Unit-3

D.Venkata Vara Prasad
SSN College of Engineering

Session Meta Data

Author	D.Venkata Vara Prasad
Version No	1.1
Release Date	7.03.2021
Reviewer	

Revision History

Date of Revision	Details	Version Number

Session Objectives

- To explain the Logic Design Conventions

Session Outcomes

- At the end of the session, students will be able to
  - Understand the Logic design conventions

The Processor

Agenda

1. Introduction
2. Logic Design Convention
3. Building a Datapath
4. A Simple Implementation Scheme
5. An Overview of Pipelining
6. Pipelined Datapath and Control
7. Data Hazards: Forwarding versus Stalling
8. Control Hazards
9. Exception

Datapath Design

We will design a simplified MIPS processor

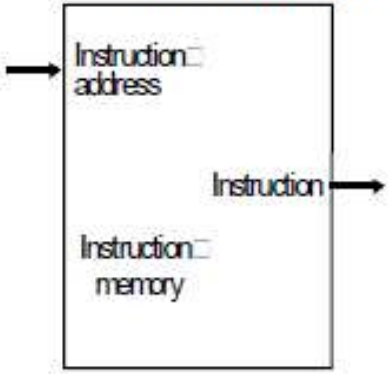
- The instructions supported are
  - memory-reference instructions: lw, sw
  - arithmetic-logical instructions: add, sub, and, or, slt
  - control flow instructions: beq, j
- Generic Implementation: – use the program counter (PC) to supply instruction address – get the instruction from memory – read registers – use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers Why? memory-reference? arithmetic? control flow?

What blocks we need

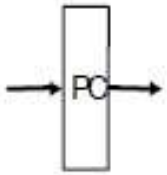
- We need an ALU
- We need memory to store inst and data – Instruction memory takes address and supplies inst – Data memory takes address and supply data for lw – Data memory takes address and data and write into memory
- We need to manage a PC and its update mechanism
- We need a register file to include 32 registers – We read two operands and write a result back in register file
- Some times part of the operand comes from instruction
- We may add support of immediate class of instructions
- We may add support for J, JR, JAL

Simple Implementation

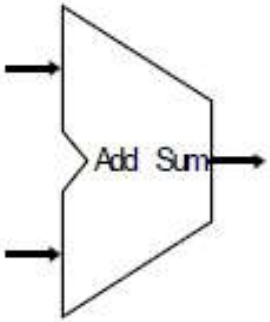
- **Include the functional units we need for each instruction**



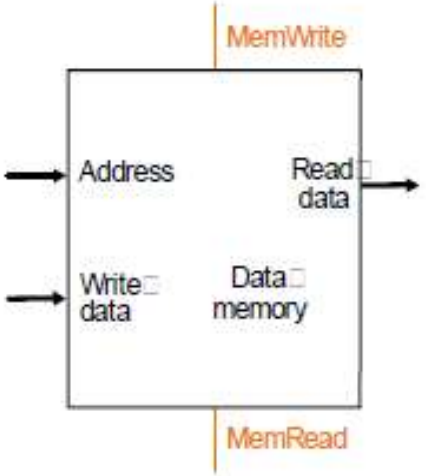
a. Instruction memory



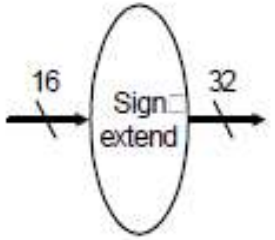
b. Program counter



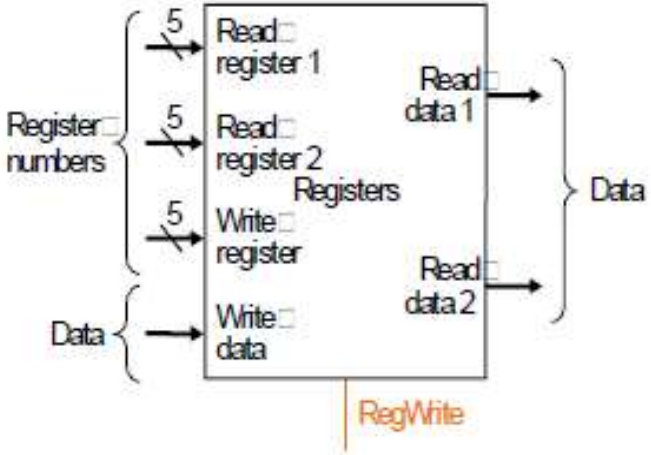
c. Adder



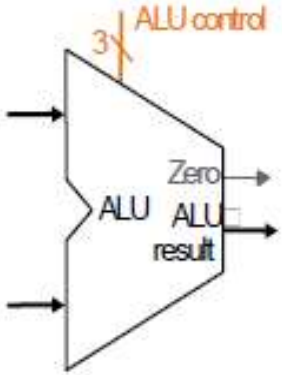
a. Data memory unit



b. Sign-extension unit



a. Registers



b. ALU

Why do we need this stuff?

Introduction

- Performance of a computer is determined by three key factors:
  - Instruction count — Determined by compiler and the instruction set architecture
  - Clock cycle time
  - Clock cycles per instruction (CPI)

  } Determined by the implement of processor

- The main purpose of this chapter:
  - Explanation of the principles and techniques used in implementing a processor with MIPS instruction set.
  - Building up a datapath and constructing a simple version of a processor sufficient to implement an instruction set like MIPS.
  - Covering a more realistic pipeline MIPS implementation, followed by a section that develops the concepts necessary to implement more complex instruction sets, like the x86.

A Basic MIPS Implementation

Examining an implementation that includes a subset of the core MIPS

instruction set:

- The memory-reference instructions load word (**lw**) and store word (**sw**)
- The arithmetic-logical instructions **add, sub, AND, OR**, and **slt**
- The instructions branch equal (**beq**)

(missing: **shift, multiply, divide, floating-point** instructions)

Introduction

An overview of the implementation

Memory-reference instruction:

Fetch the instruction → read one/two registers → use ALU →

access the memory to read/write data

Arithmetic-logical instruction:

Fetch the instruction → read one/two registers → use ALU →
write data to register

Brach instruction:

Fetch the instruction → read one/two registers → use ALU

→ change the next instruction address based on the comparison

More Implementation Details

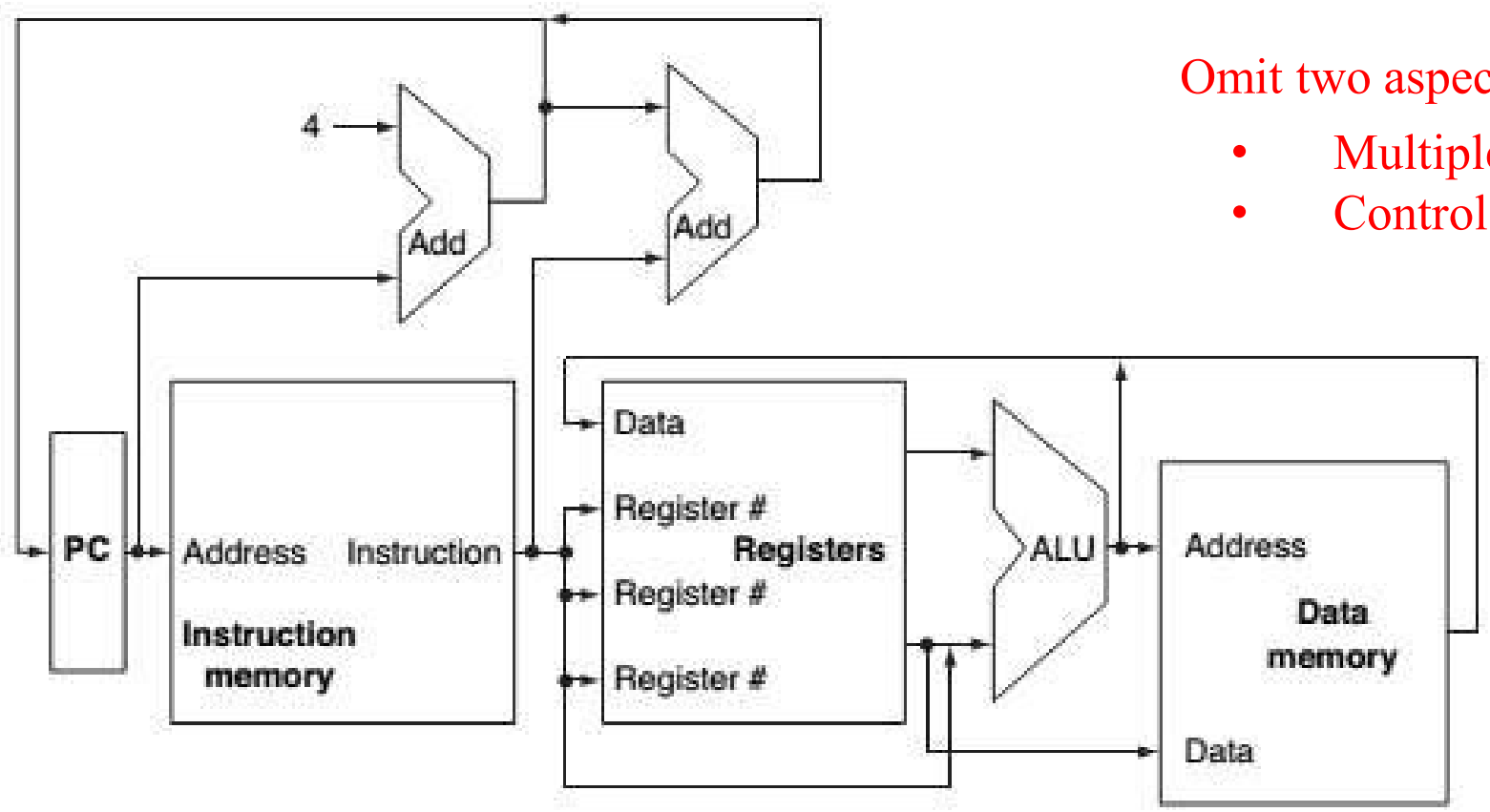
For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.

2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

Introduction

An overview of the implementation



Omit two aspects ???

- Multiplexor
- Control unit

Fig.1 An abstract view of the implementation of the MIPS subset showing the major function units and the major connections between them

More Implementation Details

- The program counter supply the instruction address to the instruction memory
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction
- Once the register operands have been fetched, they can be operated on

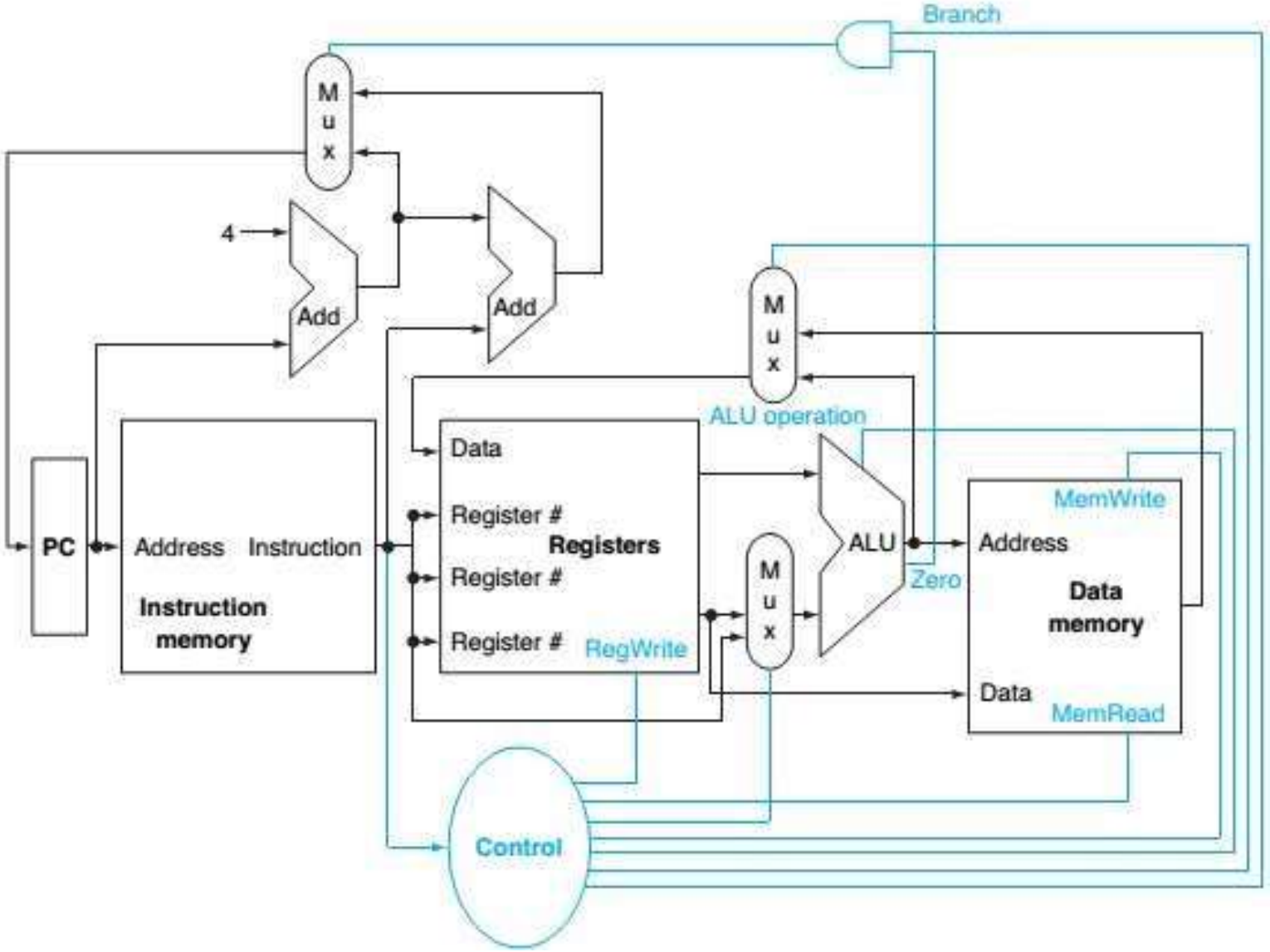
  - to compute a memory address (for a load or store),
  - to compute an arithmetic result (for an integer arithmetic- logical instruction)
  -to compare (for a branch).

More Implementation Details

- If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
- If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file.
- Branches require the use of the ALU output to determine the next instruction address:
  - either from the ALU (where the PC and branch off set are summed)
  - from an adder that increments the current PC by 4.

Introduction

An overview of the implementation



How many of the five classic components of a computer shown in Fig.1 and Fig.2 ???

Fig.2 The basic implementation of the MIPS subset, including the necessary multiplexors and control lines

More Implementation Details

- The top multiplexor (“Mux”) controls what value replaces the PC (PC + 4 or the branch destination address).
- The multiplexor is controlled by the gate that “ANDs” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch.
- The middle multiplexor, whose output returns to the register file:
  - used to steer the output of the ALU (for ALU operation)
  - the output of the data memory (load) for writing into the register file.

More Implementation Details

- The bottommost multiplexor is used to determine whether the second ALU input is :
  - from the registers (for an ALU instruction or a branch)
  - from the off set field of the instruction (for a load or store).

  - The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation

Logic Design Convention

❖To discuss the design of a computer, we must decide how the logic implementing the computer will operate and how the computer is clocked.

❖This section reviews a few key ideas in digital logic that will be used extensively in this chapter.

- **Combinational:** the elements that operate on data values (ALU)
- **State elements (sequential):** the elements contains state if it has some
  internal storage (instruction, data memories and registers)

❖ The below terms are used in this subject:

- **Asserted (assert)**: the signal is logically high or true.
- **Deasserted (deassert):** the signal is logically low or false.

Logic Design Convention

Clocking Methodology

- A **clocking methodology** defines when signals can be read or written. This approach used to determine when data valid and stable relative to the clock.

- **Edge-triggered clocking methodology** is a clocking scheme in which *all state changes occur on a clock edge*. That means that any values stored in a sequential logic element are updated only on a clock edge.

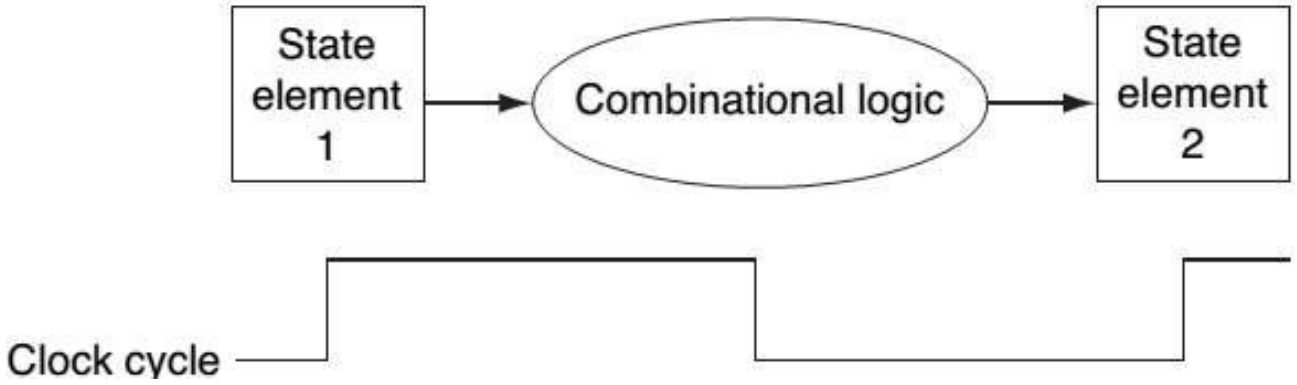


Fig.3 Combinational logic, state elements, and the clock are closely related. The time necessary for the signals to reach state element 2 def ines the **length of the clock cycle**.

Logic Design Convention

Clocking Methodology

- **Control signal**: A signal used for multiplexor selection or for directing the operation of a functional unit.

  Data signal: a signal contains information that is operated on by a functional unit.

- **Bus:** is signals wider than 1 bit, with thicker lines. Several buses combine to from a wider bus. For example, 32-bit bus is obtained by combining two 16-bit buses

Building a Datapath

1. Fetching Instruction

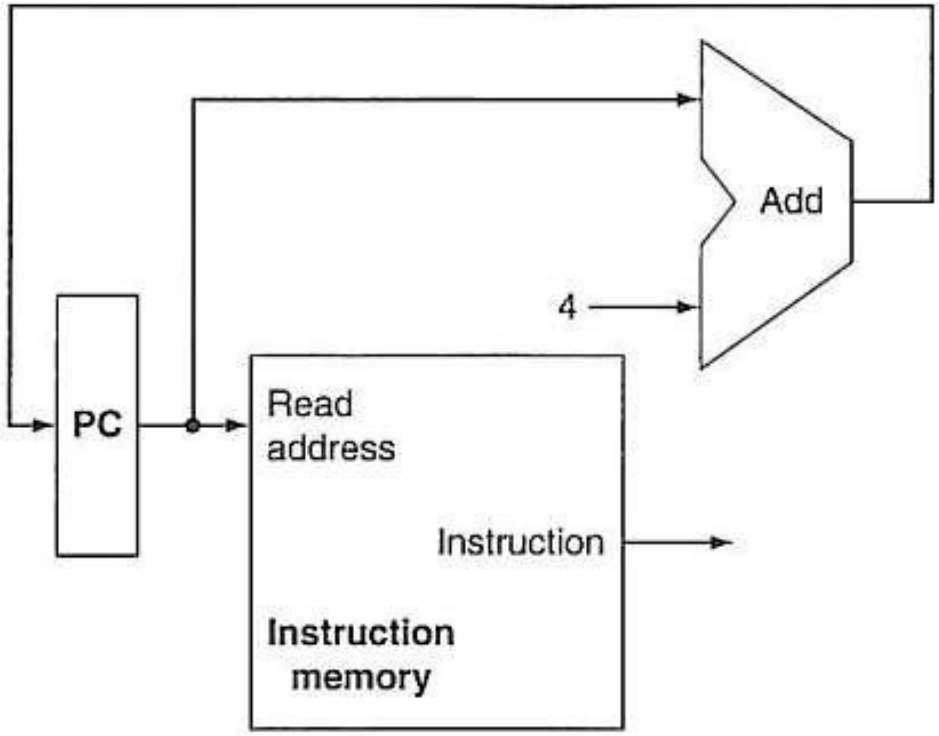


Fig.5 A portion of the datapath used for fetching instructions and incrementing the program counter (PC).

Building a Datapath

- **Datapath element:** A unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include *the instruction and data, the register file, the ALU,* and *adders*.

- **Program Counter (PC):** The register containing the address of the instruction in the program being executed.

- **Register file:** A state element that consists of a set of register that can be read and written by supplying a register number to be accessed.

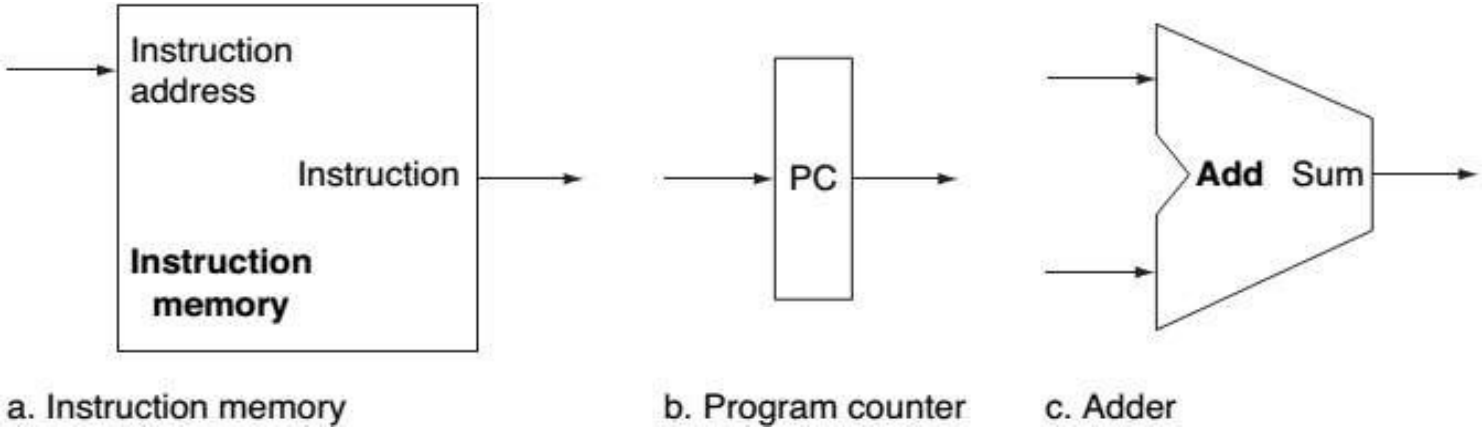


Fig.4 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

Building a Datapath

- We need memory unit to store the instructions of a program
- Need a **program counter (PC)**, holds the address of the current instruction
- Need an adder to increment the PC to the address of the next instruction.
- To execute any instruction, we must start by fetching the instruction from memory.
- To prepare for executing the next instruction, we must also increment the PC to point at the next instruction, 4 bytes later

Building a Datapath

2. R-format instruction (arithmetic-logical instruction)
(add, sub, AND, OR and **slt)**

Example: **add** $\$t_1$, $\$t_2$, $\$t_3$

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

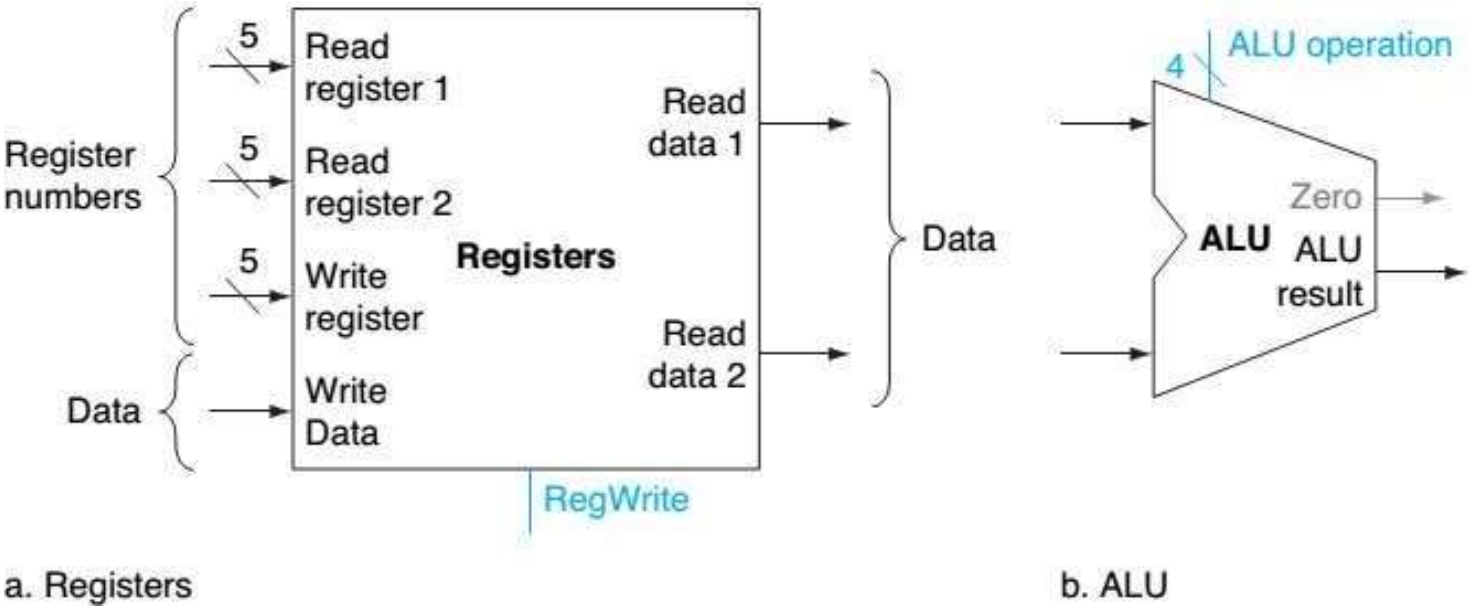


Fig.6 The two elements needed to implement R-format ALU operations are the register file and the ALU.

Building a Datapath

- a total of four inputs (three for register numbers and one for data) and two outputs (both for data).
- The register number inputs are 5 bits wide to specify one of 32 registers.
- the data input and two data output buses are each 32 bits wide.
- ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0.
- The register fi le contains all the registers and has two read ports and one write port.
- The register file outputs the contents of the registers corresponding to the Read register inputs on the outputs.
- A register write must be explicitly indicated by asserting the write control signal and the writes are edge-triggered.

Building a Datapath

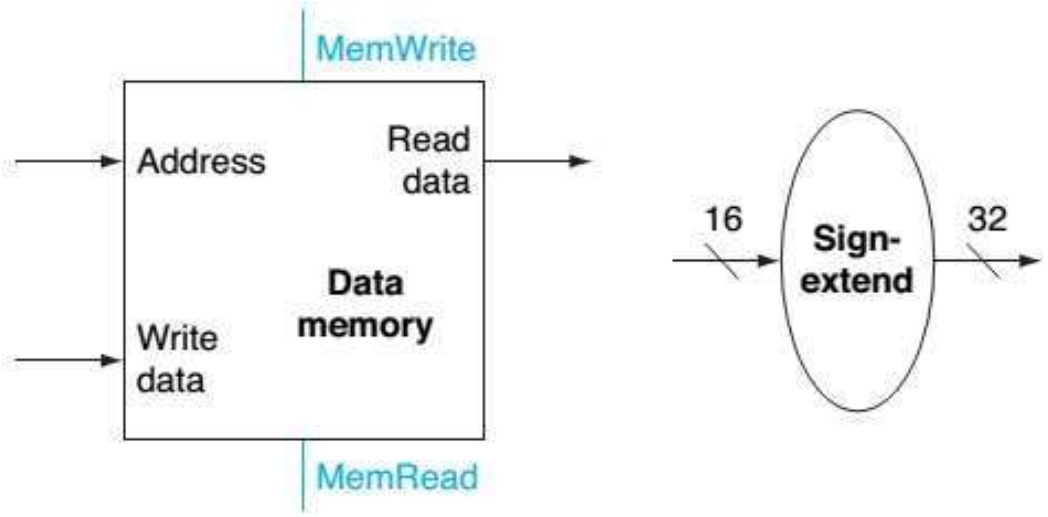
3. Load word and store word instruction

Example:

lw $t1, offset_value($t2)

sw $t1, offset_value($t2)

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0



a. Data memory unit

b. Sign extension unit

Fig.7 The two units needed to implement loads and stores, in addition to the register file and ALU of Fig.6, are the data memory unit and the sign extension unit.

Building a Datapath

- we need a unit to **sign-extend** the 16-bit off set field in the instruction to a 32-bit signed value,
- A Data memory unit
  - to read from or write to the data memory
  - data memory has read and write control signals
  - an address input, and an input for the data to be written into memory.

Building a Datapath

4. branch instruction (beq)

Example: **beq** $\$t_1$, $\$t_2$, offset

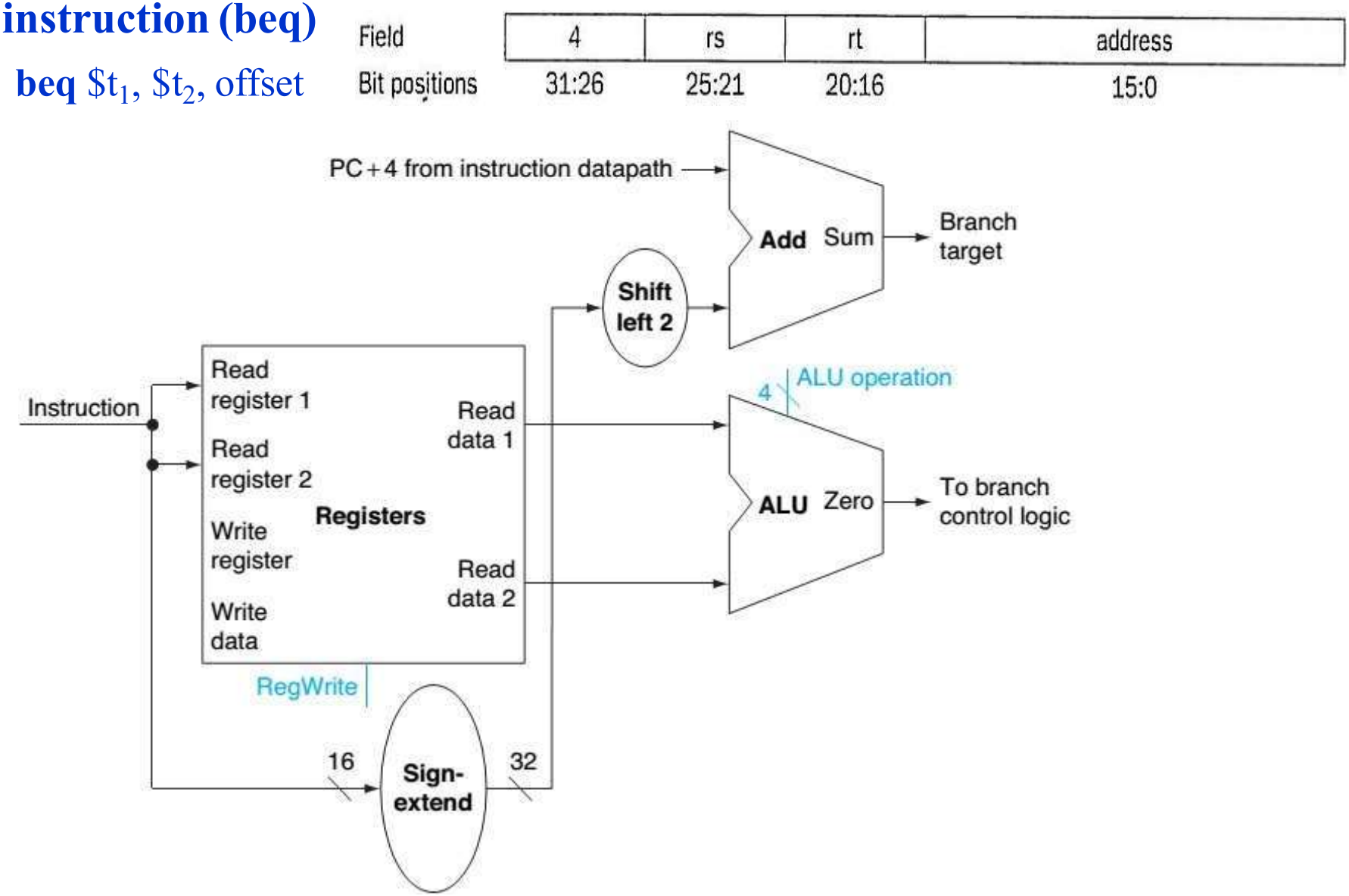


Fig.8 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bit..

Building a Datapath

- The **beq** instruction has three operands, two registers that are compared for equality, and a 16-bit off set used to compute the **branch target address** relative to the branch instruction address.
- Its form is **beq $t1,$t2,offset**.
- Compute the branch target address by adding the sign-extended off set field of the instruction to the PC.
- The ISA specifies that the base for the branch address calculation is the address of the instruction following the branch i.e we compute PC + 4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.
- The architecture also states that the off set field is shifted left 2 bits so that it is a word off set; this shift increases the effective range of the off set field by a factor of 4.

Building a Datapath

Example 1:

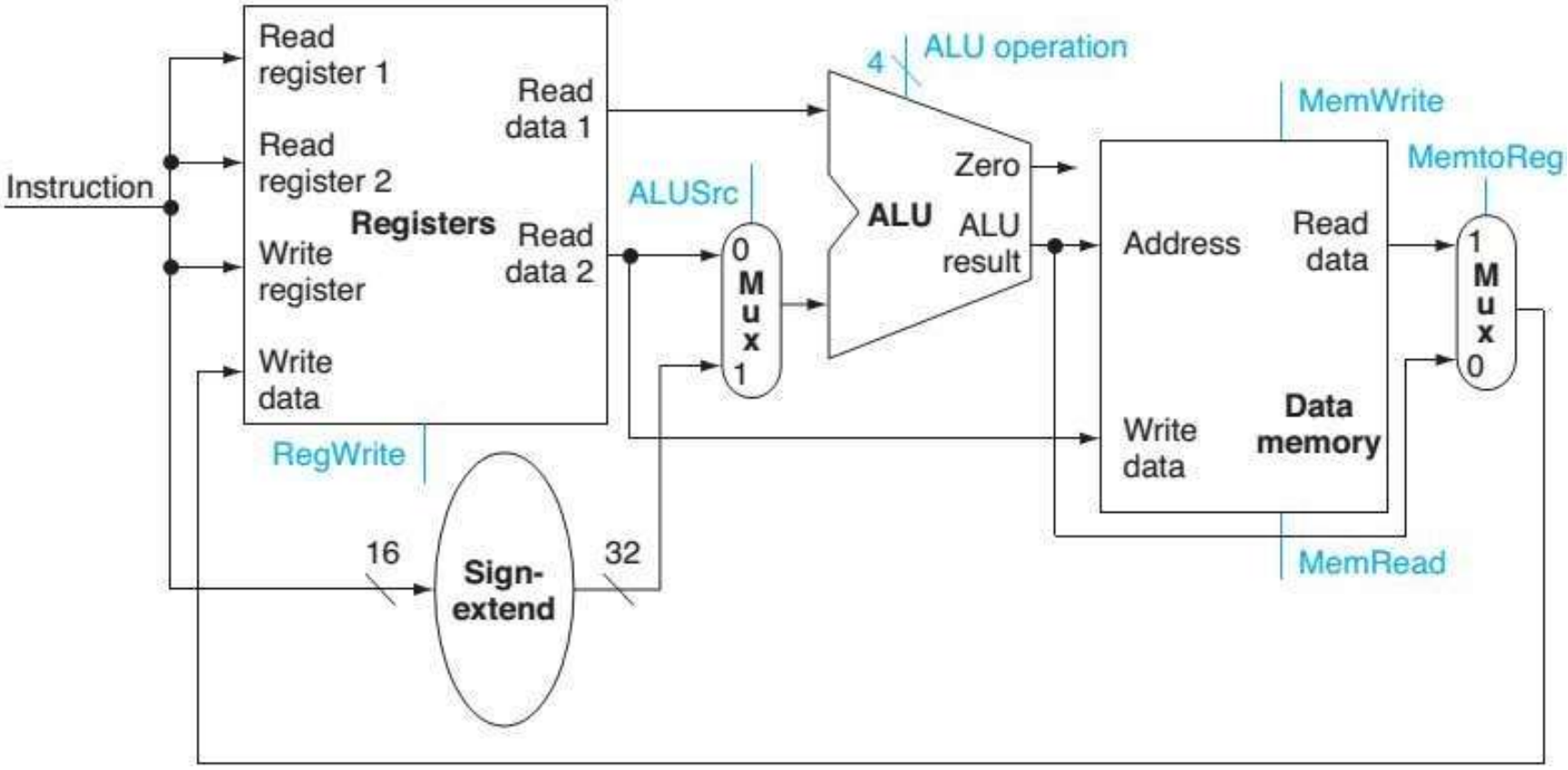


Fig.9 The datapath for the memory instructions and the R-type instructions.

Building a Datapath

Example 2:

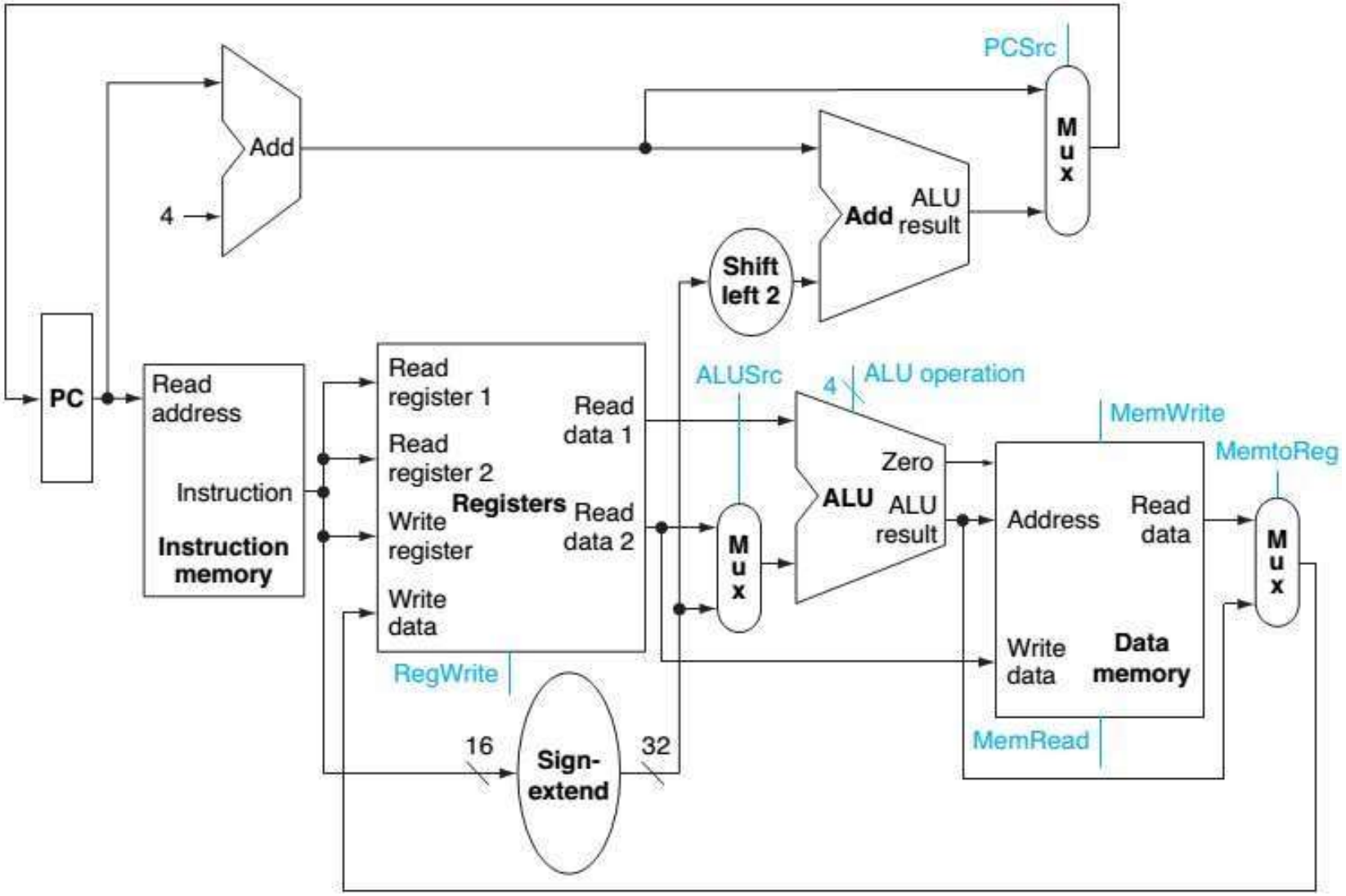


Fig.10 The simple datapath for the MIPS architecture combines the elements required by different instruction classes. The components come form Fig.5, 8, and 9. This datapath can execute the basic instructions (load-store word, ALU operation, and branches) in a single clock cycle.

A Simple Implementation Scheme

The ALU Control

The MIPS ALU defines the **6 combinations** of **4 control inputs**:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Depending on the **instruction class**, the ALU will need to perform one of these first five functions. (NOR is needed for other parts)

- For **load word** and **store word** instructions, the ALU computes the *memory address by addition*.
- For the **R-type instructions**, the ALU needs to perform one of the five actions (*AND, OR, subtract, add, or set on less than*), depending on the value of the *6-bit funt (or function) field in the low-order bits of the instruction*.
  - For **branch equal**, the ALU must *perform a subtraction*

A Simple Implementation Scheme

The ALU Control

- To generate the **4-bit ALU control input,** we can use a small control unit that has as inputs *the function field of the instruction* and *a 2-bit control field* called **ALUOp.**
- **ALUOp** indicates the operation performed by ALU as add (00) for loads and stores, subtract (01) for beq, or determinded by the operation encoded in the funct field (10).
- The **4-bit signal**, the output of *ALU control unit* can control directly the operation of ALU

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.11 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction.

A Simple Implementation Scheme

- The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary.
- when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we “don’t care” about the value of the function code, and the funct field is shown as XXXXXX
- When the ALUOp value is 10, then the function code is used to set the ALU control input

A Simple Implementation Scheme

The ALU Control

❖**Truth Table:** From logic, a representation of a logical operation by listing all values of the inputs and the in each case showing what the resulting outputs should be.

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

Fig.12 The truth table for the ALU control bits (call Operation)

Don't-care term: An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

A Simple Implementation Scheme

- The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown.

- Some don't-care entries have been added.
- Ex: ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01.

- when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

A Simple Implementation Scheme

Design the Main Control Unit

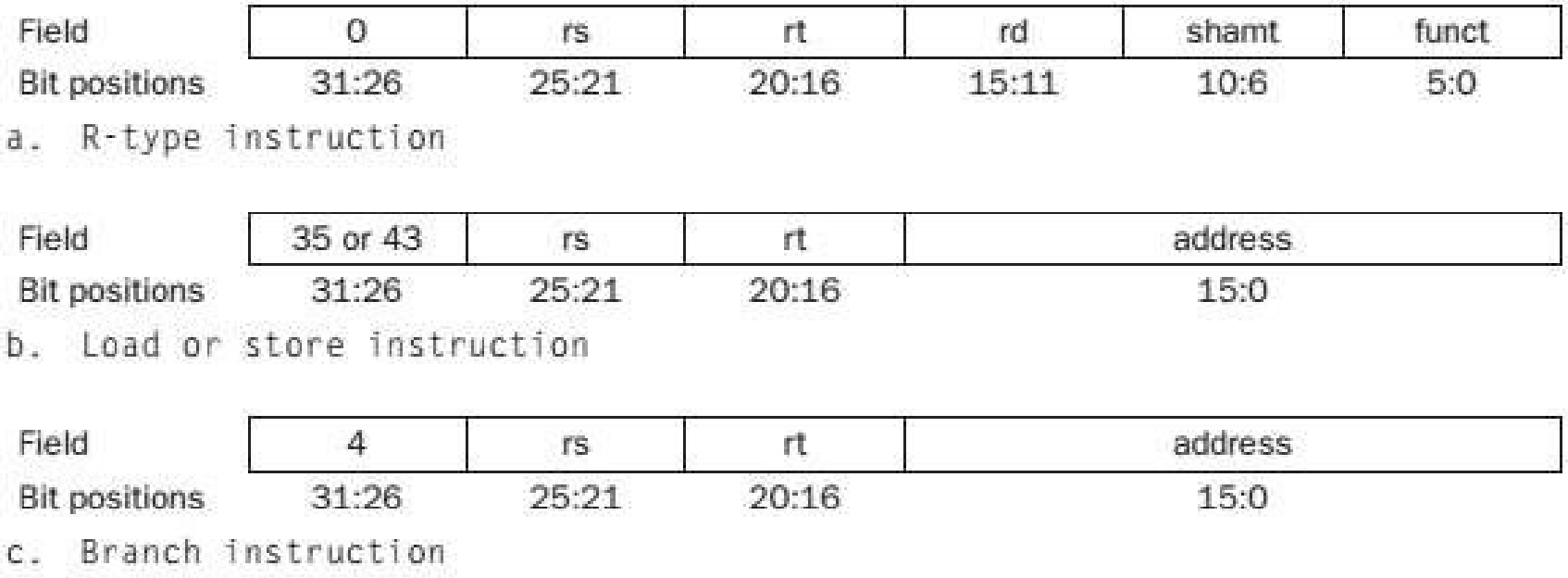
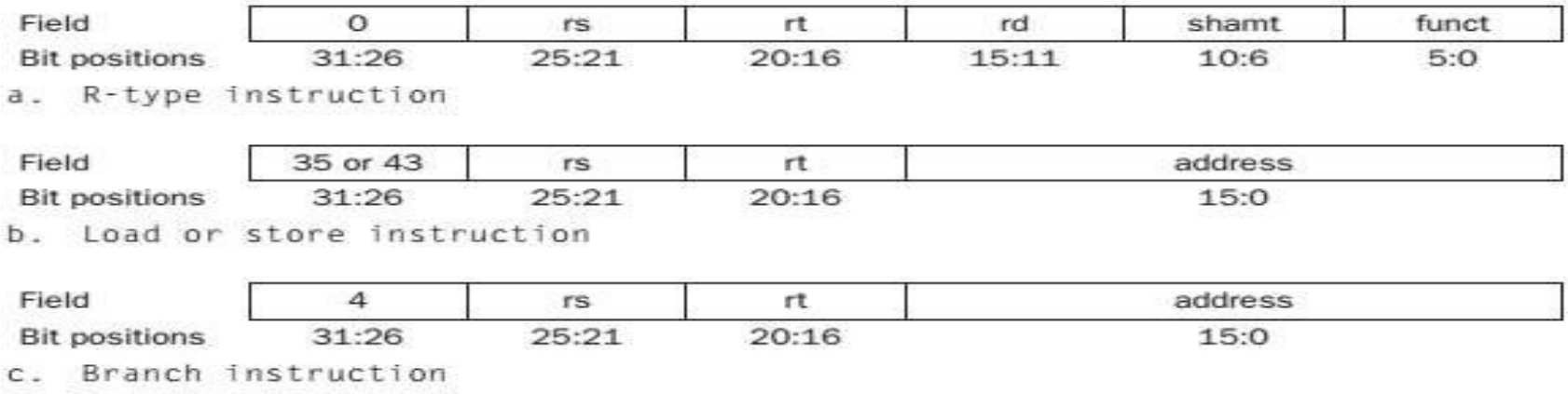


Fig.13 The three instruction classes use to different instruction formats.

A Simple Implementation Scheme



There are several major observations about this instruction format that we will rely on:

- The op field, also called the **opcode**, is always contained bits 31:26. We will refer to this field as **Op[5:0].**
- The two registers to be read are always specified by the **rs** and **rt** fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
- The base register for load and store instructions is always in bit positions **25:21(rs).**
- The 16-bit offset for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load, it is in bit positions **20:16(rt),** while for an R-type instruction it is in bit positions **15:11(rd).** Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

A Simple Implementation Scheme

Design the Main Control Unit

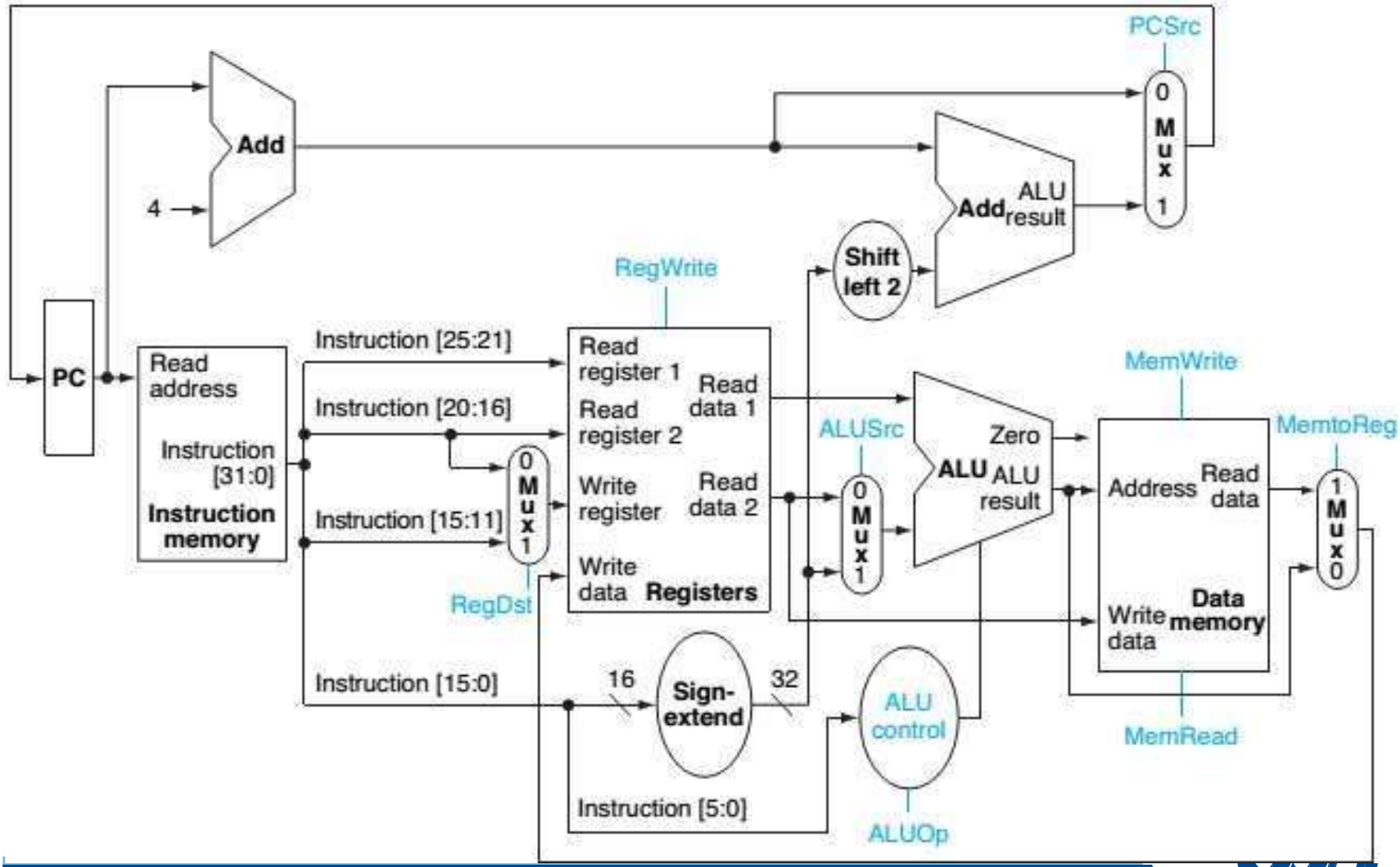


Fig.14 The datapath of Fig.10 with all necessary multiplexors and all control lines identified.

A Simple Implementation Scheme

- The control lines are shown in color.
- The ALU control block has also been added.
- The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.
- ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors.
- All the multiplexors have two inputs, they each require a single control line.

A Simple Implementation Scheme

Design the Main Control Unit

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.

Fig.15 The effect of each of the seven control signals.

A Simple Implementation Scheme

Operation of the Datapath

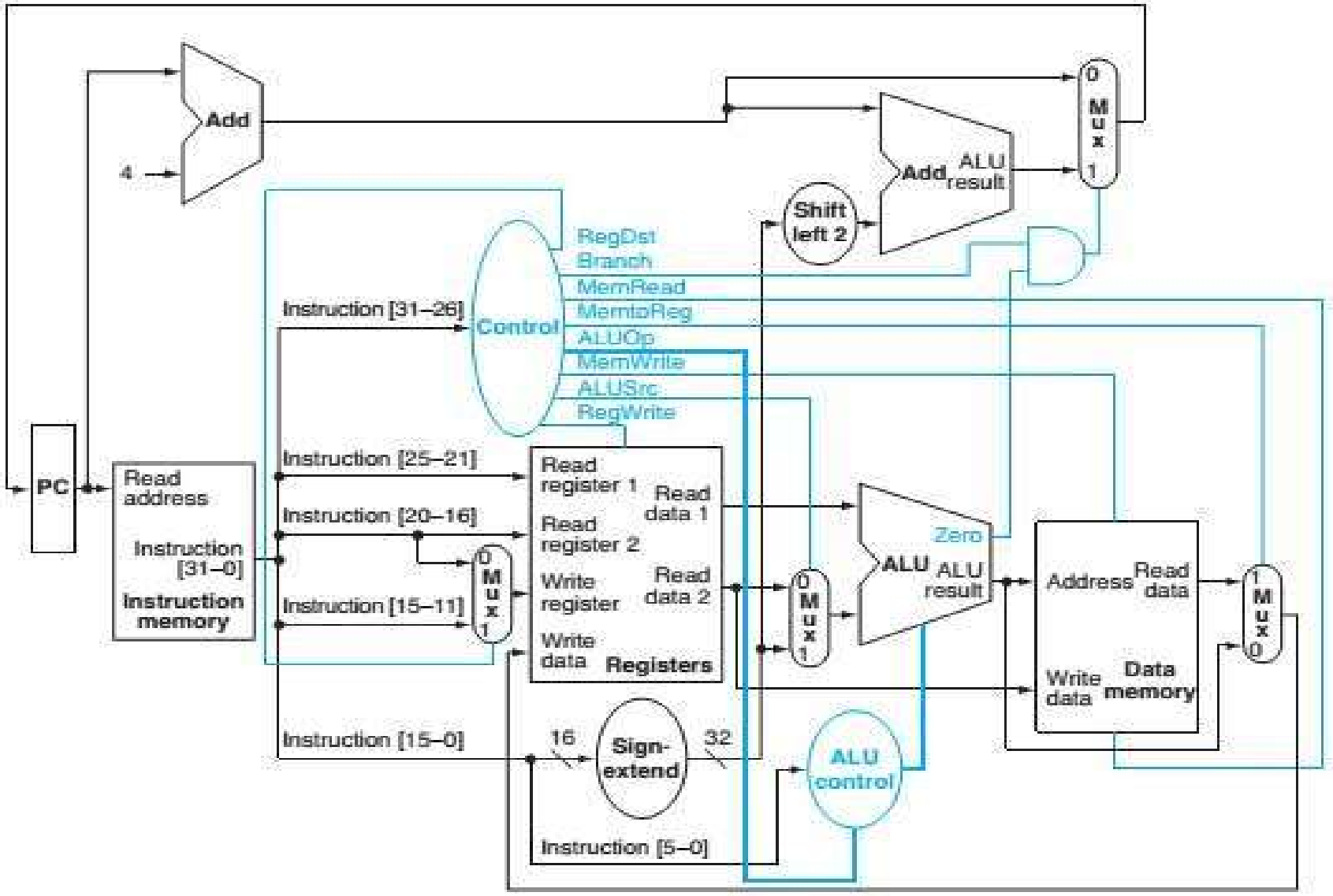


Fig.16 Simple datapath with the control unit.

A Simple Implementation Scheme

Operation of the Datapath

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

Fig.17 The setting of the control lines is completely determined by the opcode fields of the instruction

A Simple Implementation Scheme

- The first row of the table corresponds to the R-format instructions (add, sub, AND, OR, and slt).
- For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set.
- R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory.
- When the Branch control signal=0 : PC ← PC+4 (unconditional)
  otherwise : PC ← branch target
- The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field.
- The second and third rows of this table give the control signal settings for lw & sw.
- These ALUSrc and ALUOp fields are set to perform the address calculation.

A Simple Implementation Scheme

- The MemRead and MemWrite are set to perform the memory access.
- Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register.
- The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality.
- The MemtoReg field is irrelevant when the RegWrite signal is 0:
- Since the register is not being written, the value of the data on the register data write port is not used.
- Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

A Simple Implementation Scheme

Operation of the Datapath

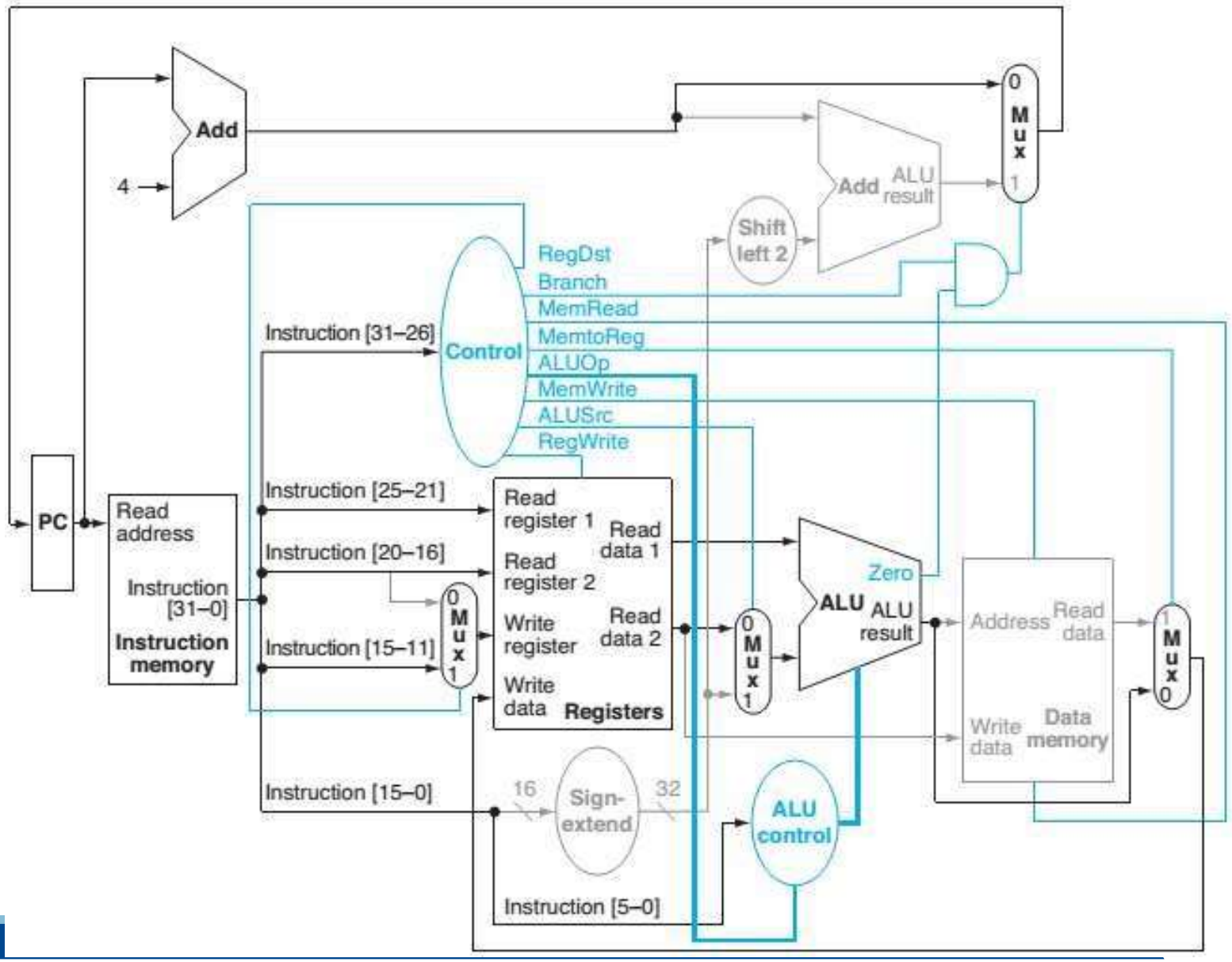


Fig.18 The datapath in operation for an R-type instruction, such as **add $t1, $t2, $t3**.

A Simple Implementation Scheme

R-type instruction : add $t1,$t2,$t3.

Four steps to execute the instruction:

1.The instruction is fetched, and the PC is incremented.

2.Two registers, $t2 and $t3, are read from the register file

3. The ALU operates on the data read from the register fi le, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.

4.The result from the ALU is written into the register fi le using bits 15:11 of the instruction to select the destination register ($t1).

A Simple Implementation Scheme

Operation of the Datapath

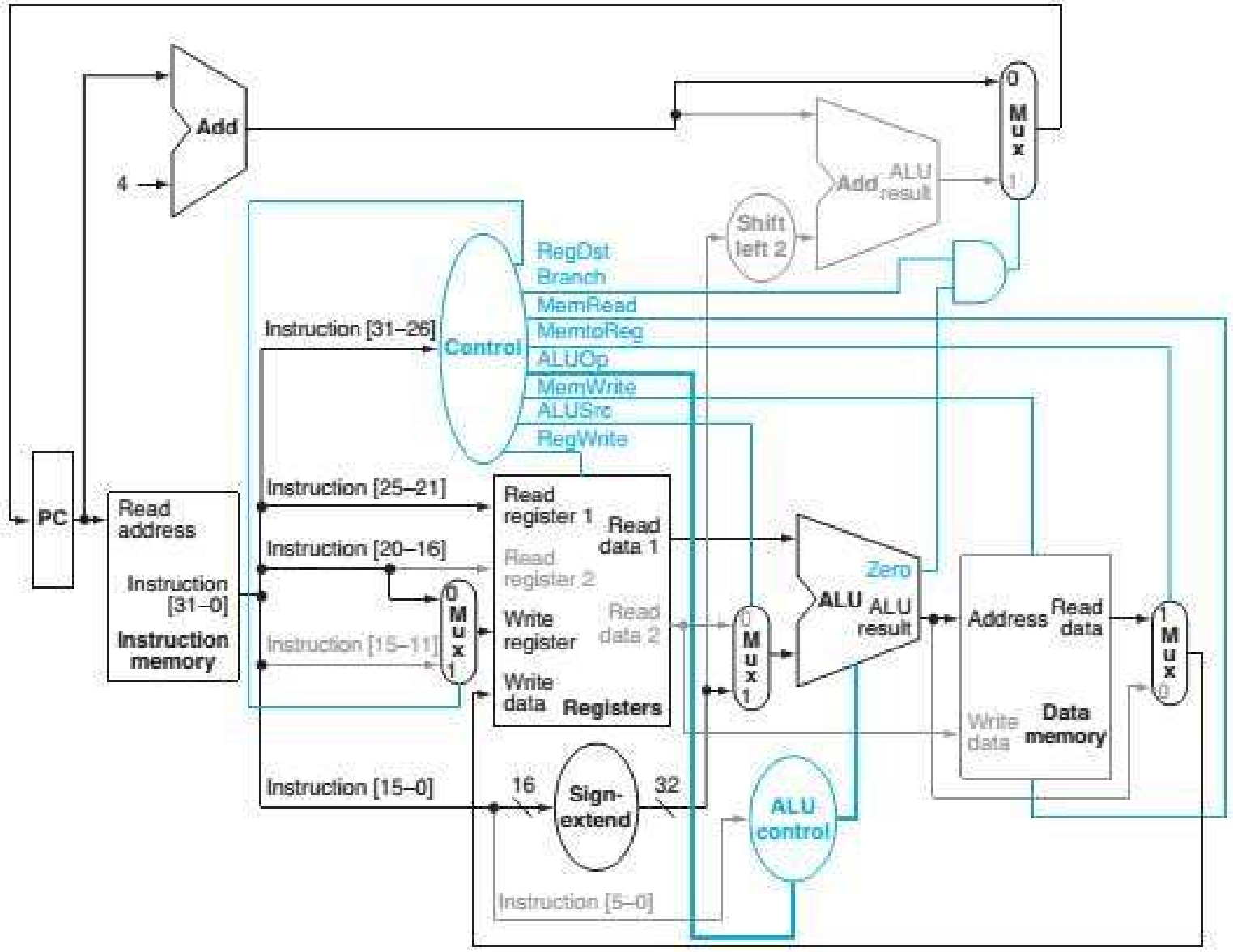


Fig.19 The datapath in operation for a **load instruction.**

A Simple Implementation Scheme

- Memory Ref instruction :lw $t1, offset($t2)

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register ($t2) value is read from the register file.
The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

A Simple Implementation Scheme

Operation of the Datapath

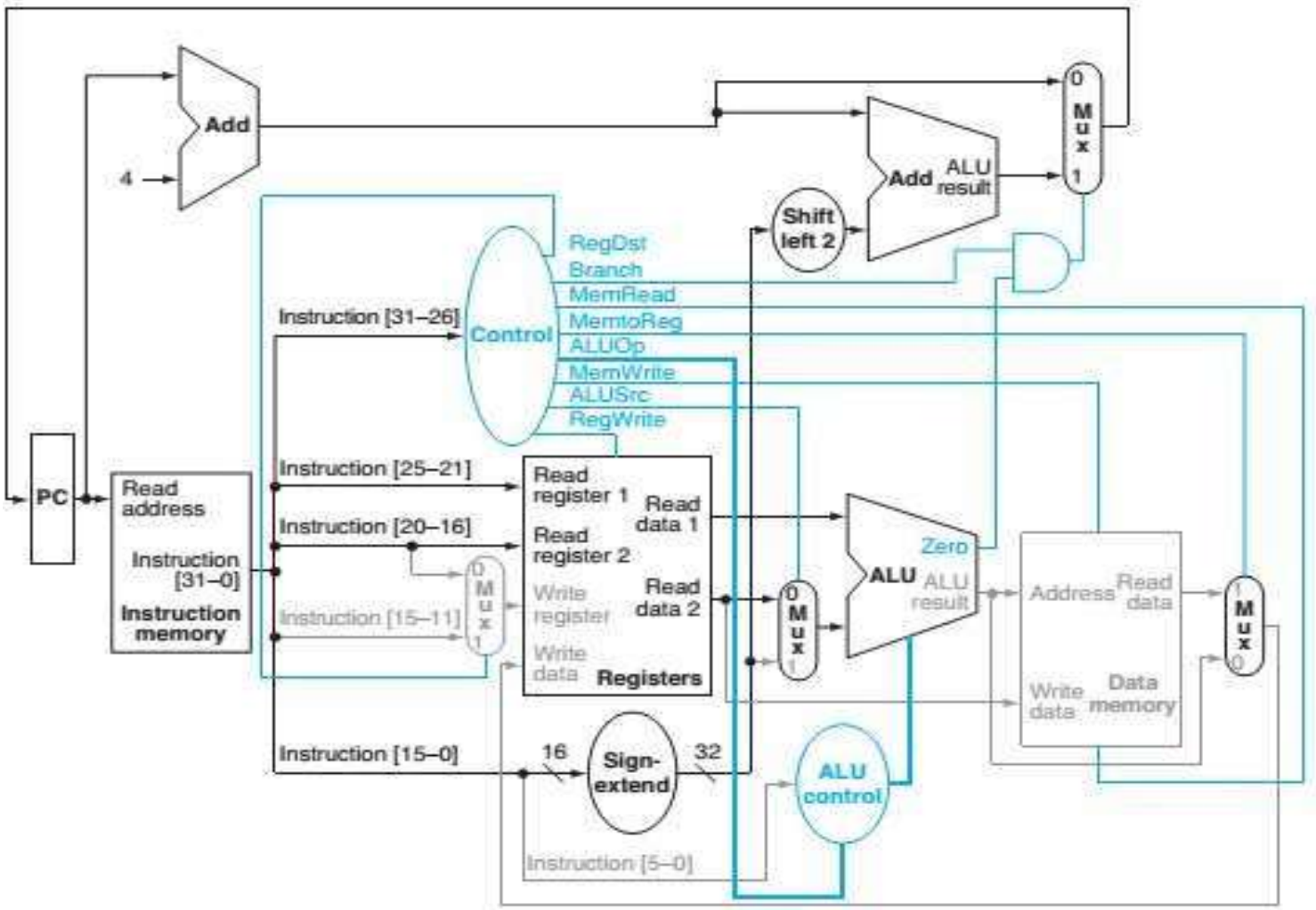


Fig.20 The datapath in operation for a **branch-on-equal instruction.**

A Simple Implementation Scheme

- Branch Instruction :beq $t1, $t2, offset,

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, $t1 and $t2, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shift ed left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

A Simple Implementation Scheme

Finalizing Control

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Fig.21 The control function for the simple single-cycle implementation is completely specified by this truth table.

Single-cycle implementation: Also called single clock cycle implementation. An implementation in which an instruction is *executed in one clock cycle.*

A Simple Implementation Scheme

Implementing JUMPinstruction

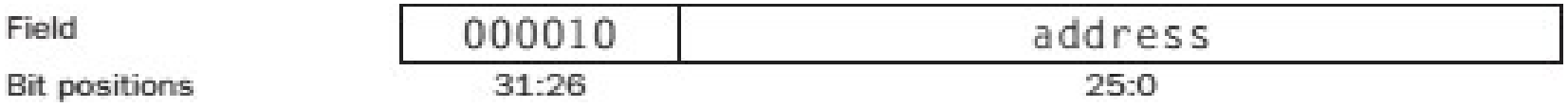


Fig.22 Instruction format for the **jump** instruction (**opcode = 2**)

The 32-bit target PC address of the **jump instruction** can be computed as follows:

- The **low-order 2 bits** are always 00_{two} , like one of the branch instruction.
- **The next lower 26 bits** come from the 26-bit immediate field in the instruction.
- The **upper 4 bits** are obtained by replacing the one of the current PC+4 (31:28).

A Simple Implementation Scheme

Implementing JUMP instruction

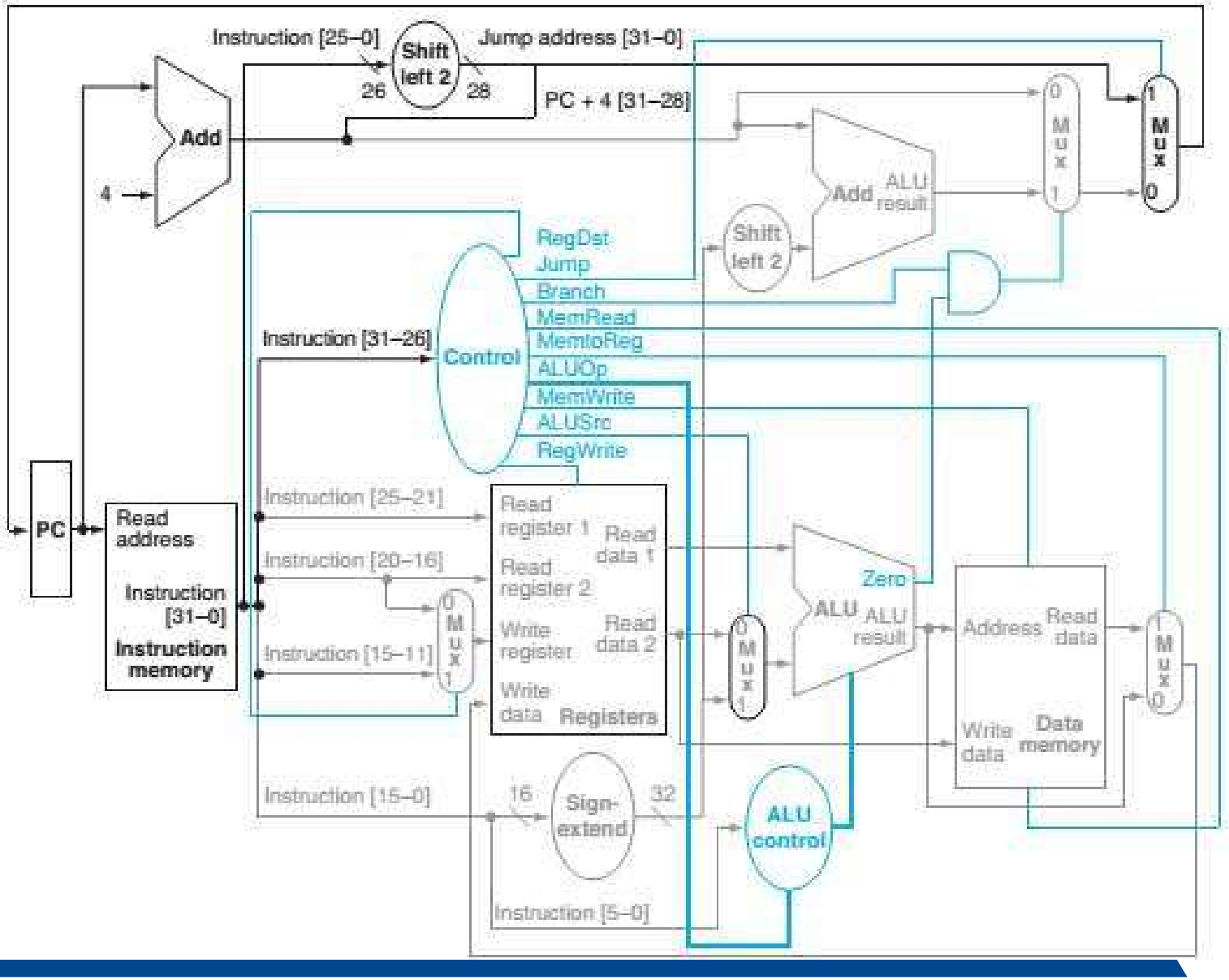


Fig.23 The simple control and datapath are extended to handle the **jump instruction**.

A Simple Implementation Scheme

Why a Single-Cycle Implementation is not used today?

In this single-cycle design:

The clock cycle must have the *same length for every instruction*

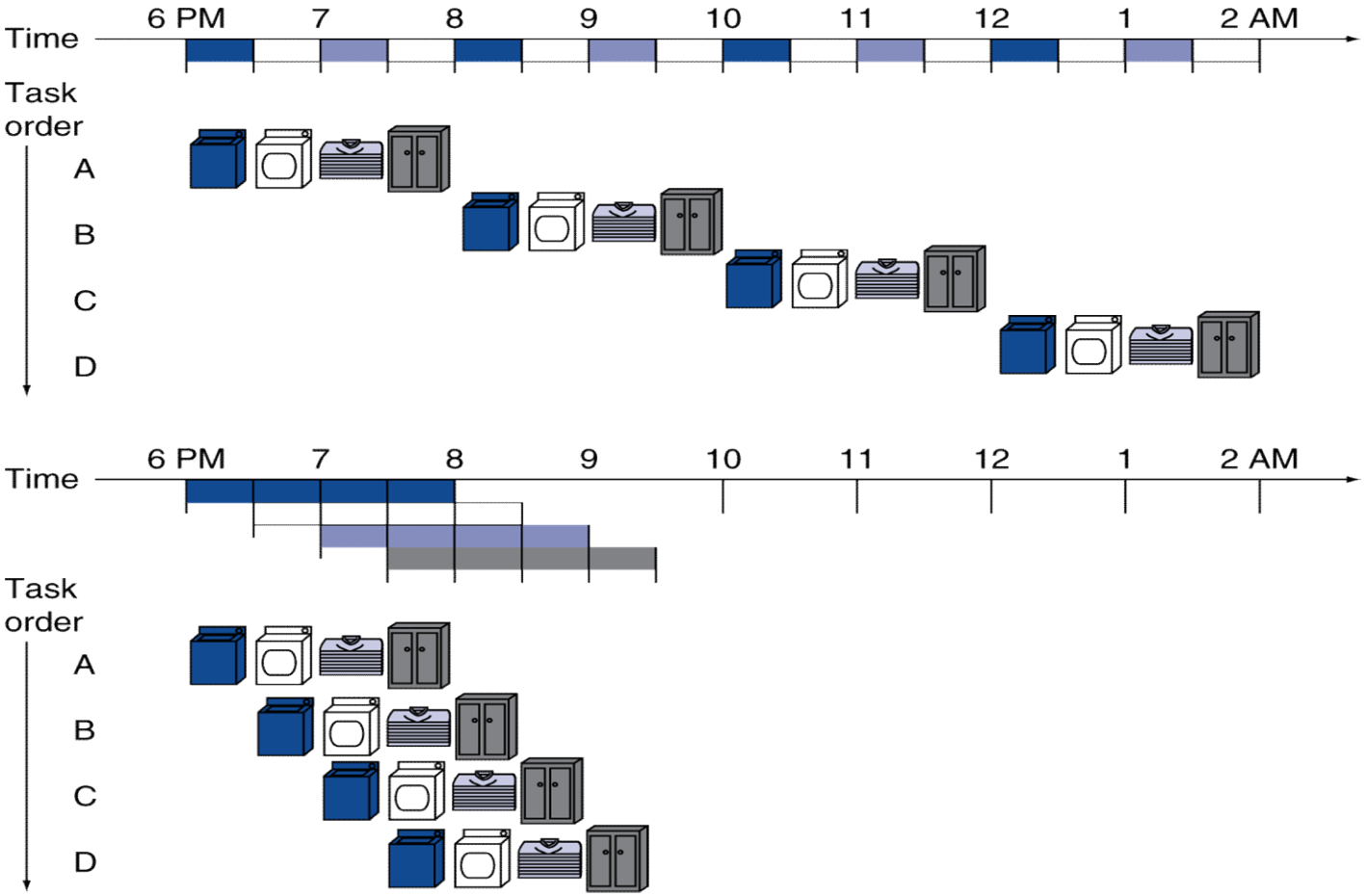
→ the clock cycle is determined by *the longest possible path in the processor* (load instruction – using 5 function units in series: the instruction memory, the register file, the ALU, the data memory, and the register file)

→ *Clock cycle is too long*, performance is poor.

Note: A single-cycle design might be considered acceptable for the small instruction set.

MIPS PIPELINE

Pipelined laundry: overlapping execution
Parallelism improves performance



- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

MIPS PIPELINE

The *nonpipelined* approach to laundry would be as follows:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

- When your roommate is done, start over with the next dirty load.

MIPS PIPELINE

pipelined approach takes much less time,

- As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load.
- When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and put the next dirty load into the washer.
- Next you have your roommate put the fi rst load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer.
- At this point all steps—called stages in pipelining—are operating concurrently.

MIPS PIPELINE

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

MIPS PIPELINE

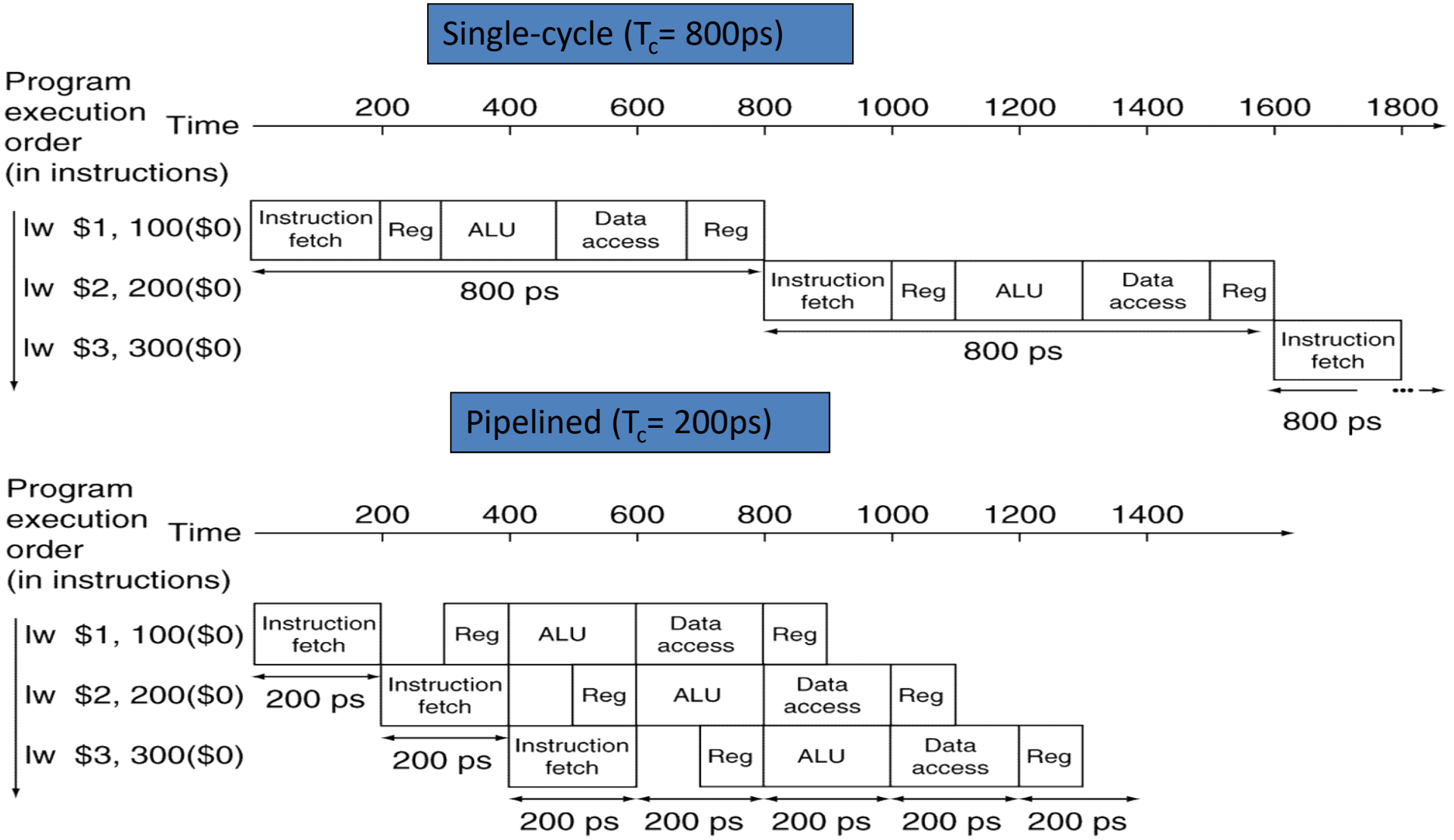
Assume time for stages is

- 100ps for register read or write
- 200ps for other stages

Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

MIPS PIPELINE



Pipeline Speedup

If all stages are balanced i.e., all take the same time

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipeline Hazards

Situations that prevent starting the next instruction in the next cycle

Structure hazards

A required resource is busy

Data hazard

Need to wait for previous instruction to complete its data read/write

Control hazard

Deciding on control action depends on previous instruction

Structure Hazards

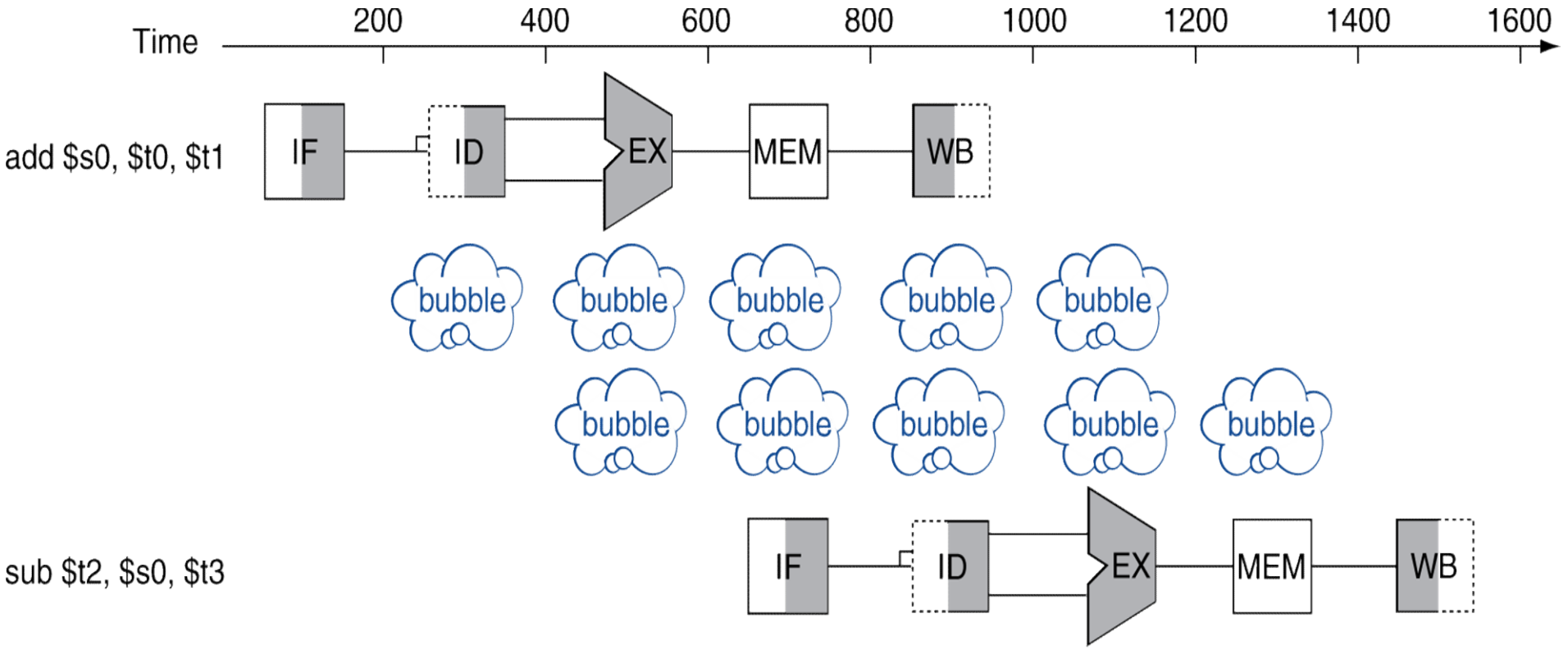
Conflict for use of a resource

- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined data paths require separate instruction/data memories Or separate instruction/data caches

Data Hazards

An instruction depends on completion of data access by a previous instruction

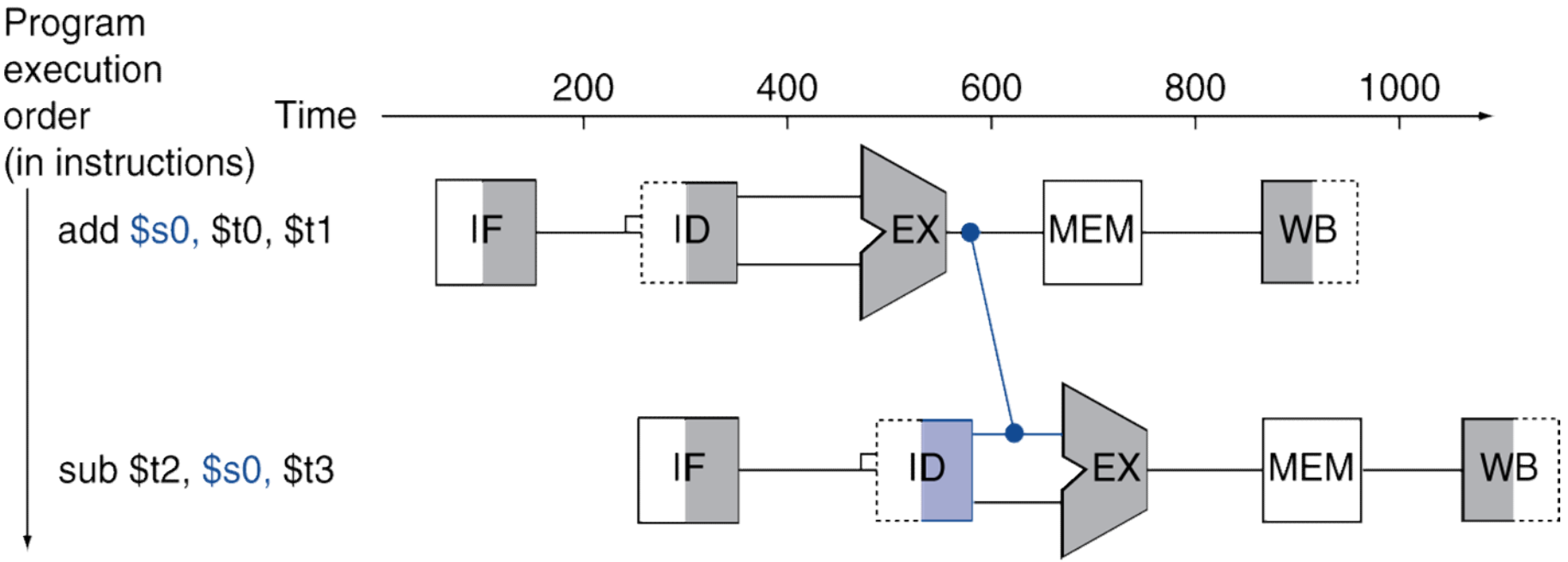
```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```



Forwarding (Bypassing)

Use result when it is computed

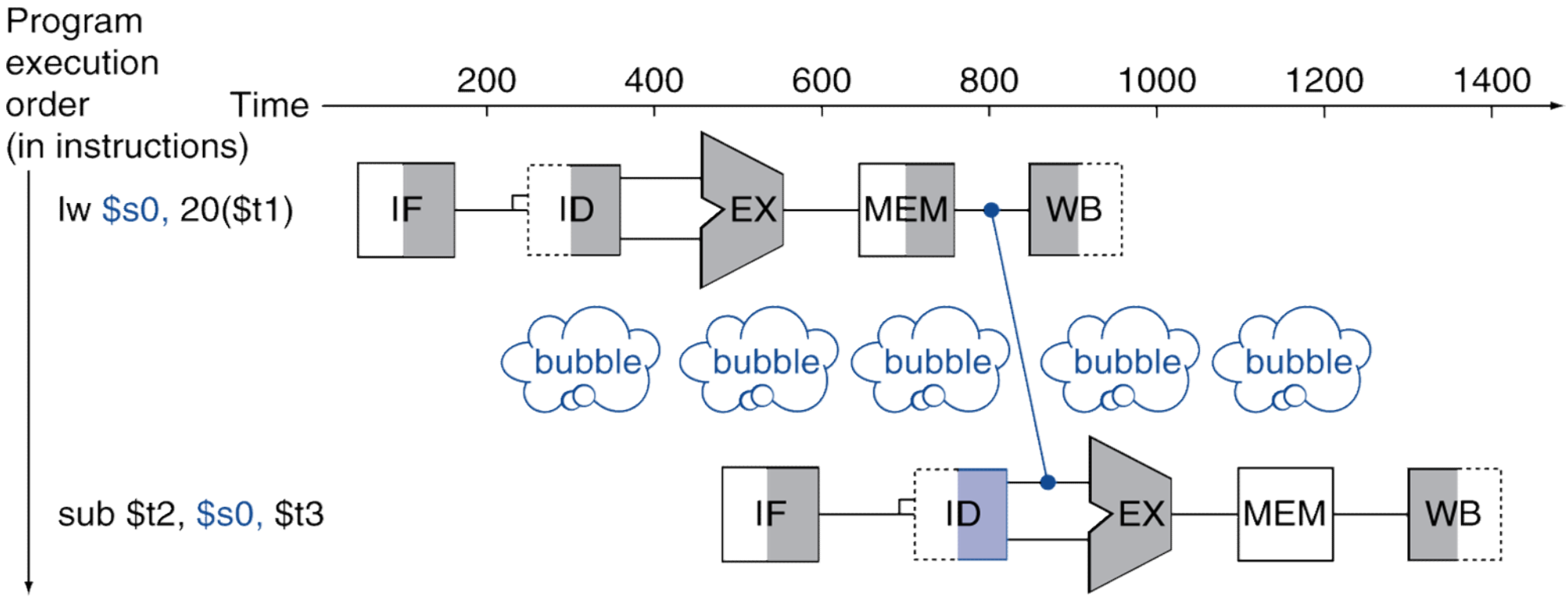
- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath



Load-Use Data Hazard

Can’t always avoid stalls by forwarding

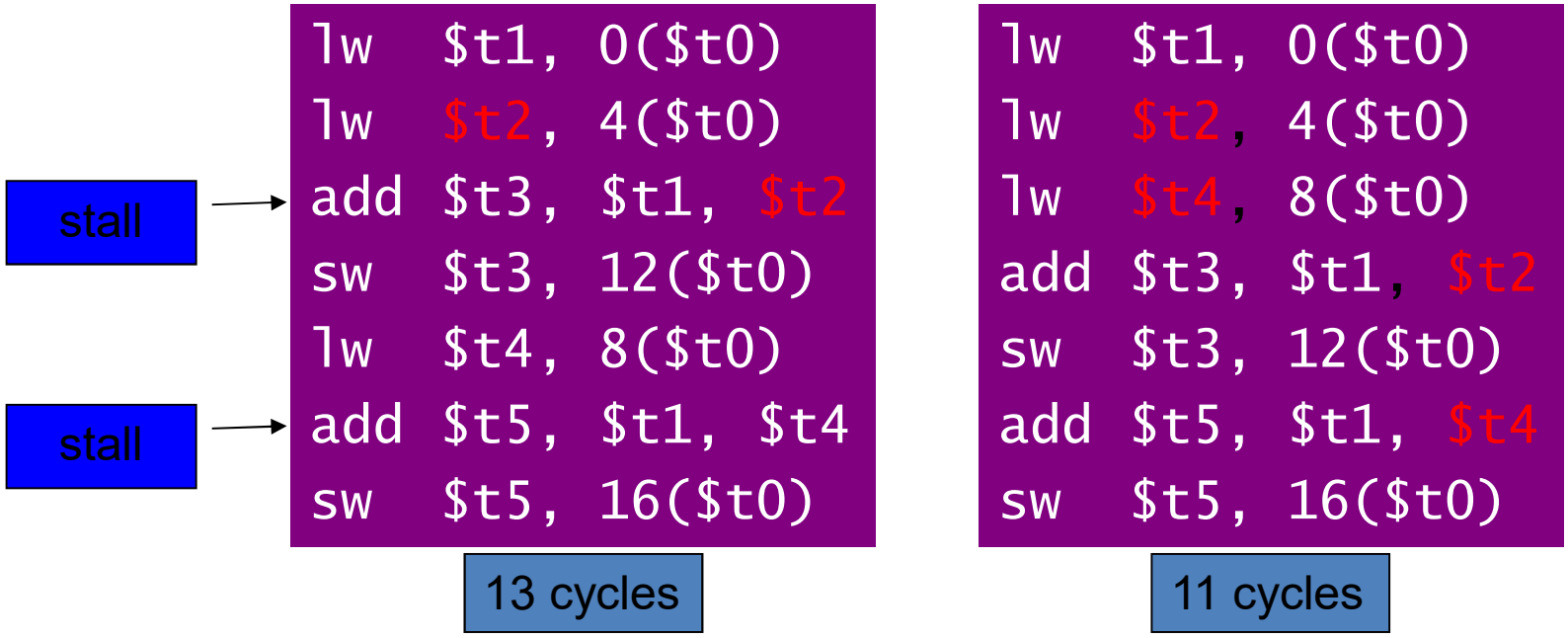
- If value not computed when needed
- Can’t forward backward in time!



Code Scheduling to Avoid Stalls

Reorder code to avoid use of load result in the next instruction

C code for `A = B + E; C = B + F;`



Control Hazards

Branch determines flow of control

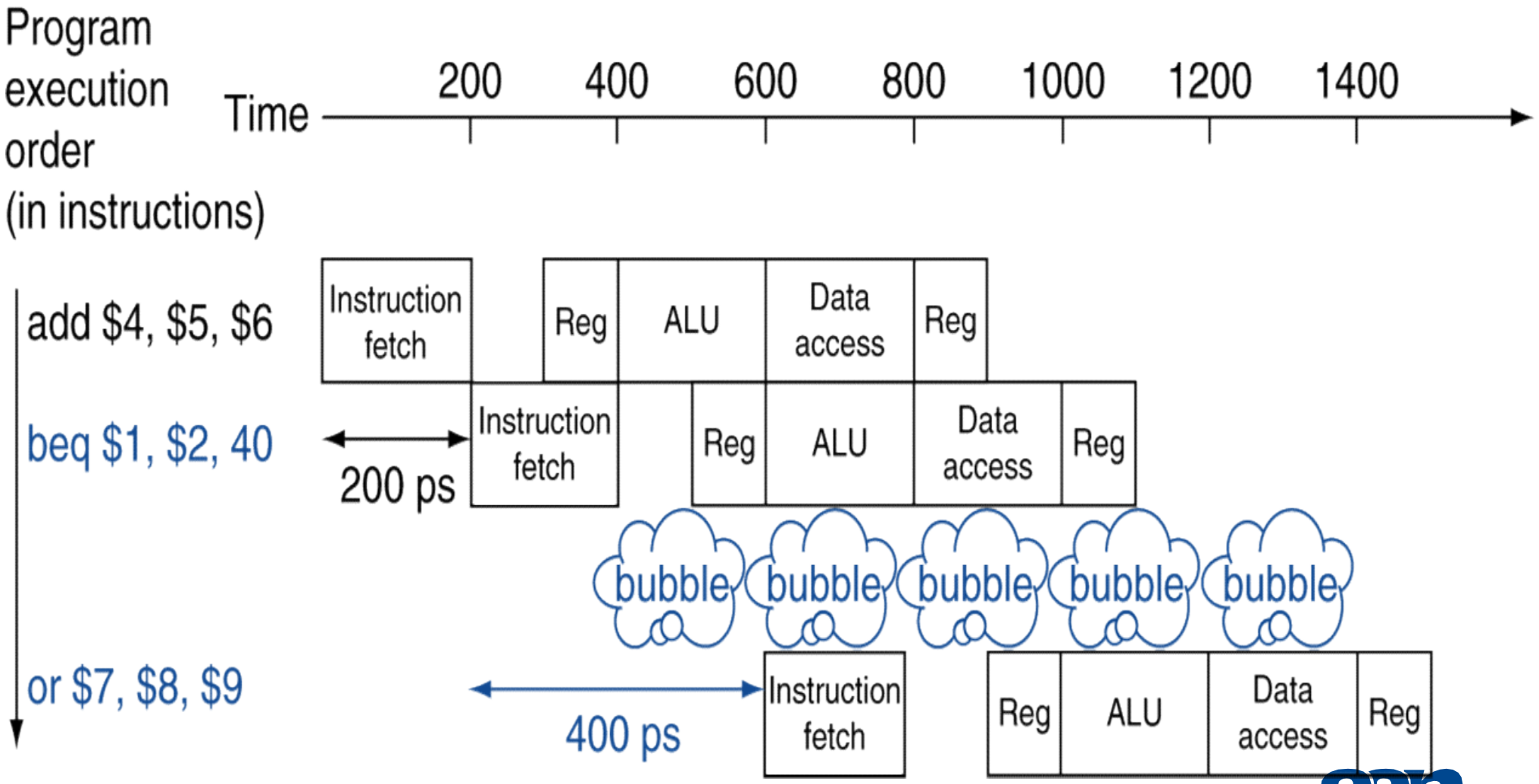
- Fetching next instruction depends on branch outcome
- Pipeline can't always fetch correct instruction
  Still working on ID stage of branch

In MIPS pipeline

- Need to compare registers and compute target early in the pipeline
- Add hardware to do it in ID stage

Stall on Branch

Wait until branch outcome determined before fetching next instruction

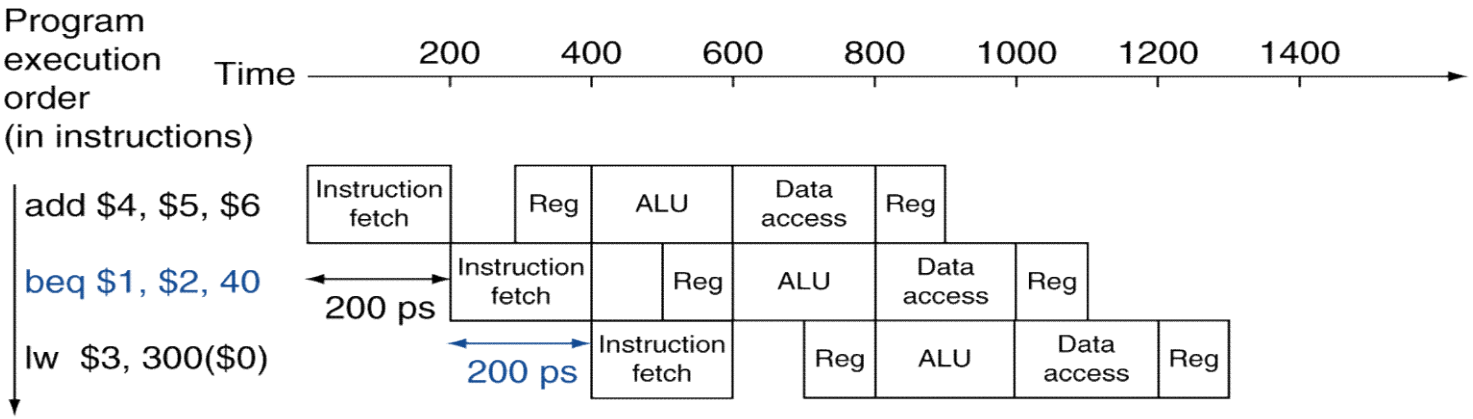


Branch Prediction

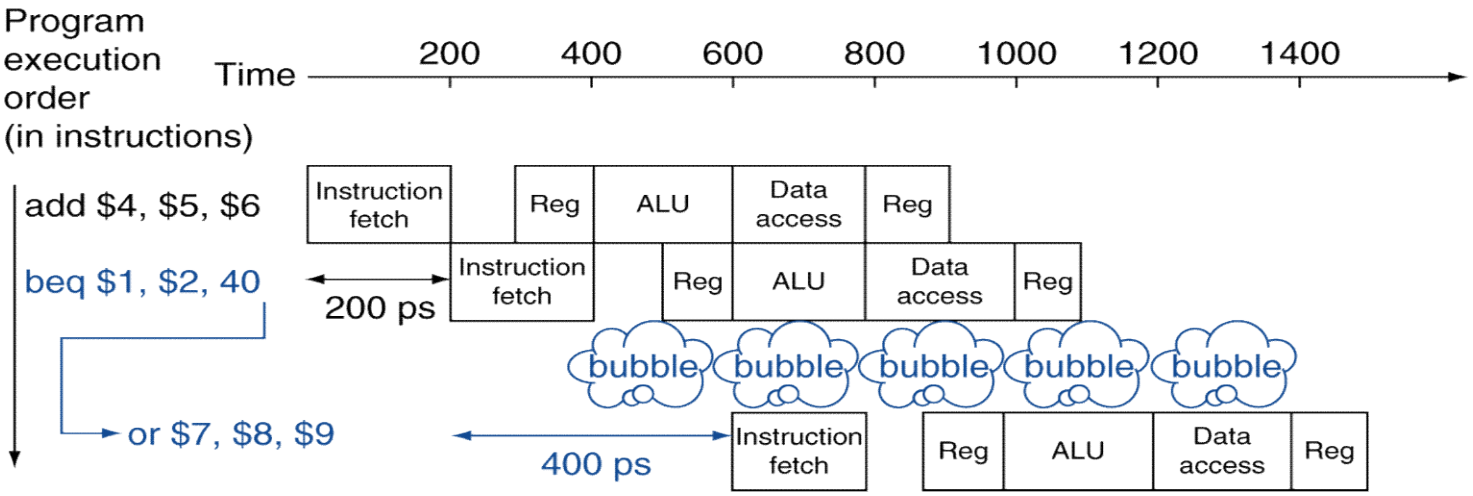
- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



References

- David A. Patterson and John L. Hennessey, “Computer Organization and Design‟, Fifth edition, Morgan Kauffman / Elsevier, 2014.
- V.Carl Hamacher, Zvonko G. Varanesic and Safat G. Zaky, “Computer Organisation“, VI edition, Mc Graw-Hill Inc, 2012.
- William Stallings “Computer Organization and Architecture”, Seventh Edition , Pearson Education, 2006.
- Vincent P. Heuring, Harry F. Jordan, “Computer System Architecture”, Second Edition, Pearson Education, 2005.

Thank you