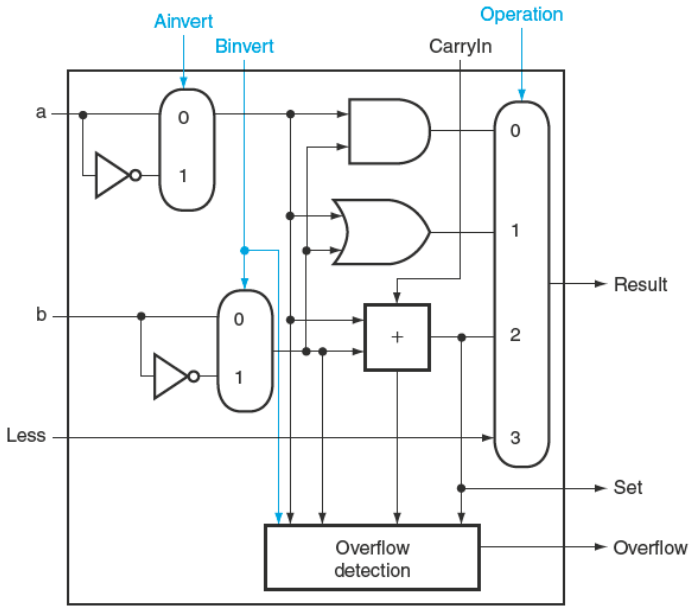
```
000 = and
001 = or
010 = add
110 = subtract(beq/bne)
111 = slt
```



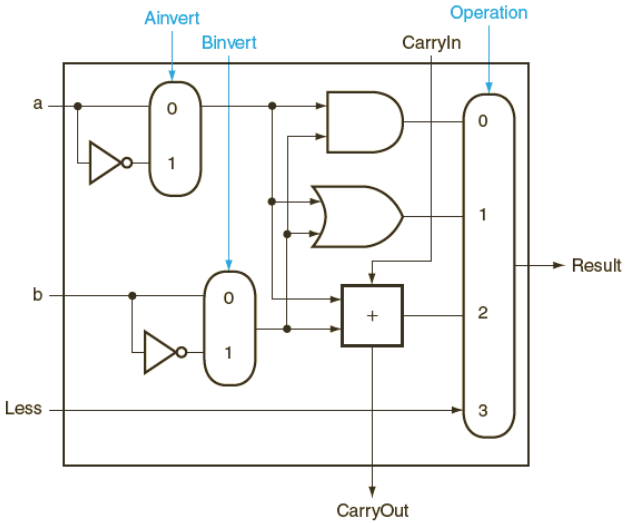


What about (a nor b) ?

- To do: NOT $t0 => use: nor $t0, $t0, $zero
- (a nor b) = not (a or b) = (not a) and (not b) ---> DeMorgan's law



bit 31



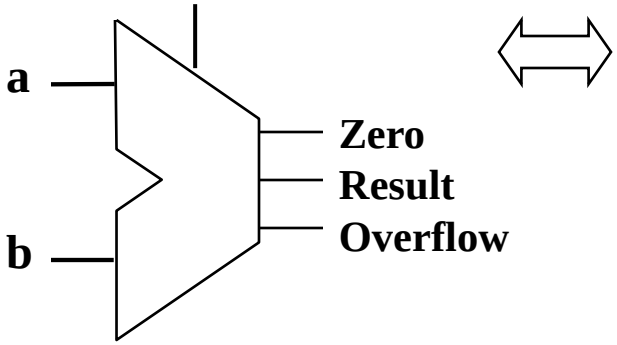
bit 0 to bit 30

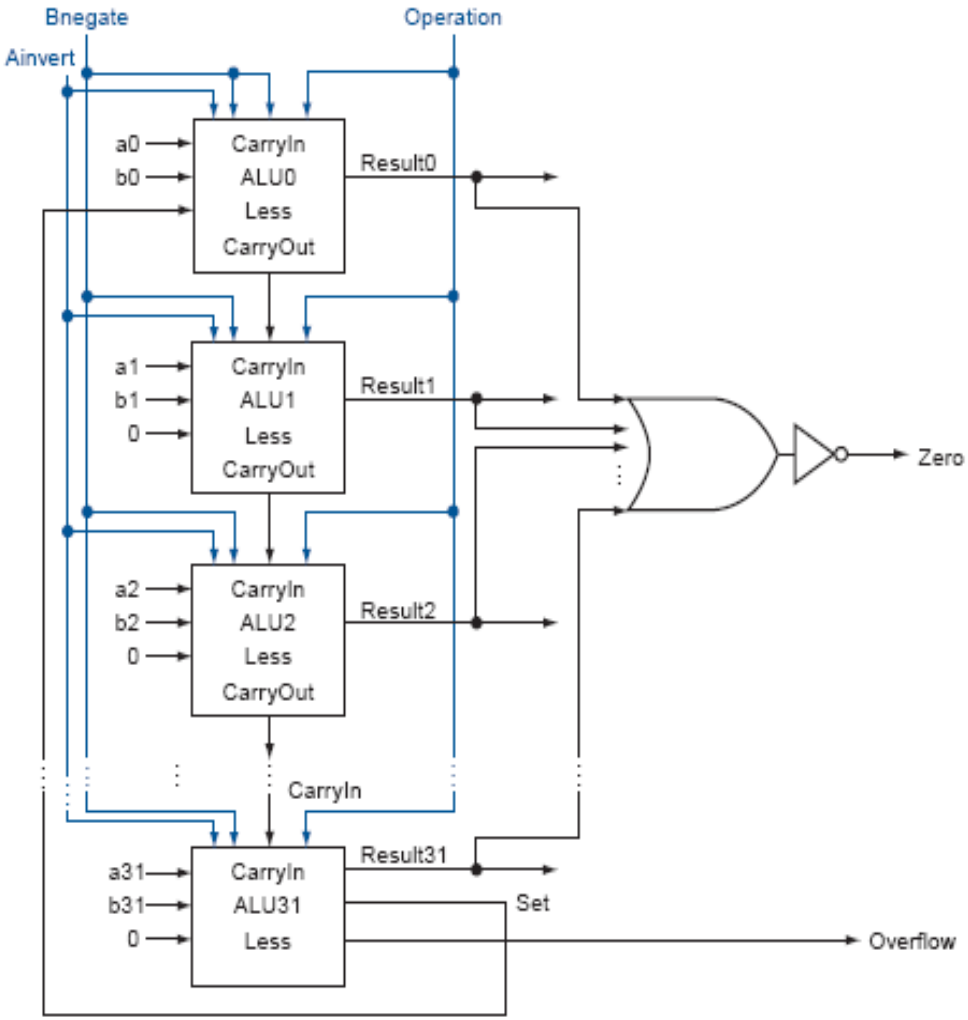
Final Version: 32-bit MIPS ALU

- **ALU control lines:**

```
0000 = and
0001 = or
0010 = add
0110 = subtract(beq/bne)
0111 = slt
1100 = nor
```

ALU operation





Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication

Register Number	Alternative Name	Description
0	zero	the value 0
1	$at	(**a**ssembler **t**emporary) reserved by the assembler
2-3	$v0 - $v1	(**v**alues) from expression evaluation and function results
4-7	$a0 - $a3	(**a**rguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(**s**aved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	**g**lobal **p**ointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	**s**tack **p**ointer Points to last location on the stack.
30	$s8/$fp	**s**aved value / **f**rame **p**ointer Preserved across procedure calls
31	$ra	**r**eturn **a**ddress

Service	Code in $v0	Arguments	Results
print_int	1	$a0 = integer to be printed	
print_float	2	$f12 = float to be printed	
print_double	3	$f12 = double to be printed	
print_string	4	$a0 = address of string in memory	
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	
sbrk	9	$a0 = amount	address in $v0
exit	10		
print char	11	$a0 = character to print	
read char	12		$v0 contains char read

Chapter 2

Instructions

Instruction Set:

- **Language of the Computer**
- **Different computers have different instruction sets**
  - **But with many aspects in common**
- **Early computers had very simple instruction sets**
  - **Simplified implementation**
- **Many modern computers also have simple instruction sets**

The MIPS Instruction Set:

- We'll be working with the MIPS instruction set architecture
- MIPS (Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture developed by MIPS Technologies (formerly MIPS Computer Systems, Inc.)
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo 64, Sony PlayStation, Cisco, Sun Micro Systems …
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Computer architecture courses in universities often study the MIPS architecture. (MIT, Stanford, Eastern, …)

MIPS arithmetic

- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first)

Example 1:

C/Java code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

MIPS arithmetic

- **Example 2:**

C/Java code:

```
A = B + C + D;
E = F - A;
```

MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

***Assume: registers $s1, $s2, $s3, $s4, and $s5 contain the <u>values</u> of B, C, D, E, and F, respectively**

MIPS arithmetic

- **Example 3:**

C/Java code:

```
f = (g + h) - (i + j);
```

MIPS code:

```
add $t0, $s1, $s2  # temp t0 = g + h
add $t1, $s3, $s4  # temp t1 = i + j
sub $s0, $t0, $t1  # f = t0 - t1
```

***Assume: registers $s0, $s1, $s2, $s3, and $s4 contain the <u>values</u> of f, g, h, i, and j, respectively**

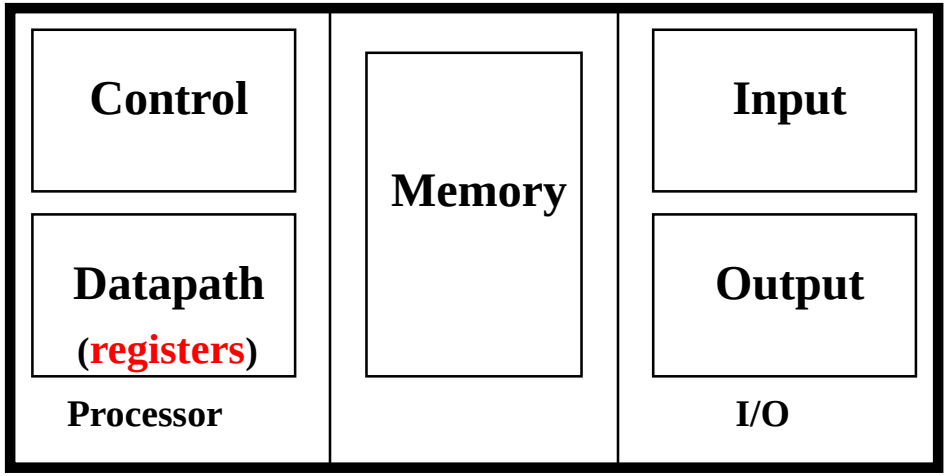
- **Operands must be registers, only 32 registers provided**
- ***Design Principle 2:* smaller is faster.**

Registers vs. Memory (1)

- **Arithmetic instructions operands must be registers,**
  - **Only 32 registers provided (a 32 × 32-bit register file)**
  - **Use for frequently accessed data**
  - **Numbered 0 to 31**
  - **32-bit data called a "word"**
- **Assembler names**
  - **$t0, $t1, …, $t9 for temporary values**
  - **$s0, $s1, …, $s7 for saved variables**
- **Compiler associates variables with registers**

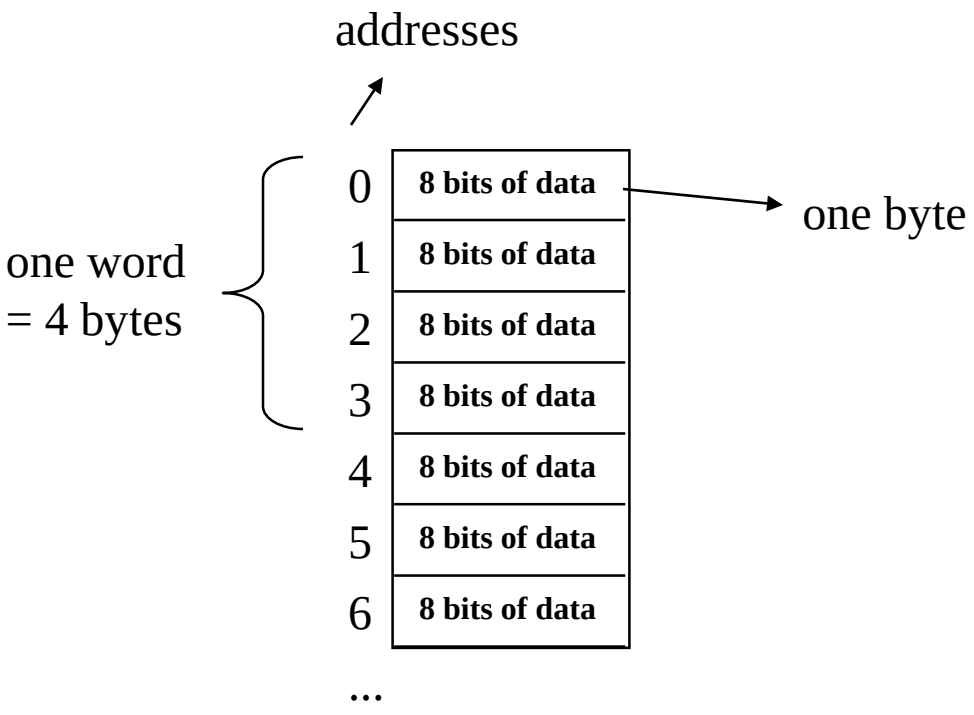
Registers vs. Memory (2)

- **Memory reference instructions:**
  - **lw (load word) : move data from memory to register**
  - **sw (store word) : move data from register to memory**



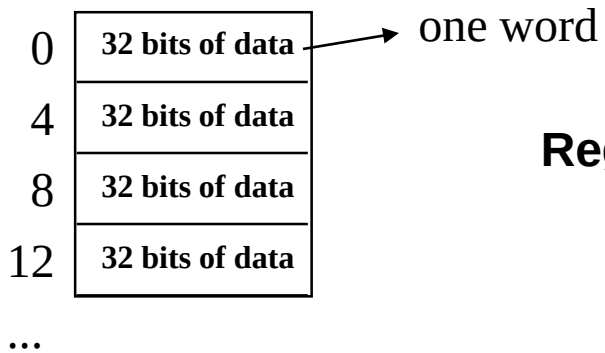
Memory Organization (1)

- **Viewed as a large, single-dimension array, with an address.**
- **A memory address is an index into the array**
- **"Byte addressing" means that the index points to a byte of memory.**



Memory Organization (2)

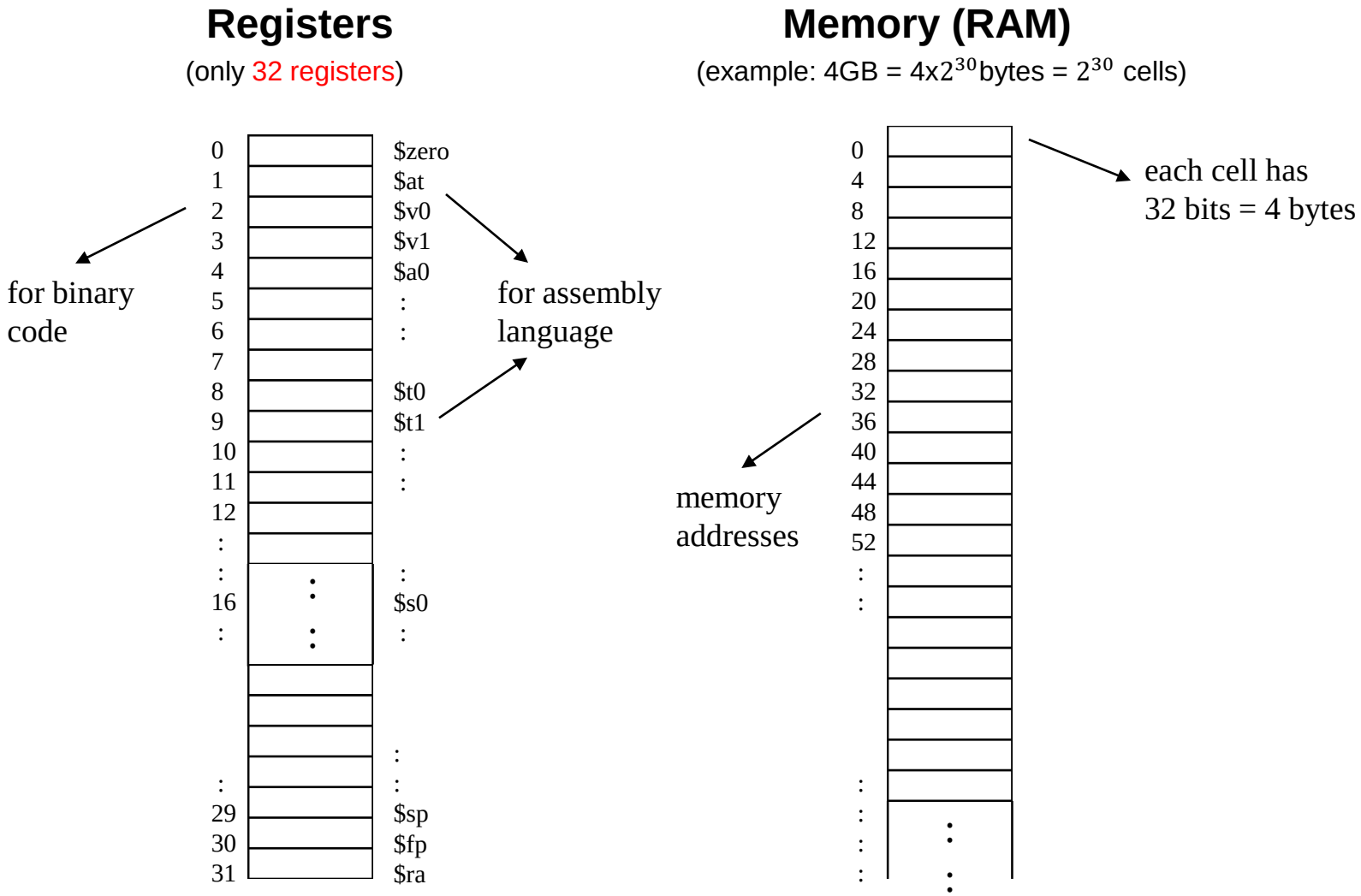
- **Bytes are nice, but most data items use larger "words"**
- **For MIPS, a word is 32 bits or 4 bytes.**



Registers hold 32 bits of data

- **4GB=4x2^{30} bytes=2^{32} bytes with byte addresses from 0,1,2, ...to 2^{32}-1**
- **4GB=2^{30} words with byte addresses 0, 4, 8, ... 2^{30}-1**
- **Main memory used for composite data (Arrays, structures …)**
- **To apply arithmetic operations**
  - **Load values from memory into registers (lw)**
  - **Store result from register to memory (sw)**

Registers vs. Memory (Summary)



Load and store Instructions

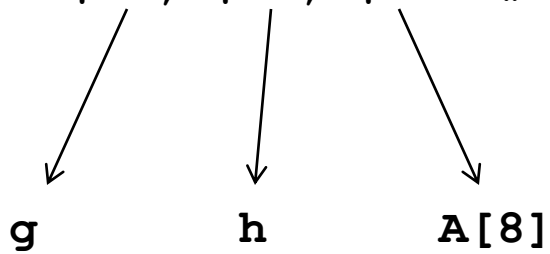
- **Example 1: (g in $s1, h in $s2, base address of A[ ] in $s3)**

C/Java code:

```
g = h + A[8];
```

Compiled MIPS code:

```
lw $t0, 32($s3)     # load word
add $s1, $s2, $t0   # add arithmetic
```



Load and store Instructions

- **Example 2: (h in $s2, base address of A[ ] in $s3)**

C/Java code:

```
A[12] = h + A[8];
```

Compiled MIPS code:

```
lw $t0, 32($s3)     # load word
add $t0, $s2, $t0   # add arithmetic
sw $t0, 48($s3)     # store word
```

- **Store word has destination last**
- **Remember arithmetic operands are registers, not memory!**

More Example

- **Can we figure out the Java/C++ code?**

  Assume: $s4 contains address of v[0] and $s3 contains value k

```
swap(int v[], int k);
{ int temp;
     temp = v[k]
     v[k] = v[k+1];
     v[k+1] = temp;
}
```

```
swap:
    add $s1, $s3, $s3
    add $s2, $s1, $s1
    add $s2, $s4, $s2
    lw $s5, 0($s2)
    lw $s6, 4($s2)
    sw $s6, 0($s2)
    sw $s5, 4($s2)
    jr $ra
```

address of v[0]

k

Summary: Registers vs. Memory

- **Registers are faster to access than memory**
- **Operating on memory data requires loads and stores**
  - **More instructions to be executed**
- **Compiler must use registers for variables as much as possible**
  - **Only spill to memory for less frequently used variables**
  - **Register optimization is important!**

Immediate Operands (constants)

- **Constant data specified in an instruction**

```
addi $s3, $s3, 4
```

- **No subtract immediate instruction (no `subi`)**
  - **Just use a negative constant**

```
addi $s2, $s1, -1
```

- **Load immediate**

```
li $s3, 4  (same as: addi $s3, $zero, 4)
```

- ***Design Principle 3:* Make the common case fast**
  - **Small constants are common**
  - **Immediate operand avoids a load instruction**

The Constant Zero

- **MIPS register 0 (`$zero`) is the constant `0`**
  - **Cannot be overwritten**

- **Useful for common operations**
  - **E.g., move between registers**

```
add $t2, $s1, $zero
addi $s3, $zero, 4
```

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**

- **<u>Instruction</u>** **<u>Meaning</u>**

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`
`addi $s1, $s2, 100`	`$s1 = $s2 + 100`
`addi $s1, $s2, -100`	`$s1 = $s2 – 100`
`add $s1, $s2, $zero`	`$s1 = $s2 + 0`
`li $s1, 4`	`$s1 = 4`

Numbers

- **Bits are just bits (no inherent meaning)**
  — conventions define relationship between bits and numbers
- **Binary numbers (base 2)**
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0...2^n-1$
- **How do we represent negative numbers?**
  i.e., which bit patterns will represent which numbers?

Bases that are a power of two

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Converting to base 10

- The value of a specific number in a specified base is: $\sum_{i=0}^{n-1} d_i * b^i$,where d is a digit and b is the base.

- An example in base 10:
  $425_{10} = 4\times10^2 + 2\times10^1 + 5\times10^0 = 400 + 20 + 5$
- An example in base 2:
  $1011_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 8 + 0 + 2 + 1 = 11_{10}$
- An example in base 8:
  $425_8 = 4\times8^2 + 2\times8^1 + 5\times8^0 = 256 + 16 + 5 = 277_{10}$
- An example in base 16:
  $17A_{16} = 1\times16^2 + 7\times16^1 + 10\times16^0 = 256 + 112 + 10 = 378_{10}$

Base Conversion Algorithm

To convert from base 10 to base *m*:

Keep dividing the base 10 number by *m* until the quotient is 0, and save all of the remainders

Example: 25 (base 10) = 11001 (base 2)

25 div 2 = 12 … 1
12 div 2 = 6 … 0
6 div 2 = 3 … 0
3 div 2 = 1 … 1
1 div 2 = 0 … 1

***Read these numbers from bottom to top**

quotient remainder

Examples of Converting between Bases

- <u>Binary to Hexadecimal:</u>

  0100 1100 0011 1101

  => 4 C 3 D

  => 4C3D (base 16)

- <u>Binary to Octal:</u>

  011 110 001

  => 3 6 1

  => 361 (base 8)

- <u>Hexadecimal to Octal:</u>

  C3F0

  => 1100 0011 1111 0000

  => 00 1100 0011 1111 0000 (note: add 2 0s to the left)

  => 001 100 001 111 110 000

  => 1 4 1 7 6 0 (base 8)

Binary Representations

- **32 bit *unsigned* numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
...
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Binary Representations - MIPS

- **Two's complement representation can represent both positive and negative values.**
- **32 bit *signed* numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — ***maxint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — ***minint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Two's Complement Operations

- **Negating a two's complement number:** ***<u>invert all bits and add 1</u>***
  - **remember: "negate" and "invert" are quite different!**

- **Example 1:**

$$3_{10} = 00000000000000000000000000000011_2$$

$$-3_{10} = 11111111111111111111111111111100_2 + 1$$

$$= 11111111111111111111111111111101_2$$

- **Example 2:**

$$-3_{10} = 11111111111111111111111111111101_2$$

$$3_{10} = 00000000000000000000000000000100_2 + 1$$

$$= 00000000000000000000000000000011_2$$

Two's Complement Operations

What are the decimal values of following 32-bit two's complement numbers?

- **Example 3:**

$$
\begin{aligned}
&11111111111111111111111111111100_2 \\
&= (1\times -2^{31}) + (1\times 2^{30}) + (1\times 2^{29}) + ... + (1\times 2^{2}) + (0\times 2^{1}) + (0\times 2^{0}) \\
&= (-2^{31} + 2^{30} + 2^{29} + ... + 2^{2} + 0 + 0)_{10} \\
&= -4_{10}
\end{aligned}
$$

- **Example 4:**

$$
\begin{aligned}
&11111111111111111111111111111101_2 \\
&= (1\times -2^{31}) + (1\times 2^{30}) + (1\times 2^{29}) + ... + (1\times 2^{2}) + (0\times 2^{1}) + (1\times 2^{0}) \\
&= (-2^{31} + 2^{30} + 2^{29} + ... + 2^{2} + 0 + 1)_{10} \\
&= -3_{10}
\end{aligned}
$$

Signed Extension Shortcut

- An integer register on the MIPS is 32 bits. When a value is loaded from memory with fewer than 32 bits, the remaining bits must be assigned.
- Example: convert 16-bit binary number of 2 into 32-bit binary numbers 2 and -2:

$$0000000000000010_2 = 2_{10}$$

$$+2: 00000000000000000000000000000010_2$$

$$-2: 11111111111111111111111111111110_2$$

$$Note:$$

$$1111111111111101_2 + 1_2 = 1111111111111110_2$$

$$then\ copying\ the\ sign\ bit\ 16\ times\ and\ placing\ it\ on\ the\ left$$

$$11111111111111111111111111111110_2 = -2_{10}$$

Binary addition

		(0)	(0)	(1)	(1)	(0)	(0)	→ (*carries*)
	….	0	0	0	1	1	1	
+	….	0	0	0	1	1	0	
	….	0	0	1	1	0	1	

		(1)	(1)	(1)	(1)	(0)	(0)	→ (*carries*)
	….	0	0	1	0	1	0	
+	….	0	1	1	1	1	0	
	….	1	0	1	0	0	0	

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

Machine Language (R-format: for register)

- **Instructions, like registers and words of data, are also 32 bits long**
  - **Example:** `add $t0, $s1, $s2`
  - **registers have numbers,** `$t0=8, $s1=17, $s2=18`

- **Instruction Format:**

000000	10001	10010	01000	00000	100000

op	rs	rt	rd	shamt	funct

- ***op(opcode): basic operation of the instruction***
- ***rs: the first register***
- ***rt: the second register***
- ***rd: the destination register***
- ***shamt: sfift amount***
- ***funct: function code***

Machine Language (R-format)

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2     # $t0=8, $s1=17, $s2=18
```

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$00000100011001001000000001000000_2 = 02324020_{16}$

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding -- allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Machine Language (I-format: data transfer)

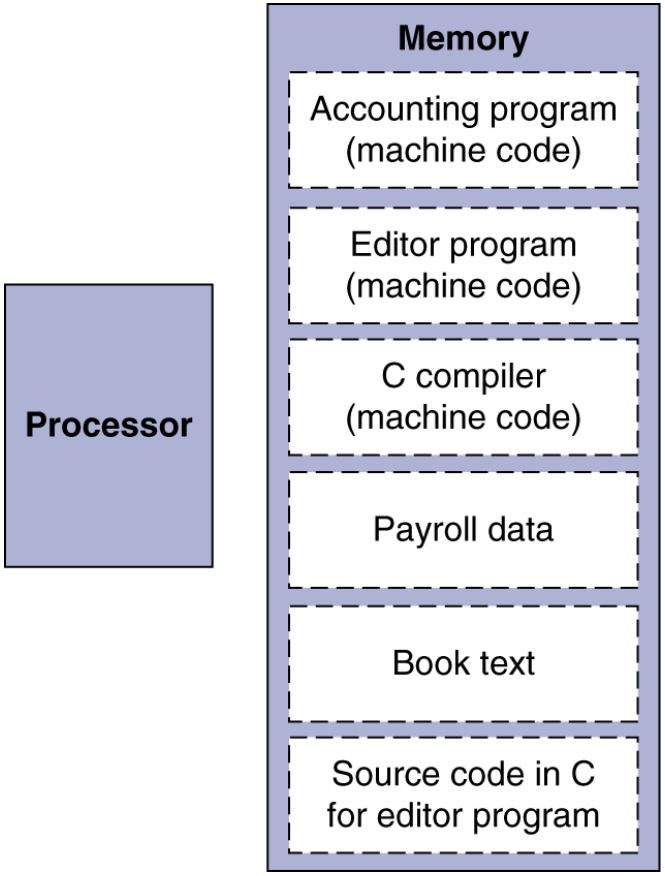
- **Introduce a new type of instruction format**
  - **I-type for data transfer instructions**
  - **other format was R-type for register**
- **Example:** `lw $t0, 32($s2)`

35	**18**	**8**	**32**
op	**rs**	**rt**	**16 bit number (address)**

Stored Program Concept

- Instructions (programs) represented in binary, just like data
- Instructions are stored in memory -- to be read or written just like data
- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

The BIG Picture



Logical Operators

	C/C++	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>	`srl`
Logical and	&	&	`and, andi`
Logical or	|	|	`or, ori`
Logical Not	~	~	`nor`

<u>Example:</u>

```
sll $t2, $s0, 8
```

// left shift $s0 8 bits and store the result to $t2
// Note: the content of $s0 is unchanged

Shift Operations

- **Use R-format**

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **shamt: how many positions to shift**
- **Shift left logical**
  - **Shift left and fill with 0 bits**
  - **Note: sll by *i* bits = multiplies by 2^i**
- **Shift right logical**
  - **Shift right and fill with 0 bits**
  - **Note: srl by *i* bits = divides by 2^i (unsigned only)**

Logical Operations - sll

- **Shift left logical (sll):**

Instruction	**Meaning**
`sll $t2, $s0, 8`	`$t2 ← $s0 << 8 bits`

R-Format

op	rs	rt	rd	shamt	funct	
0	0	16	10	8	0	(decimal)
000000	00000	10000	01010	01000	000000	(binary)

unused

always 0

Example: **$s0 contains: 0000 0000 0000 0000 0000 0000 0000 1101**

after "`sll $t2, $s0, 8`"

$t2 contains: 0000 0000 0000 0000 0000 1101 0000 0000

Fill empty bits with 0s

Logical Operations - srl

- **Right left logical (srl):**

Instruction	**Meaning**
`srl $t2, $s0, 8`	`$t2 ← $s0 >> 8 bits`

Example:

$s0 contains: 0000 0000 0000 0000 0000 0000 0000 1101

after "`srl $t2, $s0, 8`"

$t2 contains: 0000 0000 0000 0000 0000 0000 0000 0000

Fill empty bits with 0s

Logical Operations - and

Instruction	Meaning
`and $t0, $t1, $t2`	`$t0 ← $t1 and $t2`

Example:

$t1 contains: 0000 0000 0000 0000 0000 0000 0010 1101

$t2 contains: 0000 0000 0000 0000 0000 0000 0011 0101

after "and $t0, $t1, $t2"

$t0 contains: 0000 0000 0000 0000 0000 0000 0010 0101

Logical Operations - or

Instruction	Meaning
`or $t0, $t1, $t2`	`$t0 ← $t1 or $t2`

Example:

$t1 contains: 0000 0000 0000 0000 0000 0000 0010 1101

$t2 contains: 0000 0000 0000 0000 0000 0000 0011 0101

after "or $t0, $t1, $t2"

$t0 contains: 0000 0000 0000 0000 0000 0000 0011 1101

Logical Operations - NOT

- **Useful to invert bits in a word: Change 0 to 1, and 1 to 0**
- **Note: NOT (*a* OR *b*) ≡ *a* NOR *b***
- **MIPS has NOR instruction only**

  So, Not *a* ≡ NOT (*a* OR *0*) ≡ *a* NOR *0*

Instruction	**Meaning**
`nor $t0, $t1, $zero`	`$t0 ← NOT ($t1 or $zero)`
	`$t0 ← NOT $t1`

Example:

$t1 contains: 0000 0000 0000 0000 0000 0000 0010 1101

$zero contains: 0000 0000 0000 0000 0000 0000 0000 0000

after "nor $t0, $t1, $zero"

$t0 contains: 1111 1111 1111 1111 1111 1111 1101 0010

Control (if statement)

- **Decision making instructions**
  - **alter the control flow,**
  - **i.e., change the "next" instruction to be executed**

- **MIPS conditional branch instructions (bne, beq: I-format):**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

→ **if ($t0 != $t1) go to Label**
→ **if ($t0 == $t1) go to Label**

- **Example:** `if (i==j) h = i + j;`

-

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```

h i j

Control (if..else.. Statement)

- **MIPS unconditional branch instructions:**

```
j   label
```

- **J-format:**

op	26 bit address

- **Example:**

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

```
                i     j        h
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1:  sub $s3, $s4, $s5
Lab2:  ...
```

Control (while loops)

- **C/Java code (assuming *i* in $s3, *k* in $s5, address of A in $s6):**

```
while (A[i] == k) i = i+1;
```

- **Compiled MIPS code:**

i

```
Loop: sll  $t1, $s3, 2      # $t1 ← 4 * i
      add  $t1, $t1, $s6    # $t1 ← address A[i]
      lw   $t0, 0($t1)      # $t0 ← value A[i]
      bne  $t0, $s5, Exit   # if (A[i] != k)
      addi $s3, $s3, 1      # else i = i+1
      j    Loop             # go to Loop
Exit: …
```

So far:

- **Instruction** **Meaning**

```
add $s1,$s2,$s3    $s1 = $s2 + $s3
sub $s1,$s2,$s3    $s1 = $s2 - $s3
lw $s1,100($s2)    $s1 = Memory[$s2+100]
sw $s1,100($s2)    Memory[$s2+100] = $s1
bne $s4,$s5,L      Next instr. is at Label if $s4 ° $s5
beq $s4,$s5,L      Next instr. is at Label if $s4 = $s5
j Label            Next instr. is at Label
```

- **Formats:**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Control Flow (slt, slti)

- **We have: beq, bne, what about Branch-if-less-than?**
- **New instructions:**

MIPS:	**Meaning:**

`slt $t0, $s1, $s2`
(R-format)
	`if ($s1 < $s2) then`
	`  $t0 = 1`
	`else`
	`  $t0 = 0`

`slti $t0, $s1, 10`
(I-format)
	`if ($s1 < 10) then`
	`  $t0 = 1`
	`else`
	`  $t0 = 0`

Control Flow (`slt`, `slti`)

- `slt` and `slti` are used in combination with `beq` and `bne`

- **Example 1:**

```
slt $t0, $s1, $s2          # if ($s1 < $s2)
bne $t0, $zero, Label      # branch to Label, skip {do ...}
   {do something ...}         same as
 Label:                    # if ($s1 >= $s2)
                           # {do something ...}
```

- **Example 2:**

```
slti $t0, $s1, 100         # if ($s1 >= 100)
beq $t0, $zero, Label      # branch to Label, skip {do...}
   {do something ...}         same as
 Label:                    # if ($s1 < 100)
                           # {do something ...}
```

Control Flow (`slt`, `slti`)

Example 3:

Java/C++:

```
if (x < y)
   { label1: do something ...}
else
   { label2: do something ...}
```

MIPS:

```
                x     y
        slt $t0, $s0, $s1
        bne $t0, $zero, Label1   # if x < y, go to Label1
        j Label2                 # else go to Label2
Label1: {do something ...}
        j Exit
Label2: {do something ...}
  Exit:
```

Control (for loops)

- C/Java code (assuming *i* in $s3, *k* in $s5, address of A in $s6):

```
for (i=0; i<k; i++)
    A[i]=0;
```

- Compiled MIPS code:

```
                                i   k

      li    $S3, 0                  # i=0 (addi $s3, $zero, 0)
Loop: slt   $t0, $s3, $s5           # test if i<k
      beq   $t0, $zero, Exit        # if i>=k, go to Exit
      sll   $t1, $s3, 2             # $t1 ← (4*i)
      add   $t1, $t1, $s6           # $t1 ← address A[i]
      sw    $zero, 0($t1)           # A[i]← 0
      addi  $s3, $s3, 1             # i = i+1
      j     Loop                    # go to Loop
Exit: …

                                     A[]
```

Branch Instruction Design

- Why not `blt` (>), `bge` (>=), etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Signed vs. Unsigned

- **Signed comparison:** `slt, slti`
- **Unsigned comparison:** `sltu, sltui`

- **Example**
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt  $t0, $s0, $s1  # signed`
    - `–1 < +1` ⇒ `$t0 = 1`
  - `sltu $t0, $s0, $s1  # unsigned`
    - `+4,294,967,295 > +1` ⇒ `$t0 = 0`

Register Usage

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Constants

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B - 1;
  C = 100;

- Solutions
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

- MIPS Instructions:

```
addi $s2, $s2, 5      → $s2 = $s2 + 5
slti $s1, $s2, 10     → if ($s2 < 10) then $s1 = 1 else $s1 = 0
addi $s2, $s2, -6     → $s2 = $2 – 6
li $s1, 100           → $s1 = 100
```

- Most constants are small → 16-bit immediate is sufficient
- For the occasional 32-bit constant → see next slide

How about larger constants (32-bit)?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 0000101010101010
```

filled with zeros

0000101010101010	0000000000000000

- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 0000000000000111
```

0000101010101010	0000000000000000
0000000000000000	0000000000000111

ori

0000101010101010	0000000000000111

this is the value we need

Branch Addressing

- **Branch instructions specify**
  - **Opcode, two registers, target address**
- **Most branch targets are near branch**
  - **Forward or backward**

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- **PC-relative addressing**
  - **Target address = PC + offset × 4**
  - **PC already incremented by 4 by this time**

Program Counter

Jump Addressing

- **Jump (`j` and `jal`) targets could be anywhere in text segment**
  - **Encode full address in instruction**

op	address
6 bits	26 bits

- **(Pseudo)Direct jump addressing**
  - **Target address = PC : (address × 4)**

Program Counter

Target Addressing Example

- **Loop code from earlier example**
  - **Assume Loop at location 80000**

20000*4=80000 (for jump)

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

0	0	19	9	2	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

PC (80016)+2*4=80024

Assembler Pseudoinstructions

- **Most assembler instructions represent machine instructions one-to-one**
- **Pseudoinstructions: figments of the assembler's imagination**

MIPS:		Meaning:
`move $t0, $t1`	→	`add $t0, $zero, $t1`
`li $t0, 100`	→	`addi $t0, $zero, 100`
`blt $t0, $t1, L`	→	`slt $at, $t0, $t1` `bne $at, $zero, L`

Note: $at (register 1): assembler temporary

Procedure Calling

- **Steps required**
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call

Procedure Calling Instructions

- **Procedure call: jump and link**

```
jal ProcedureLabel
```

  - **Address of following instruction put in `$ra`**
  - **Jumps to target address**

- **Procedure return: jump register**

```
jr $ra
```

  - **Copies `$ra` to program counter**
  - **Can also be used for computed jumps**
    - e.g., for case/switch statements

Procedure Test

- **C/C++ code:**

```
int Test(int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  – Arguments g, h, i, j in $a0, $a1, $a2, $a3
  – f in $s0 (hence, need to save $s0 on stack $sp)
  – Result in $v0

Procedure Example

- **MIPS code:**

```
Test:
    addi $sp, $sp, -8
    sw   $s0, 0($sp)
    sw   $ra, 4($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $ra, 4($sp)
    lw   $s0, 0($sp)
    addi $sp, $sp, 8
    jr   $ra
```

Procedure Label

Adjust stack pointer
Save $ra, $s0 on stack

Procedure body
f = (g + h) - (i + j);

Result

Restore $ra and $s0
Adjust stack pointer

Return

C/C++ Sort Example

- **Illustrates use of assembly instructions for a C/C++ bubble sort function**

- **Swap procedure to swap v[k] and v[k+1]:**

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  – ***v*** **in** ***$a0***, ***k*** **in** ***$a1***, ***temp*** **in** ***$t0***

The MIPS Procedure Swap

```
swap:
      addi $sp, $sp, -4 # Adjust stack pointer
      sw   $ra, 0($sp)  # save return address
      sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #  (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      lw   $ra, 0($sp)  # Adjust stack pointer
      addi $sp, $sp, 4  # restore return address
      jr $ra            # return to calling
                        #  routine
```

The Sort Procedure in C/C++

- **Calls swap**

```
void sort (int v[], int n)
{
int i, j;

for (i = 0; i < n; i += 1){
 for (j = i-1; j >= 0 && v[j]>v[j+1]; j -= 1){
     swap(v,j); // to swap v[k] and v[k+1]
    }
  }
}
```

  – ***v*** **in** ***$a0***, ***k*** **in** ***$a1***, ***i*** **in** ***$s0***, ***j*** **in** ***$s1***

The MIPS Procedure Body

```
         move $s2, $a0          # save $a0 into $s2
         move $s3, $a1          # save $a1 into $s3
         move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
         beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, -1      # j = i - 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2       # $t1 = j * 4
         add  $t2, $s2, $t1     # $t2 = v + (j * 4)
         lw   $t3, 0($t2)       # $t3 = v[j]
         lw   $t4, 4($t2)       # $t4 = v[j + 1]
         slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
         move $a0, $s2          # 1st param of swap is v (old $a0)
         move $a1, $s1          # 2nd param of swap is j
         jal  swap              # call swap procedure
         addi $s1, $s1, -1      # j -= 1
         j    for2tst           # jump to test of inner loop
exit2:   addi $s0, $s0, 1       # i += 1
         j    for1tst           # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

The Full MIPS Procedure

```
sort:    addi $sp,$sp, -20      # make room on stack for 5 registers
         sw $ra, 16($sp)        # save $ra on stack
         sw $s3,12($sp)         # save $s3 on stack
         sw $s2, 8($sp)         # save $s2 on stack
         sw $s1, 4($sp)         # save $s1 on stack
         sw $s0, 0($sp)         # save $s0 on stack
         …                      # procedure body
         …
exit1:   lw $s0, 0($sp)         # restore $s0 from stack
         lw $s1, 4($sp)         # restore $s1 from stack
         lw $s2, 8($sp)         # restore $s2 from stack
         lw $s3,12($sp)         # restore $s3 from stack
         lw $ra,16($sp)         # restore $ra from stack
         addi $sp,$sp, 20       # restore stack pointer
         jr $ra                 # return to calling routine
```

add the MIPS procedure body and swap procedure here

Lessons Learned

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Overview of MIPS

- **simple instructions all 32 bits wide**
- **very structured, no unnecessary baggage**
- **only three instruction formats**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- ***op(opcode): basic operation of the instruction***
- ***rs: the first register***
- ***rt: the second register***
- ***rd: the destination register***
- ***shamt: sfift amount***
- ***funct: function code***

Addresses in Branches and Jumps

- **Instructions:**

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4` $\neq$ `$t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4` = `$t5`
`j Label`	Next instruction is at Label

- **Formats:**

I	op	rs	rt	16 bit address
J	op	26 bit address		

- **Addresses are not 32 bits**
  - **How do we handle this with load and store instructions?**

To summarize:

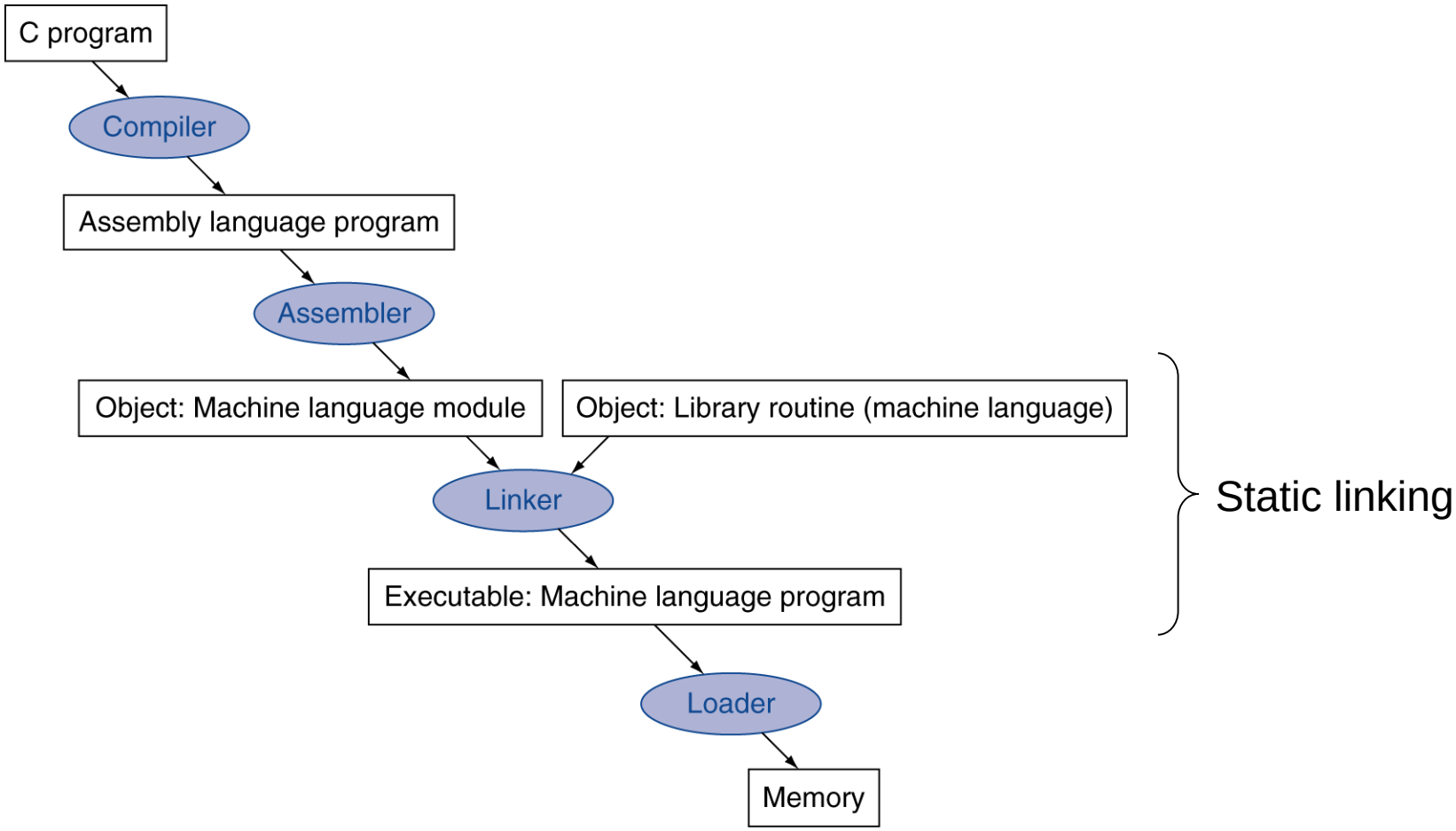
MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Translation and Starting a Program:



Fallacies

- Fallacy: More powerful instructions mean higher performance.
- Fallacy: Write in assembly language to obtain the highest performance.

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!

MIPS assembly language

Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo
Data transfer	load word	lw	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw	$s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half unsigned	lhu	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh	$s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte unsigned	lbu	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb	$s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store conditional word	sc	$s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half atomic swap
	load upper immediate	lui	$s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	AND	AND	$s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	OR	OR	$s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	NOR	NOR	$s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	AND immediate	ANDi	$s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	OR immediate	ORi	$s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR with constant
	shift left logical	sll	$s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl	$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq	$s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne	$s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Unconditional jump	jump	j	2500	go to 10000	Jump to target address
	jump register	jr	$ra	go to $ra	For switch, procedure return
	jump and link	jal	2500	$ra = PC + 4; go to 10000	For procedure call

MIPS machine language

Name	Format	Example						Comments
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
add	R	0	2	3	1	0	32	add $1,$2,$3
sub	R	0	2	3	1	0	34	sub $1,$2,$3
addi	I	8	2	1	100			addi $1,$2,100
addu	R	0	2	3	1	0	33	addu $1,$2,$3
subu	R	0	2	3	1	0	35	subu $1,$2,$3
addiu	I	9	2	1	100			addiu $1,$2,100
mfc0	R	16	0	1	14	0	0	mfc0 $1,$epc
mult	R	0	2	3	0	0	24	mult $2,$3
multu	R	0	2	3	0	0	25	multu $2,$3
div	R	0	2	3	0	0	26	div $2,$3
divu	R	0	2	3	0	0	27	divu $2,$3
mfhi	R	0	0	0	1	0	16	mfhi $1
mflo	R	0	0	0	1	0	18	mflo $1
and	R	0	2	3	1	0	36	and $1,$2,$3
or	R	0	2	3	1	0	37	or $1,$2,$3
andi	I	12	2	1	100			andi $1,$2,100
ori	I	13	2	1	100			ori $1,$2,100
sll	R	0	0	2	1	10	0	sll $1,$2,10
srl	R	0	0	2	1	10	2	srl $1,$2,10
lw	I	35	2	1	100			lw $1,100($2)
sw	I	43	2	1	100			sw $1,100($2)
lui	I	15	0	1	100			lui $1,100
beq	I	4	1	2	25			beq $1,$2,100
bne	I	5	1	2	25			bne $1,$2,100
slt	R	0	2	3	1	0	42	slt $1,$2,$3
slti	I	10	2	1	100			slti $1,$2,100
sltu	R	0	2	3	1	0	43	sltu $1,$2,$3
sltiu	I	11	2	1	100			sltiu $1,$2,100
j	J	2	2500					j 10000
jr	R	0	31	0	0	0	8	jr $31
jal	J	3	2500					jal 10000

MIPS instruction formats

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Main MIPS machine language. Formats and examples are shown, with values in each field: op and funct fields form the opcode (each 6 bits), rs field gives a source register (5 bits), rt is also normally a source register (5 bits), rd is the destination register (5 bits), and shamt supplies the shift amount (5 bits). The field values are all in decimal. Floating-point machine language instructions are shown in Figure 4.47 on page 291. Appendix A gives the full MIPS machine language.

Register Number	Alternative Name	Description
0	zero	the value 0
1	$at	(**a**ssembler **t**emporary) reserved by the assembler
2-3	$v0 - $v1	(**v**alues) from expression evaluation and function results
4-7	$a0 - $a3	(**a**rguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(**s**aved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	**g**lobal **p**ointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	**s**tack **p**ointer Points to last location on the stack.
30	$s8/$fp	**s**aved value / **f**rame **p**ointer Preserved across procedure calls
31	$ra	**r**eturn **a**ddress

Service	Code in $v0	Arguments	Results
print_int	1	$a0 = integer to be printed	
print_float	2	$f12 = float to be printed	
print_double	3	$f12 = double to be printed	
print_string	4	$a0 = address of string in memory	
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	
sbrk	9	$a0 = amount	address in $v0
exit	10		
print char	11	$a0 = character to print	
read char	12		$v0 contains char read

MIPS Reference Data

①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31–26	25–21	20–16	15–11	10–6	5–0
I	opcode	rs	rt	immediate		
	31–26	25–21	20–16	15–0		
J	opcode	address				
	31–26	25–0				

ARITHMETIC CORE INSTRUCTION SET

②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/y
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/y

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct
	31–26	25–21	20–16	15–11	10–6	5–0
FI	opcode	fmt	ft	immediate		
	31–26	25–21	20–16	15–0		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

© 2014 by Elsevier, Inc. All rights reserved. From Patterson and Hennessy, *Computer Organization and Design*, 5th ed.