IT16301 –
Computer Organization and Architecture

Prepared by

N.Uma & V.Ranjith

Assistant professor/IT

Course Outcomes

CO1 - Build the basic structure of computer, operations and instructions

CO2 - Design arithmetic and logic unit

CO3 - Discuss the pipelined execution and design control unit

CO4 - Evaluate performance of memory systems

CO5 - Construct the parallel processing architectures

Text Books:

- 1. M. Moris Mano, “ Computer System Architecture”, 3rd Edition, Pearson/ PHI, 2007
- 2. David A. Patterson and John L. Hennessey, “Computer organization and design‟, Morgan kauffman / elsevier, Fifth edition, 2014.

Unit 1

BASIC COMPUTER ORGANIZATION AND DESIGN

- Instruction codes,
- Computer registers,
- computer instructions,
- Timing and Control,
- Instruction cycle,
- Memory-Reference Instructions,
- Input-output and interrupt,
- Complete computer description,
- Design of Basic computer,
- design of Accumulator Unit

COMPUTER ORGANISATION AND ARCHITECTURE

- The components from which computers are built, i.e., computer organization.
- In contrast, computer architecture is the science of integrating those components to achieve a level of functionality and performance.
- It is as if computer organization examines the lumber, bricks, nails, and other building material
- While computer architecture looks at the design of the house.

Basic Terminology

- Input - Whatever is put into a computer system.
- Data - Refers to the symbols that represent facts, objects, or ideas.

Basic Terminology

- Assembly language program (ALP) –Programs are written using mnemonics
- Mnemonic –Instruction will be in the form of English like form
- Assembler –is a software which converts ALP to MLL (Machine Level Language)

Basic Terminology

- Interpreter – Converts HLL to MLL, does this job statement by statement

- System software – Program routines which aid the user in the execution of programs eg: Assemblers, Compilers

- Operating system – Collection of routines responsible for controlling and coordinating all

Computers have two kinds of components:

- *Hardware,*
  - *consisting of its physical devices (CPU, memory, bus, storage devices, ...)*
- *Software,*
  - *consisting of the programs it has (Operating system, applications, utilities, ...)*

All computer functions are:

- Data PROCESSING
- Data STORAGE
- Data MOVEMENT
  - Data = Information

- CONTROL--- Coordinates How Information is Used

- **INPUT UNIT:**

•Converts the external world data to a binary format, which can be understood by CPU

– Mouse, Joystick etc

- **OUTPUT UNIT:**

•Converts the binary format data to a format that a common man can understand

CPU

- **The Brain** of the machine
- carrying out computational task
- ALU, CU, Registers
- ALU Arithmetic and logical operations

MEMORY

- Stores data, results, programs
- Two classes of storage

(i) Primary

(ii) Secondary

(•) Two types are RAM or R/W memory and ROM read only memory

(•) ROM is used to store data and program which is not going to change.

Instruction Codes

- The Internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers

- The user of a computer can control the process by means of a program

- A *program* is a set of instructions that specify the operations, operands, and the processing sequence

Instruction Codes

- A *computer instruction* is a binary code that specifies a sequence of micro-operations for the computer. Each computer has its unique instruction set

- Instruction codes and data are stored in memory

- The computer reads each instruction from memory and places it in a control register

- The control unit interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of micro-operations

Instruction Codes

- An Instruction code is a group of bits that instructs the computer to perform a specific operation (sequence of microoperations). It is divided into parts (basic part is the operation part)

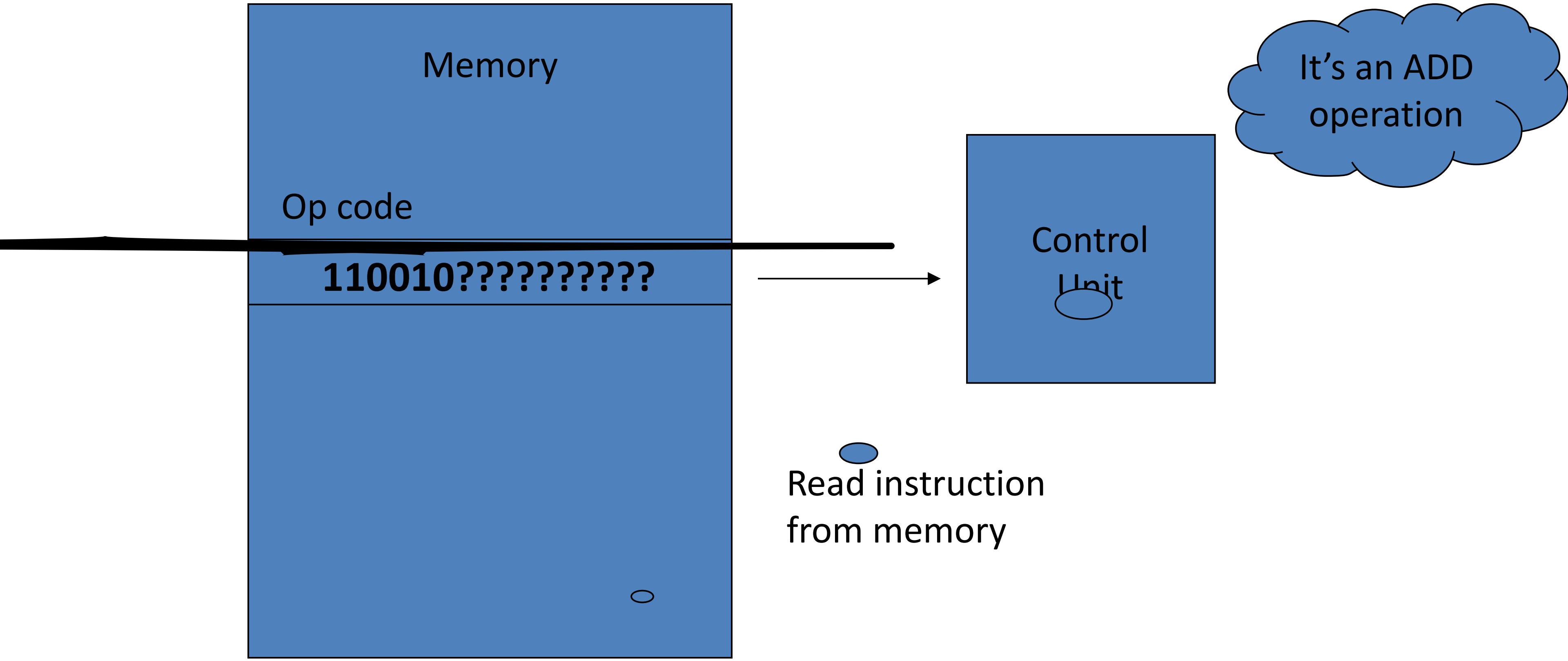
- The operation code of an instruction is a group of bits that defines certain operations such as add, subtract, shift, and complement

Instruction Codes

- The number of bits required for the operation code depends on the total number of operations available in the computer

- 2n (or little less) distinct operations → n bit operation code

Instruction Codes



Memory

Op code

110010??????????

Control Unit

It's an ADD operation

Read instruction from memory

Instruction Codes

- An operation must be performed on some data stored in processor registers or in memory

- An instruction code must therefore specify not only the operation, but also the location of the operands (in registers or in the memory), and where the result will be stored (registers/memory)

Instruction Codes

- Memory words can be specified in instruction codes by their address

- Processor registers can be specified by assigning to the instruction another binary code of k bits that specifies one of 2k registers

- Each computer has its own particular instruction code format

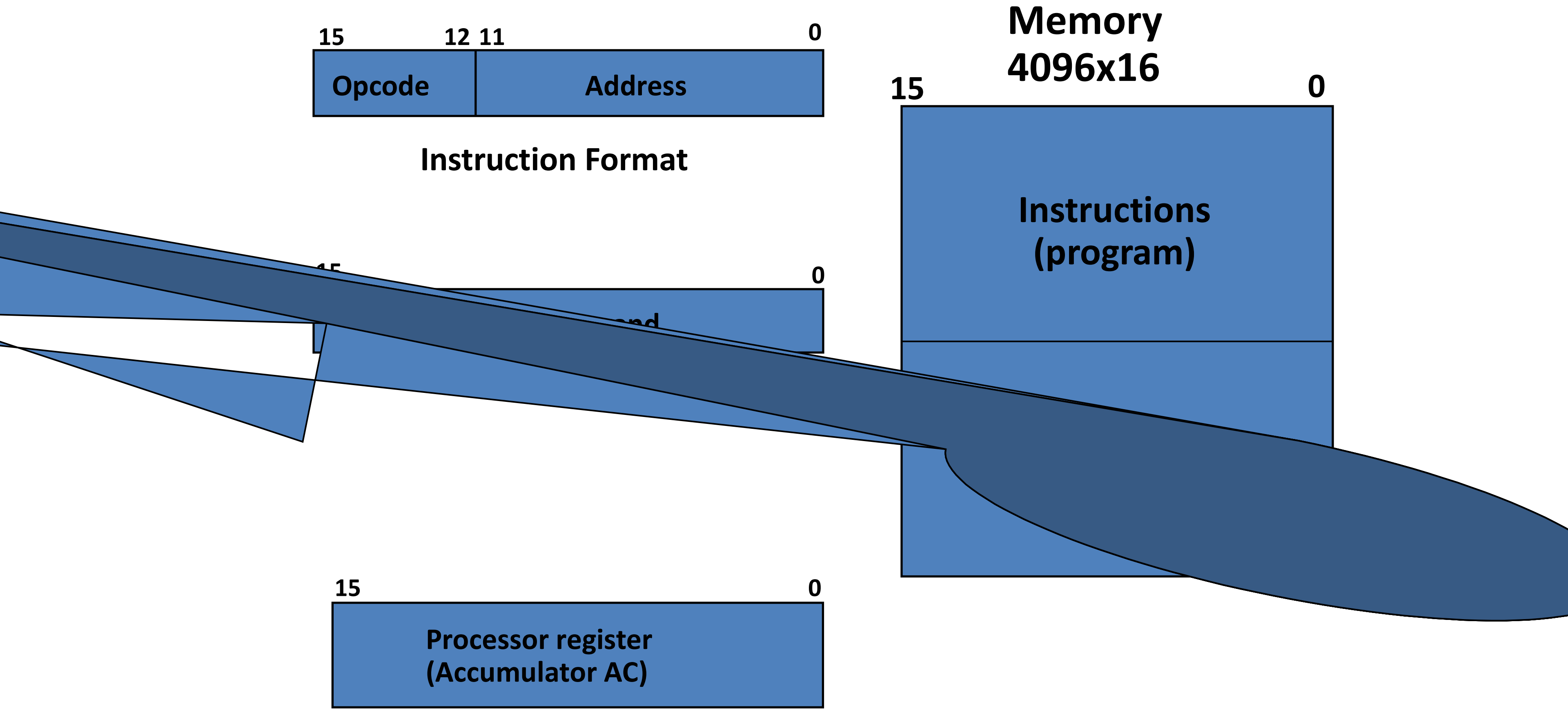
- Instruction code formats are conceived by computer designers who specify the architecture of the computer

Instruction Codes
Stored Program Organization

- An instruction code is usually divided into *operation code*, *operand address*, *addressing mode*, etc.

- The simplest way to organize a computer is to have one processor register (accumulator AC) and an instruction code format with two parts (op code, address)

Instruction Codes
Stored Program Organization



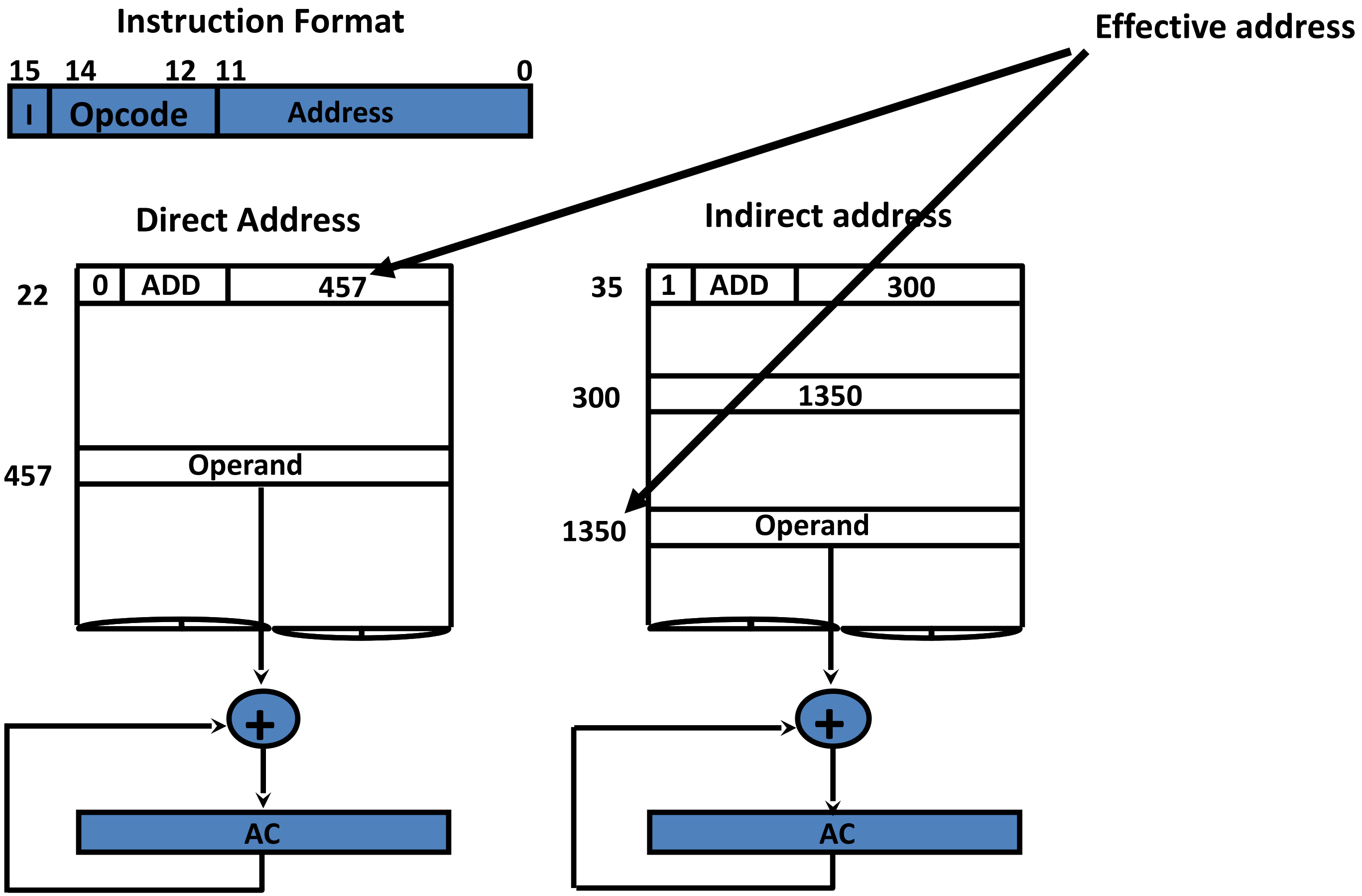
Instruction Format

Memory 4096x16

Instruction Codes
Indirect Address

- There are three **Addressing Modes** used for address portion of the instruction code:
  - Immediate: the operand is given in the address portion (constant)
  - Direct: the address points to the operand stored in the memory
  - Indirect: the address points to the pointer (another address) stored in the memory that references the operand in memory
- One bit of the instruction code can be used to distinguish between direct & indirect addresses

Instruction Codes
Indirect Address



Instruction Format

15 14 12 11 0

I Opcode Address

Effective address

Direct Address

22 0 ADD 457

457 Operand

AC

Indirect address

35 1 ADD 300

300 1350

1350 Operand

AC

Instruction Codes - Indirect Address

- Effective address: the address of the operand in a computation-type instruction or the target address in a branch-type instruction

- The pointer can be placed in a processor register instead of memory as done in commercial computers

Computer Registers

- Computer instructions are normally stored in consecutive memory locations and executed sequentially one at a time

- The control reads an instruction from a specific address in memory and executes it, and so on

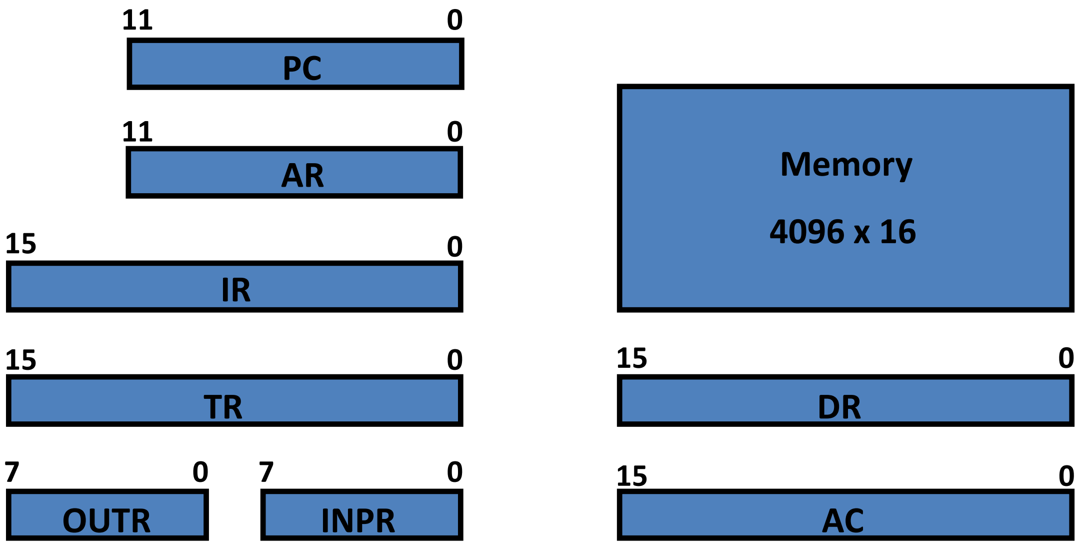
- This type of sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed

Computer Registers

- It is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory

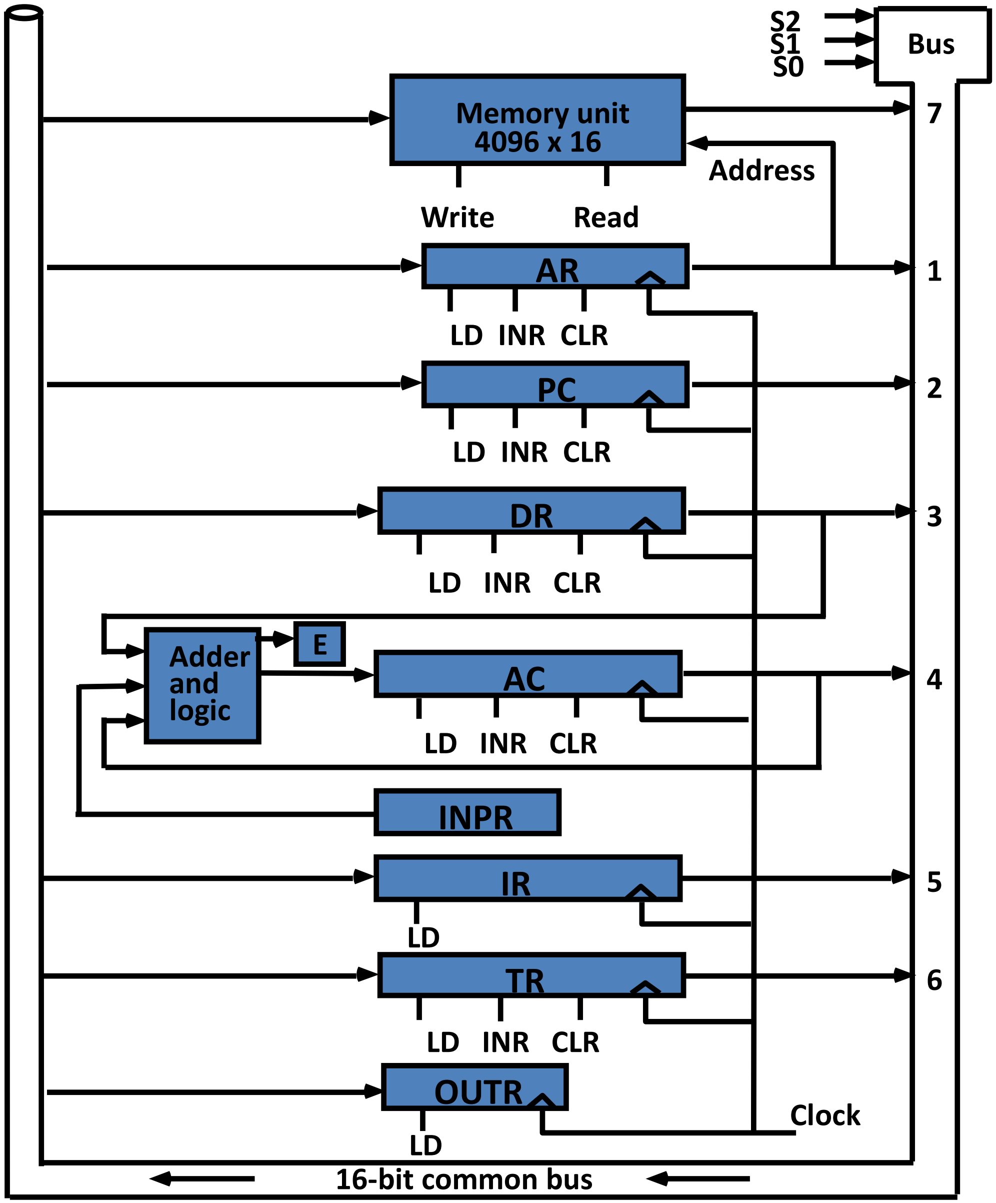
- The computer needs processor registers for manipulating data and a register for holding a memory address

Registers in the Basic Computer



List of BC Registers

DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character



Computer Registers
Common Bus System

Computer Registers- Common Bus System

- S2S1S0: Selects the register/memory that would use the bus
- LD (load): When enabled, the particular register receives the data from the bus during the next clock pulse transition
- E (extended AC bit): flip-flop holds the carry
- DR, AC, IR, and TR: have 16 bits each
- AR and PC: have 12 bits each since they hold a memory address

Computer Registers-Common Bus System

- When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to zeros

- When AR or PC receives information from the bus, only the 12 least significant bits are transferred into the register

- INPR and OUTR: communicate with the eight least significant bits in the bus

Computer Registers-Common Bus System

- INPR: Receives a character from the input device (keyboard,…etc) which is then transferred to AC

- OUTR: Receives a character from AC and delivers it to an output device (say a Monitor)

- Five registers have three control inputs: LD (load), INR (increment), and CLR (clear)

- Register ≡ binary counter with parallel load and

Computer Registers-Memory Address

- The input data and output data of the memory are connected to the common bus

- But the memory address is connected to AR

- Therefore, AR must always be used to specify a memory address

- By using a single register for the address, we eliminate the need for an address bus that would have

Computer Registers- Memory Address

- Register → Memory: Write operation
- Memory → Register: Read operation (note that AC cannot directly read from memory!!)
- Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle

Computer Registers-Memory Address

- The transition at the end of the cycle transfers the content of the bus into the destination register, and the output of the adder and logic circuit into the AC

- For example, the two microoperations

  DR←AC and AC←DR (Exchange)

can be executed at the same time

Computer Registers
Memory Address

- 1- place the contents of AC on the bus (S2S1S0=100)

- 2- enabling the LD (load) input of DR

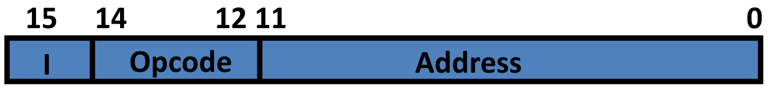
- 3- Transferring the contents of the DR through the adder and logic circuit into AC

- 4- enabling the LD (load) input of AC

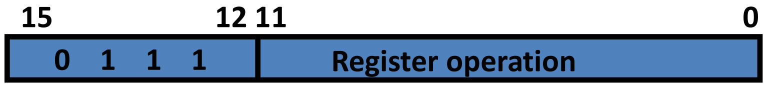
Computer Instructions

Basic Computer Instruction code format

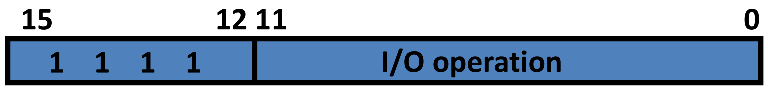
Memory-Reference Instructions (OP-code = 000 ~ 110)



Register-Reference Instructions (OP-code = 111, I = 0)



Input-Output Instructions (OP-code =111, I = 1)



BASIC COMPUTER INSTRUCTIONS

	Hex Code		
Symbol	*I = 0*	*I = 1*	*Description*
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Computer Instructions
Instruction Set Completeness

- The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:
  - Arithmetic, logical, and shift instructions
  - Instructions for moving information to and from memory and processor registers

Timing & Control

- The timing for all registers in the basic computer is controlled by a master clock generator

- The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit

- The clock pulses do not change the state of a register unless the register is enabled by a

Timing & Control

- The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator

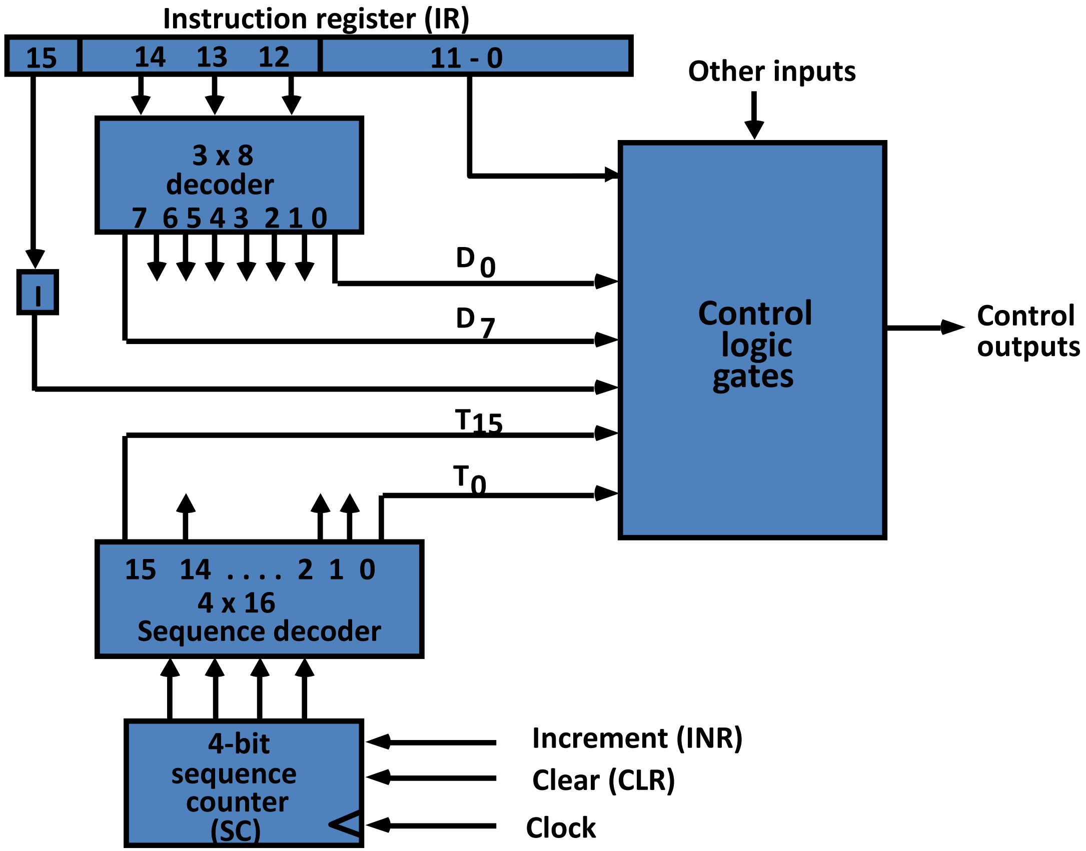
- There are two major types of control organization:
  - Hardwired control
  - Microprogrammed control

Timing & Control

- In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits.

- In the microprogrammed organization, the control information is stored in a control memory (if the design is modified, the microprogram in control memory has to be updated)
- D3T4: SC←0

The Control Unit for the basic computer



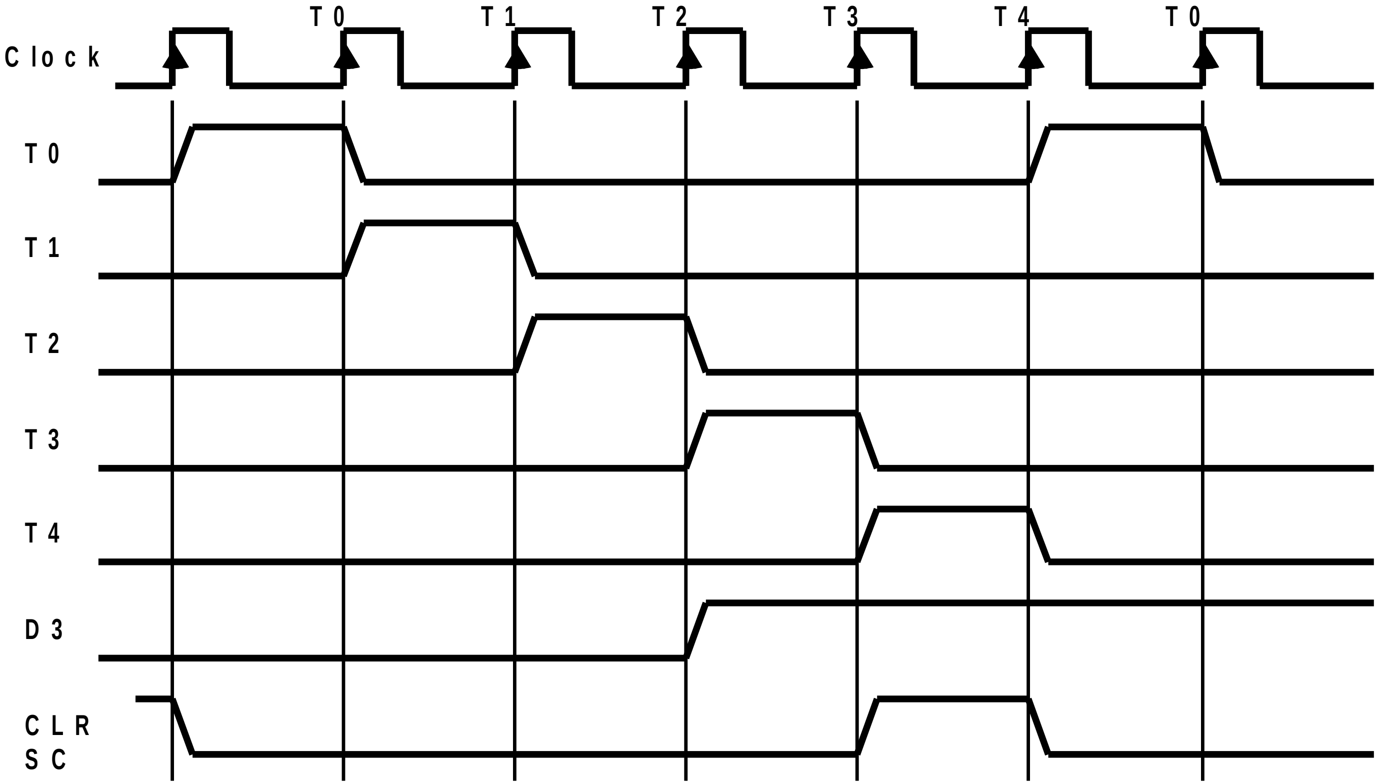
Hardwired Control Organization

- Generated by 4-bit sequence counter and 4x16 decoder
- The SC can be incremented or cleared.

- Example: T0, T1, T2, T3, T4, T0, T1, . . .

  Assume: At time T4, SC is cleared to 0 if decoder output D3 is active.

D3T4: SC ← 0



Timing & Control

- A memory read or write cycle will be initiated with the rising edge of a timing signal

- Assume: memory cycle time < clock cycle time!

- So, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive edge

- The clock transition will then be used to load the

Timing & Control

- T0: AR←PC
  - Transfers the content of PC into AR if timing signal T0 is active
  - T0 is active during an entire clock cycle interval
  - During this time, the content of PC is placed onto the bus (with S2S1S0=010) and the LD (load) input of AR is enabled
  - The actual transfer does not occur until the end of the clock

Instruction Cycle

- A program is a sequence of instructions stored in memory

- The program is executed in the computer by going through a cycle for each instruction (in most cases)

- Each instruction in turn is subdivided into a sequence of sub-cycles or phases

Instruction Cycle

- Instruction Cycle Phases:
  - 1- Fetch an instruction from memory
  - 2- Decode the instruction
  - 3- Read the effective address from memory if the instruction has an indirect address
  - 4- Execute the instruction
- This cycle repeats indefinitely unless a HALT

Instruction Cycle
Fetch and Decode

- Initially, the Program Counter (PC) is loaded with the address of the first instruction in the program

- The sequence counter SC is cleared to 0, providing a decoded timing signal T0

- After each clock pulse, SC is incremented by one, so that the timing signals go through a

Instruction Cycle Fetch and Decode

- T0: AR←PC (this is essential!!)

The address of the instruction is moved to AR.

- T1: IR←M[AR], PC←PC+1

The instruction is fetched from the memory to IR ,

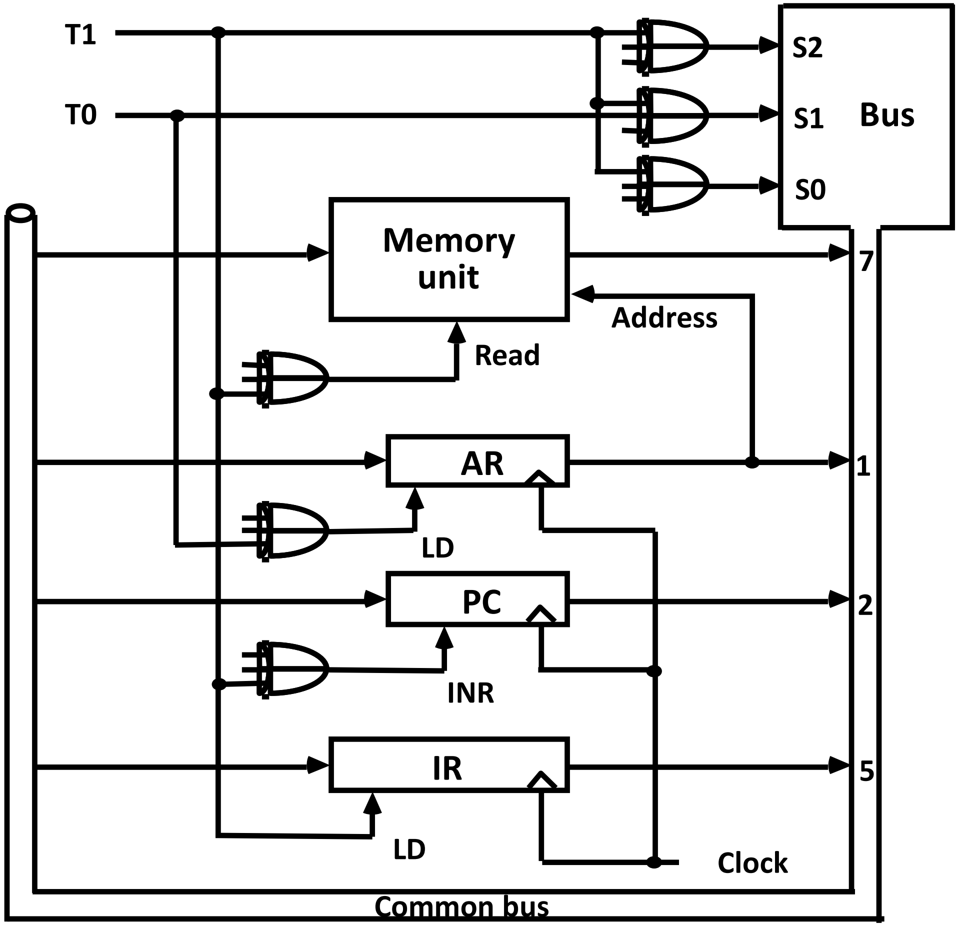
and the PC is incremented.

- T2: D0,…, D7←Decode IR(12-14), AR←IR(0-11), I←IR(15)

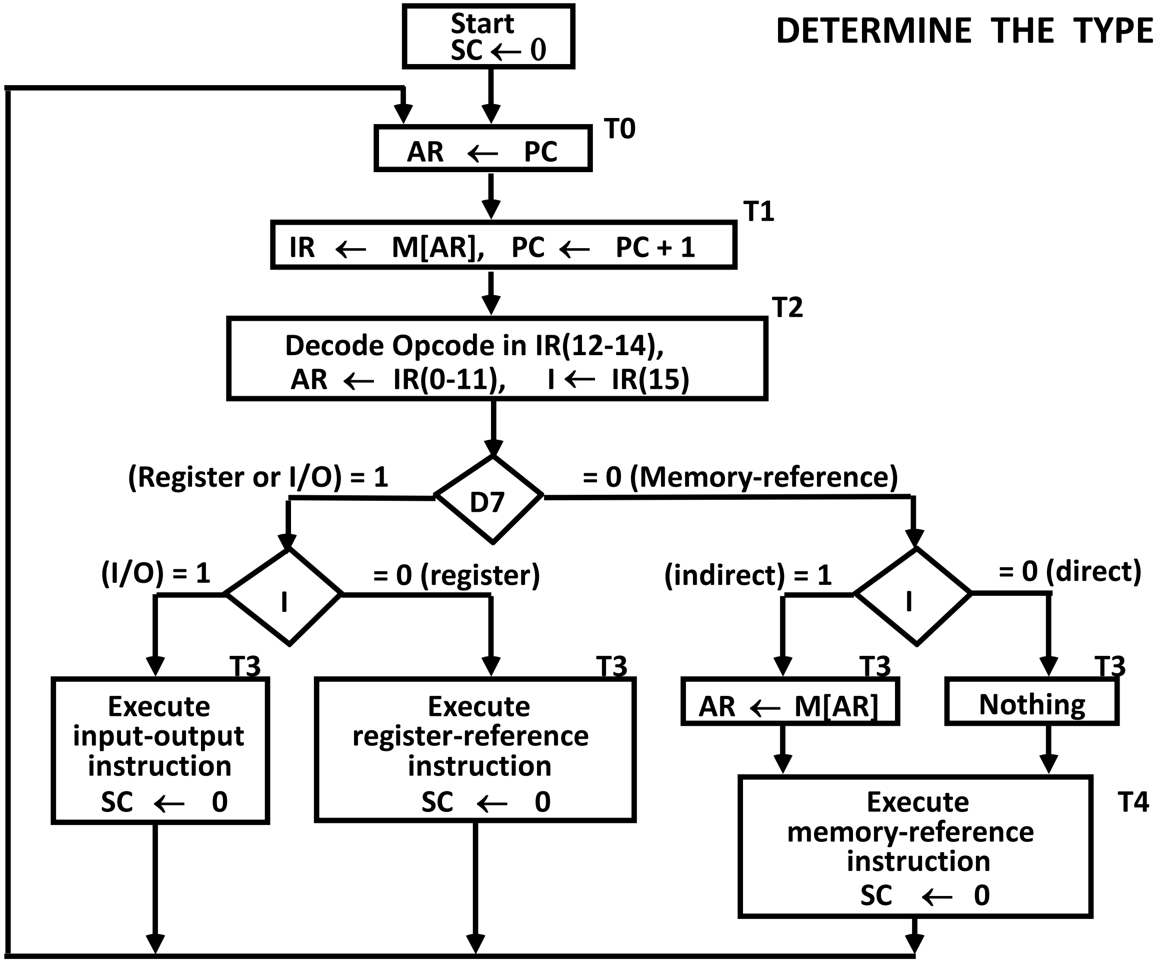
BC Instruction cycle: [Fetch Decode [Indirect] Execute]*

- **Fetch and Decode**

T0: AR ← PC (S0S1S2=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1   (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)



DETERMINE THE TYPE OF INSTRUCTION



Start
SC ← 0

T0
AR ← PC

T1
IR ← M[AR], PC ← PC + 1

T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)

D7

(Register or I/O) = 1

= 0 (Memory-reference)

I

(I/O) = 1

= 0 (register)

I

(indirect) = 1

= 0 (direct)

T3
Execute input-output instruction
SC ← 0

T3
Execute register-reference instruction
SC ← 0

T3
AR ← M[AR]

T3
Nothing

T4
Execute memory-reference instruction
SC ← 0

D'_7IT_3: AR ← M[AR]

$D'_7I'T_3$: Nothing

$D_7I'T_3$: Execute a register-reference instr.

D_7IT_3: Execute an input-output instr.

REGISTER REFERENCE INSTRUCTIONS

Register Reference Instructions are identified when

- **D7 = 1, I = 0**
- **Register Ref. Instr. is specified in B0 ~ B11 of IR**
- **Execution starts with timing signal T3**

r = D7 I' T3 => Register Reference Instruction
Bi = IR(i) , i=0,1,2,...,11, the ith bit of IR.

	r:	**SC ← 0**
CLA	**rB11:**	**AC ← 0**
CLE	**rB10:**	**E ← 0**
CMA	**rB9:**	**AC ← AC'**
CME	**rB8:**	**E ← E'**
CIR	**rB7:**	**AC ← shr AC, AC(15) ← E, E ← AC(0)**
CIL	**rB6:**	**AC ← shl AC, AC(0) ← E, E ← AC(15)**
INC	**rB5:**	**AC ← AC + 1**
SPA	**rB4:**	**if (AC(15) = 0) then (PC ← PC+1)**
SNA	**rB3:**	**if (AC(15) = 1) then (PC ← PC+1)**
SZA	**rB2:**	**if (AC = 0) then (PC ← PC+1)**
SZE	**rB1:**	**if (E = 0) then (PC ← PC+1)**
HLT	**rB0:**	**S ← 0 (S is a start-stop flip-flop)**

5.6 MEMORY REFERENCE INSTRUCTIONS

Symbol	Operation Decoder	Symbolic Description
	AND D0	$AC \leftarrow AC \wedge M[AR]$
	ADD D1	$AC \leftarrow AC + M[AR], E \leftarrow Cout$
	LDA D2	$AC \leftarrow M[AR]$
	STA D3	$M[AR] \leftarrow AC$
	BUN	D4 $PC \leftarrow AR$
	BSA D5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
	ISZ D6	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC+1$

- The effective address of the instruction is in AR and was placed there during timing signal T2 when I = 0, or during timing signal T3 when I = 1
- Memory cycle is assumed to be short enough to be completed in a CPU cycle
- The execution of MR Instruction starts with T4

AND to AC

D0T4: $DR \leftarrow M[AR]$ Read operand

D0T5: $AC \leftarrow AC \wedge DR, SC \leftarrow 0$ AND with AC

ADD to AC

D1T4: $DR \leftarrow M[AR]$ Read operand

D1T5: $AC \leftarrow AC + DR, E \leftarrow Cout, SC \leftarrow 0$ Add to AC and store carry in E

MEMORY REFERENCE INSTRUCTIONS

LDA: Load to AC

D2T4: DR ← M[AR]

D2T5: AC ← DR, SC ← 0

STA: Store AC

D3T4: M[AR] ← AC, SC ← 0

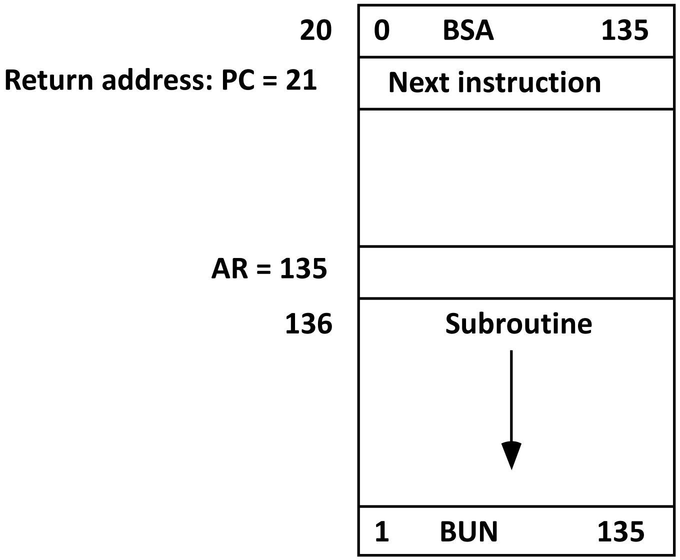
BUN: Branch Unconditionally

D4T4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

M[AR] ← PC, PC ← AR + 1

Memory, PC, AR at time T4



20 | 0 BSA 135

Return address: PC = 21 | Next instruction

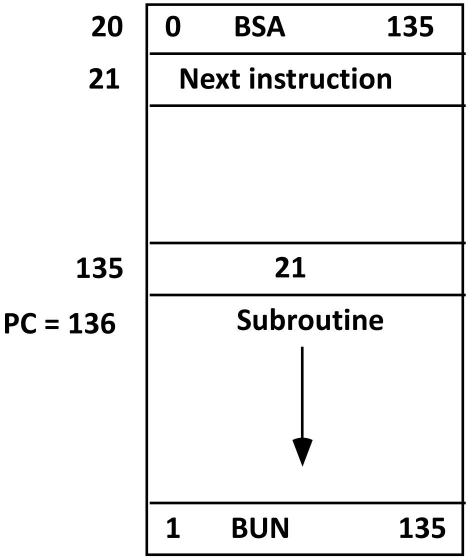
AR = 135 |

136 | Subroutine

1 BUN 135

Memory

Memory, PC after execution



20 | 0 BSA 135

21 | Next instruction

135 | 21

PC = 136 | Subroutine

1 BUN 135

Memory

Memory Reference Instructions

BSA: executed in a sequence of two micro-operations:

D5T4: M[AR] ← PC, AR ← AR + 1

D5T5: PC ← AR, SC ← 0

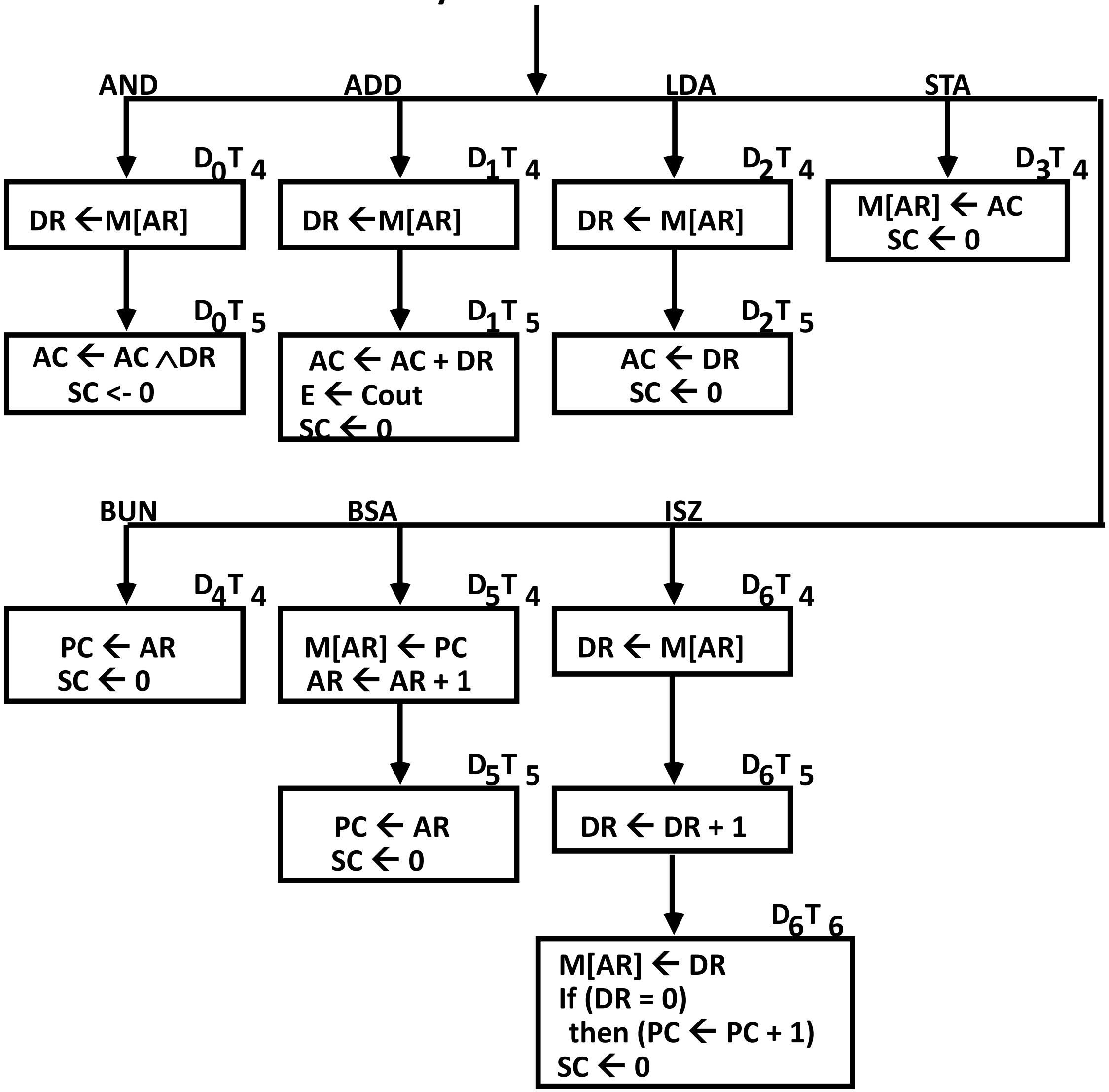
ISZ: Increment and Skip-if-Zero

D6T4: DR ← M[AR]

D6T5: DR ← DR + 1

D6T6: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

Memory-reference instruction



AND

D_0T_4 DR ← M[AR]

D_0T_5 AC ← AC ∧ DR
SC <- 0

ADD

D_1T_4 DR ← M[AR]

D_1T_5 AC ← AC + DR
E ← Cout
SC ← 0

LDA

D_2T_4 DR ← M[AR]

D_2T_5 AC ← DR
SC ← 0

STA

D_3T_4 M[AR] ← AC
SC ← 0

BUN

D_4T_4 PC ← AR
SC ← 0

BSA

D_5T_4 M[AR] ← PC
AR ← AR + 1

D_5T_5 PC ← AR
SC ← 0

ISZ

D_6T_4 DR ← M[AR]

D_6T_5 DR ← DR + 1

D_6T_6 M[AR] ← DR
If (DR = 0)
then (PC ← PC + 1)
SC ← 0

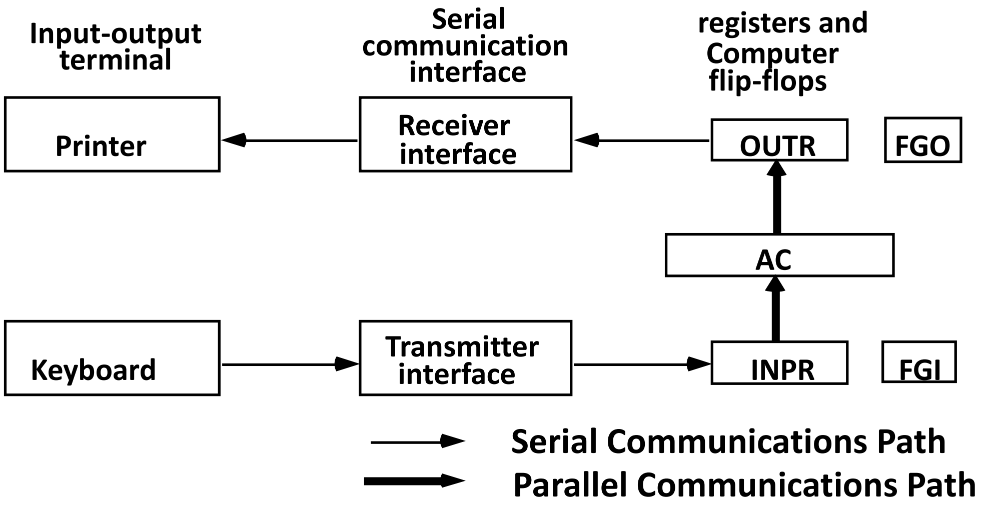
Input-Output and Interrupt

- Instructions and data stored in memory must come from some input device

- Computational results must be transmitted to the user through some output device

- For the system to communicate with an input device, serial information is shifted into the input register INPR

Input-Output and Interrupt



Input-output terminal

Serial communication interface

registers and Computer flip-flops

Printer

Receiver interface

OUTR

FGO

AC

Keyboard

Transmitter interface

INPR

FGI

Serial Communications Path

Parallel Communications Path

Input-Output and Interrupt

- INPR and OUTR communicate with a communication interface serially and with the AC in parallel. They hold an 8-bit alphanumeric information

- I/O devices are slower than a computer system → we need to synchronize the timing rate difference between the input/output device and the computer.

- FGI: 1-bit input flag (Flip-Flop)

Input-Output and Interrupt

- FGI is set to 1 when a new information is available in the input device and is cleared to 0 when the information is accepted by the computer

- FGO: 1-bit output flag used as a control flip-flop to control the output operation

- If FGO is set to 1, then this means that the computer can send out the information from

Input-Output and Interrupt

- The process of input information transfer:
  - Initially, FGI is cleared to 0
  - An 8-bit alphanumeric code is shifted into INPR (Keyboard key strike) and the input flag FGI is set to 1
  - As long as the flag is set, the information in INPR cannot be changed by another data entry

Input-Output and Interrupt

- Once the flag is cleared, new information can be shifted into INPR by the input device (striking another key)

- The process of outputting information:
  - Initially, the output flag FGO is set to 1
  - The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0

Input-Output and Interrupt

- When the operation is completed, the output device sets FGO back to 1

- The computer does not load a new data information into OUTR when FGO is 0 because this condition indicates that the output device is busy to receive another information at the moment!!

Input-Output Instructions

- Needed for:
  - Transferring information to and from AC register
  - Checking the flag bits
  - Controlling the interrupt facility

- The control unit recognize it when D7=1 and I = 1

- The remaining bits of the instruction specify the particular operation

Input-Output Instructions

D7IT3 = p
IR(i) = Bi, i = 6, …, 11

INP	pB11:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB10:	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	pB9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB7:	IEN ← 1	Interrupt enable on
IOF	pB6:	IEN ← 0	Interrupt enable off

Program Interrupt

- The process of communication just described is referred to as ***Programmed Control Transfer***

- The computer keeps checking the flag bit, and when it finds it set, it initiates an information transform (this is sometimes called ***Polling***)

- This type of transfer is in-efficient due to the difference of information flow rate between the computer and the I/O device

Program Interrupt

- The computer is wasting time while checking the flag instead of doing some other useful processing task

- An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer

- This type of transfer uses the interrupt facility

Program Interrupt

- While the computer is running a program, it does not check the flags

- Instead:
  - When a flag is set, the computer is immediately interrupted from proceeding with the current program

  - The computer stops what it is doing to take care of the input or output transfer

  - Then, it returns to the current program to continue what it

Program Interrupt

- The interrupt facility can be enabled or disabled via a flip-flop called IEN

- The interrupt enable flip-flop IEN can be set and cleared with two instructions (IOF, ION):
  - IOF: IEN ← 0 (the computer cannot be interrupted)
  - ION: IEN ← 1 (the computer can be interrupted)

Program Interrupt

- Another flip-flop (called the interrupt flip-flop **R**) is used in the computer's interrupt facility to decide when to go through the interrupt cycle

- **FGI** and **FGO** are different here compared to the way they acted in an earlier discussion!!

Program Interrupt

- The interrupt cycle is a hardware implementation of a branch and save return address operation (BSA)

- The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted

- This location may be a processor register, a

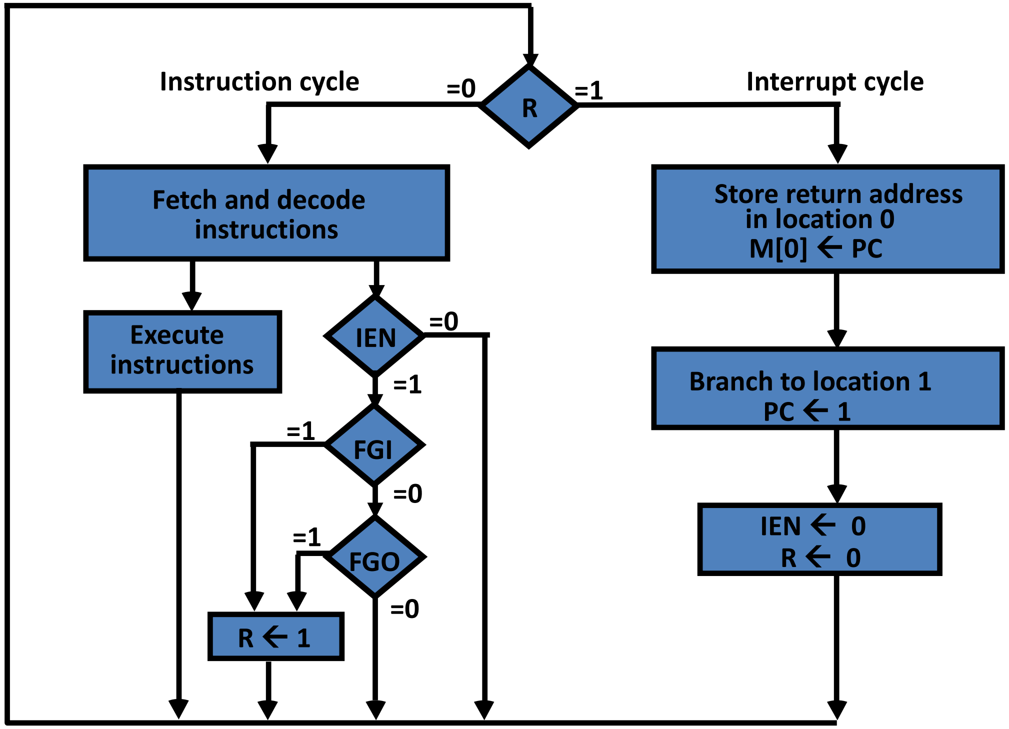
Program Interrupt

- For our computer, we choose the memory location at address 0 as a place for storing the return address

- Control then inserts address 1 into PC:
  - this means that the first instruction of the interrupt service routine should be stored in memory at address 1,
  - or, the programmer must store a branch instruction that sends the control to an interrupt service routine!!

Program Interrupt

R = Interrupt flip-flop



Flowchart for interrupt cycle

Program Interrupt

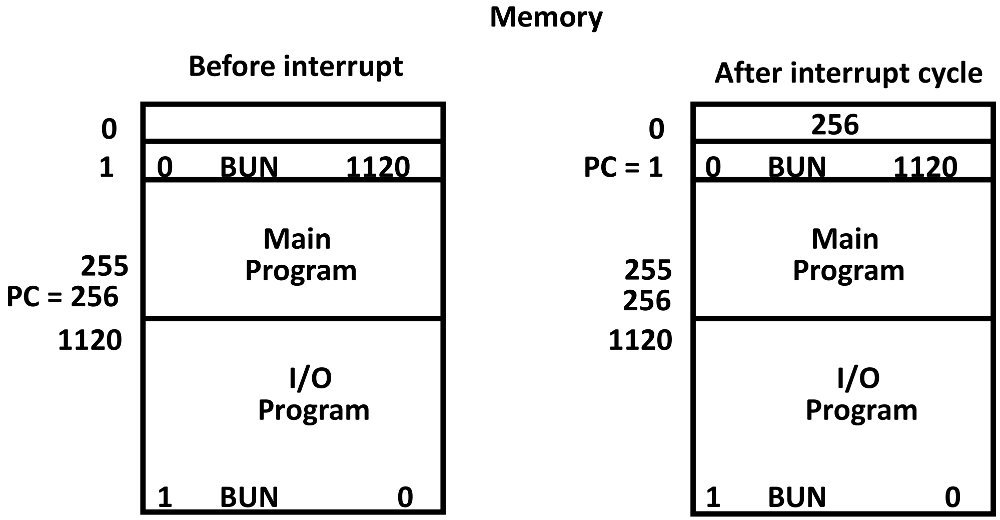
- IEN, R ← 0: no more interruptions can occur until the interrupt request from the flag has been serviced

- The service routine must end with an instruction that re-enables the interrupt (IEN ← 1) and an instruction to return to the instruction at which the interrupt occurred

- The instruction that returns the control to the original program is "indirect BUN 0"

Program Interrupt

- Example: the computer is interrupted during execution of the instruction at address 255



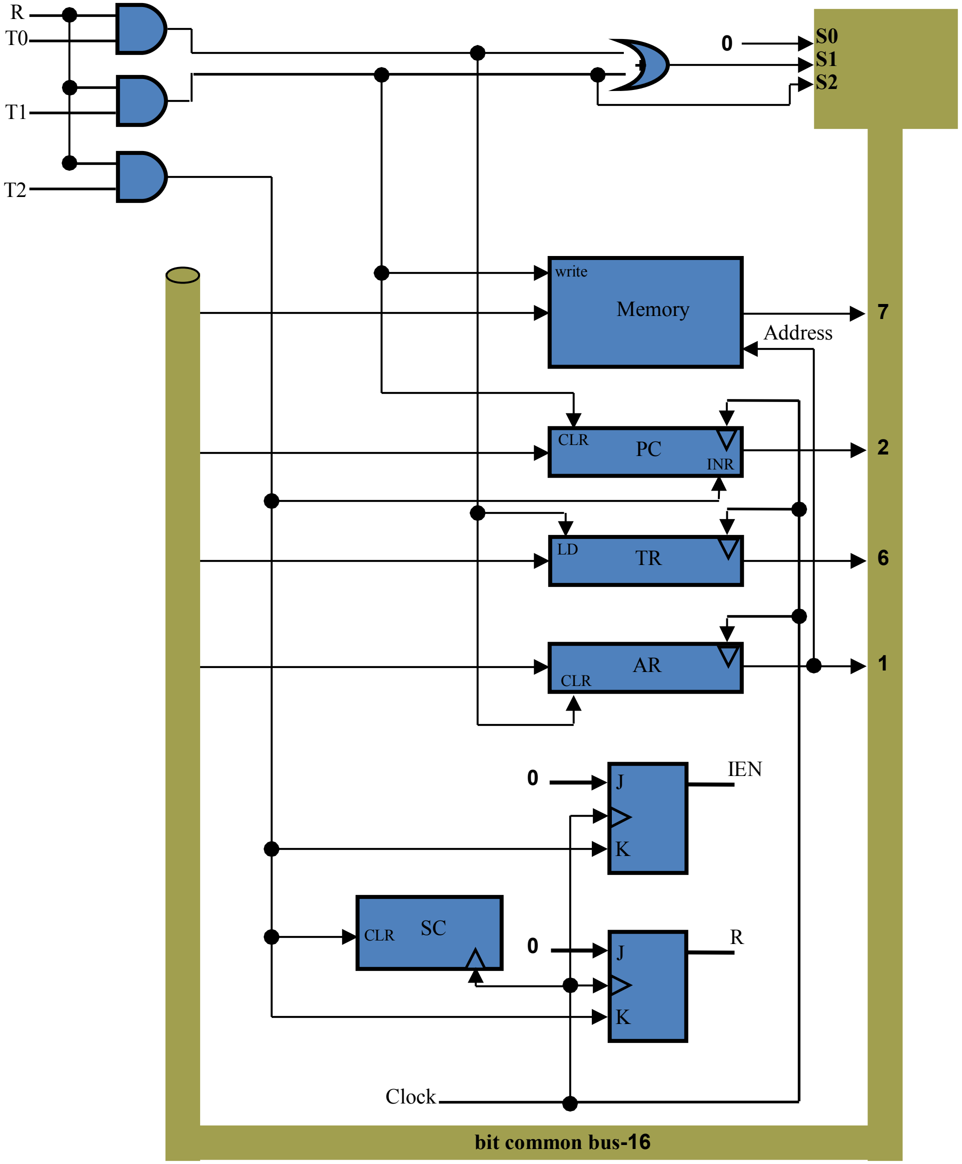
Interrupt Cycle

- The fetch and decode phases of the instruction cycle must be :

(Replace T0, T1, T2 → R'T0, R'T1, R'T2 (fetch and decode phases occur at the instruction cycle when R = 0)

- Interrupt Cycle:
  - **RT0: AR ← 0, TR ← PC**
  - **RT1: M[AR] ← TR, PC ← 0**

Register transfers for the Interrupt Cycle

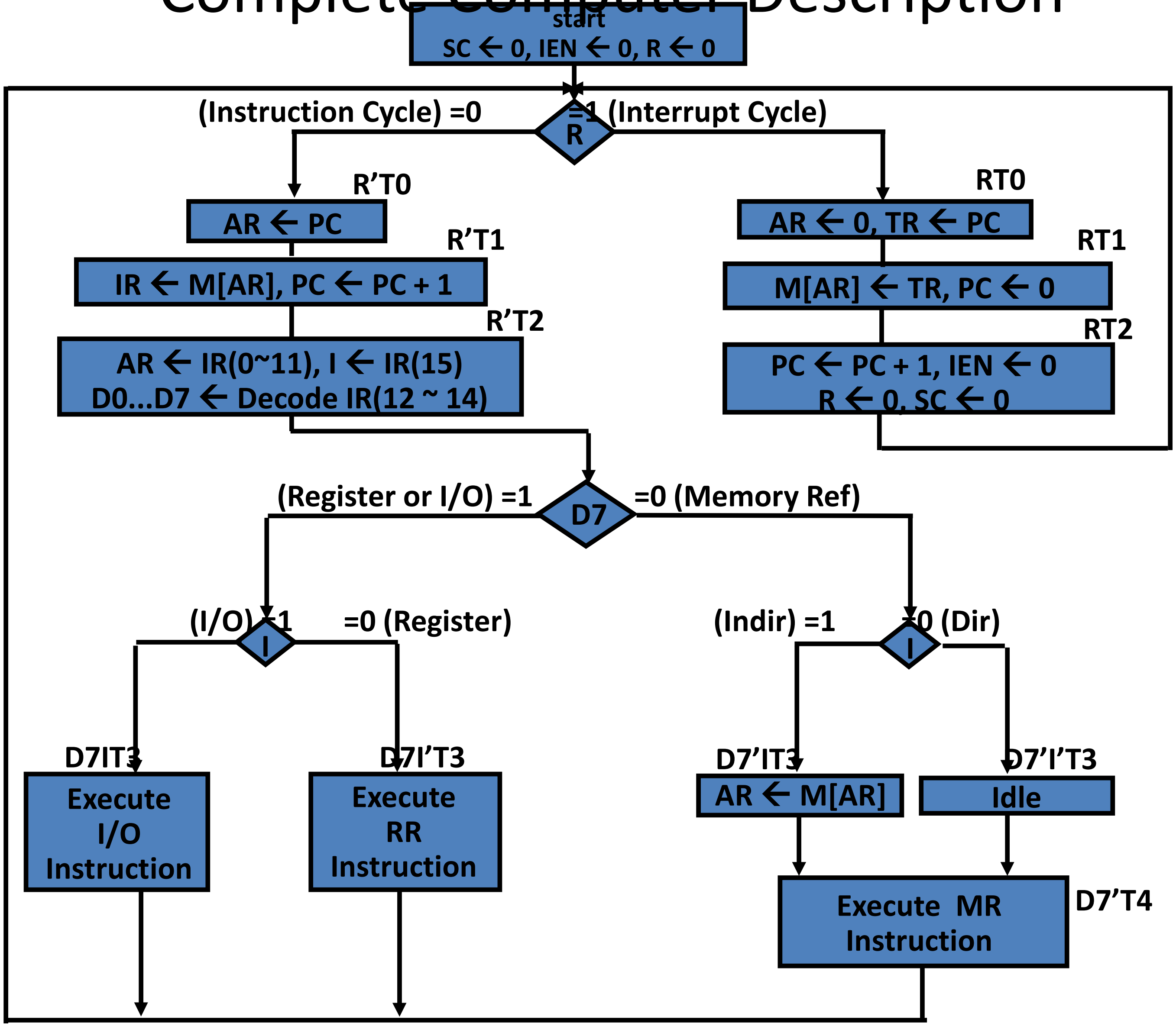


Interrupt

- **Further Questions:**
  - **How can the CPU recognize the device requesting an interrupt?**
  - **Since different devices are likely to require different interrupt service routines, how can the CPU obtain the starting address of the appropriate routine in each case?**
  - **Should any device be allowed to interrupt the CPU while another interrupt is being serviced?**
  - **How can the situation be handled when two or more interrupt requests occur simultaneously?**

Complete Computer Description

Fig 5-15



Complete Computer Description

Fetch	R’T0:	AR ← PC
	R’T1:	IR ← M[AR], PC ← PC + 1
Decode	R’T2:	D0, ..., D7 ← Decode IR(12 ~ 14), AR ← IR(0 ~ 11), I ← IR(15)
Indirect	D7’IT3:	AR ← M[AR]
Interrupt:		
T0’T1’T2’(IEN)(FGI + FGO):		R ← 1
	RT0:	AR ← 0, TR ← PC
	RT1:	M[AR] ← TR, PC ← 0
	RT2:	PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0
Memory-Reference:		
AND	D0T4:	DR ← M[AR]
	D0T5:	AC ← AC . DR, SC ← 0
ADD	D1T4:	DR ← M[AR]
	D1T5:	AC ← AC + DR, E ← Cout, SC ← 0
LDA	D2T4:	DR ← M[AR]
	D2T5:	AC ← DR, SC ← 0
STA	D3T4:	M[AR] ← AC, SC ← 0
BUN	D4T4:	PC ← AR, SC ← 0
BSA	D5T4:	M[AR] ← PC, AR ← AR + 1
	D5T5:	PC ← AR, SC ← 0
ISZ	D6T4:	DR ← M[AR]
	D6T5:	DR ← DR + 1
	D6T6:	M[AR] ← DR, if(DR=0) then (PC ← PC + 1), SC ← 0

Complete Computer Description

Register-Reference:

	D7I’T3 = r	(Common to all register-reference instructions)
	IR(i) = Bi	(i = 0,1,2, ..., 11)
	r:	SC ← 0
CLA	rB11:	AC ← 0
CLE	rB10:	E ← 0
CMA	rB9:	AC ← AC’
CME	rB8:	E ← E’
CIR	rB7:	AC ← shr AC, AC(15) ← E, E ← AC(0)
CIL	rB6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB5:	AC ← AC + 1
SPA	rB4:	If(AC(15) =0) then (PC ← PC + 1)
SNA	rB3:	If(AC(15) =1) then (PC ← PC + 1)
SZA	rB2:	If(AC = 0) then (PC ← PC + 1)
SZE	rB1:	If(E=0) then (PC ← PC + 1)
HLT	rB0:	S ← 0

Input-Output:

	D7IT3 = p	(Common to all input-output instructions)
	IR(i) = Bi	(i = 6,7,8,9,10,11)
	p:	SC ← 0
INP	pB11:	AC(0-7) ← INPR, FGI ← 0
OUT	pB10:	OUTR ← AC(0-7), FGO ← 0
SKI	pB9:	If(FGI=1) then (PC ← PC + 1)
SKO	pB8:	If(FGO=1) then (PC ← PC + 1)
ION	pB7:	IEN ← 1
IOF	pB6:	IEN ← 0

Table

Design of Basic Computer

1. **A memory unit: 4096 x 16.**
2. **Registers: AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC**
3. **Flip-Flops (Status): I, S, E, R, IEN, FGI, and FGO**
4. **Decoders:**
    1. **a 3x8 Opcode decoder**
    2. **a 4x16 timing decoder**
5. **Common bus: 16 bits**
6. **Control logic gates**
7. **Adder and Logic circuit: Connected to AC**

Design of Basic Computer

- The control logic gates are used to control:
  - Inputs of the nine registers
  - Read and Write inputs of memory
  - Set, Clear, or Complement inputs of the flip-flops
  - S2, S1, S0 that select a register for the bus
  - AC Adder and Logic circuit

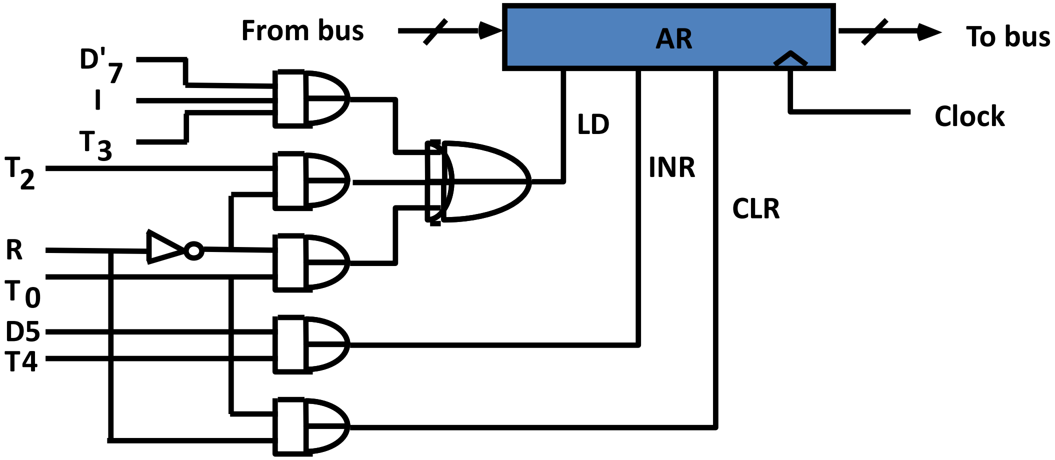
Design of Basic Computer

- Control of registers and memory
  - The control inputs of the registers are LD (load), INR (increment), and CLR (clear)
  - To control AR We scan table to find out all the statements that change the content of AR:

R’T0:	**AR ← PC**	**LD(AR)**
R’T2:	**AR ← IR(0-11)**	**LD(AR)**
D’7IT3:	**AR ← M[AR]**	**LD(AR)**
RT0:	**AR ← 0**	**CLR(AR)**
D5T4:	**AR ← AR + 1**	**INR(AR)**

Design of Basic Computer

Control Gates associated with AR



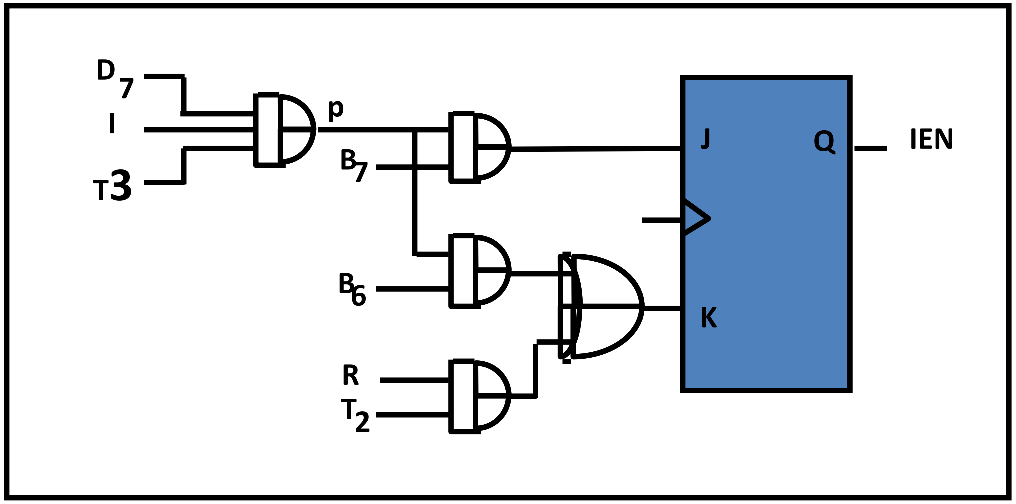
Design of Basic Computer

- To control the Read input of the memory we scan the table again to get these:
    - D0T4: DR ← M[AR]
    - D1T4: DR ← M[AR]
    - D2T4: DR ← M[AR]
    - D6T4: DR ← M[AR]
    - D7′IT3: AR ← M[AR]
    - R′T1: IR ← M[AR]
- → Read = R′T1 + D7′IT3 + (D0 + D1 + D2 + D6) T4

Design of Basic Computer

- Control of Single Flip-flops (IEN for example)
  - **pB7: IEN ← 1 (I/O Instruction)**
  - **pB6: IEN ← 0 (I/O Instruction)**
  - **RT2: IEN ← 0 (Interrupt)**
    - **where p = D7IT3 (Input/Output Instruction)**
  - If we use a JK flip-flop for IEN, the control gate logic will be as shown in the following slide:

Design of Basic Computer

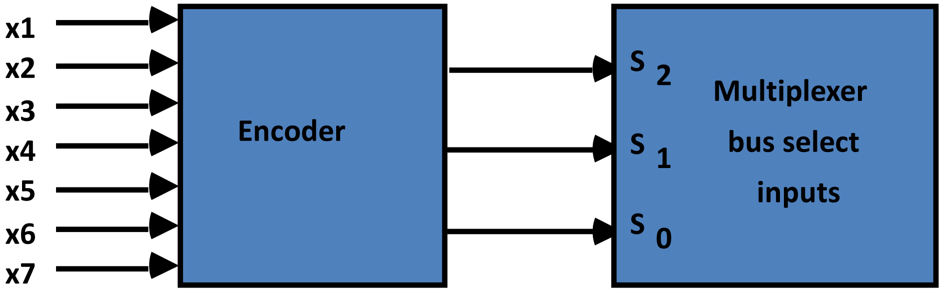


J	*K*	*Q(t+1)*
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

JK FF Characteristic Table

Design of Basic Computer

- Control of Common bus is accomplished by placing an encoder at the inputs of the bus selection logic and implementing the logic for each encoder input



Design of Basic Computer

- To select AR on the bus then x1 must be 1. This is happen when:
  - **D4T4: PC ← AR**
  - **D5T5: PC ← AR**
- ⇒ x1 = D4T4 + D5T5

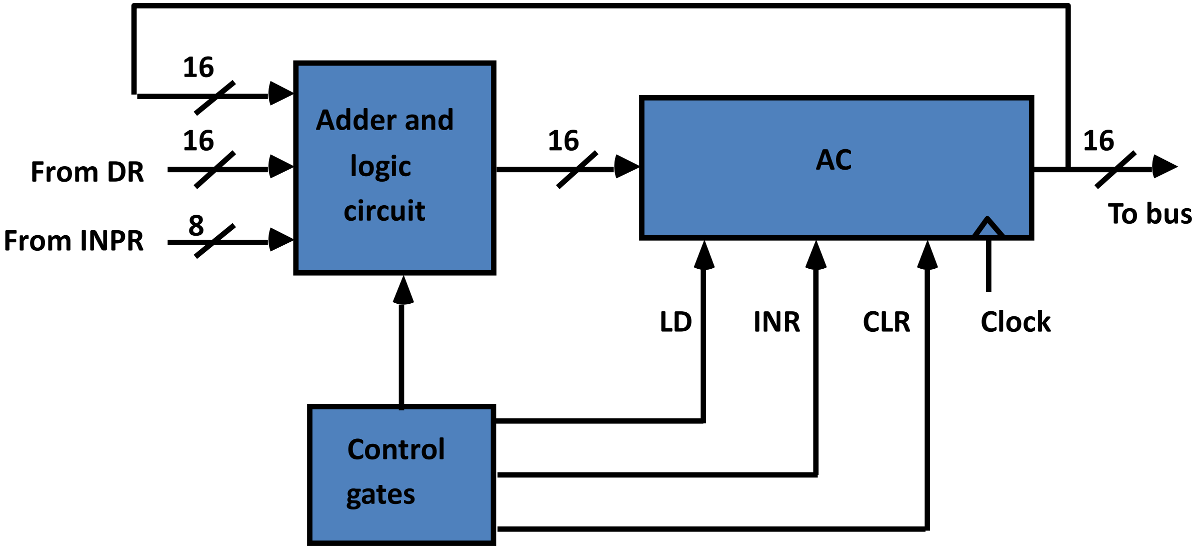
x1	x2	x3	x4	x5	x6	x7	S2	S1	S0	selected register
0	0	0	0	0	0	0	0	0	0	none
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	Memory

Design of Basic Computer

- For x7:
  - $X7 = R'T1 + D7'IT3 + (D0 + D1 + D2 + D6\ )T4$ where it is also applied to the read input

Design of Accumulator Logic

Circuits associated with AC

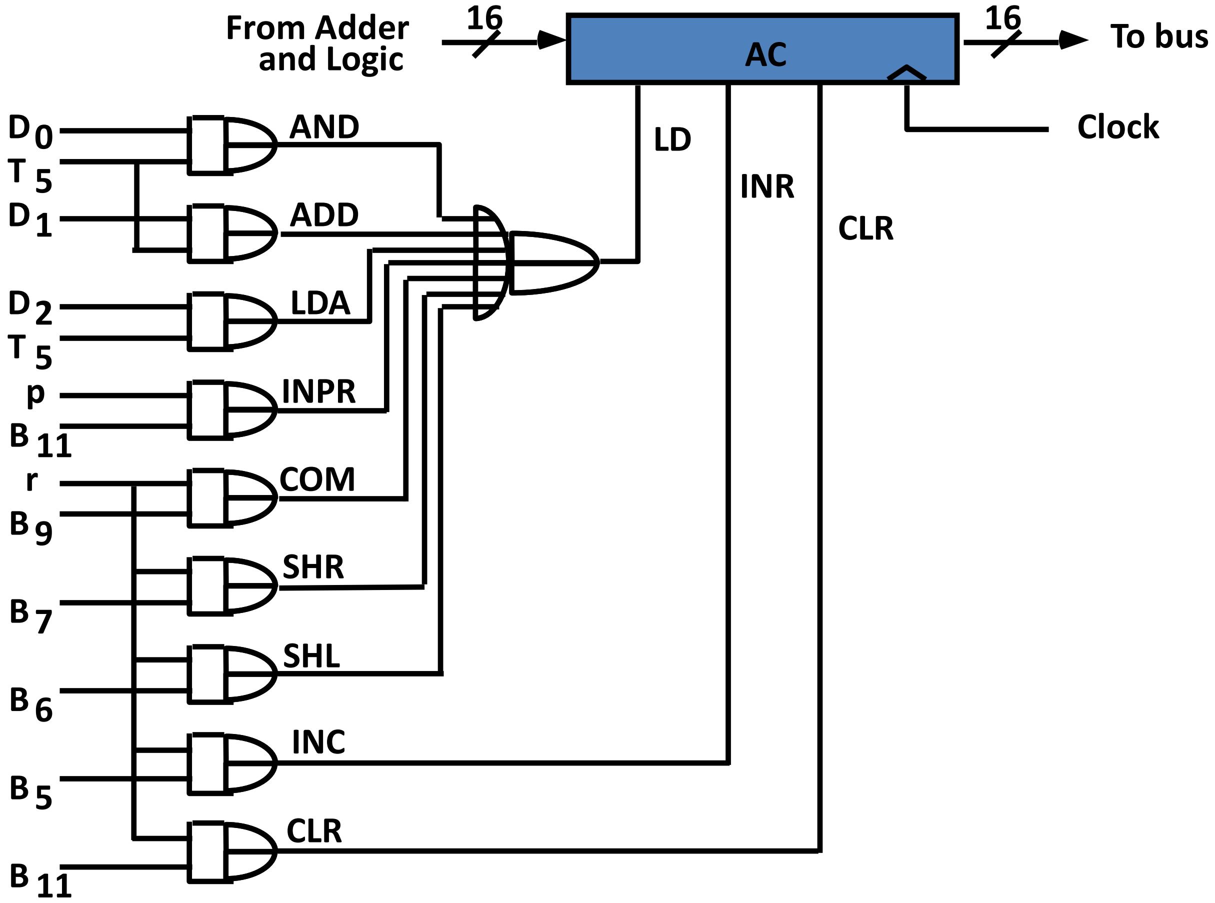


All the statements that change the content of AC

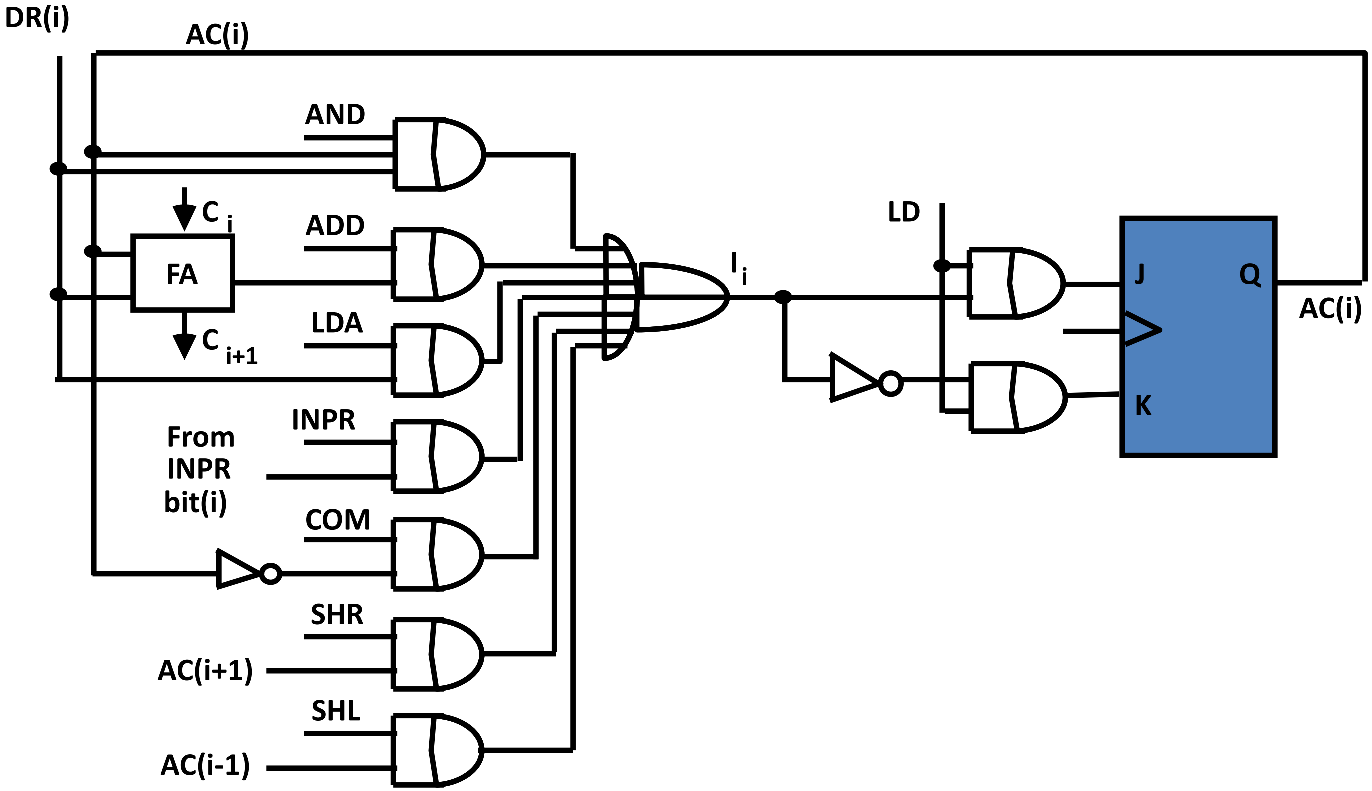
D0T5:	AC ← AC ∧ DR	AND with DR
D1T5:	AC ← AC + DR	Add with DR
D2T5:	AC ← DR	Transfer from DR
pB11:	AC(0-7) ← INPR	Transfer from INPR
rB9:	AC ← AC’	Complement
rB7 :	AC ← shr AC, AC(15) ← E	Shift right
rB6 :	AC ← shl AC, AC(0) ← E	Shift left
rB11 :	AC ← 0	Clear
rB5 :	AC ← AC + 1	Increment

Design of Accumulator Logic

Gate structures for controlling the LD, INR, and CLR of AC



Adder and Logic Circuit



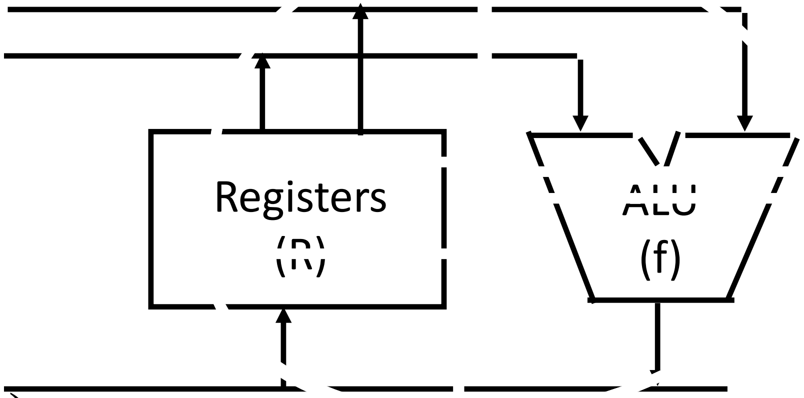
REGISTER TRANSFER AND MICROOPERATIONS

- Register Transfer Language
- Register Transfer
- Bus and Memory Transfers
- Arithmetic Microoperations
- Logic Microoperations

SIMPLE DIGITAL SYSTEMS

- Combinational and sequential circuits can be used to create simple digital systems.
- These are the low-level building blocks of a digital computer.
- Simple digital systems are frequently characterized in terms of
  - the registers they contain, and
  - the operations that they perform.
- Typically,
  - What operations are performed on the data in the registers
  - What information is passed between registers
- The operations on the data in registers are called microoperations.
- The functions built into registers are examples of microoperations
  - Shift
  - Load

- An elementary operation performed (during one clock pulse), on the information stored in one or more registers
- 1 clock cycle:



- R ← f(R, R)
- f: shift, load, clear, increment, add, subtract, complement,and, or, xor, …

ORGANIZATION OF A DIGITAL SYSTEM

- Definition of the (internal) organization of a computer
  - Set of registers and their functions
  - Micro operations set

Set of allowable micro operations provided by the organization of the computer

- Control signals that initiate the sequence of micro operations (to perform the functions)

REGISTER TRANSFER LEVEL

- Viewing a computer, or any digital system, in this way is called the register transfer level

- This is because we’re focusing on
  - The system’s registers
  - The data transformations in them, and
  - The data transfers between them.

REGISTER TRANSFER LANGUAGE

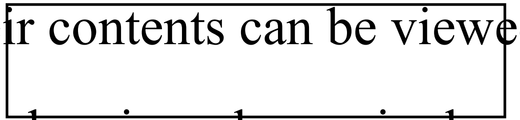
- Rather than specifying a digital system in words, a specific notation is used, *register transfer language*

- For any function of the computer, the register transfer language can be used to describe the (sequence of) microoperations

- Register transfer language
  - A symbolic language
  - A convenient tool for describing the internal organization of digital computers
  - Can also be used to facilitate the design process of digital

DESIGNATION OF REGISTERS

- Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)
- Often the names indicate function:
  - MAR - memory address register
  - PC - program counter
  - IR - instruction register
- Registers and their contents can be viewed and represented in *various ways*
  - A register can be viewed as a single entity:



  - Registers may also be represented showing the bits of data they contain

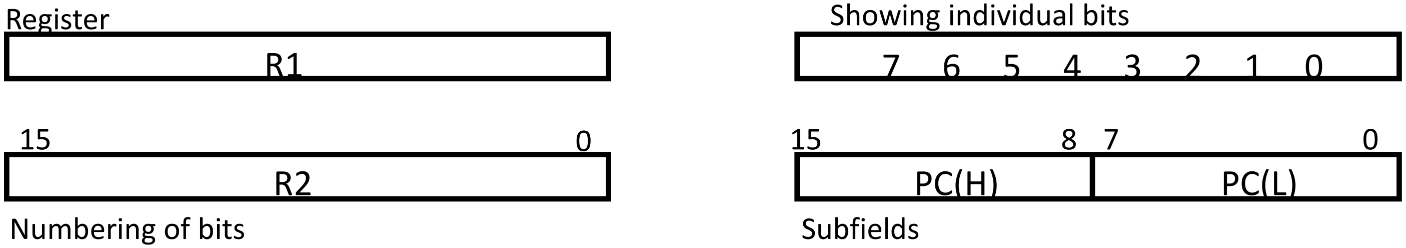
- Designation of a register

  - a register

  - portion of a register

  - a bit of a register

- Common ways of drawing the block diagram of a register



REGISTER TRANSFER

- Copying the contents of one register to another is a register transfer

- A register transfer is indicated as

R2 ← R1

  - In this case the contents of register R1 are copied (loaded) into register R2
  - A simultaneous transfer of all bits from the source R1 to the destination register R2, during one clock pulse
  - Note that this is a non-destructive; i.e. the contents of R1 are not altered by copying (loading) them to R2

- A register transfer such as

R3 ← R5

Implies that the digital system has

  - the data lines from the source register (R5) to the destination register (R3)
  - Parallel load in the destination register (R3)
  - Control lines to perform the action

CONTROL FUNCTIONS

- Often actions need to only occur if a certain condition is true
- This is similar to an “if” statement in a programming language
- In digital systems, this is often done via a *control signal*, called a *control function*
  - If the signal is 1, the action takes place
- This is represented as:

P: R2 ← R1

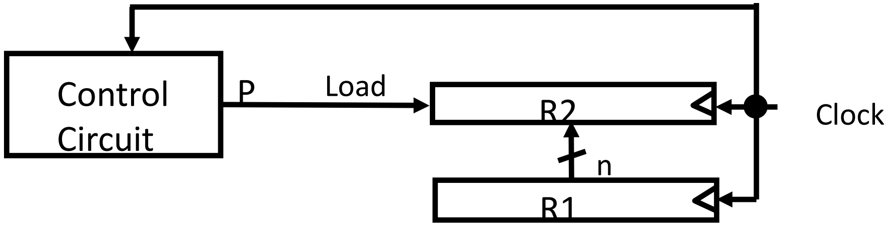
Which means “if P = 1, then load the contents of register R1 into register R2”, i.e., if (P = 1) then (R2 ← R1)

HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

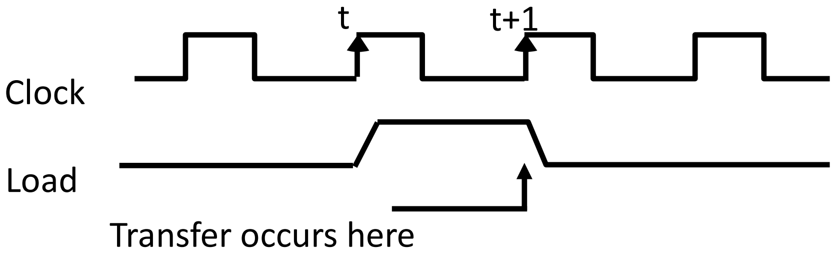
- Implementation of controlled transfer

P: R2 ←R1

Block diagram



Timing diagram



- The same clock controls the circuits that generate the control function and the destination register

- Registers are assumed to use *positive-edge-triggered* flip-flops

SIMULTANEOUS OPERATIONS

- If two or more operations are to occur simultaneously, they are separated with commas

P: R3 ← R5, MAR ← IR

- Here, if the control function P = 1, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR

BASIC SYMBOLS FOR REGISTER TRANSFERS

Symbols	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A ← B, B ← A

CONNECTING REGISTRS

- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers

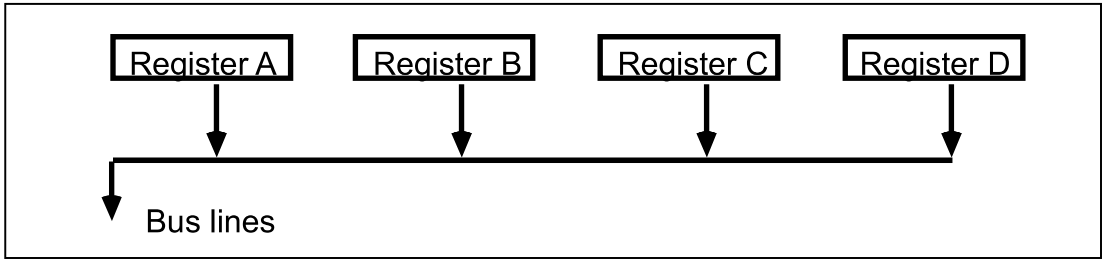
- To completely connect n registers → n(n-1) lines
- O(n2) cost
  - This is not a realistic approach to use in a large digital system

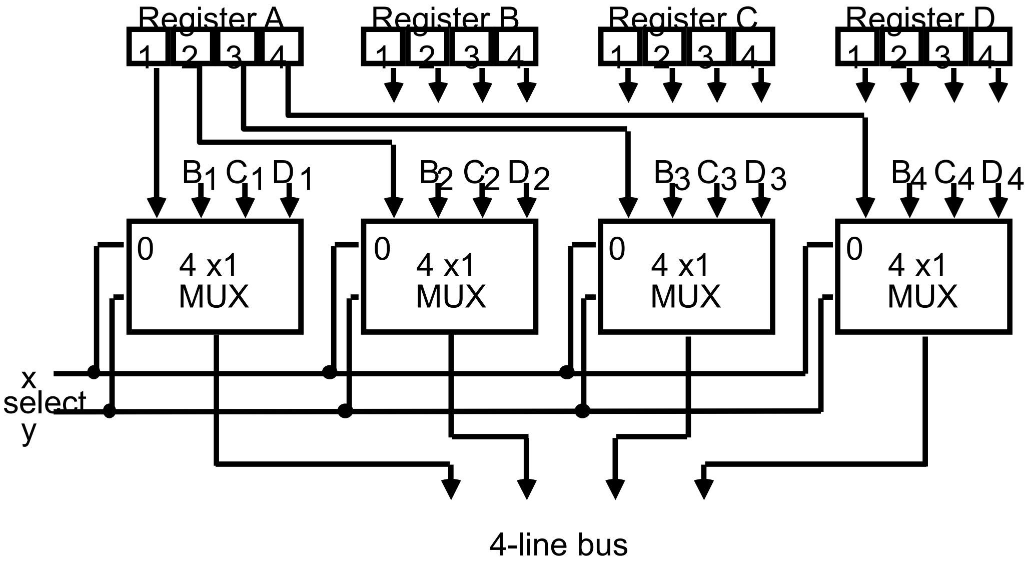
- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the bus
- Have control circuits to select which register is the source, and which is the destination

BUS AND BUS TRANSFER

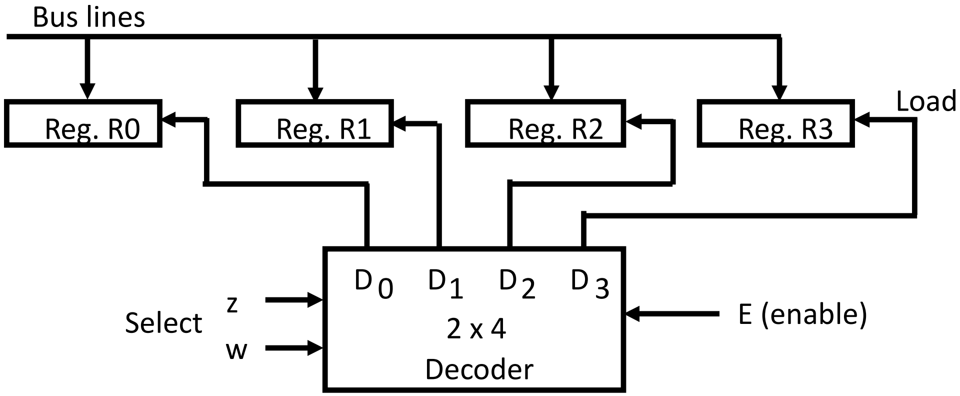
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

From a register to bus: BUS ← R





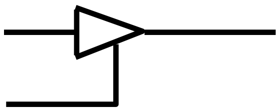
TRANSFER FROM BUS TO A DESTINATION REGISTER



Three-State Bus Buffers

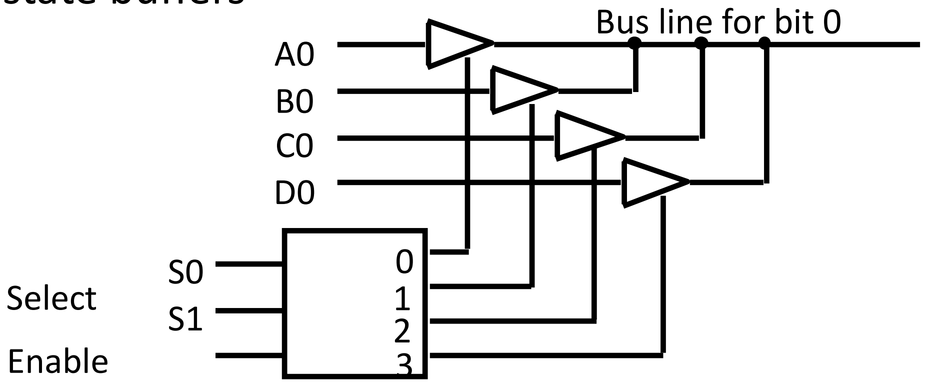
Normal input A

Control input C



Output Y=A if C=1
High-impedence if C=0

Bus line with three-state buffers



BUS TRANSFER IN RTL

- Depending on whether the bus is to be mentioned explicitly or not, register transfer can be indicated as either

$$R2 \leftarrow R1$$

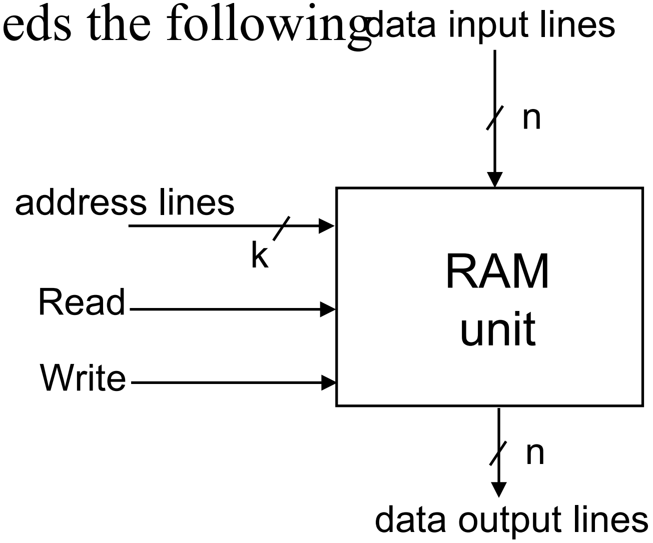
or

$$BUS \leftarrow R1,\ R2 \leftarrow BUS$$

- In the former case the bus is implicit, but in the latter, it is explicitly indicated

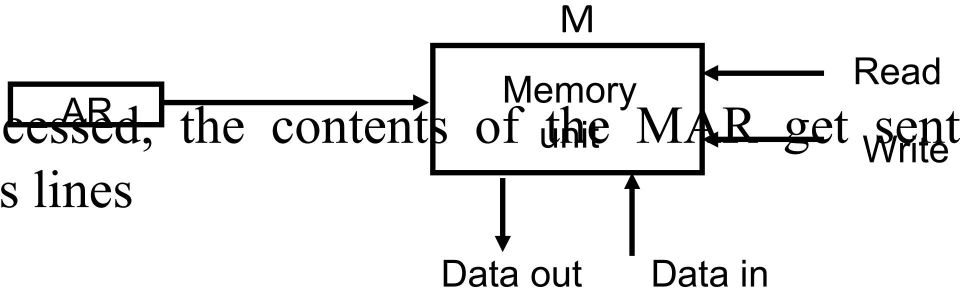
MEMORY (RAM)

- Memory (RAM) can be thought as a sequential circuits containing some number of registers
- These registers hold the *words* of memory
- Each of the r registers is indicated by an *address*
- These addresses range from 0 to r-1
- Each register (word) can hold n bits of data
- Assume the RAM contains r = 2k words. It needs the following
  - n data input lines
  - n data output lines
  - k address lines
  - A Read control line
  - A Write control line



MEMORY TRANSFER

- Collectively, the memory is viewed at the register level as a device, M.
- Since it contains multiple locations, we must specify which address in memory we will be using
- This is done by indexing memory references
- Memory is usually accessed in computer systems by putting the desired address in a special register, the *Memory Address Register* (*MAR*, or *AR*)
- When memory is accessed, the contents of the MAR get sent to the memory unit's address lines



MEMORY READ

- To read a value from a location in memory and load it into a register, the register transfer language notation looks like this:

$$R1 \leftarrow M[MAR]$$

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Read (= 1) gets sent to the memory unit
  - The contents of the specified address are put on the memory's output data lines
  - These get sent over the bus to be loaded into register R1

MEMORY WRITE

- To write a value from a register to a location in memory looks like this in register transfer language:

$$M[MAR] \leftarrow R1$$

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Write (= 1) gets sent to the memory unit
  - The values in register R1 get sent over the bus to the data input lines of the memory
  - The values get loaded into the specified address in the memory

SUMMARY OF R. TRANSFER MICROOPERATIONS

A ← B	Transfer content of reg. B into reg. A
AR ← DR(AD)	Transfer content of AD portion of reg. DR into reg. AR
A ← constant	Transfer a binary constant into reg. A
ABUS ← R1,	Transfer content of R1 into bus A and, at the same time,
R2 ← ABUS	transfer content of bus A into R2
AR	Address register
DR	Data register
M[R]	Memory word specified by reg. R
M	Equivalent to M[AR]
DR ← M	Memory *read* operation: transfers content of memory word specified by AR into DR
M ← DR	Memory *write* operation: transfers content of DR into memory word specified by AR

MICROOPERATIONS

- Computer system microoperations are of four types:

  - Register transfer microoperations
  - Arithmetic microoperations
  - Logic microoperations
  - Shift microoperations

ARITHMETIC MICROOPERATIONS

- The basic arithmetic microoperations are
  - Addition
  - Subtraction
  - Increment
  - Decrement

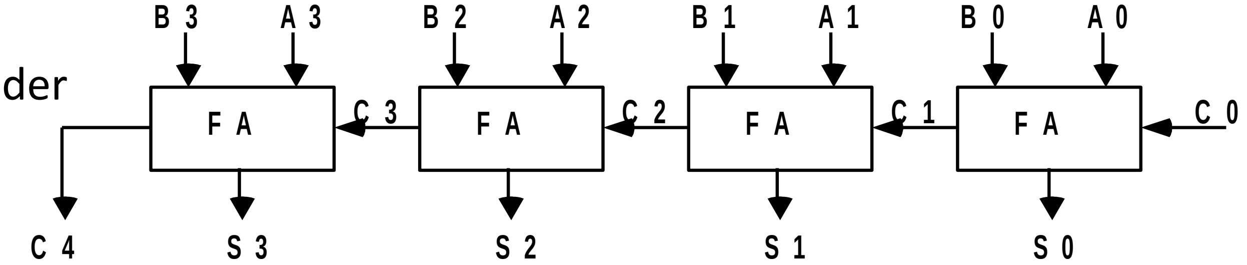
- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. ..

Summary of Typical Arithmetic Micro-Operations

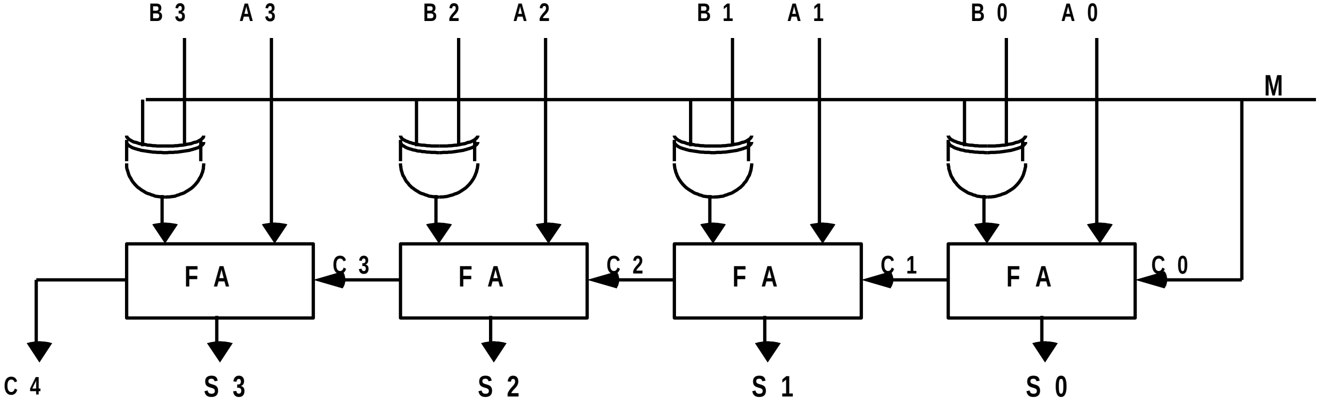
R3 ← R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ← R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ← R2’	Complement the contents of R2
R2 ← R2’+ 1	2's complement the contents of R2 (negate)
R3 ← R1 + R2’+ 1	subtraction
R1 ← R1 + 1	Increment
R1 ← R1 - 1	Decrement

BINARY ADDER / SUBTRACTOR / INCREMENTER

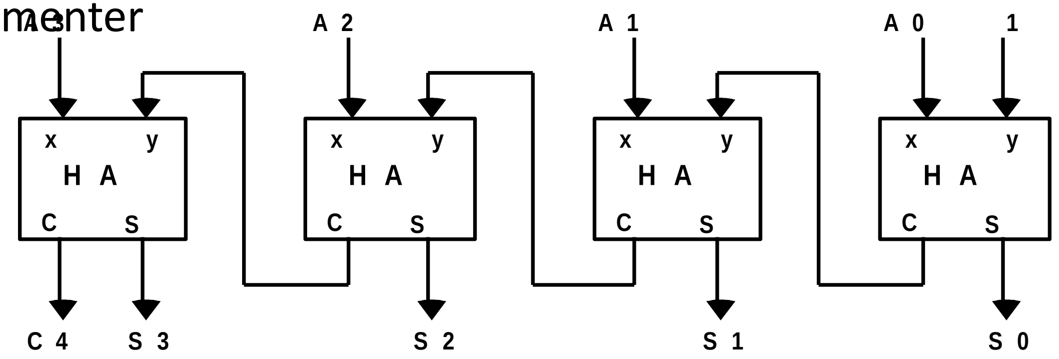
Binary Adder



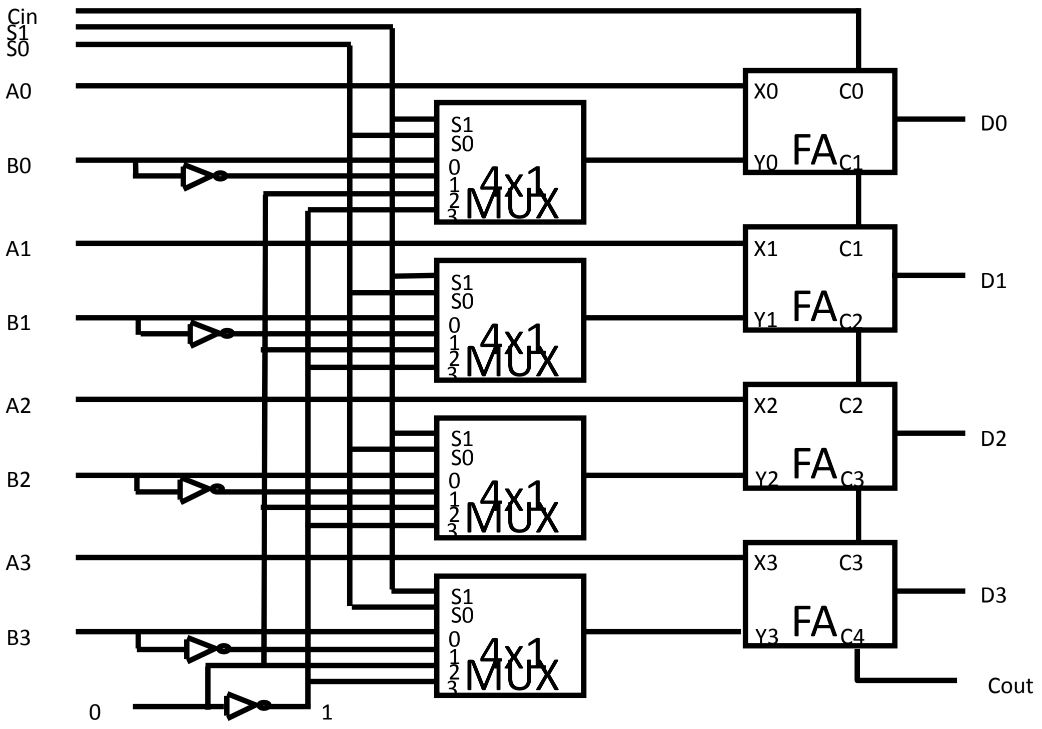
Binary Adder-Subtractor



Binary Incrementer



ARITHMETIC CIRCUIT



S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	0	B’	D = A + B’	Subtract with borrow
0	1	1	B’	D = A + B’+ 1	Subtract
1	0	0	0	D = A	Transfer A
1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A - 1	Decrement A
1	1	1	1	D = A	Transfer A

LOGIC MICROOPERATIONS

- Specify binary operations on the strings of bits in registers
  - Logic micro operations are bit-wise operations, i.e., they work on the individual bits of data
  - useful for bit manipulations on binary data
  - useful for making logical decisions based on the bit value
- There are, in principle, 16 different logic functions that can be defined over two binary input variables

A	B	F0	F1	F2 …	F13	F14	F15
0	0	0	0	0 …	1	1	1
0	1	0	0	0 …	1	1	1
1	0	0	0	1 …	0	1	1
1	1	0	1	0 …	1	0	1

- However, most systems only implement four of these
  - AND (∧), OR (∨), XOR (⊕), Complement/NOT

LIST OF LOGIC MICROOPERATIONS

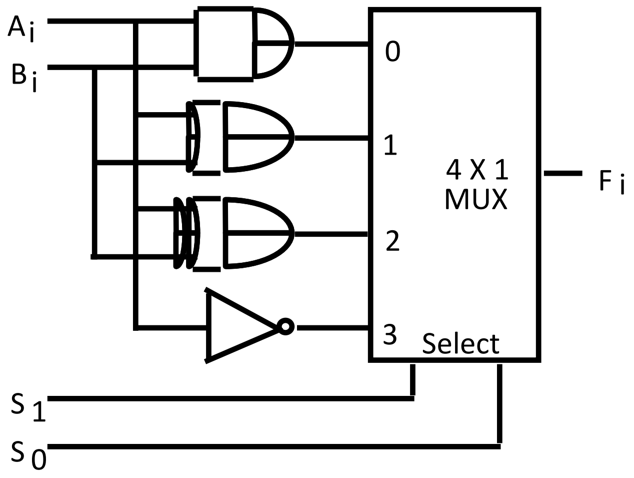
- List of Logic Microoperations

  - 16 different logic operations with 2 binary vars.
  - n binary vars → 2^{2^n} functions

- Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations

x y	0 0 1 1 0 1 0 1	*Boolean Function*	*Micro-Operations*	*Name*
	0 0 0 0	F0 = 0	F ← 0	Clear
	0 0 0 1	F1 = xy	F ← A ∧ B	AND
	0 0 1 0	F2 = xy'	F ← A ∧ B’	
	0 0 1 1	F3 = x	F ← A	Transfer A
	0 1 0 0	F4 = x'y	F ← A’∧ B	
	0 1 0 1	F5 = y	F ← B	Transfer B
	0 1 1 0	F6 = x ⊕ y	F ← A ⊕ B	Exclusive-OR
	0 1 1 1	F7 = x + y	F ← A ∨ B	OR
	1 0 0 0	F8 = (x + y)'	F ← (A ∨ B)’	NOR
	1 0 0 1	F9 = (x ⊕ y)'	F ← (A ⊕ B)’	Exclusive-NOR
	1 0 1 0	F10 = y'	F ← B’	Complement B
	1 0 1 1	F11 = x + y'	F ← A ∨ B	
	1 1 0 0	F12 = x'	F ← A’	Complement A
	1 1 0 1	F13 = x' + y	F ← A’∨ B	
	1 1 1 0	F14 = (xy)'	F ← (A ∧ B)’	NAND
	1 1 1 1	F15 = 1	F ← all 1's	Set to all 1's

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



Function table

S1 S0	Output	μ-operation
0 0	F = A ∧ B	AND
0 1	F = A ∨ B	OR
1 0	F = A ⊕ B	XOR
1 1	F = A’	Complement

APPLICATIONS OF LOGIC MICROOPERATIONS

- Logic micro operations can be used to manipulate individual bits or a portions of a word in a register
- Consider the data in a register A. In another register, B, is bit data that will be used to modify the contents of A

  - Selective-set $A \leftarrow A + B$
  - Selective-complement $A \leftarrow A \oplus B$
  - Selective-clear $A \leftarrow A \bullet B'$
  - Mask (Delete) $A \leftarrow A \bullet B$
  - Clear $A \leftarrow A \oplus B$
  - Insert $A \leftarrow (A \bullet B) + C$

SELECTIVE SET

- In a selective set operation, the bit pattern in B is used to *set* certain bits in A

1 1 0 0   At

---

1 0 1 0   B

1 1 1 0   At+1   (A ← A + B)

- If a bit in B is set to 1, that same position in A gets set to 1, otherwise that bit in A keeps its previous value

SELECTIVE COMPLEMENT

- In a selective complement operation, the bit pattern in B is used to *complement* certain bits in A

1 1 0 0   At

---

1 0 1 0   B

0 1 1 0   At+1 $(A \leftarrow A \oplus B)$

- If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged

SELECTIVE CLEAR

- In a selective clear operation, the bit pattern in B is used to *clear* certain bits in A

1 1 0 0 At

1 0 1 0 B

0 1 0 0 At+1 (A ← A · B’)

- If a bit in B is set to 1, that same position in A gets set to 0, otherwise it is unchanged

MASK OPERATION

- In a mask operation, the bit pattern in B is used to *clear* certain bits in A

1 1 0 0 At

---

1 0 1 0 B

1 0 0 0 At+1 $(A \leftarrow A \cdot B)$

- If a bit in B is set to 0, that same position in A gets set to 0, otherwise it is unchanged

CLEAR OPERATION

- In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A

1 1 0 0   At

1 0 1 0   B

0 1 1 0   At+1 $(A \leftarrow A \oplus B)$

INSERT OPERATION

- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged

- This is done as
  - A mask operation to clear the desired bit positions, followed by
  - An OR operation to introduce the new bits into the desired positions
  - Example
    - Suppose you wanted to introduce 1010 into the low order four bits of A: 1101 1000 1011 0001 A (Original) 1101 1000 1011 1010 A (Desired)
    - 1101 1000 1011 0001 A (Original)

1111 1111 1111 0000 Mask

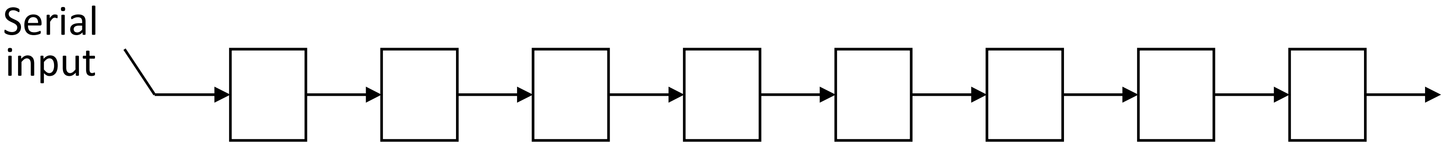
1101 1000 1011 0000 A (Intermediate)

0000 0000 0000 1010 Added bits

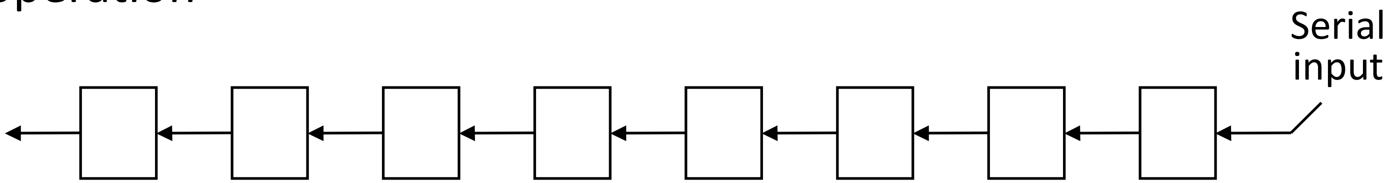
1101 1000 1011 1010 A (Desired)

SHIFT MICROOPERATIONS

- There are three types of shifts
  - *Logical shift*
  - *Circular shift*
  - *Arithmetic shift*
- What differentiates them is the information that goes into the serial input
- A right shift operation



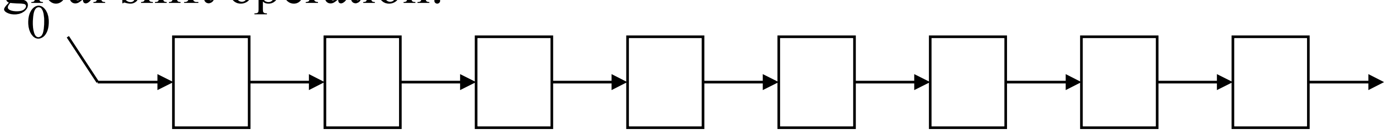
- A left shift operation



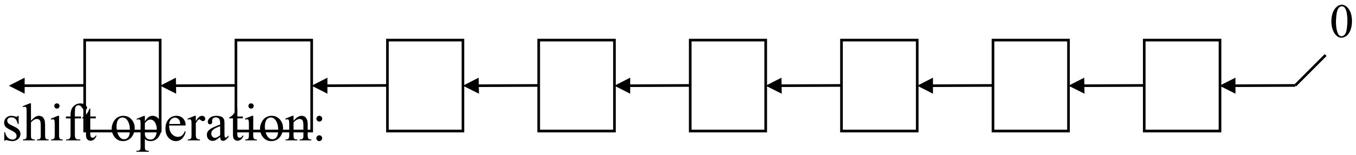
LOGICAL SHIFT

- In a logical shift the serial input to the shift is a 0.

- A right logical shift operation:



- A left logical shift operation:

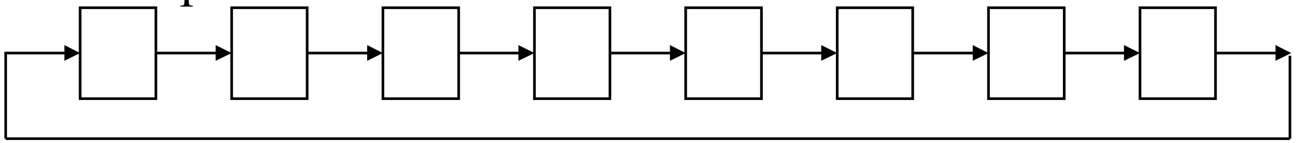


- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right

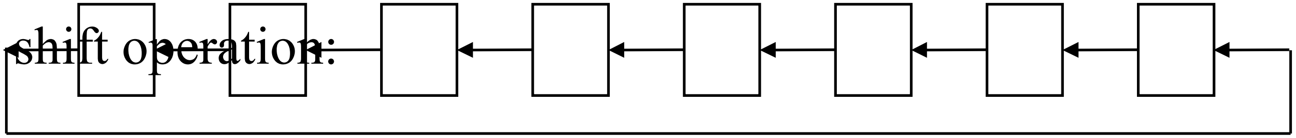
CIRCULAR SHIFT

- In a circular shift the serial input is the bit that is shifted out of the other end of the register.

- A right circular shift operation:



- A left circular shift operation:

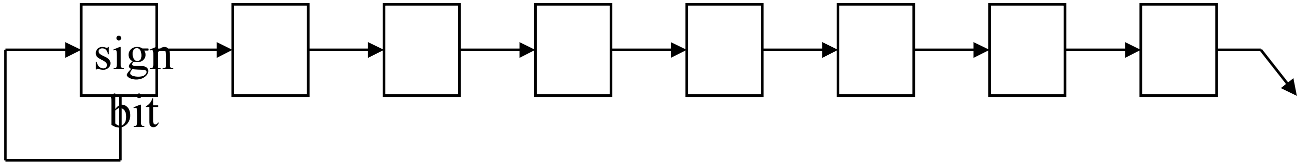


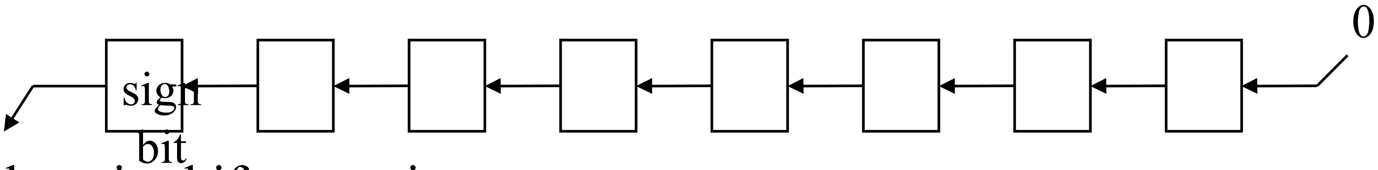
- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right

ARITHMETIC SHIFT

- An arithmetic shift is meant for signed binary numbers (integer)
- An arithmetic left shift multiplies a signed number by two
- An arithmetic right shift divides a signed number by two
- The main distinction of an arithmetic shift is that it must keep the sign of the number the same as it performs the multiplication or division

- A right arithmetic shift operation:

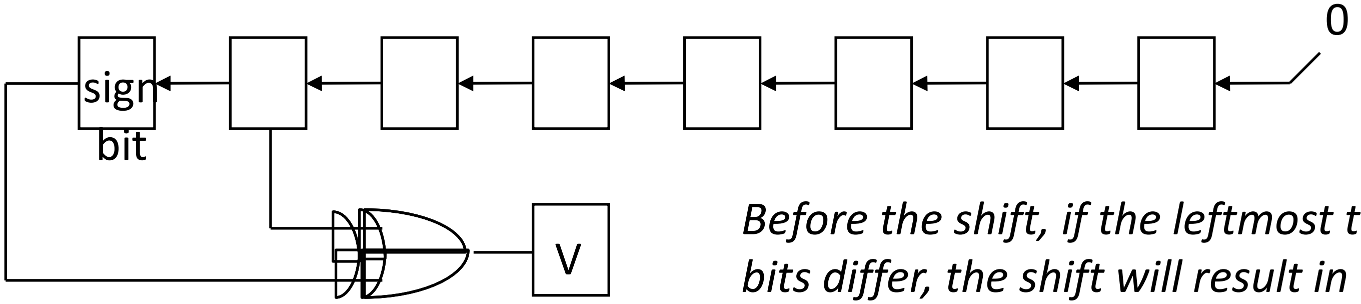




- A left arithmetic shift operation:

ARITHMETIC SHIFT

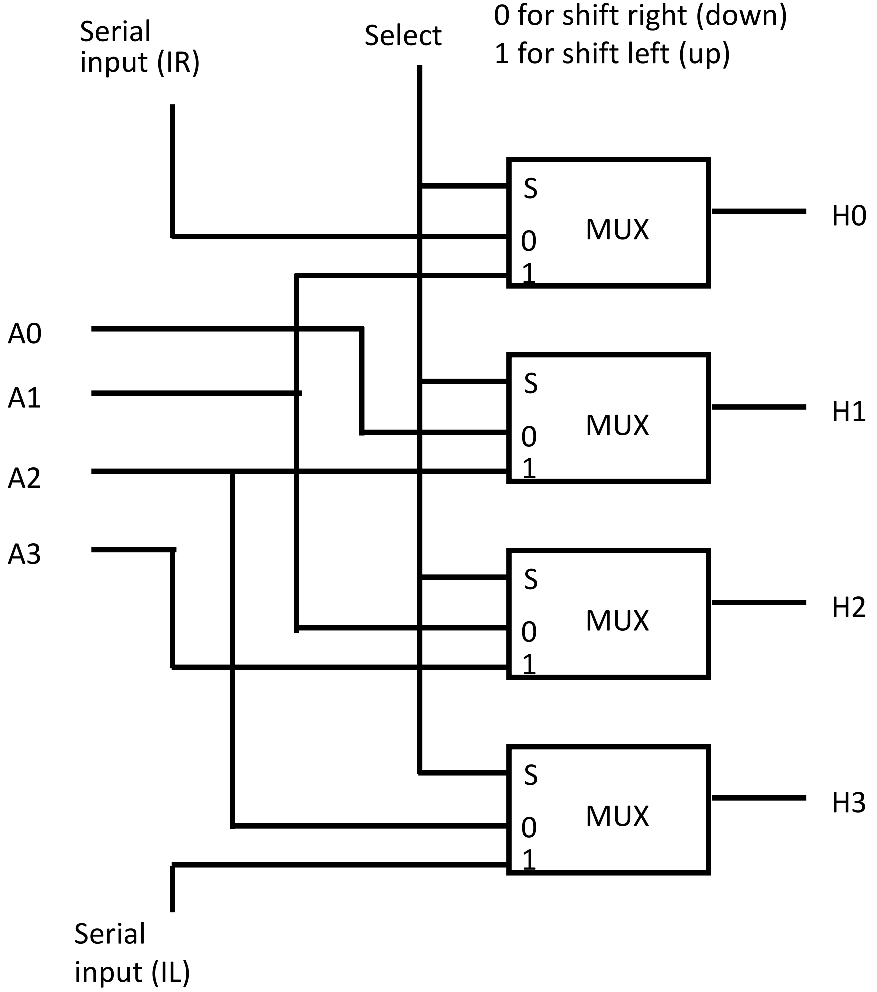
- An left arithmetic shift operation must be checked for the overflow



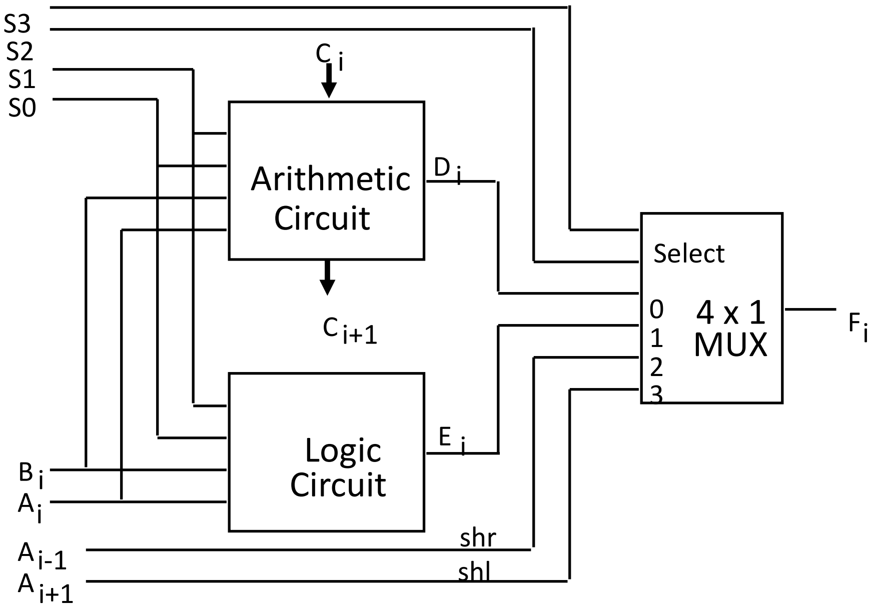
Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- In a RTL, the following notation is used
  - *ashl* for an arithmetic shift left
  - *ashr* for an arithmetic shift right
  - Examples:
    - » R2 ← *ashr* R2
    - » R3 ← *ashl* R3

HARDWARE IMPLEMENTATION OF SHIFT MICROOPERATIONS



ARITHMETIC LOGIC SHIFT UNIT



S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B’	Subtract with borrow
0	0	1	0	1	F = A + B’+ 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F = A	TransferA
0	1	0	0	X	F = A ∧ B	AND
0	1	0	1	X	F = A ∨ B	OR
0	1	1	0	X	F = A ⊕ B	XOR
0	1	1	1	X	F = A’	Complement A
1	0	X	X	X	F = shr A	Shift right A into F
1	1	X	X	X	F = shl A	Shift left A into F

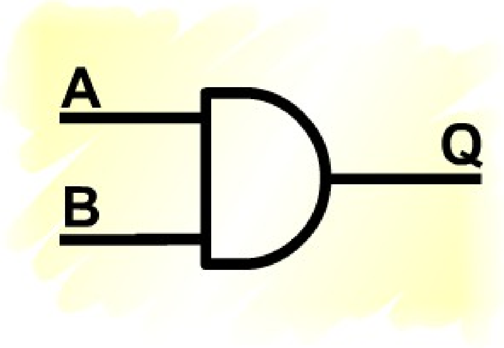
- **UNIT II ARITHMETIC OPERATIONS**
  ALU (Arthimetic Logic Unit)

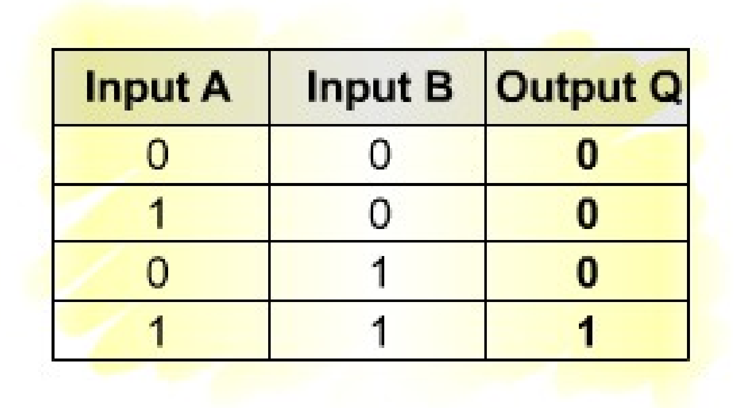
Reference

- **Appendix B: The Basics of Logic Design**
- Book - David A. Patterson and John L. Hennessey, “Computer organization and design‟, Morgan kauffman / Elsevier, Fifth edition, 2014.

Logic Gates

AND

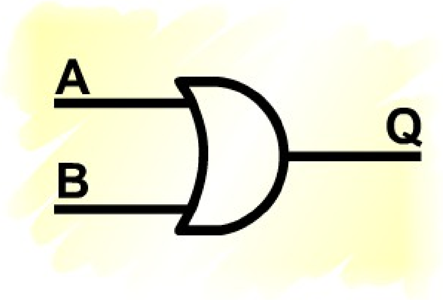


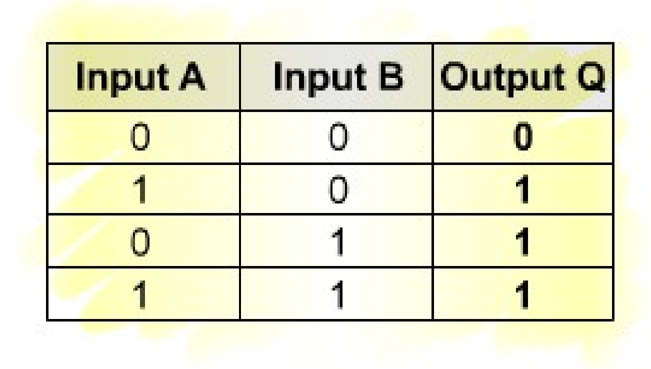


Input A	Input B	Output Q
0	0	**0**
1	0	**0**
0	1	**0**
1	1	**1**

Q = A.B

OR

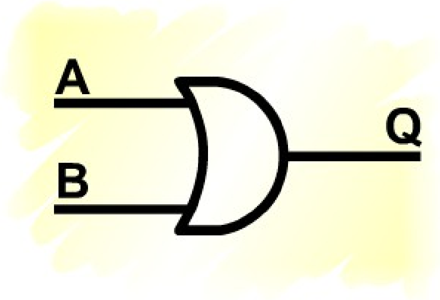


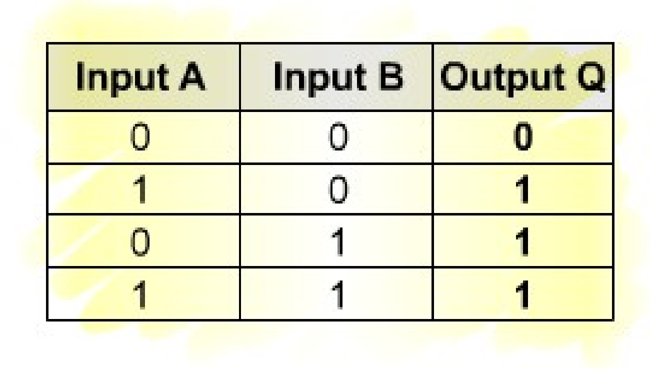


Input A	Input B	Output Q
0	0	**0**
1	0	**1**
0	1	**1**
1	1	**1**

Q = A+B

OR

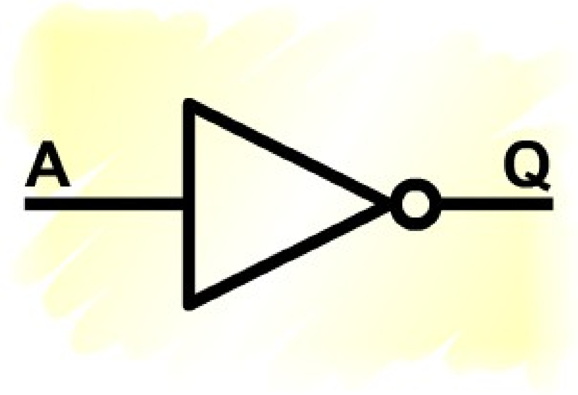




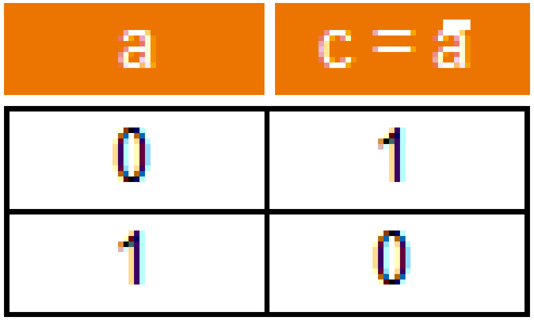
Input A	Input B	Output Q
0	0	**0**
1	0	**1**
0	1	**1**
1	1	**1**

Q = A+B

NOT (Inversion)

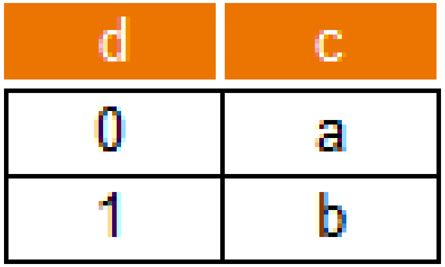


$$Q = \overline{A}$$

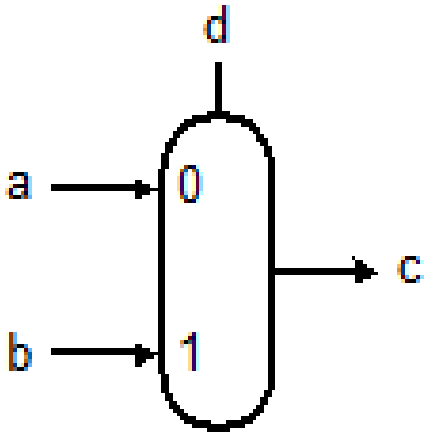


a	$c = \bar{a}$
0	1
1	0

Multiplexor



d	c
0	a
1	b



If d == 0
Then c = a
Else if d ==1
Then c = b

Used to select an operation

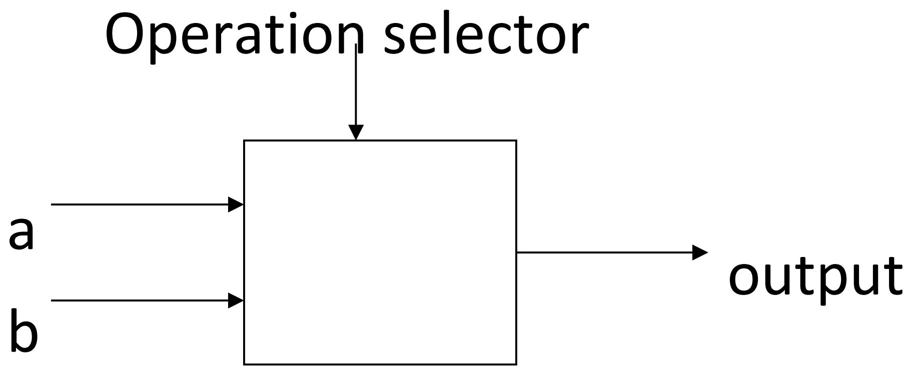
ALU

ALU definition

- **Arithmetic Logic Unit (ALU) - Hardware** that performs addition, subtraction, and usually logical operations such as AND and OR.

ALU

- Operation selector – output from multiplexor



Operation selector

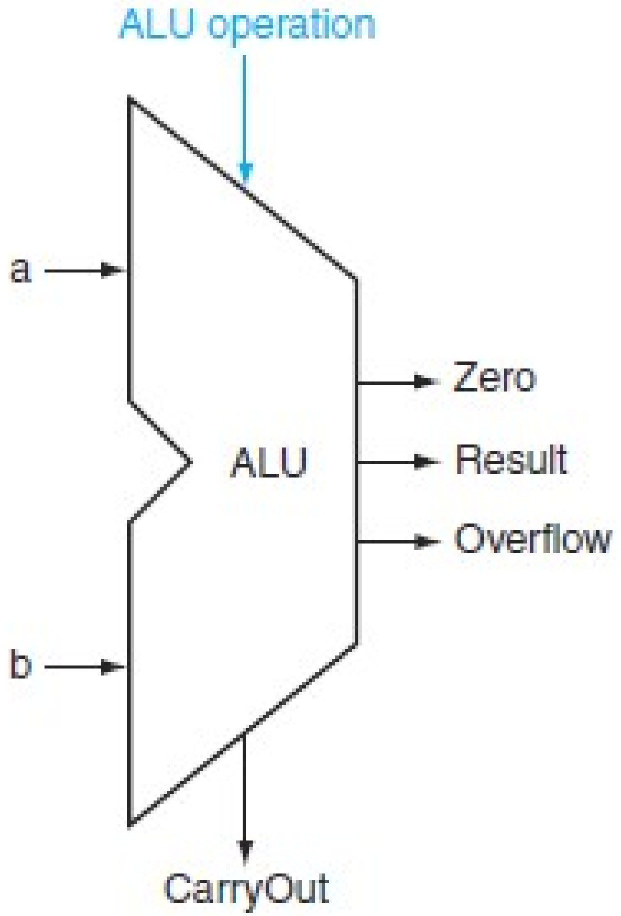
a

b

output

ALU symbol

a, b – inputs
ALU operation – operation selector
Carry out – add, sub
Zero – slt,beq,bne
Result – add, sub, and, or ….
Overflow - Exception



Agenda

- Signed Numbers
  - 1s complement
  - 2s complement
- Binary Addition
- Binary Subtraction
- Multiplication
  - Flow chart – algorithm
  - Hardware design
  - Problem

Reference

- **Chapter 2 : Instructions: Language of the Computer**
  - **2.4 Signed and Unsigned Numbers**
- **Chapter 3 : Arithmetic for Computers**
- Book - David A. Patterson and John L. Hennessey, “Computer organization and design‟, Morgan kauffman / Elsevier, Fifth edition, 2014.

Signed Numbers

- Unsigned numbers – Ex: 0, 100, 999
- Signed numbers – Ex: -100, 0 , +100
- How computers represent + or – in 0’s and 1’s?
  - 1s complement
  - 2s complement

1s complement

- Question – find the 1s complement of 10001111_2 -
- Solution – inverting the bits

$1000\ 1111_2$

$0111\ 0000_2$

2s complement

- Question – find the 2s complement of 10001111_2 -

- Solution – inverting the bits and adding 1

$1000\ 1111_2$

$0111\ 0000_2$ → 1s complement

1 → add 1

$0111\ 0001_2$

Another problem

- Question : find 1s and 2s complement for the following
  - 00000110_2
  - 00011000_2

Binary Addition

A	**B**	**A+B**	**Carry Out**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Binary Addition Problem

```
      0111 two = 7ten
+     0110 two = 6ten
-----------------------
=     1101 two = 13ten
```

Binary Subtraction Problem

- Subtraction via addition using the two’s complement representation of 6:

```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
+ 1111 1111 1111 1111 1111 1111 1111 1010two = –6ten
-----------------------------------------------------
= 0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
```

Two’s Complement Addition: Verifying Carry/Borrow method

- Two (n+1)-bit integers: $X = x_n X'$, $Y = y_n Y'$

Carry/borrow add X + Y	$0 \le X' + Y' < 2^n$ (*no CarryIn to last bit*)	$2^n \le X' + Y' < 2^{n+1} - 1$ (*CarryIn to last bit*)
$x_n = 0$, $y_n = 0$	ok	not ok(overflow!)
$x_n = 1$, $y_n = 0$	ok	ok
$x_n = 0$, $y_n = 1$	ok	ok
$x_n = 1$, $y_n = 1$	not ok(overflow!)	ok

- *Prove the cases above!*
- Prove if there is *one more bit* (total n+2 then) available for the result then there is no problem with overflow in add!

Two's Complement Operations

- Now verify the negation shortcut!
  - consider $X + (\overline{X} +1) = (X + \overline{X}) + 1$:

    associative law – but what if there is overflow in one of the adds on either side, i.e., the result is wrong…!
  - think minint !
  - Examples:
    - –0101 = 1010 + 1 = 1011
    - –1100 = 0011 + 1 = 0100
    - –1000 = 0111 + 1 = 1000

Detecting Overflow

- *No overflow* when adding a positive and a negative number
- *No overflow* when subtracting numbers with the same sign
- *Overflow occurs* when **the result has “wrong” sign** :

Operation	Operand A	Operand B	Result Indicating **Overflow**	Actual **without Overflow**
A + B	≥ 0(+ve)	≥ 0 (+ve)	< 0 (-ve)	+ve
A + B	< 0 (-ve)	< 0 (-ve)	≥ 0 (+ve)	-ve
A – B	≥ 0 (+ve)	< 0 (-ve)	< 0 (-ve)	+ve
A – B	< 0 (-ve)	≥ 0 (+ve)	≥ 0 (+ve)	-ve

- Consider the operations A + B, and A – B
  - *can overflow occur* if B is 0 ?
  - can overflow occur *if A is 0 ?*

Effects of Overflow

- If an *exception* (interrupt) occurs
  - control jumps to predefined address for exception
  - interrupted address is saved for possible resumption
- Don't always want to cause exception on overflow
  - `add, addi, sub` *cause exceptions* on overflow
  - `addu, addiu, subu` *do not cause exceptions* on overflow

Multiply

- Binary multiple of 2ten x 3ten = 6ten

- Multiplicand – 2
- Multiplier – 3
- Product - 6

Multiply

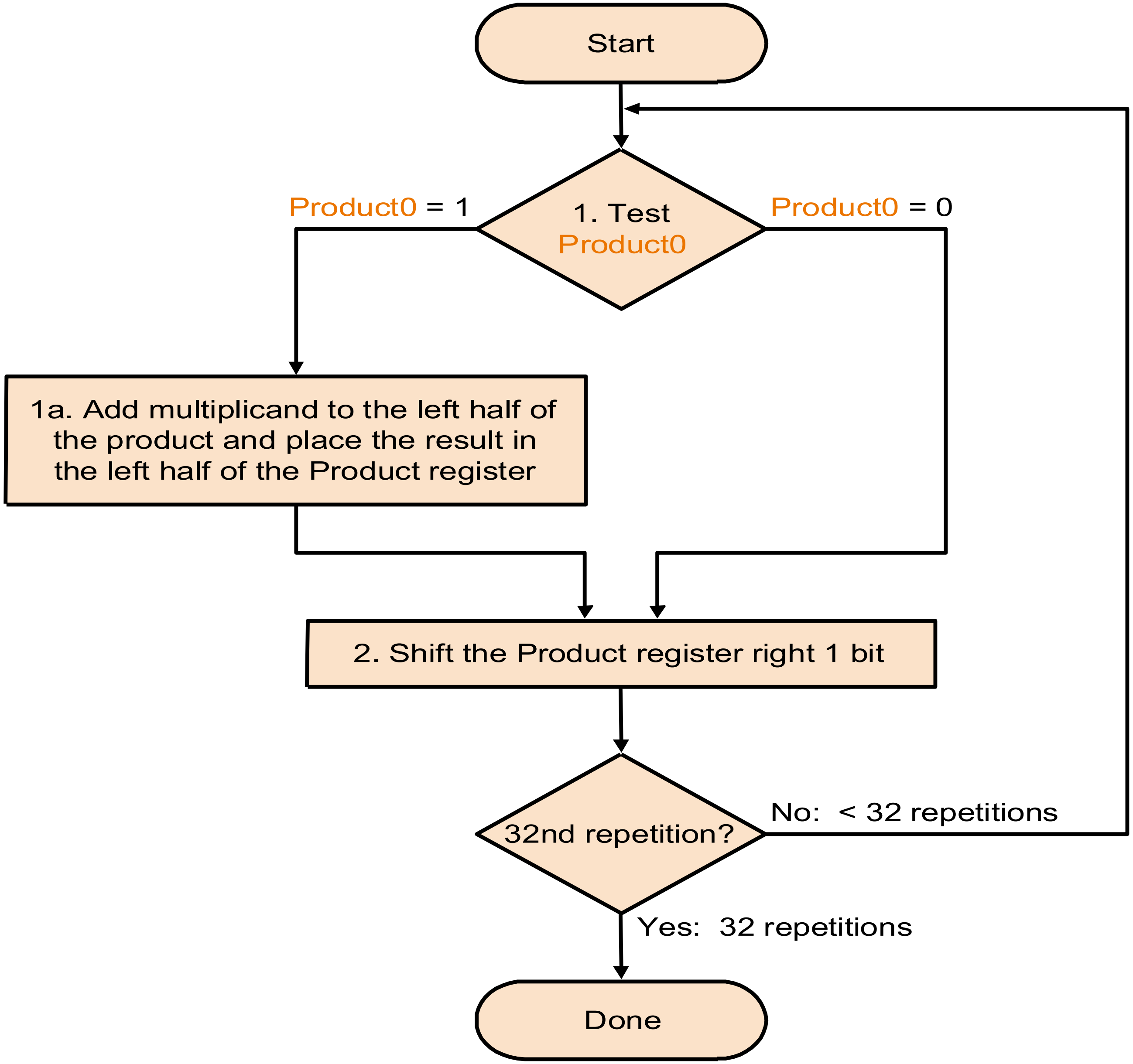
- Grade school shift-add method:

```
Multiplicand        1000
Multiplier       x1001
                  --------
                1000
                    0000
                   0000
                  1000
                --------
Product         01001000
```

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - multiplier bit 1 => copy multiplicand (1 x multiplicand)
  - multiplier bit 0 => place 0 (0 x multiplicand)

Multiplication -
Sequential Refined Version
Algorithm

Sequential(Refined) version of Multiplication Algorithm


Start
1. Test Product0
Product0 = 1
Product0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Sequential(Refined) Version of Multiplication Problem

2 x 3 = 6; 0010 x 0011 = 0110

Iteration	Steps	Multiplicand	Product
0	Initialize	0010	0000 0011
1	1a. 1=> Prod = Prod + Mcand 2. Shift Right Product	0010 0010	0010 0011 0001 0001
2	1a. 1=> Prod = Prod + Mcand 2. Shift Right Product	0010 0010	0011 0001 0001 1000
3	1a. 0=> No operation 2. Shift Right Product	0010 0010	0001 1000 0000 1100
4	1a. 0=> No operation 2. Shift Right Product	0010 0010	0000 1100 0000 0110

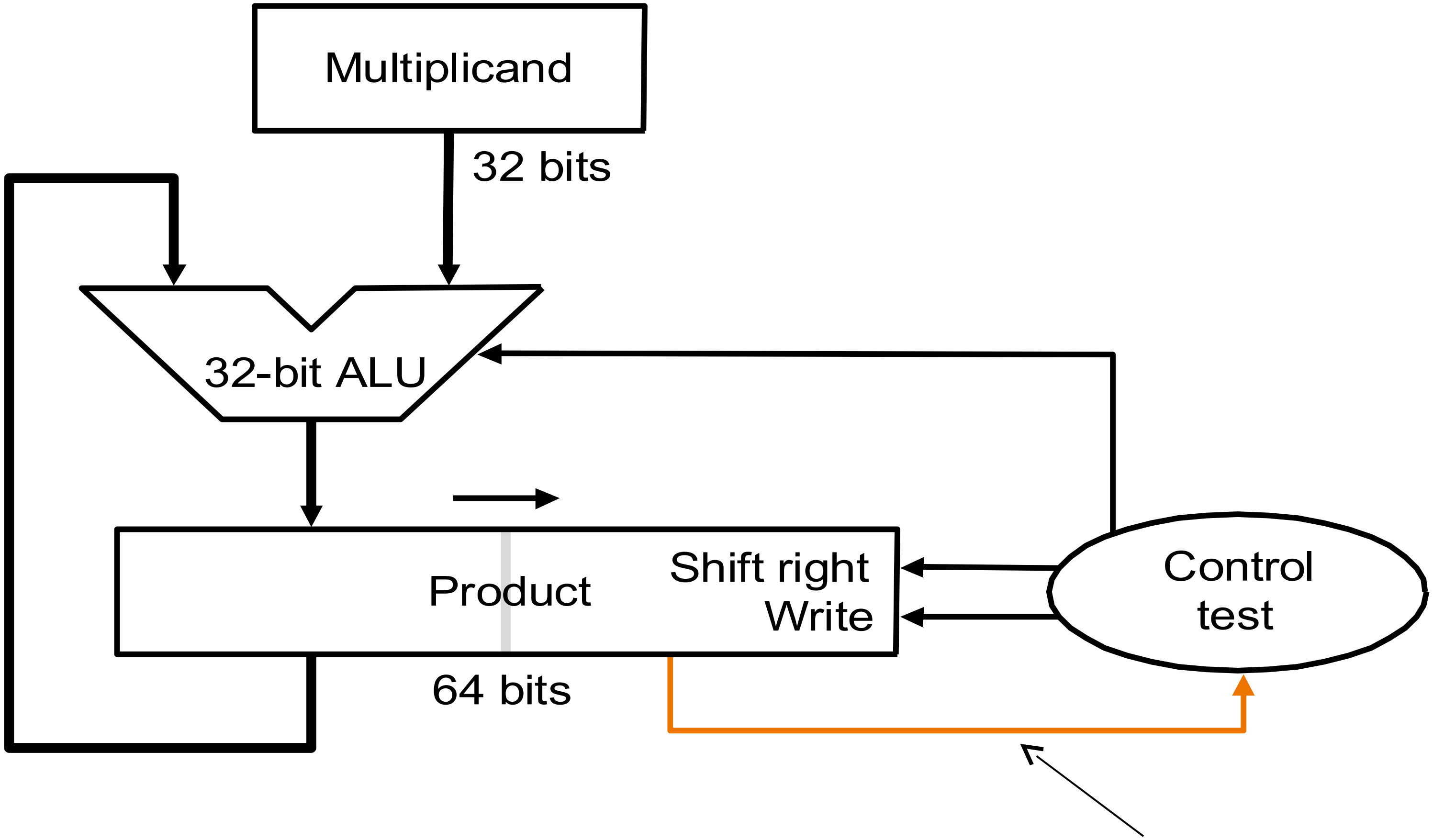
Last Class Summary

- Signed Numbers Representation
  - 1s complement
  - 2s complement
- Binary Addition
- Binary Subtraction
- Multiplication
  - Flow chart – algorithm
  - Hardware design - pending
  - Problem

Multiplication

- Two versions of multiplication:
  - Sequential refined version algorithm
  - Sequential first version algorithm
- Booth’s algorithm
  - Signed and Unsigned multiplication

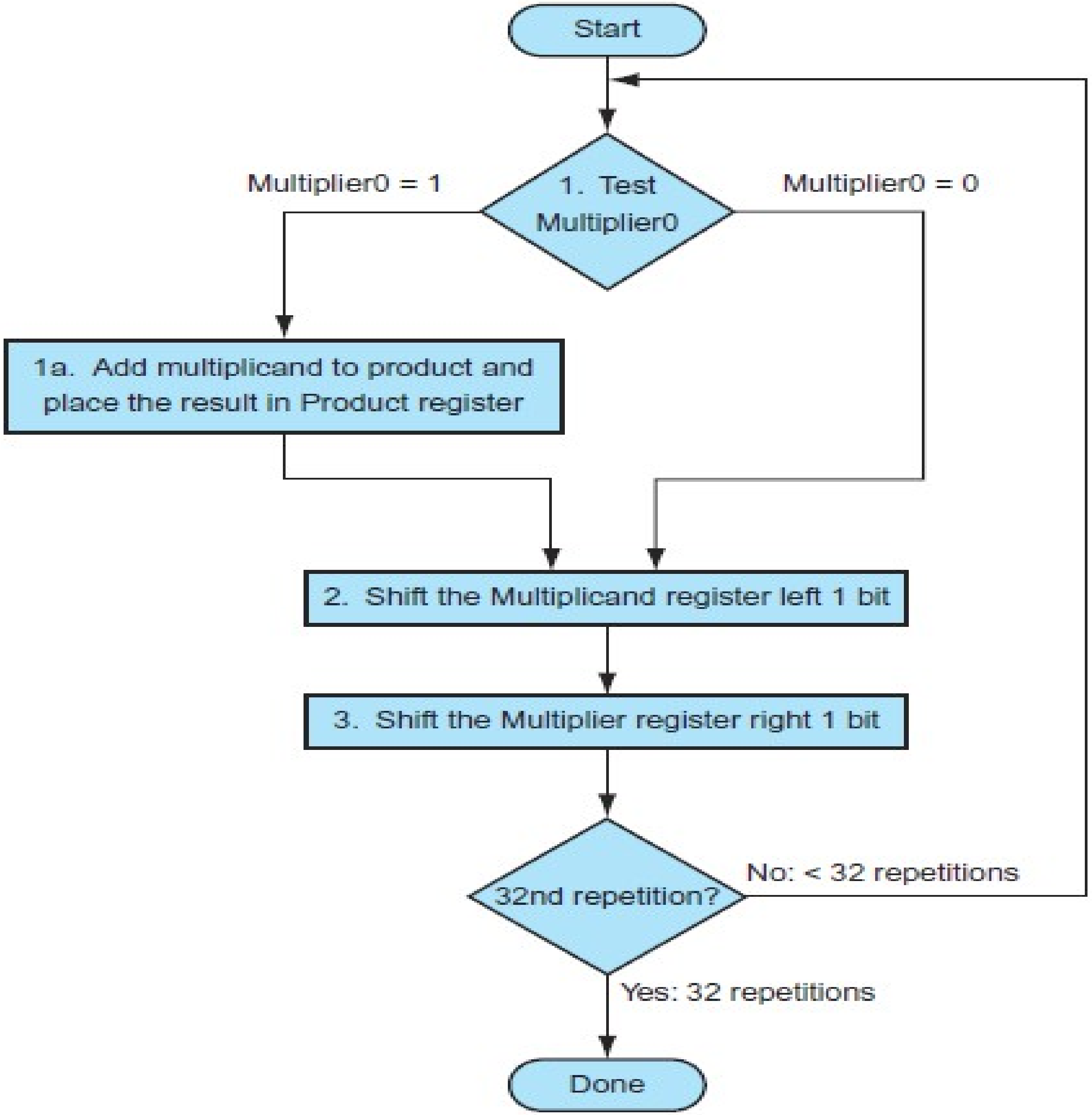
Sequential(Refined) Version of Multiplication Hardware



Product register is initialized with multiplier on right

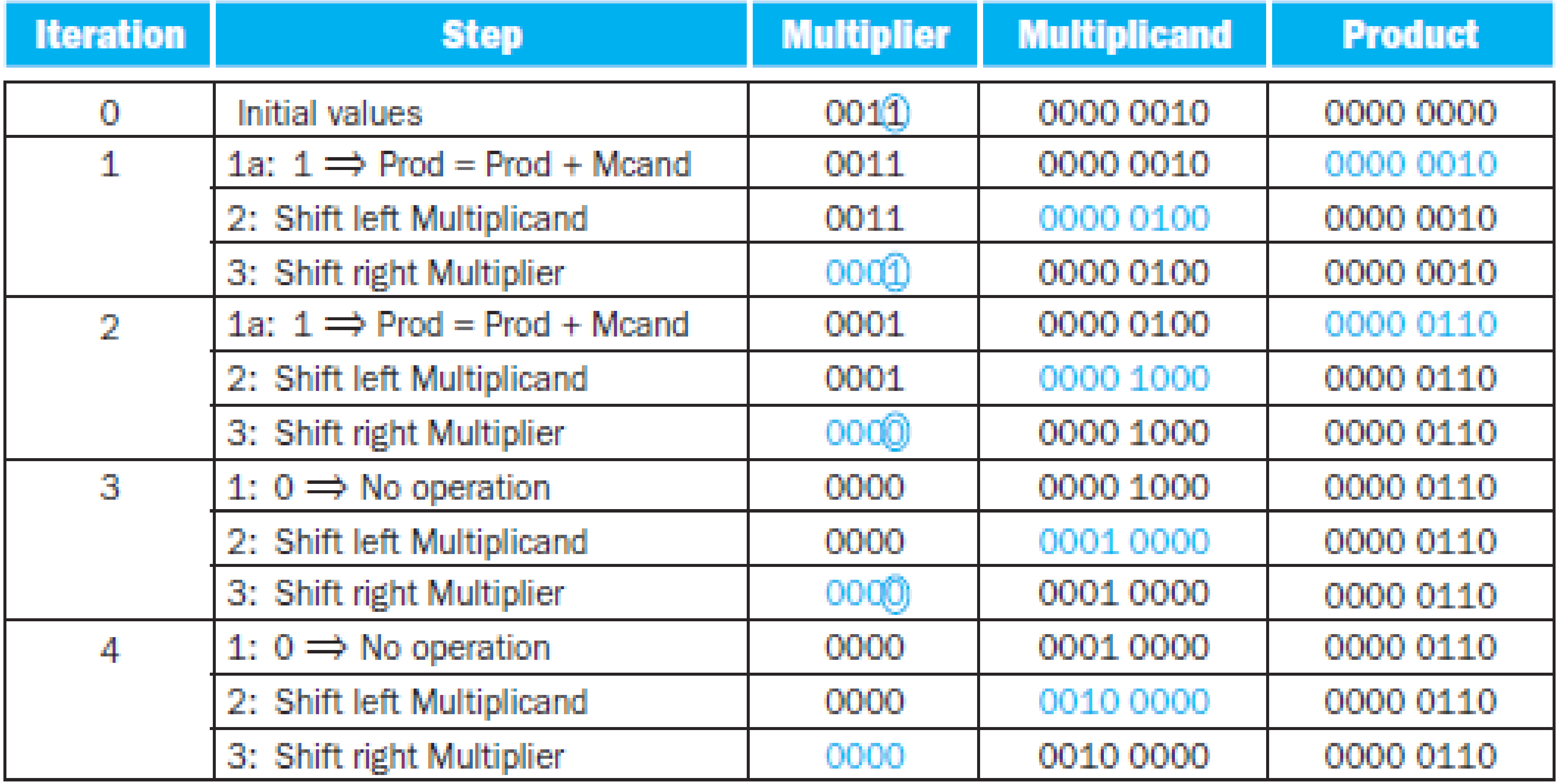
Multiplication - Sequential First Version Algorithm

Sequential First Version of Multiplication Algorithm


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

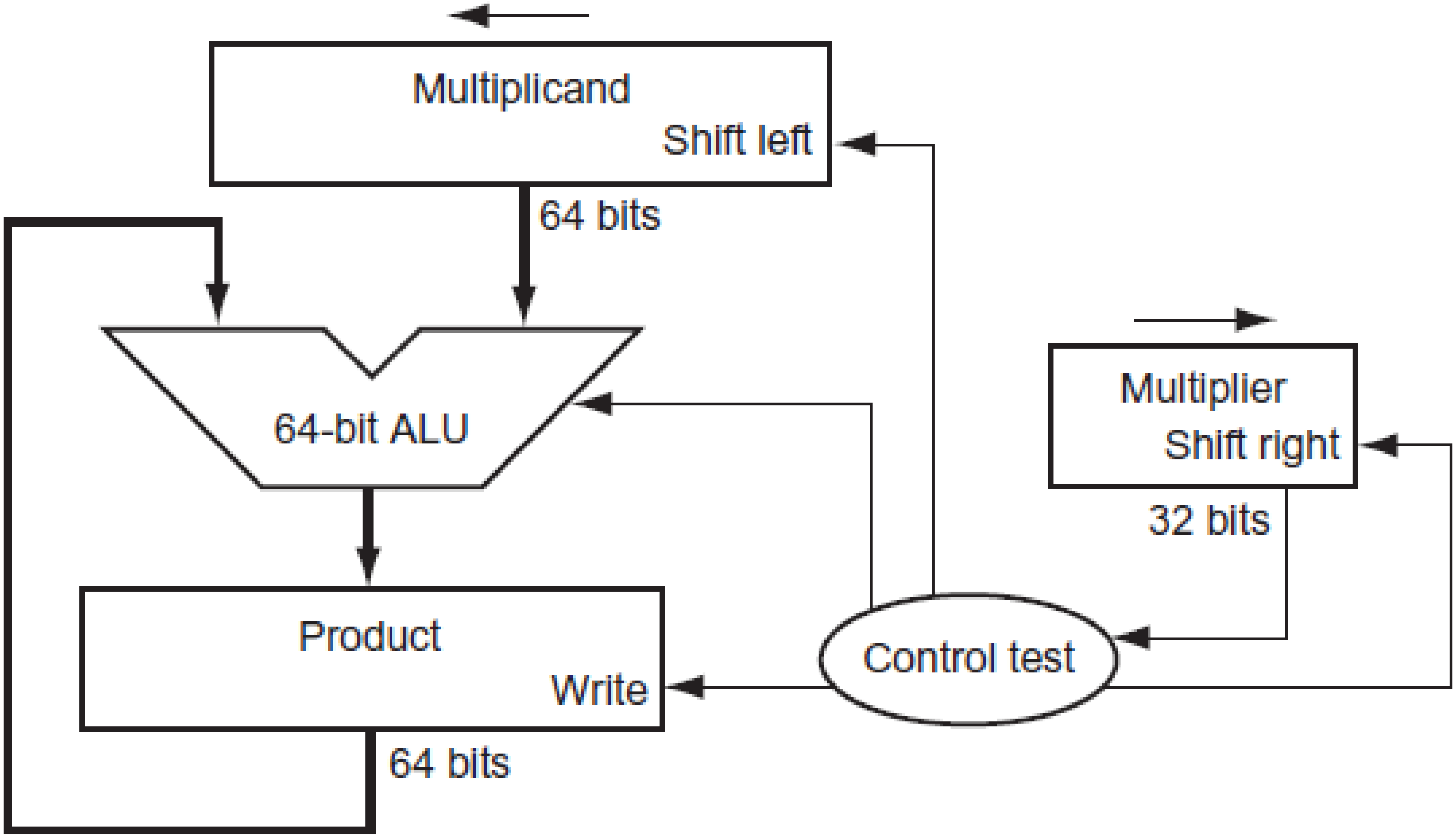
Sequential First Version of Multiplication Problem

2 x 3 = 6;   0010 x 0011 = 0110



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Sequential First version of Multiplication Hardware



Multiply in MIPS

- **2 Registers:**
- separate pair of 32-bit registers to contain the 64-bit product called ***Hi and Lo.***
- **4 Instructions:**
- multiply (***mult***)
- multiply unsigned (***multu***)
- move from lo *(**mflo**) – to place the product into registers*
- Move from hi *(***mfhi**) – *to place the product into registers*

Booth’s Algorithm – Both Signed and Unsigned Multiplication

Reference

- **Chapter 6 : Arithmetic**
  - **6.4 Signed – operand multiplication**
    - **Booth algorithm**
- Book – Carl Hamacher, “Computer organization‟, Fifth edition, Mc Graw Hill.

Booth’s algorithm

- 1st step – Find the 2s complement of negative multiplier
- 2nd step – recode the 2s complement of negative multiplier
- 3rd step – multiply multiplicand and recoded multiplier using long hand method

Basic concept - 2s complement

3 bits - Unsigned numbers – 8 - (0-7)

3 bits - Signed numbers – 8 - (-4,3,-2,-1,0,1,2,3)

3 bits - unsigned numbers	
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

3 bits - signed numbers	
100	-4
101	-3
110	-2
111	-1
000	0
001	+1
010	+2
011	+3

Example problem

- Multiply +13 x -6
- Where +13 -> multiplicand -> 5bits
- -6 -> multiplier -> 5 bits

1st step -> find the 2s complement of negative multiplier

Assume : 5bit multiplier

+6 -> 00110 (sign extension)

1s complement -> 11001

Add 1 -> 1

2s complement -> 11010 = -6

Why 5 bits for +13?

- +13 is in which range
- -13 … 0 … +13 = total 27 numbers
- Nearest power of 2 number is 32 = 2^5
- So 5 bits to represent +13 .

How 5 bits for -6?

- Find binary number of -6?
- Range of -6 is -6…0..+6 = total 13 numbers
- Nearest power of 2 is 2^4=16
- 6 -> 0110
- Invert 1001
- +1-> 1
- -6 -> 1010

11010 -> **sign extension** (Make -6 extend to 5 bits)

Problem solving

- 2nd step -> recode the 2s complement of negative multiplier

Booth multiplier recoding table:

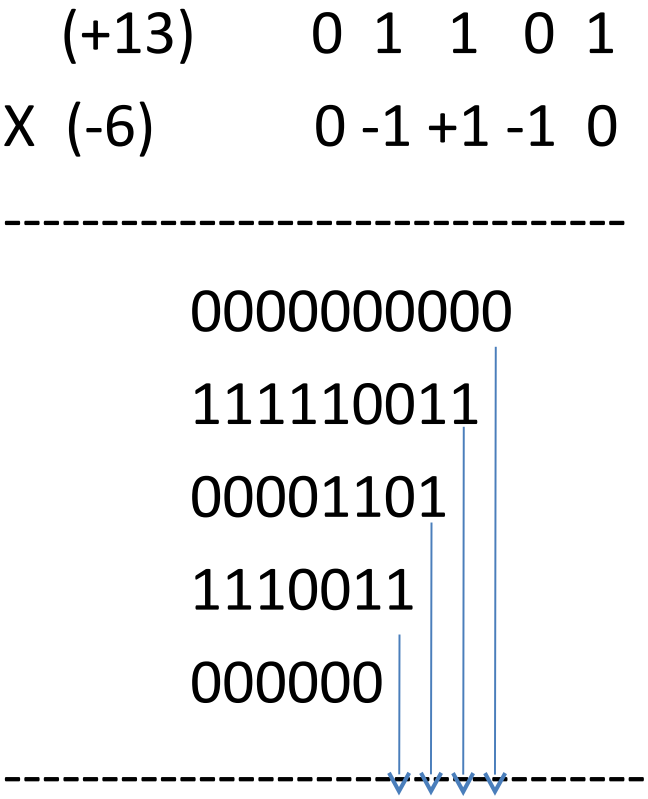
Multiplier		Version of multiplicand selected by bit i
Bit I	Bit i-1	
0	0	0 x M
0	1	+1 x M
1	0	-1 x M
1	1	0 x M

-6 => 1 1 0 1 0 (0)

booth recoded multiplier -6 => 0 -1 +1 -1 0

Booth's Algorithm Problem solving

- 3rd step -> multiply multiplicand and recoded multiplier using long hand method.



```
   (+13)        0  1   1   0  1
X  (-6)          0 -1 +1 -1  0
--------------------------------
           0000000000
           111110011
           00001101
           1110011
           000000
----------------------------------
```

(-78)   1110110010   **-> how to verify this result**

Booth’s multiplication features

- Handles both signed and unsigned multipliers uniformly.
- Faster multiplication – fewer additions

How Speed affected by multiplier??

- Worst case multiplier

0 1 0 1 0 1 0 1 0 1 0 1 0 1

+1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1

- Ordinary multiplier

1 1 0 0 0 1 0 1 1 0 1 1 1 1 0 0

0 -1 0 0 +1 -1 +1 0 -1 +1 0 0 0 -1 0 0

- Good multiplier

0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1

0 0 0 +1 0 0 0 0 -1 0 0 0 +1 0 0 -1

Computer Architecture Division

Last Class Summary

- Booth’s Algorithm

Agenda

- Booth’s algorithm example
  - Why 5 bits?
  - How to verify the result?
- Division
  - Hardware
  - Algorithm
  - Problem

Basic concept - 2s complement

3 bits - Unsigned numbers – 8 - (0-7)

3 bits - Signed numbers – 8 - (-4,3,-2,-1,0,1,2,3)

3 bits - unsigned numbers	
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

3 bits - signed numbers	
100	-4
101	-3
110	-2
111	-1
000	0
001	+1
010	+2
011	+3

Example problem

- Multiply +13 x -6
- Where +13 -> multiplicand -> 5bits
- -6 -> multiplier -> 5 bits

1st step -> find the 2s complement of negative multiplier

Assume : 5bit multiplier

+6 -> 00110 (sign extension)

1s complement -> 11001

Add 1 -> 1

2s complement -> 11010 = -6

Why 5 bits for +13?

- +13 is in which range
- -13 … 0 … +13 = total 27 numbers
- Nearest power of 2 number is 32 = 2^5
- So 5 bits to represent +13 .

How 5 bits for -6?

- Find binary number of -6?
- Range of -6 is -6…0..+6 = total 13 numbers
- Nearest power of 2 is 2^4=16
- 6 -> 0110
- Invert 1001
- +1-> 1
- -6 -> 1010

11010 -> **sign extension** **(**Make -6 extend to 5 bits)

Booth’s Algorithm Problem solving

- 3rd step -> multiply multiplicand and recoded multiplier using long hand method.

```
   (+13)       0  1   1   0  1
X  (-6)        0 -1 +1 -1  0
--------------------------------
          0000000000
          111110011
          00001101
          1110011
          000000
---------------------------------
(-78)    1110110010
```

-> how to verify this result

DIVISION

- **UNIT II ARITHMETIC OPERATIONS**
  Division

Reference

- **Chapter 3 : Arithmetic for Computers**
- Book - David A. Patterson and John L. Hennessey, “Computer organization and design‟, Morgan kauffman / Elsevier, Fifth edition, 2014.

Long Hand Divide

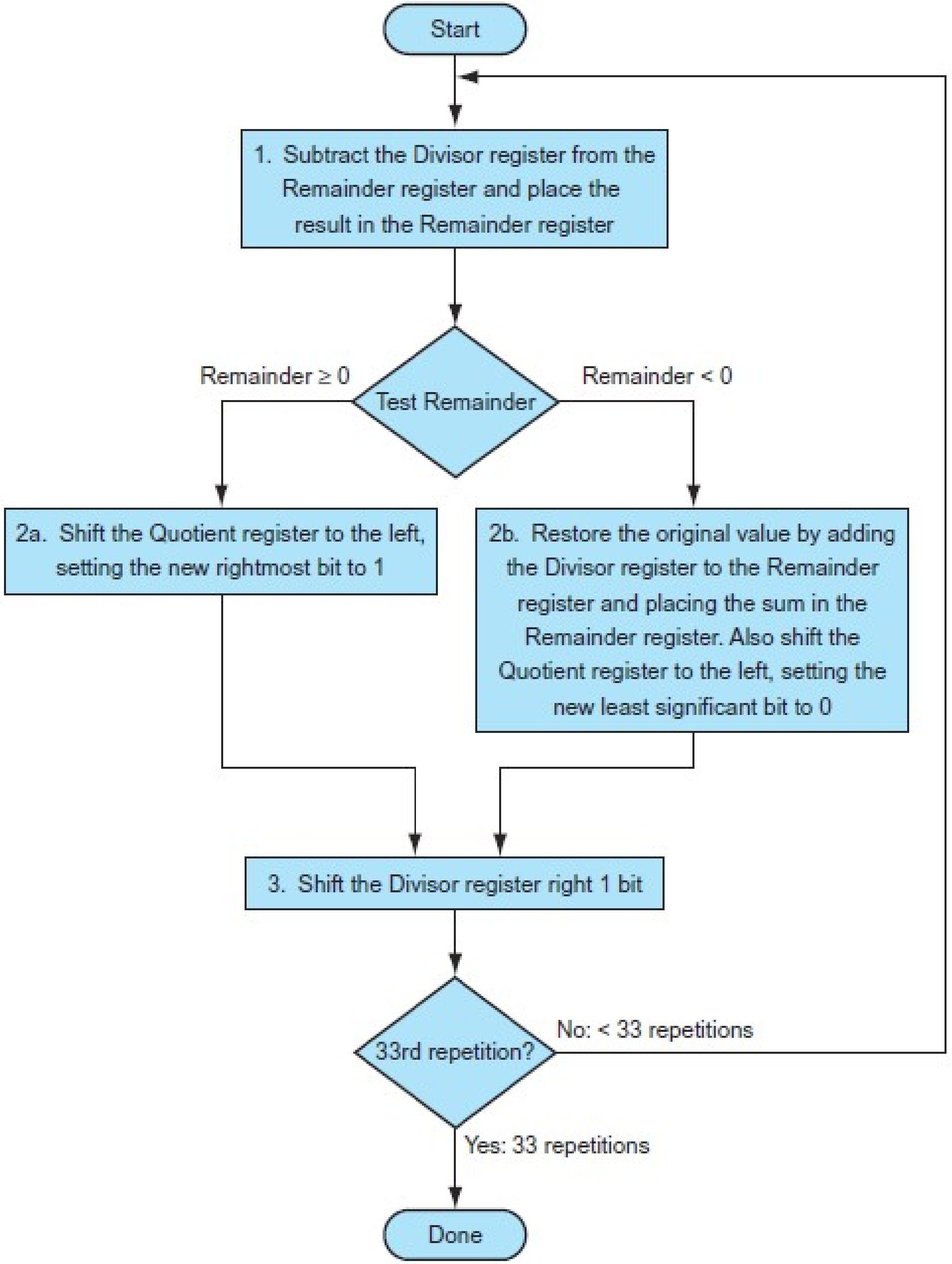
```
                   1001   Quotient
                ________
Divisor 1000   | 1001010  Dividend
               -1000
                ----
                   10
                   101
                   1010
                  -1000
                   ----
                     10   Remainder
```

- **Dividend = (Quotient * Divisor) + Remainder**

Division algorithms

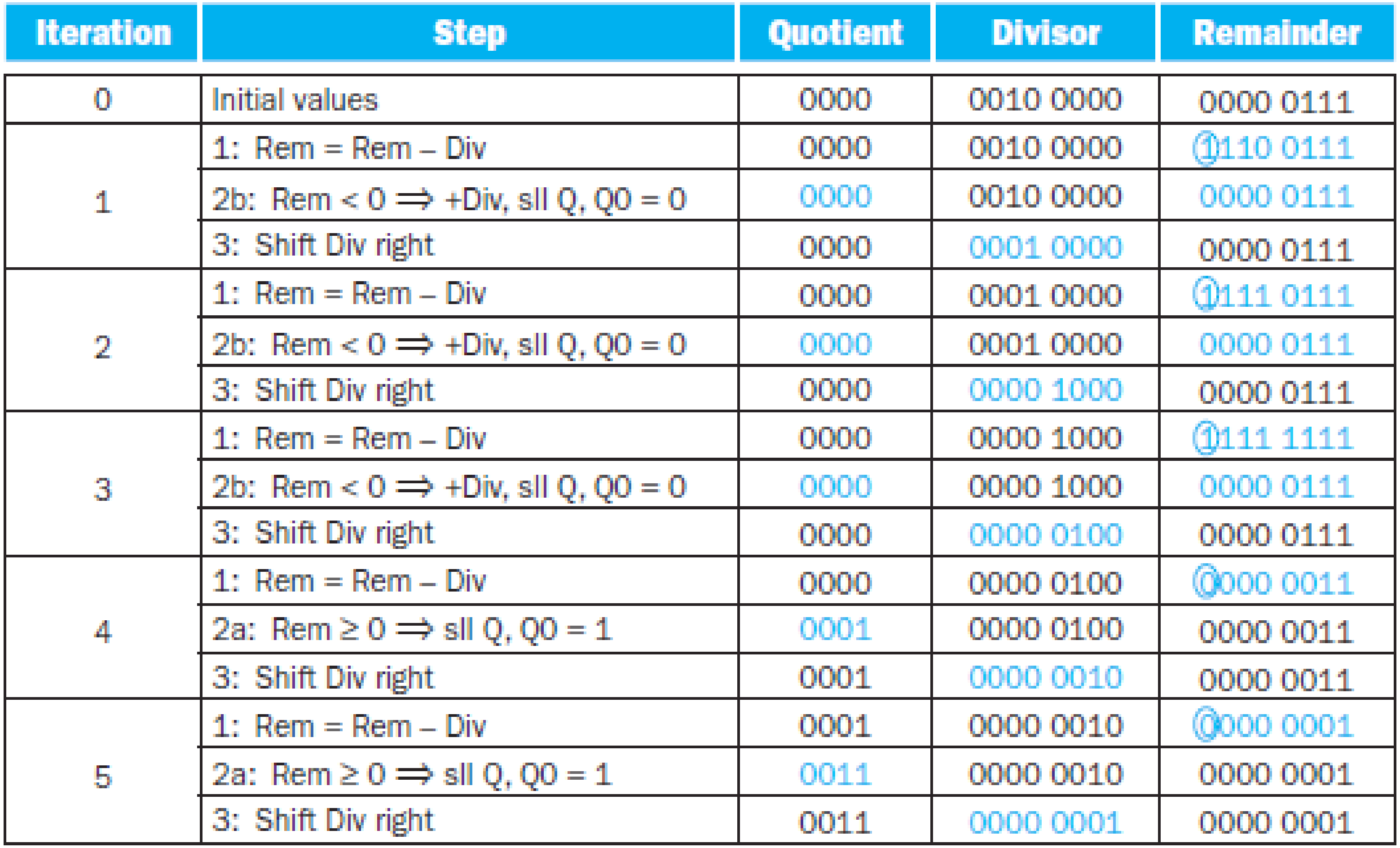
1. Restoring
   - 1st version
   - Improved version
2. Non-Restoring

Restoring Divide version 1 – Algorithm


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

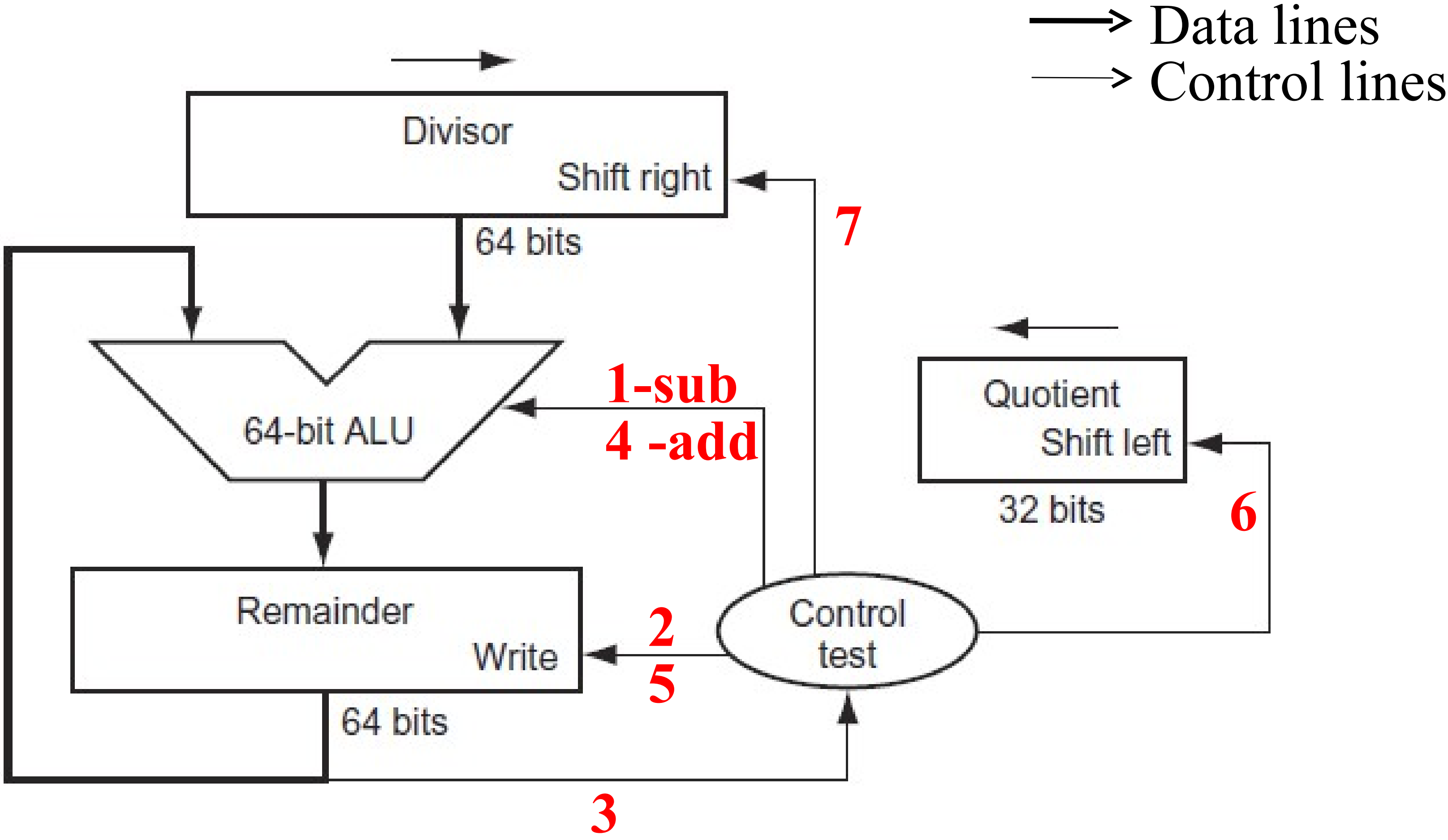
Restoring Divide Version 1 - Problem

Divide 7_{ten} by 2_{ten},
0000 0111_{two} by 0010_{two}



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Restoring Divide version 1 - Hardware

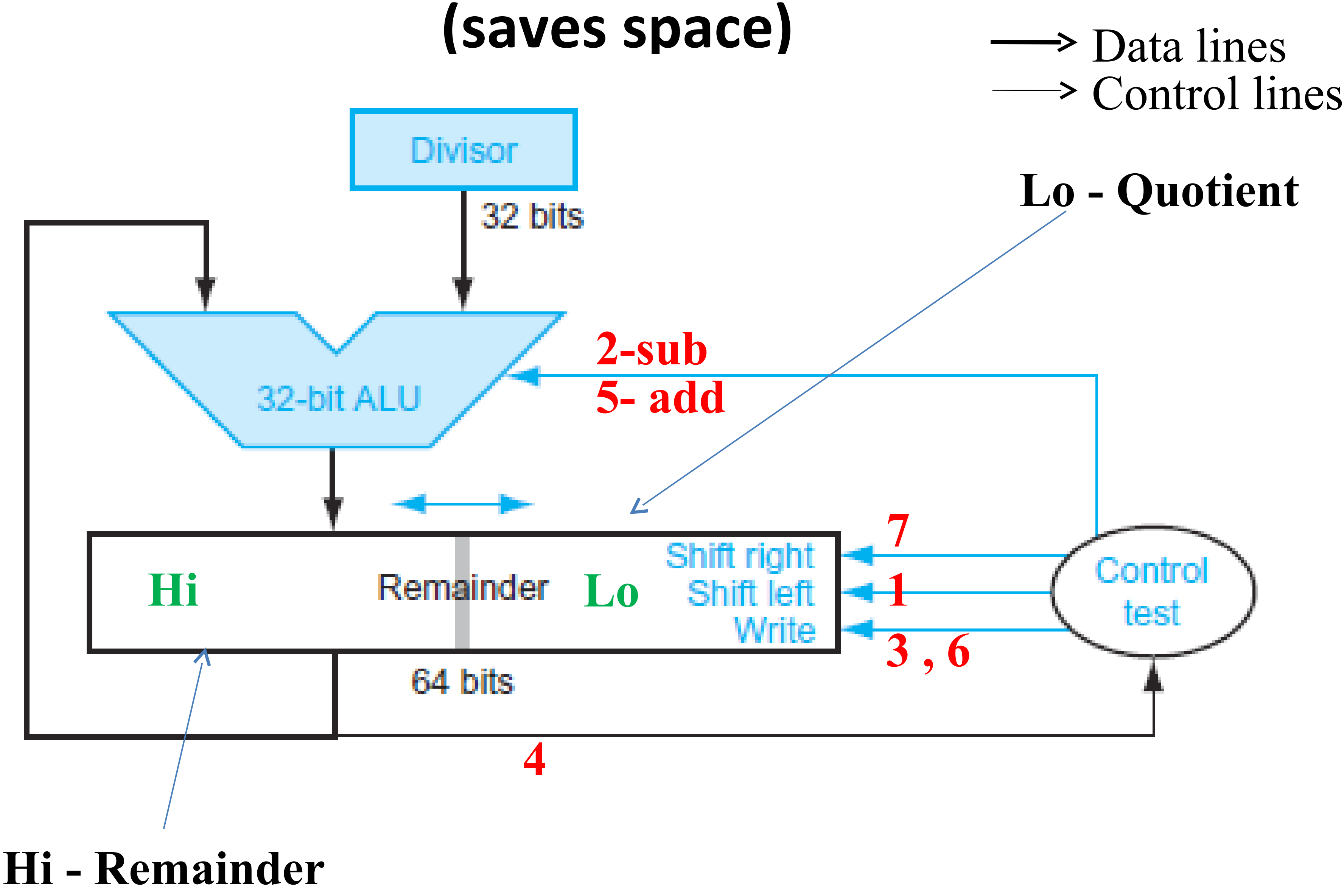


Observations on Divide Version 1

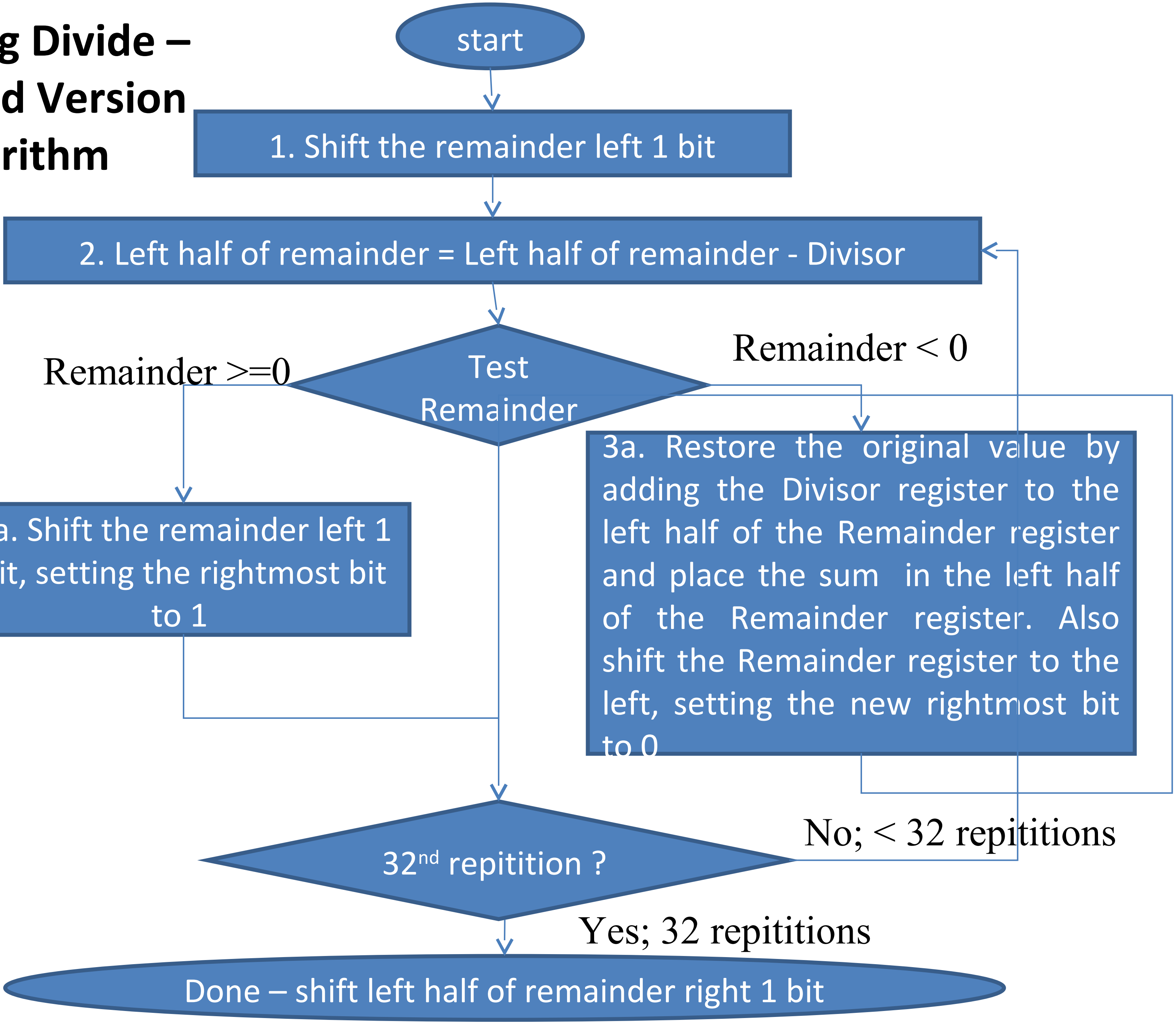
- Half the bits in divisor always 0
  - $\Rightarrow$ 1/2 of 64-bit adder is wasted
  - $\Rightarrow$ 1/2 of divisor register is wasted
- Intuition: instead of shifting divisor to right, shift remainder to left…

- Step 1 cannot produce a 1 in quotient bit – as all bits corresponding to the divisor in the remainder register are 0 (remember all operands are 32-bit)
- Intuition: switch order to shift first and then subtract – can save 1 iteration…

Restoring Divide – Improved Version (saves space)



Restoring Divide – Improved Version Algorithm



start

1. Shift the remainder left 1 bit

2. Left half of remainder = Left half of remainder - Divisor

Test Remainder

Remainder >=0

Remainder < 0

3a. Shift the remainder left 1 bit, setting the rightmost bit to 1

3a. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

32nd repitition ?

No; < 32 repititions

Yes; 32 repititions

Done – shift left half of remainder right 1 bit

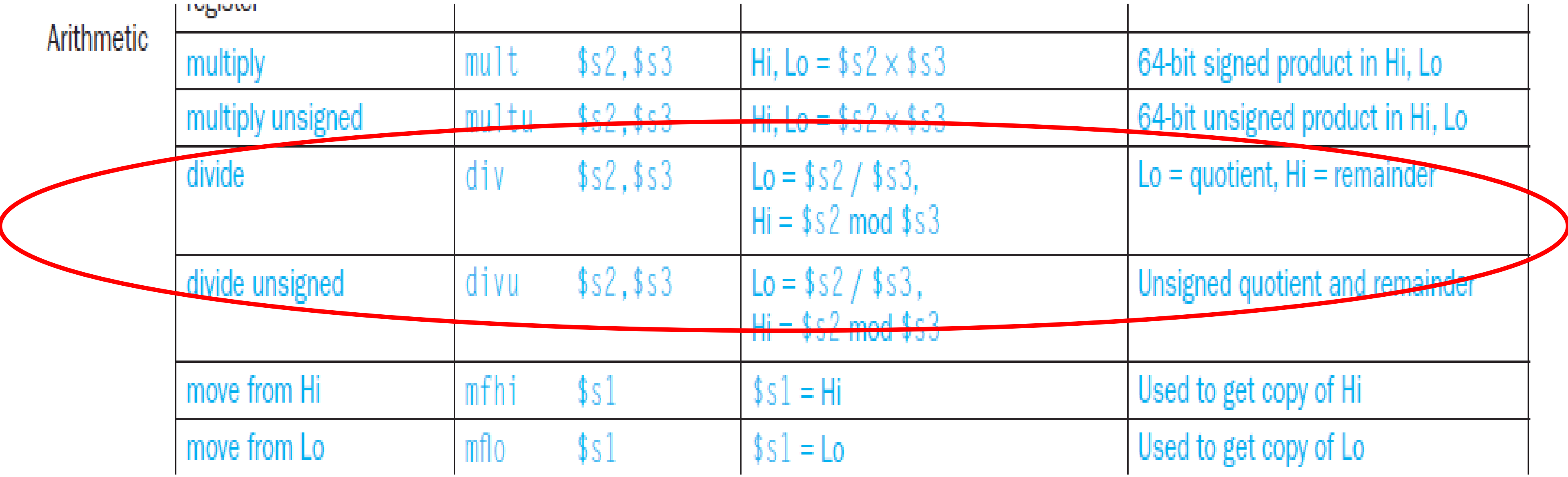
Observations on Divide Improved Version

- **Signed divide:**
  - make both divisor and dividend positive and perform division
  - negate the quotient if divisor and dividend were of opposite signs
  - make the sign of the remainder match that of the dividend
  - this ensures always
    - dividend = (quotient * divisor) + remainder
    - –quotient (x/y) = quotient (–x/y) (e.g. 7 = 3*2 + 1 & –7 = –3*2 – 1)
- **Faster divide:**
  - Based on Prediction and Moore's Law.
  - SRT division

MIPS instructions

- div (signed), divu (unsigned) –
  - with two 32-bit register operands,
  - put **remainder in Hi register and quotient in Lo**;
  - overflow is ignored in both cases
- mflo – place the quotient from Lo to general purpose register
- mfhi - place the remainder from Hi to general purpose register
- Example; `div $s2, $s1          #$s2/Ss1`
- `                                # Lo -quotient`
- `                                # Hi - Remainder`
- `      mflo $s3     # $s3 = Lo = quotient`
- `      mfhi $s4    #  $s4 = Hi = remainder`

MIPS instructions



	register			
Arithmetic	multiply	mult $s2,$s3	Hi, Lo = $s2 x $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 x $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

Non-Restoring Division

Reference:

Chapter 6 : Arithmetic

6.6 Integer Division - Nonrestoring Division

Book – Carl Hamacher, “Computer organization‟, Fifth edition, Mc Graw Hill.

Non-Restoring Divide - Algorithm

- Do the following n times:
  1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
  2. Now, if the sign of A is 0, set q0 to 1; otherwise, set q0 to 0.

Non restoring divide - problem

- Divide 8 by 3;
- Q - 8 – Dividend - 1000
- M - 3 – Divisor – 00011
- A – 00000 initially
- Where,
- A, Q, M are registers
- A and M are n+1 bits
- Q is n bits

Solution: M = 00011

Iteration	Steps	A	Q
0	Intial	00000	1000
1	1. sign(A) = 0 => a) shift A and Q left.	00001	0000
	b) A = A – M	11110	0000
	2. Sign(A) = 1 => Q0 = 0	11110	0000
2	1. sign(A) = 1 => a) shift A and Q left.	11100	0000
	b) A = A + M	11111	0000
	2. Sign(A) = 1 => Q0 = 0	11111	0000
3	1. sign(A) = 1 => a) shift A and Q left.	11110	0000
	b) A = A + M	00001	0000
	2. Sign(A) = 0 => Q0 = 1	00001	0001
4	1. sign(A) = 0 => a) shift A and Q left.	00010	0010
	b) A = A - M	11111	0010
	2. Sign(A) = 1=> Q0 = 0	11111	0010

Quotient – Q – 0010

Remainder = A + M = 11111 + 00011 = 00010

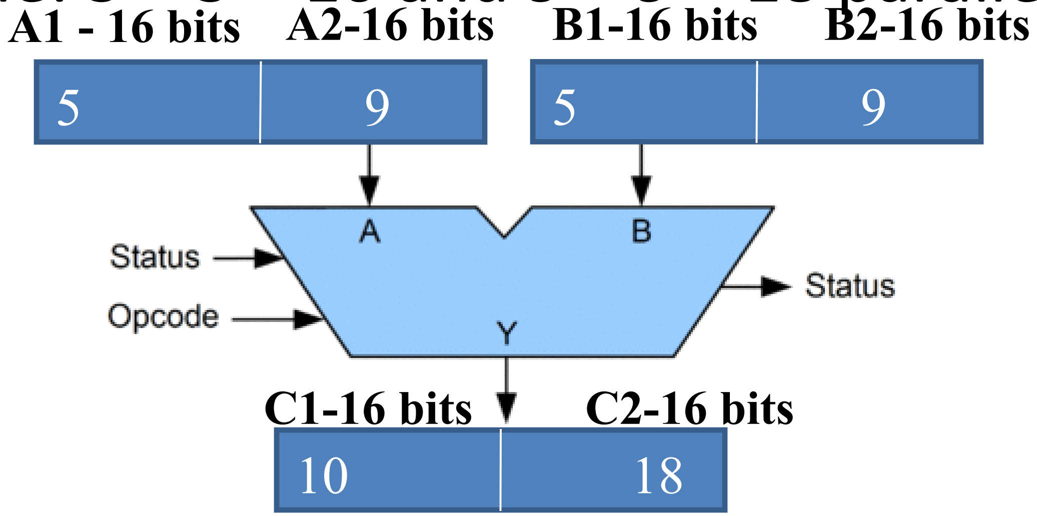
Sub Word Parallelism

SubWord Parallelism

- A subword is a lower precision unit of data contained within a word.
- In subword parallelism, multiple subwords are packed into a word and then process whole words.
- Since the same instruction is applied to all subwords within the word, This is a form of **SIMD(Single Instruction Multiple Data)** processing.
- ***Sub word parallelism or data level parallelism or vector***

SubWord Parallelism

- A 32 bit processor simultaneously execute operations on 4 eight bit operands or 2 sixteen bit operands.
- Example: 5 + 5 = 10 and 9 + 9 = 18 parallel operation



A1 - 16 bits A2-16 bits B1-16 bits B2-16 bits

5 9 5 9

C1-16 bits C2-16 bits

10 18

Application of Sub-Word parallelism

- Used in multimedia operations
  - which has many sub-word arithmetic operations (8bit , 16bit and so on)
- ARMv7,ARMv8 – processors have NEON instructions that support sub-word parallelism
- Example NEON instructions: (128 bit registers)

VADD.F32 - adds 4 32-bit data simultaneously

VMULL.S8 – multiplies 16 8-bit data simultaneously

Floating Point Operations

Floating Point

We need a way to represent

- numbers with fractions, e.g., 3.1416
- very small numbers (in absolute value),

  e.g., 0.00000000023
- very large numbers (in absolute value) ,

  e.g., –3.15576 * 10^{46}

Floating point

- Eaxmple: –3.15576 * 10^{46}
- Sign = negative – 0 bit
- Fraction = 15576
- Significand = 3.15576
- Exponent = 46

Floating Point - Representation

- ***Scientific Notation:***

  A notation that renders numbers with a single digit to the left of the decimal point.

  Convert to scientific notation 00.001_{two} x 2^{-2}

  => 0.0001_{two} x 2^{-2+1} => 0.0001_{two} x 2^{-1}

- ***Move n bits to right – add n to exponent***

Floating Point - Representation

- ***Normalized scientific representation:***

  A number in floating-point notation that has no leading 0s.

  Convert to normalized scientific notation

  $0.0001_{two} \text{ x } 2^{-1}$

  $\Rightarrow 1.0_{two} \text{ x } 2^{-1-4}$ $\Rightarrow 1.0_{two} \text{ x } 2^{-5}$

- ***Move n bits to left – subtract n from exponent***

IEEE 754 Floating-point Standard

- IEEE 754 floating point standard:
  - **single precision: one word**

31	bits 30 to 23	bits 22 to 0
sign	8-bit exponent	23-bit significand

Range of single precision:

Smallest number is -> 2.0_{ten} x 10^{-38}

Largest number is -> 2.0_{ten} x 10^{38}

Overflow – *when positive exponent becomes too large to fit in exponent field*

Underflow - *when negative exponent becomes too large to fit in exponent field*

IEEE 754 Floating-point Standard

– **double precision: two words**

31	bits 30 to 20	bits 19 to 0
sign	11-bit exponent	upper 20 bits of 52-bit significand

bits 31 to 0
lower 32 bits of 52-bit significand

Range of double precision:

Smallest is -> 2.0_{ten} x 10^{-308}

Largest is -> 2.0_{ten} x 10^{308}

General representation of IEEE 754 Floating-point Standard

$$(-1)^{\text{sign}} * (1 + \text{fraction}) * 2^{\overbrace{(\text{exponent} - \text{bias})}^{\text{equals biased exponent value}}}$$

- bias =127 for single precision and
- bias =1023 for double precision

- **Biased exponent = exponent - bias**

IEEE 754 Floating-point Standard

- **Sign bit** is 0 for positive numbers, 1 for negative numbers

- **Significand:**

  Number is assumed normalized and leading 1 bit of significand left of binary point (for non-zero numbers) is *assumed* and not shown

  - e.g., significand 1.1001… is represented as 1001…,

  - value = $(-1)^{\text{sign}} * (1 + \text{fraction}) * 2^{\text{exponent value}}$

IEEE 754 Floating-point Standard

- **Exponent** is *biased*
  - bias of 127 for single precision and 1023 for double precision
  - value = $(-1)^{\text{sign}} * (1 + \text{significand}) * 2^{\overbrace{(\text{exponent} - \text{bias})}^{\text{equals biased exponent value}}}$
  - **Biased exponent = exponent - bias**

Example problem

- **Represent –0.75$_{\text{ten}}$ in IEEE 754 single precision**

Solution:
Step1 : Convert decimal floating to binary normalized
scientific floating point
Step2 : Write the general representation of IEEE 754
Step3 : Find the sign, fraction, biased exponent
Stpe4 : Draw the IEEE 754 single precision format

Example problem

Step1 : Convert decimal floating to binary normalized scientific floating point

decimal: $-0.75 = -3/4 = -3/2^2$

binary: $-11\ \mathrm{x}\ 2^{-2} = -.11 = -1.1\ \mathrm{x}\ 2^{-1}$

Step2 : Write the general representation of IEEE 754

$(-1)^{\text{sign}} * (1 + \text{fraction}) * 2^{(\text{exponent} - \text{bias})}$

Step3 : Find the sign, fraction, biased exponent

Sign = 1

Fraction = 1_{two} =

Significand = 1.1_{two}

Biased exponent = -1_{ten}

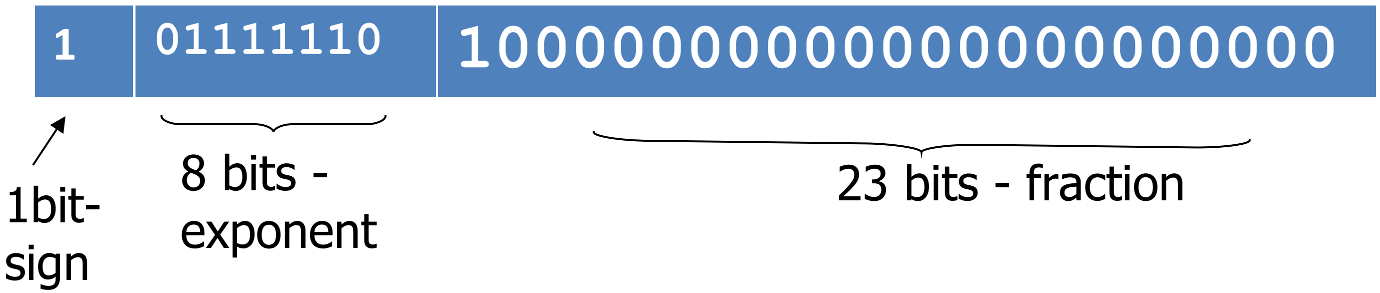
Biased exponent = exponent – bias

Exponent = biased exponent + bias

$= (-1) + 127 = 126_{\text{ten}} = 01111110_{\text{two}}$

Example problem

Stpe4 : Draw the IEEE 754 single precision format



1

01111110

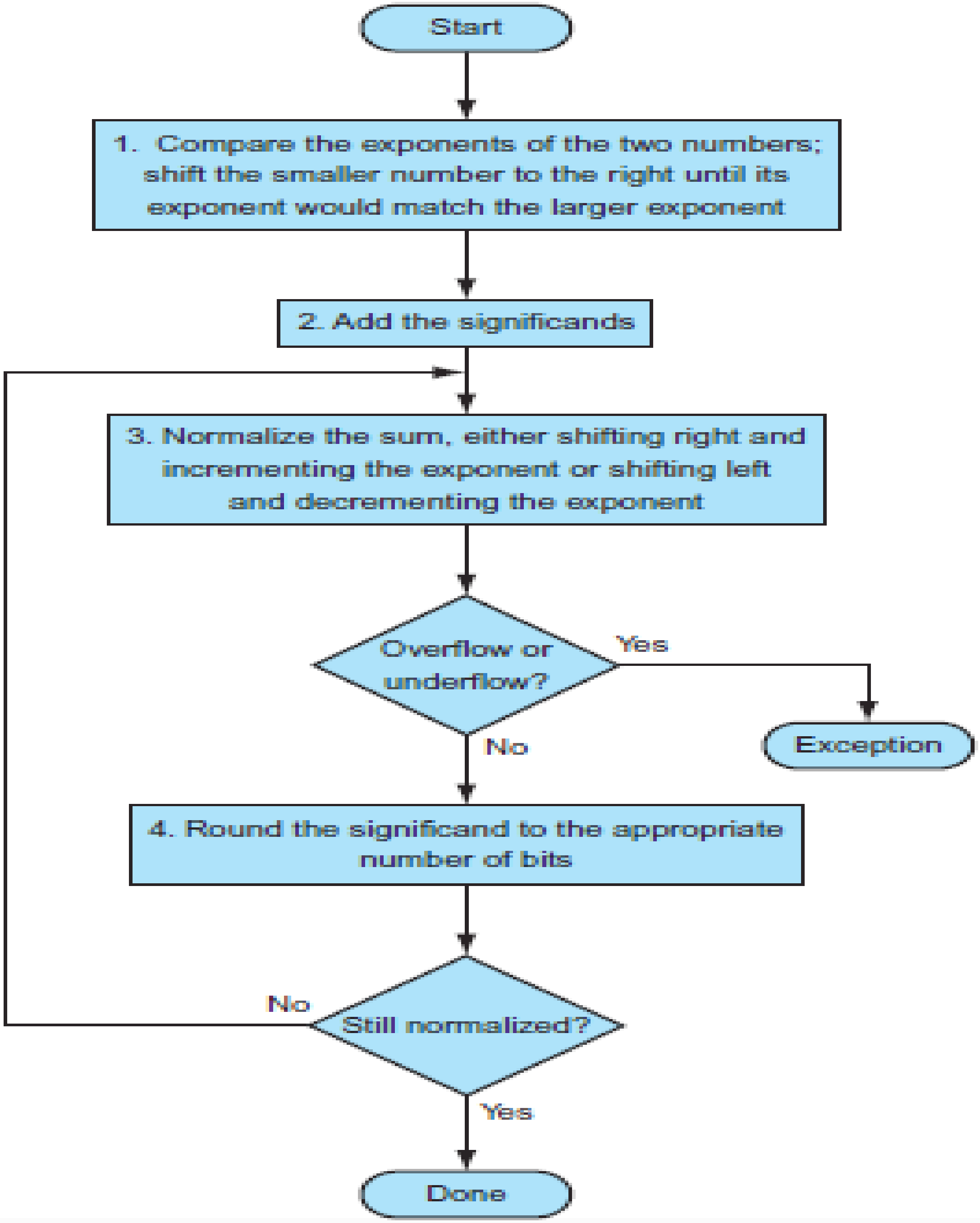
10000000000000000000000

1bit- sign

8 bits - exponent

23 bits - fraction

Floating Point Addition Algorithm



Start

1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent

2. Add the significands

3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent

Overflow or underflow?

Yes

Exception

No

4. Round the significand to the appropriate number of bits

Still normalized?

No

Yes

Done

Floating point addition problem

- Try adding the numbers 0.5ten and 0.4375ten in binary using the algorithm

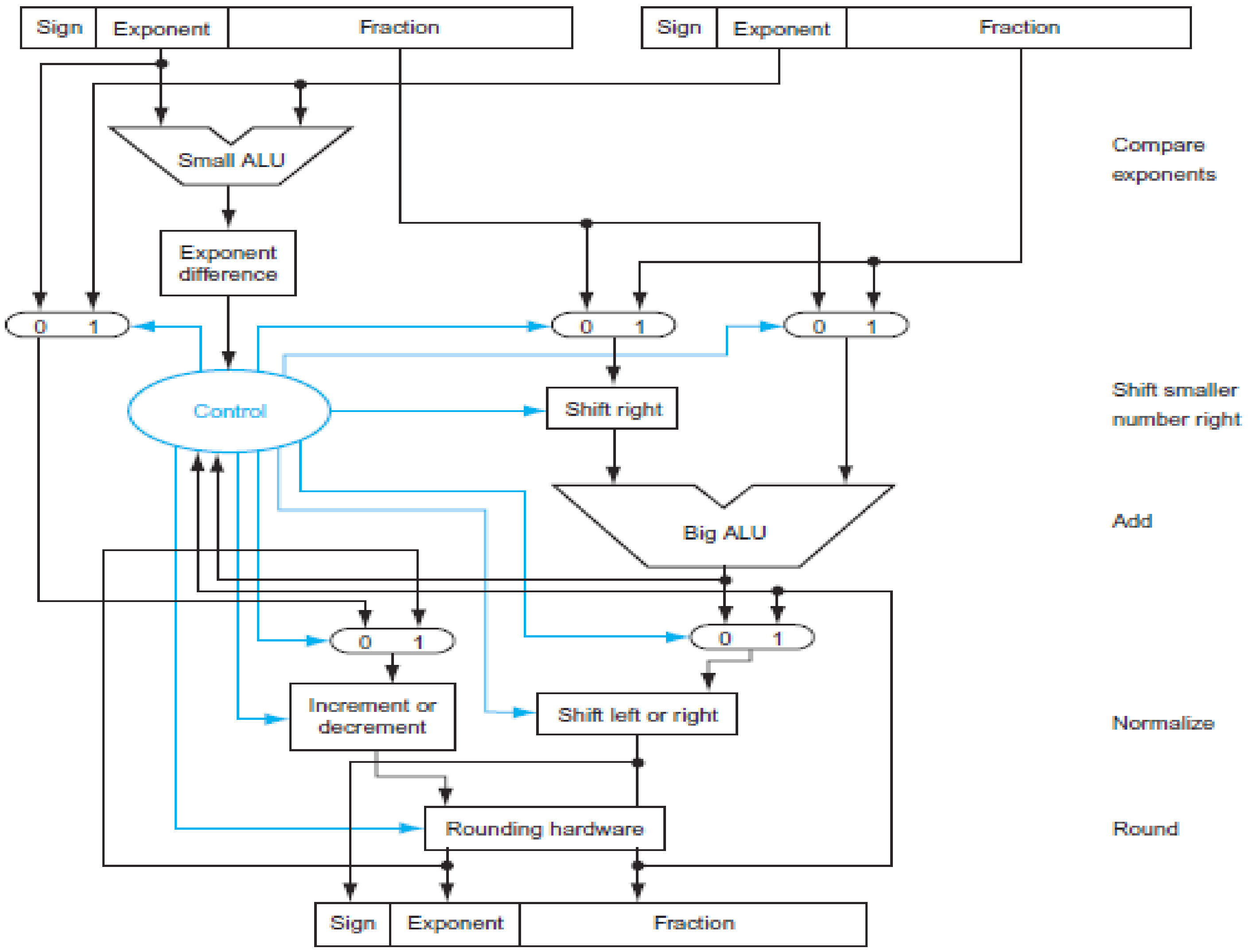
1^{st} - Convert the decimal numbers to binary numbers.

- $0.5_{ten} = ½_{ten} = 01_{two} \times 2^{-1} = 0.1_{two} = 1.0_{two} \times 2^{-1}$
- $0.4375_{ten} = 7/16_{ten} = 7/2^{4}_{ten}$

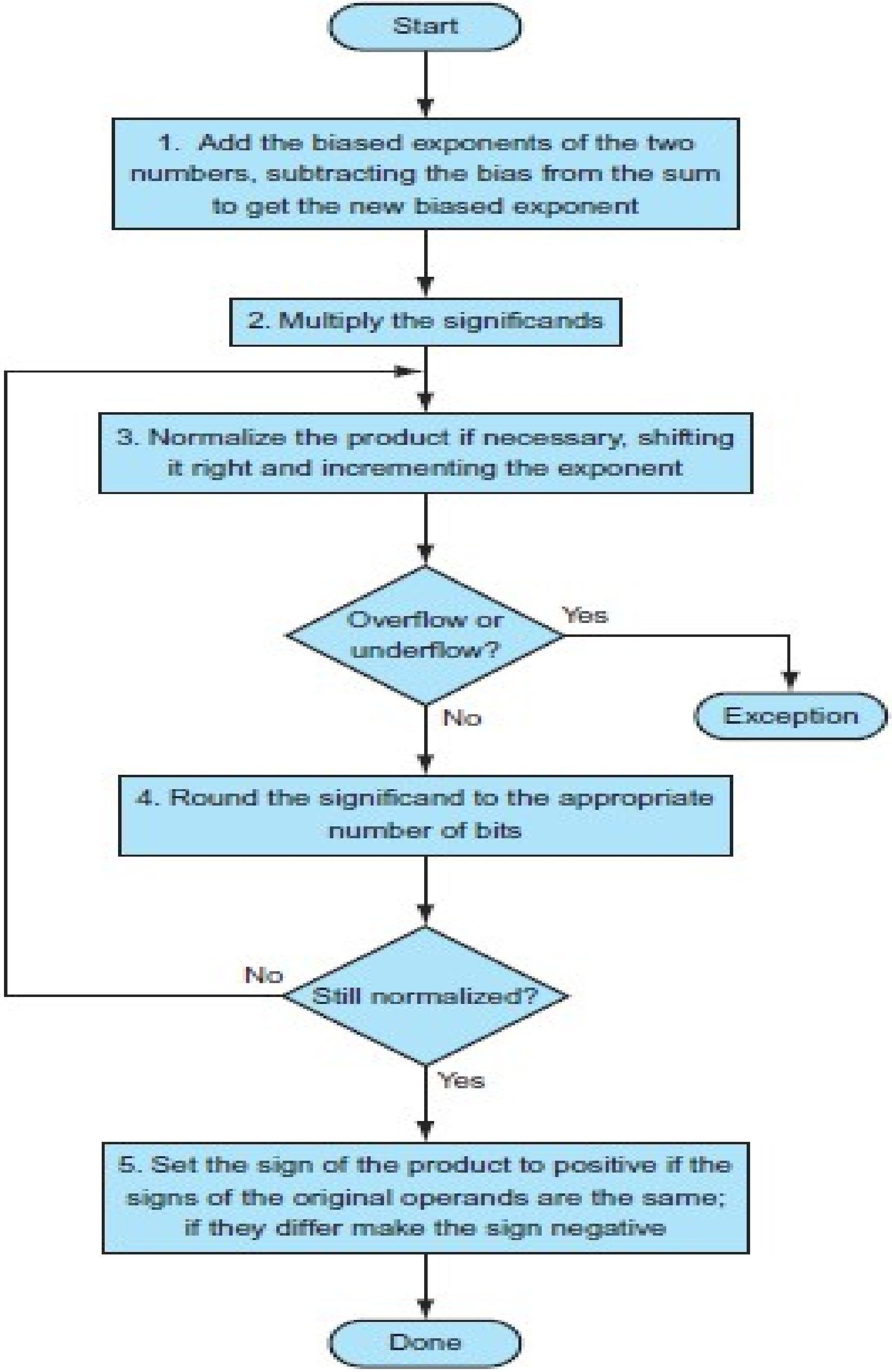
$= 111_{two} \times 2^{-4} = 0.0111_{two}$

$= 1.110_{two} \times 2^{-2}$

Floating Point Addition Hardware



Floating Point Multiplication Algorithm



Start

1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent

2. Multiply the significands

3. Normalize the product if necessary, shifting it right and incrementing the exponent

Overflow or underflow?

Yes

Exception

No

4. Round the significand to the appropriate number of bits

No

Still normalized?

Yes

5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative

Done

Floating Point Complexities

- In addition to *overflow* we can have *underflow* (number too small)

2.0_{ten} x 10^{-308} – smallest

2.0_{ten} x 10^{308} – largest

- *Accuracy* is the problem with both overflow and underflow because we have only a finite number of bits to represent numbers that may actually require arbitrarily many bits
  - limited precision $\Rightarrow$ rounding $\Rightarrow$ rounding error
  - IEEE 754 keeps *two extra bits*, *guard* and *round*

Usage of guard and round bits

- $2.56_{ten} \times 10^0 + 2.34_{ten} \times 10^2$
- Assume only 3 significand bits
- 1^{st} step : $2.56_{ten} \times 10^0 = 0.02\color{red}{56}_{ten} \times 10^2$

guard round

- 2^{nd} step :

$0.0256_{ten} \times 10^2$

$2.3400_{ten} \times 10^2$

$2.3656_{ten} \times 10^2$

- 3^{rd} step : 2.37 x 10^2

Without guard and round bits

- $2.56_{ten} \times 10^0 + 2.34_{ten} \times 10^2$
- Assume only 3 significand bits
- 1st step : $2.56_{ten} \times 10^0 = 0.02_{ten} \times 10^2$
- 2nd step :

$0.02_{ten} \times 10^2$

$2.34_{ten} \times 10^2$

$2.36_{ten} \times 10^2$

- 3rd step : $2.36_{ten} \times 10^2$

Rounding error

- **units in the last place (ulp) - The number of** bits in error in the least significant bits of the significand between the actual number and the number that can be represented.

With guard and round bits - $2.37_{ten} \times 10^2$

Without guard and round bits - $2.36_{ten} \times 10^2$

so here - it is off by 1 ulps

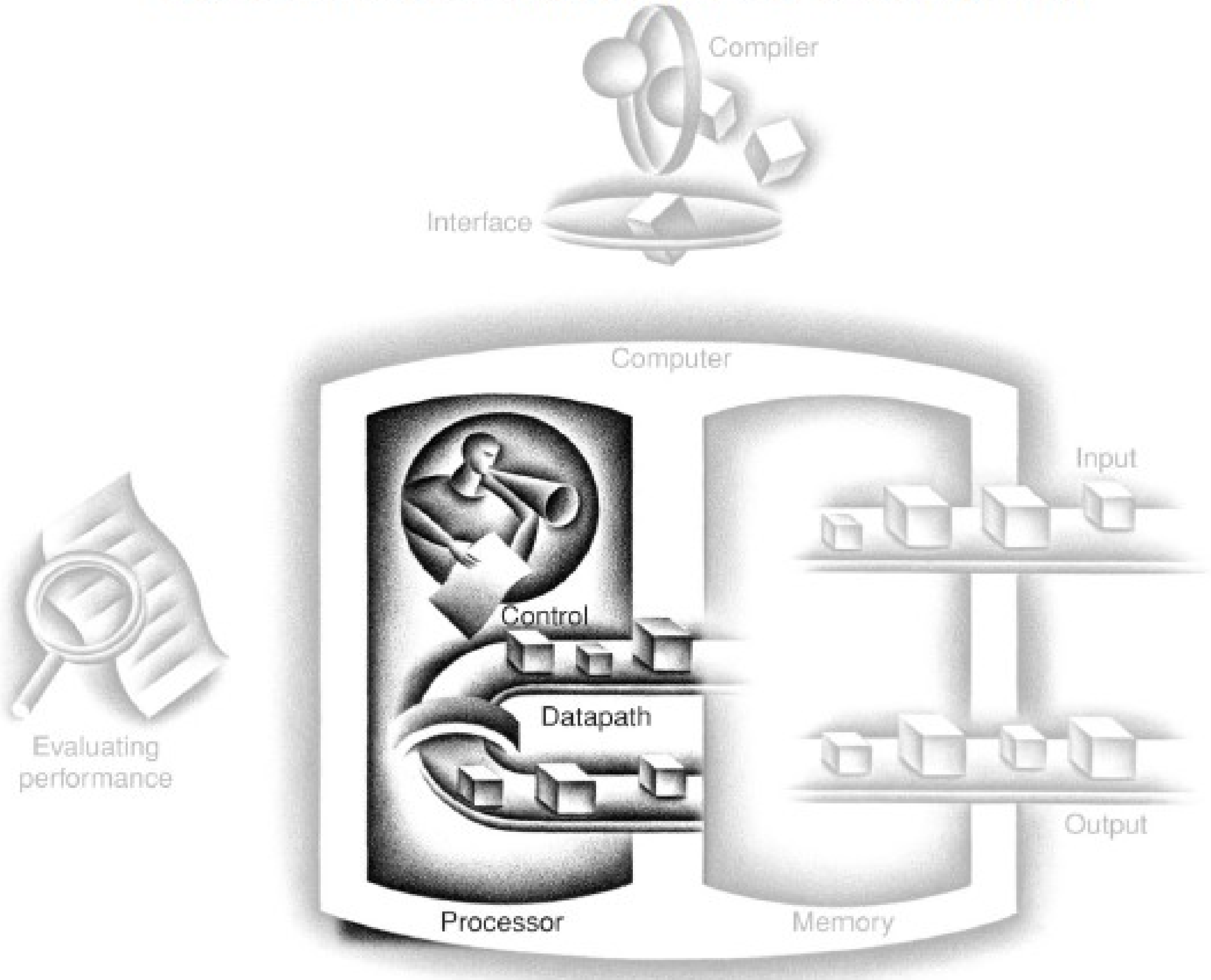
UNIT III PROCESSOR AND CONTROL UNIT

Basic MIPS implementation – Building datapath – Control Implementation scheme – Pipelining – Pipelined datapath and control – Handling Data hazards & Control hazards – Exceptions

Reference:

- Chapter 4 – The Processor
- Appendix B – The Basics of Logic Design
  - B.7 Clock
  - B.8 Memory Elements
- Book - David A. Patterson and John L. Hennessey, "Computer organization and design‟, Morgan Kauffman / Elsevier, Fifth edition, 2014.

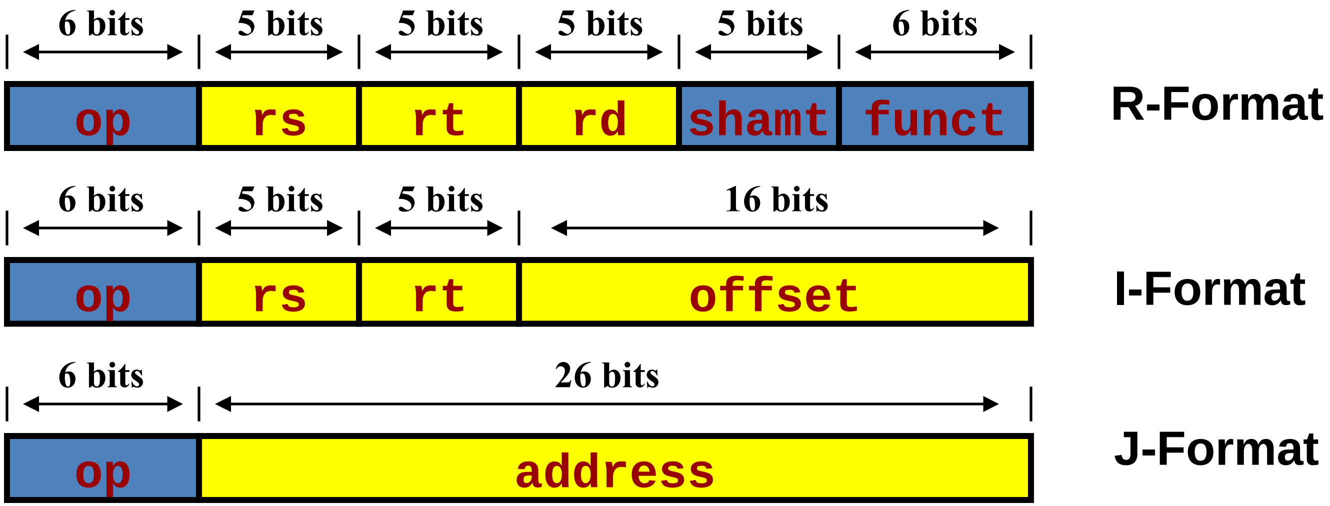
Five Classic Components of a Computer



Basic MIPS implementation

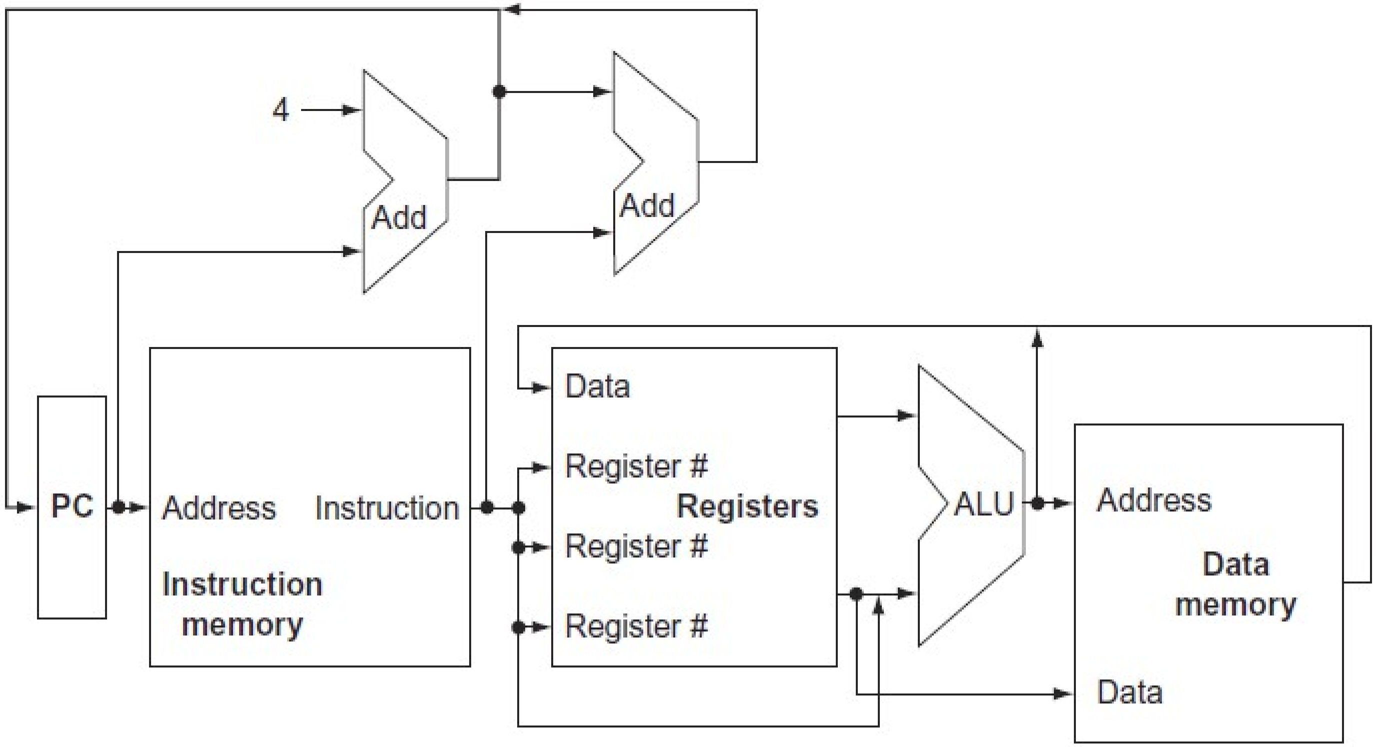
Implementing MIPS

- We're ready to look at an implementation of the MIPS instruction set
- Simplified to contain only
  - arithmetic-logic instructions: `add, sub, and, or, slt`
  - memory-reference instructions: `lw, sw`
  - control-flow instructions: `beq, j`



Overview implementing MIPS: Fetch/Execute Cycle

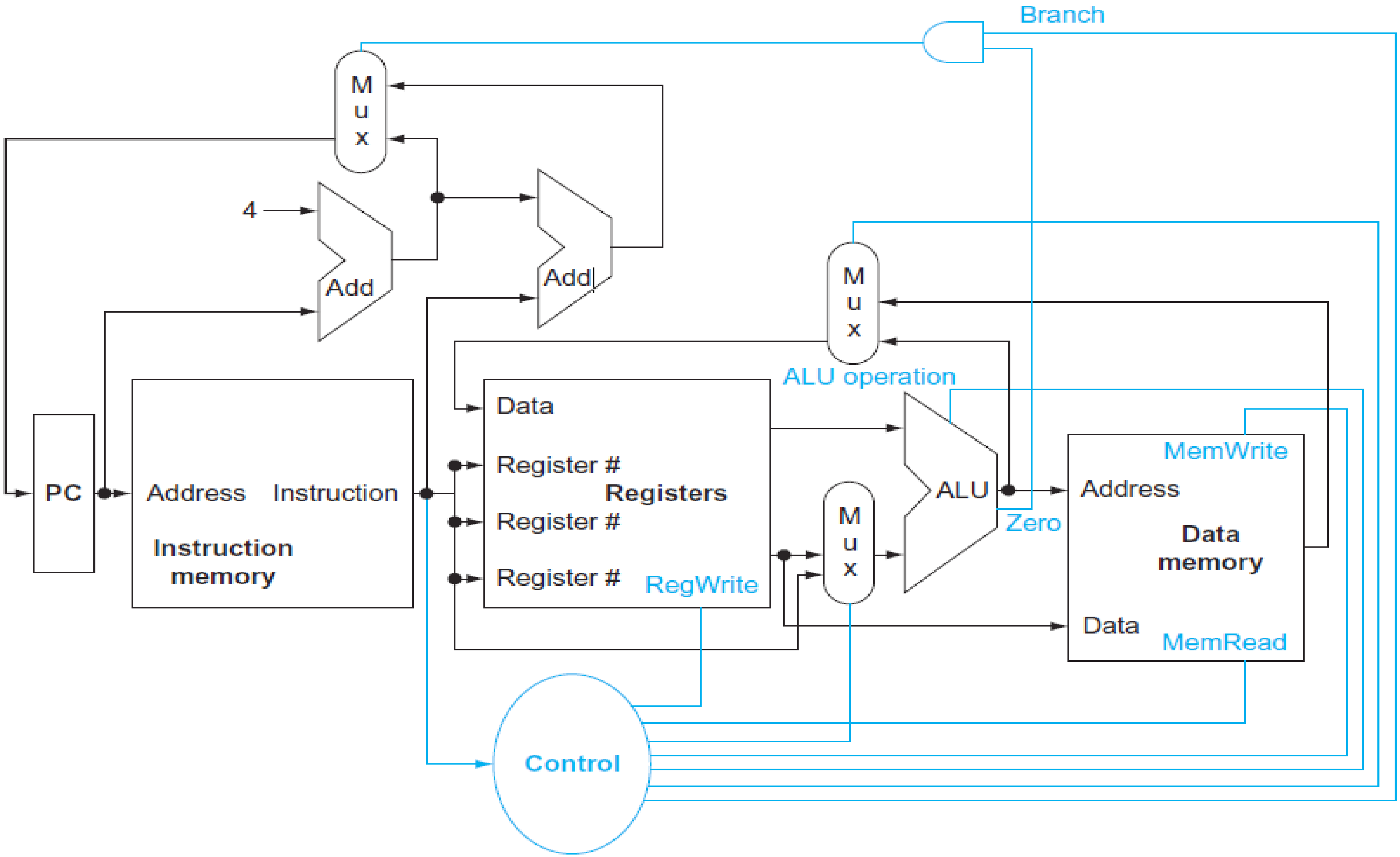
- High-level abstract view of *fetch/execute* implementation
  - use the program counter (PC) to read instruction address
  - *fetch* the instruction from memory and increment PC
  - use fields of the instruction to select registers to read
  - *execute* depending on the instruction
  - repeat.



Overview: Processor Implementation Styles

- Single Cycle
  - perform each instruction in 1 clock cycle
  - clock cycle must be long enough for slowest instruction; therefore,
  - disadvantage: only as fast as slowest instruction
- Multi-Cycle
  - break fetch/execute cycle into multiple steps
  - perform 1 step in each clock cycle
  - advantage: each instruction uses only as many cycles as it needs
- Pipelined
  - execute each instruction in multiple steps
  - perform 1 step / instruction in each clock cycle
  - process multiple instructions in parallel – assembly line

Basic Implementation of MIPS – with Data Path



Processor

- Two components of processor
  - Datapath
  - Control

Functional Elements

- Two types of functional elements:
  - elements that *operate on* data - ***combinational elements***
  - elements that *contain* data - ***state*** **or** ***sequential elements***

Building Data Path

Data path

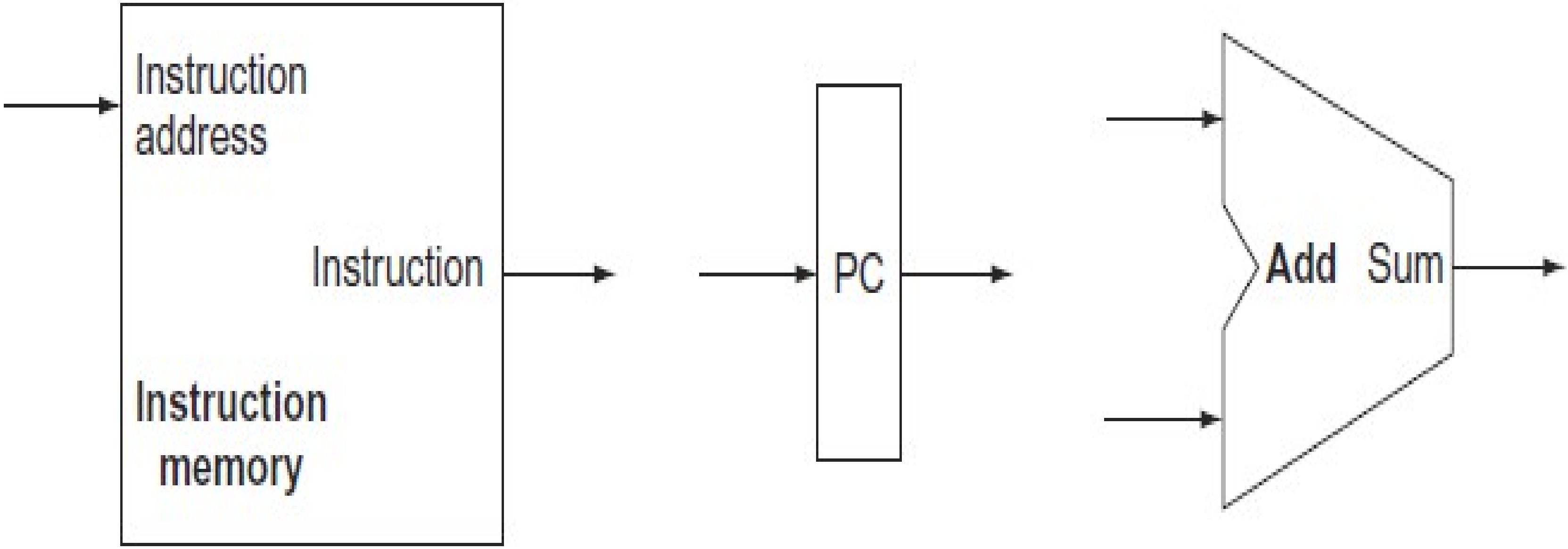
- A unit used to operate on or hold data within a processor.
- The datapath elements include
  - instruction and data memories,
  - the register file,
  - the ALU,
  - and adders.

Elements used to fetch instructions and increment the PC

3 elements are used to fetch instructions and increment PC:

- Instruction memory
  - A state element where instruction is stored.
- PC – program counter
  - A state element or register containing the address of the next instruction to be executed.
- Adder
  - an ALU wired to always add its two 32-bit inputs and place the sum on its output.

Element used to store/fetch instructions and increment the PC

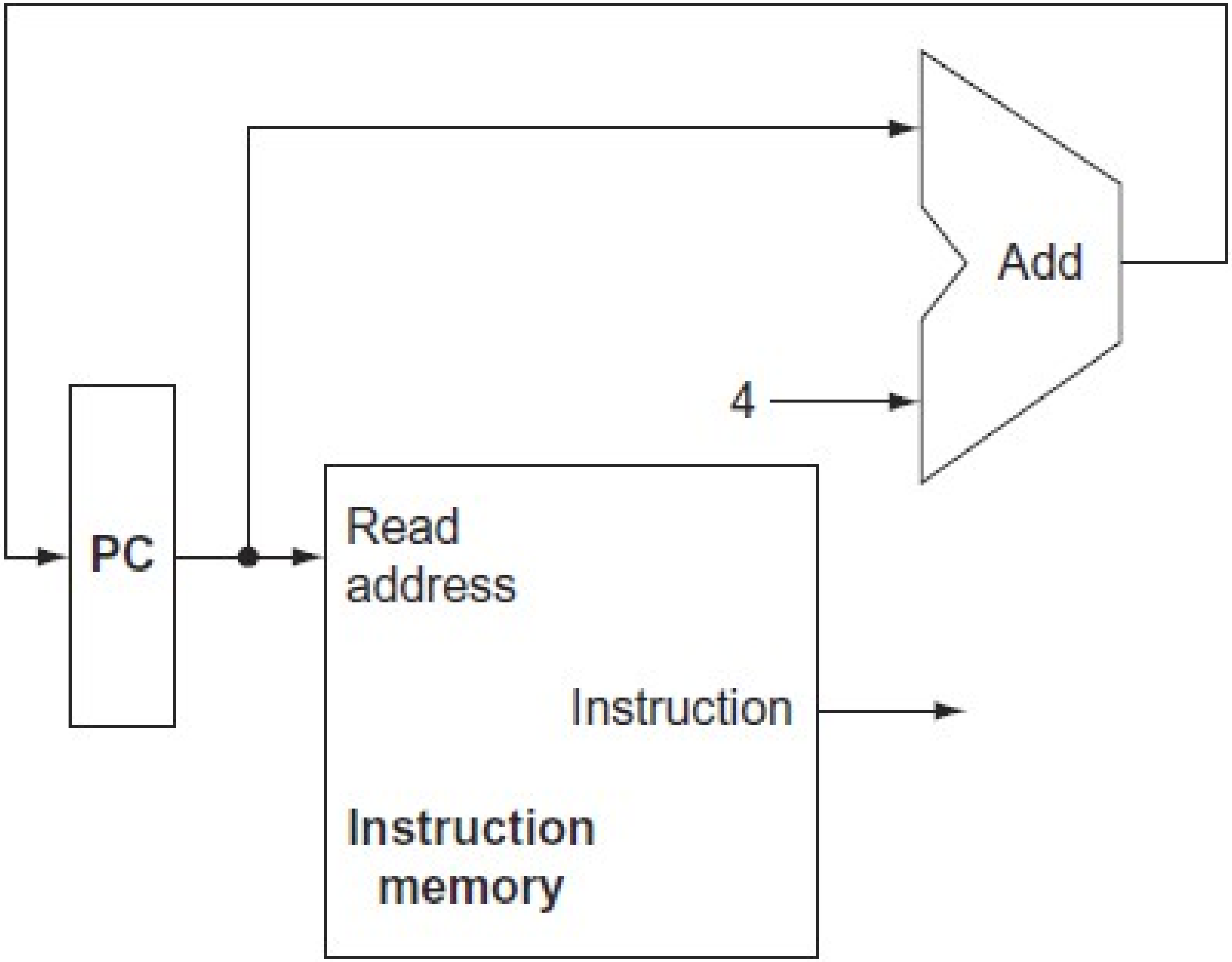


a. Instruction memory

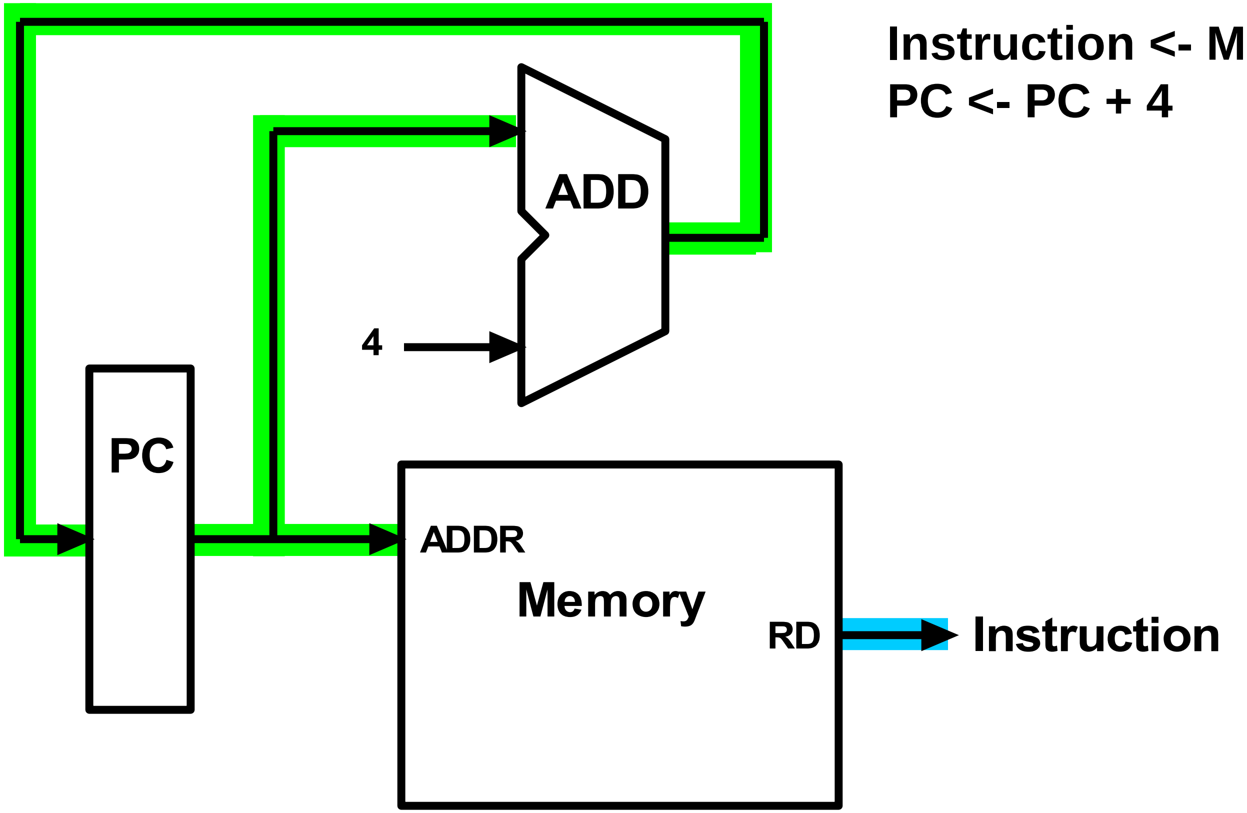
b. Program counter

c. Adder

Datapath: Instruction Store/Fetch & PC Increment



Animating the Datapath



Instruction <- MEM[PC]
PC <- PC + 4

Implementing Datapath for each instruction classes

- Arithmetic and logical instructions – R-type
- Load store instructions
- Control – conditional

I-Type

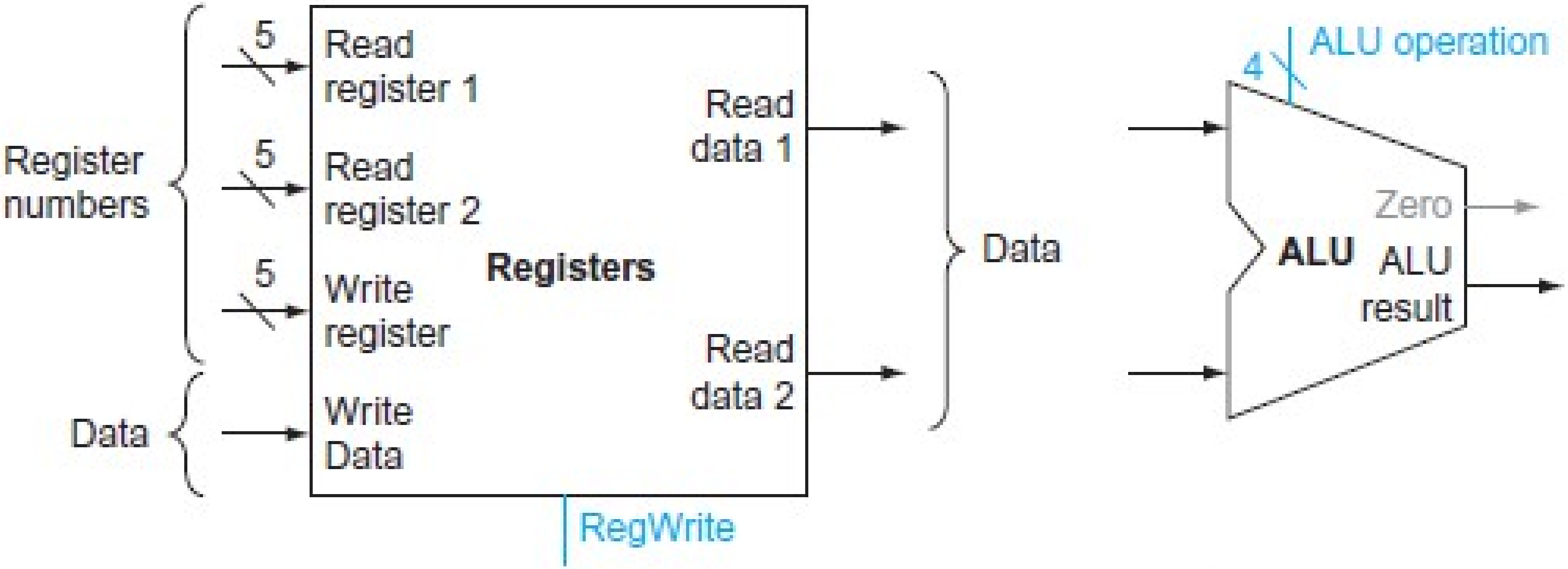
- Control – unconditional instructions - J-Type

Elements used in R-Type instructions

2 elements are used in R-Type instructions

- Register file
  - A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.
- ALU
  - A combinational element that performs arithmetic and logical operations on the input data.

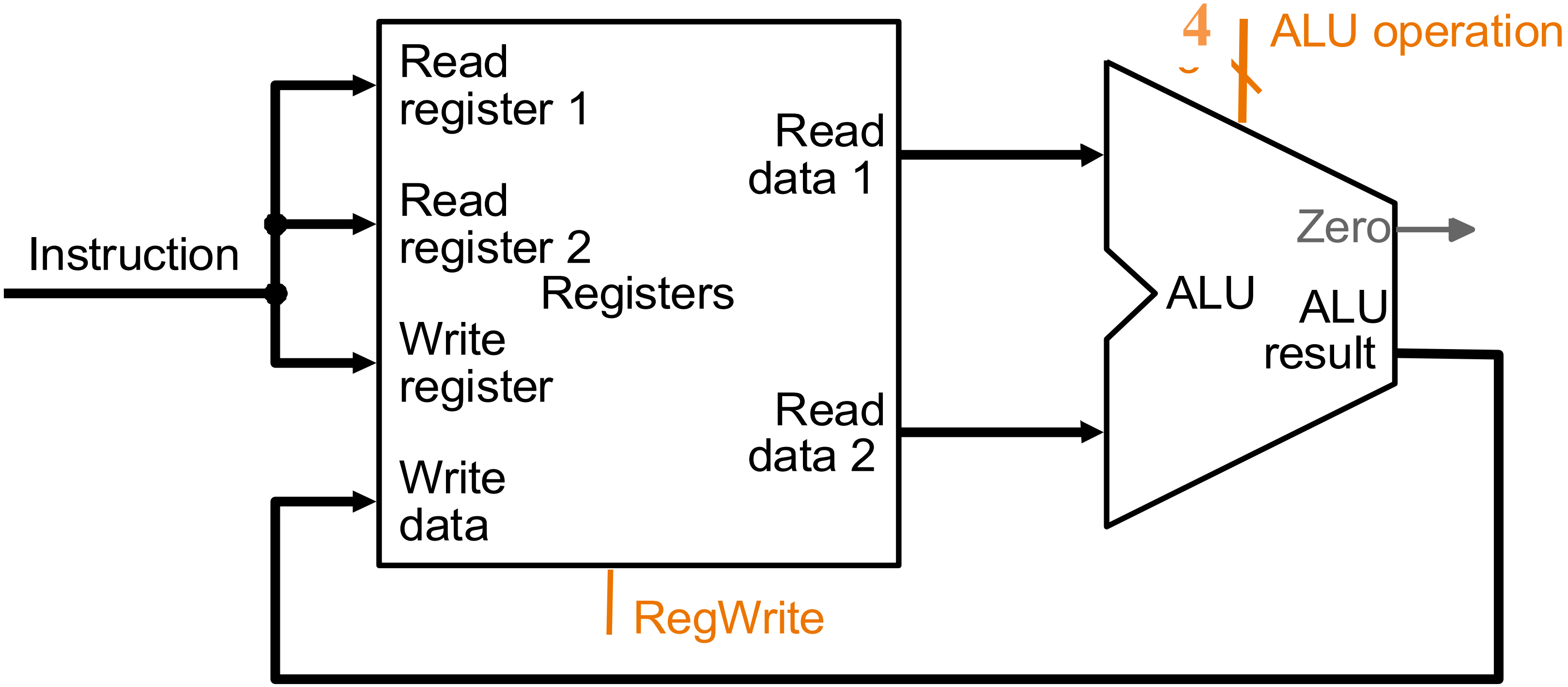
Elements used in R-Type instructions



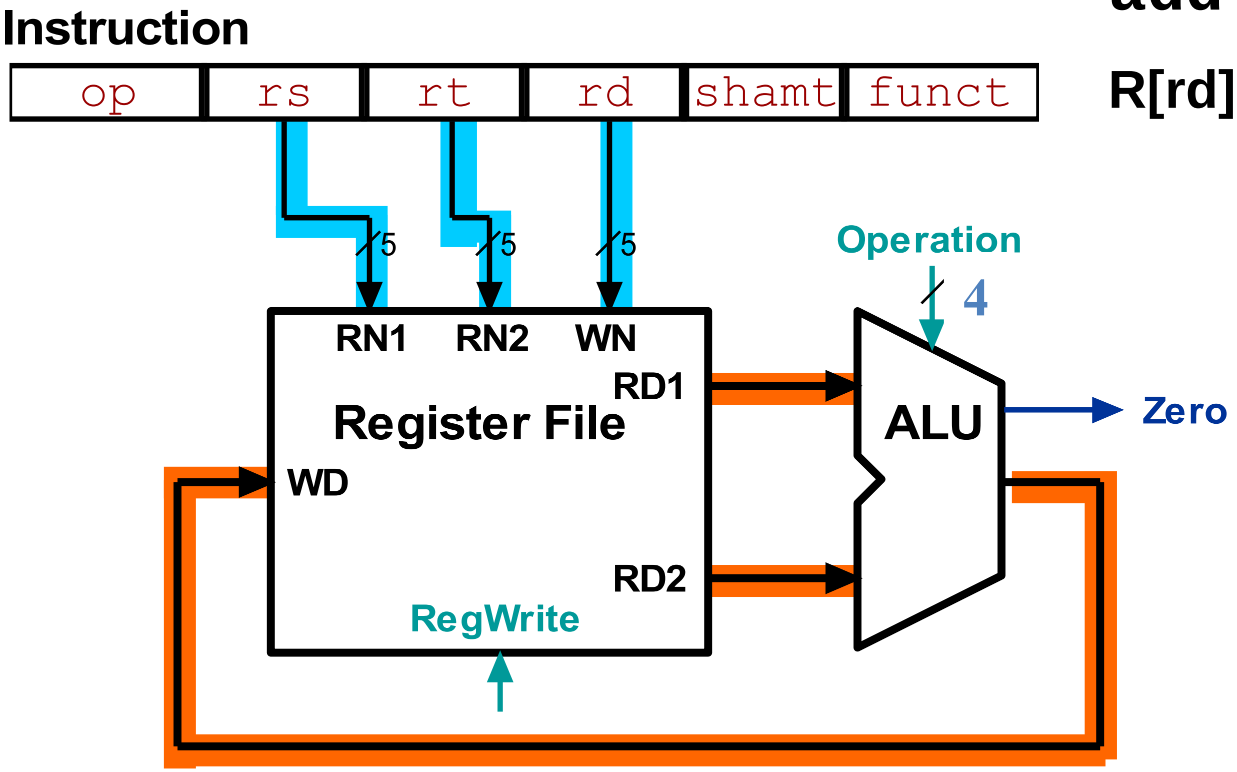
a. Registers

b. ALU

Datapath: R-Type Instruction



Animating the R-Type Instruction Datapath



`add rd, rs, rt`

R[rd] <- R[rs] + R[rt];

Implementing Datapath for each instruction classes

- Arithmetic and logical instructions – R-type
- Load store instructions
- Control – conditional

I-Type

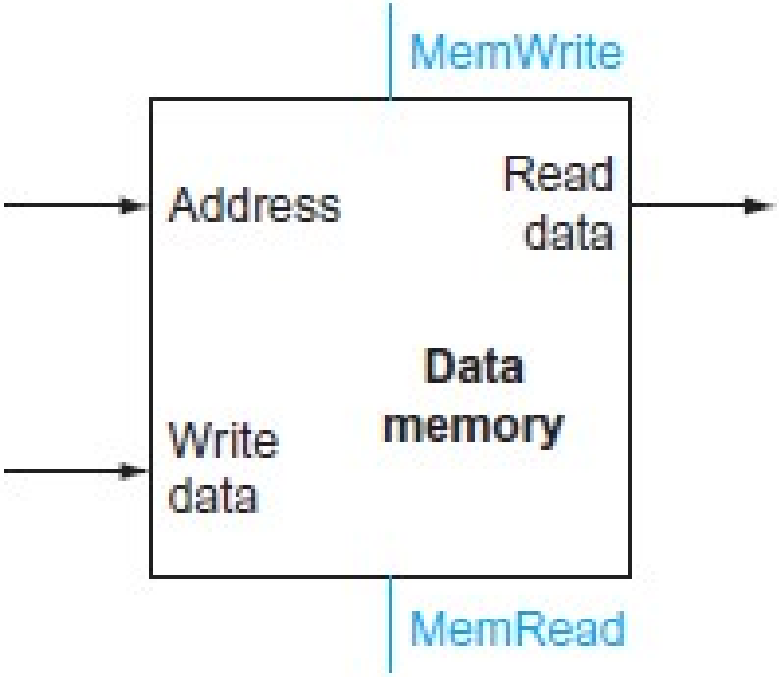
- Control – unconditional instructions - J-Type

Elements used in Load/Store instructions

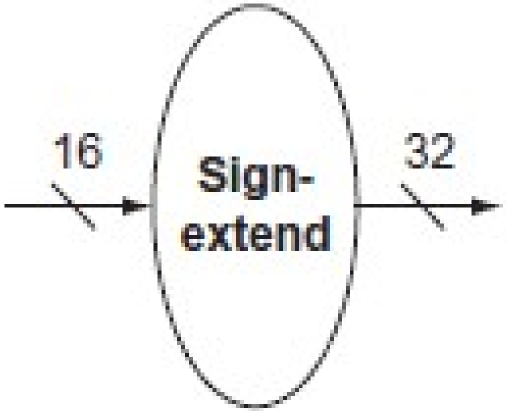
4 elements are used in Load/Store instructions

- Data memory
  - The memory unit is a state element with inputs for the address and the write data, and a single output for the read result
- Sign Extend
  - a 16-bit input that is sign-extended into a 32-bit result appearing on the output
- Register file
- ALU

Elements used in Load/Store instructions

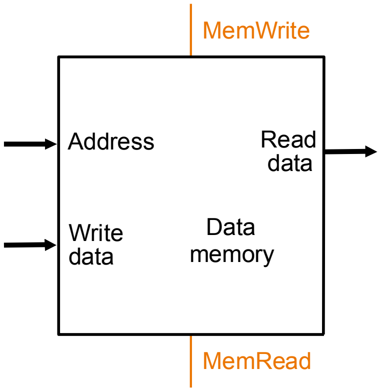


a. Data memory unit

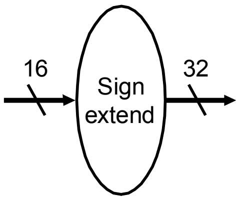


b. Sign extension unit

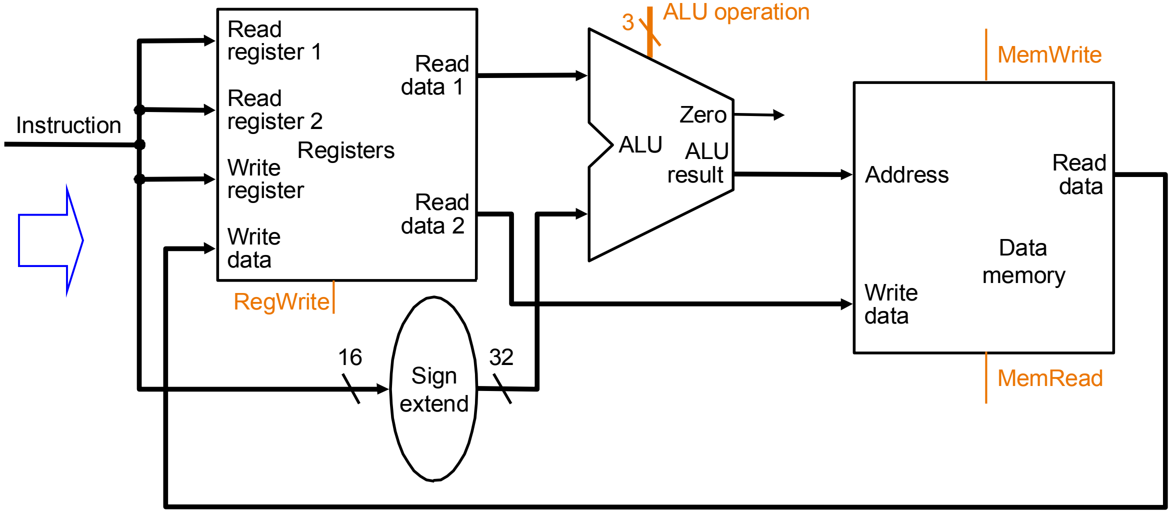
Datapath: Load/Store Instruction



a. Data memory unit



b. Sign-extension unit



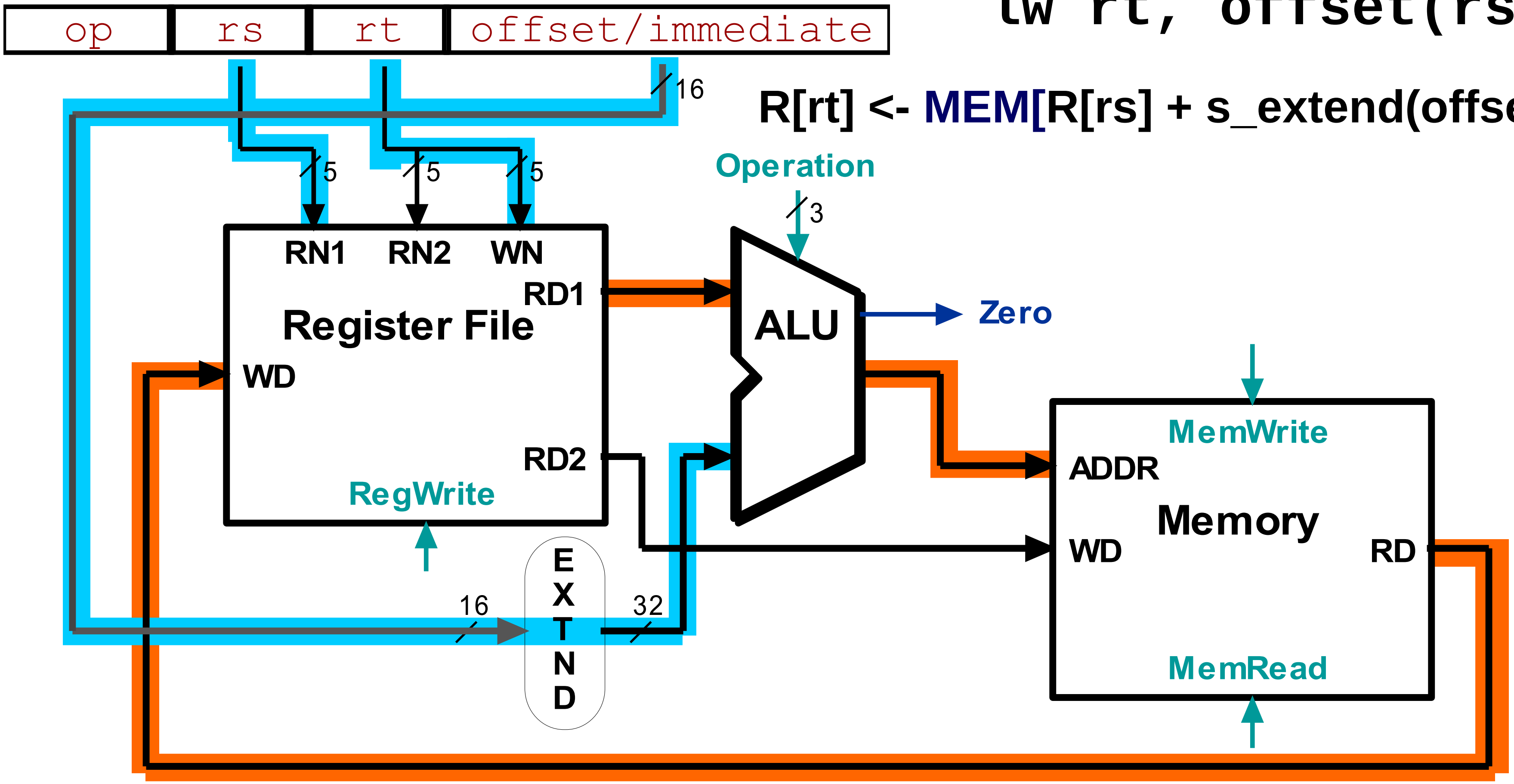
Two additional elements used To implement load/stores

Datapath

Animating the Load Instruction Datapath

`lw rt, offset(rs)`

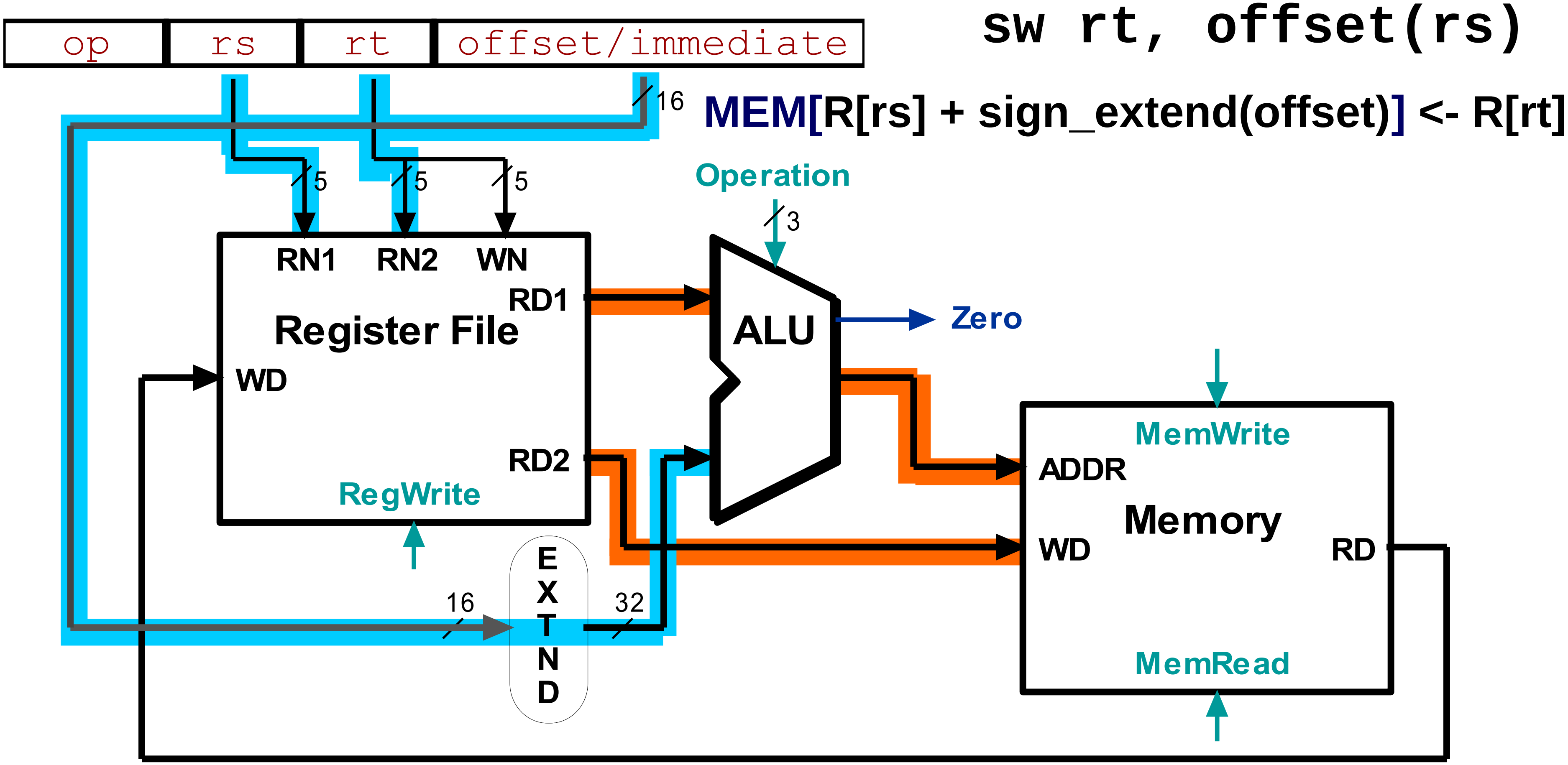
R[rt] <- MEM[R[rs] + s_extend(offset)];



Animating the Store Instruction Datapath

`sw rt, offset(rs)`

MEM[R[rs] + sign_extend(offset)] <- R[rt]



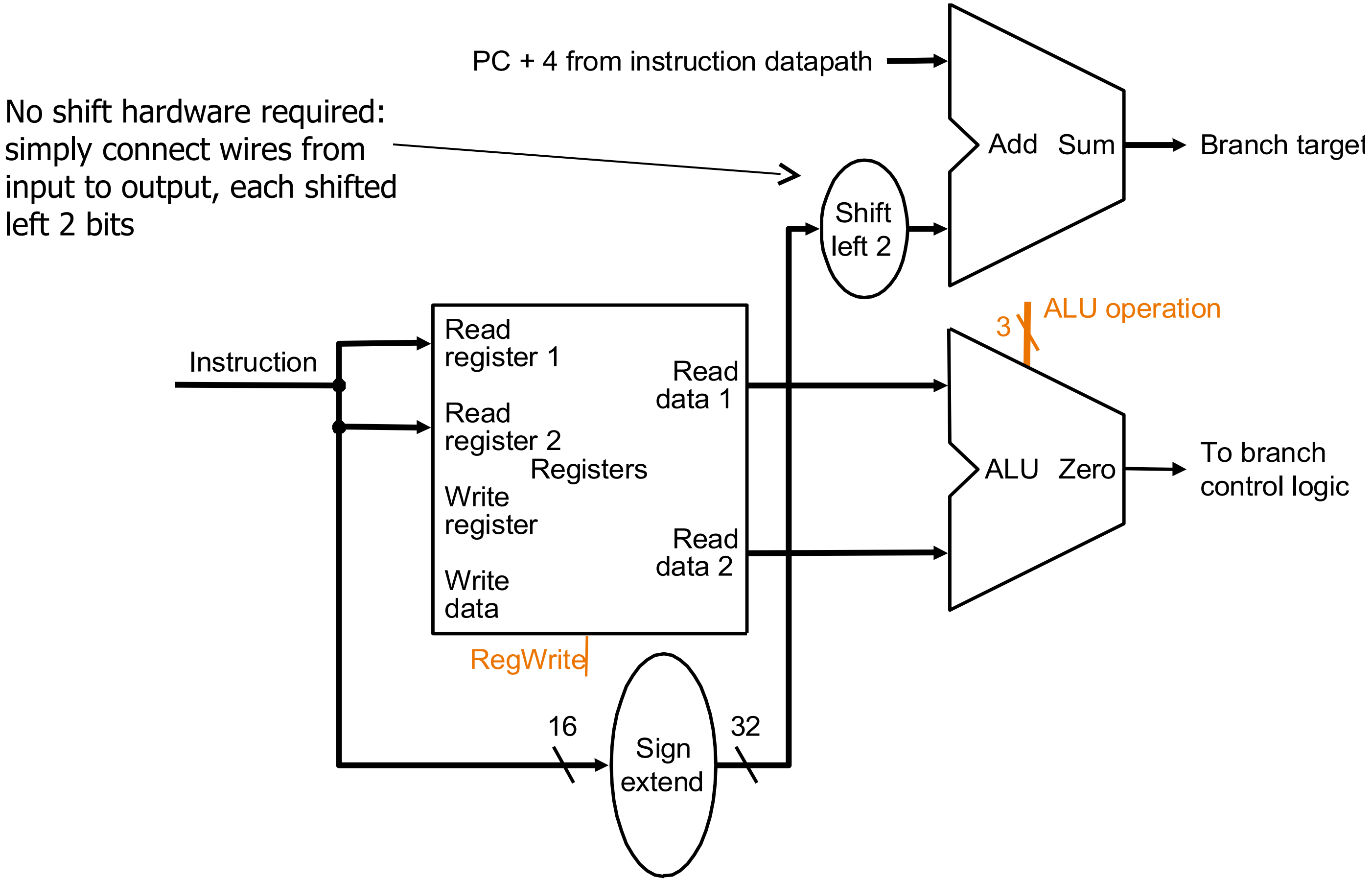
Implementing Datapath for each instruction classes

- Arithmetic and logical instructions – R-type
- Load store instructions
- Control – conditional

I-Type

- Control – unconditional - J - Type

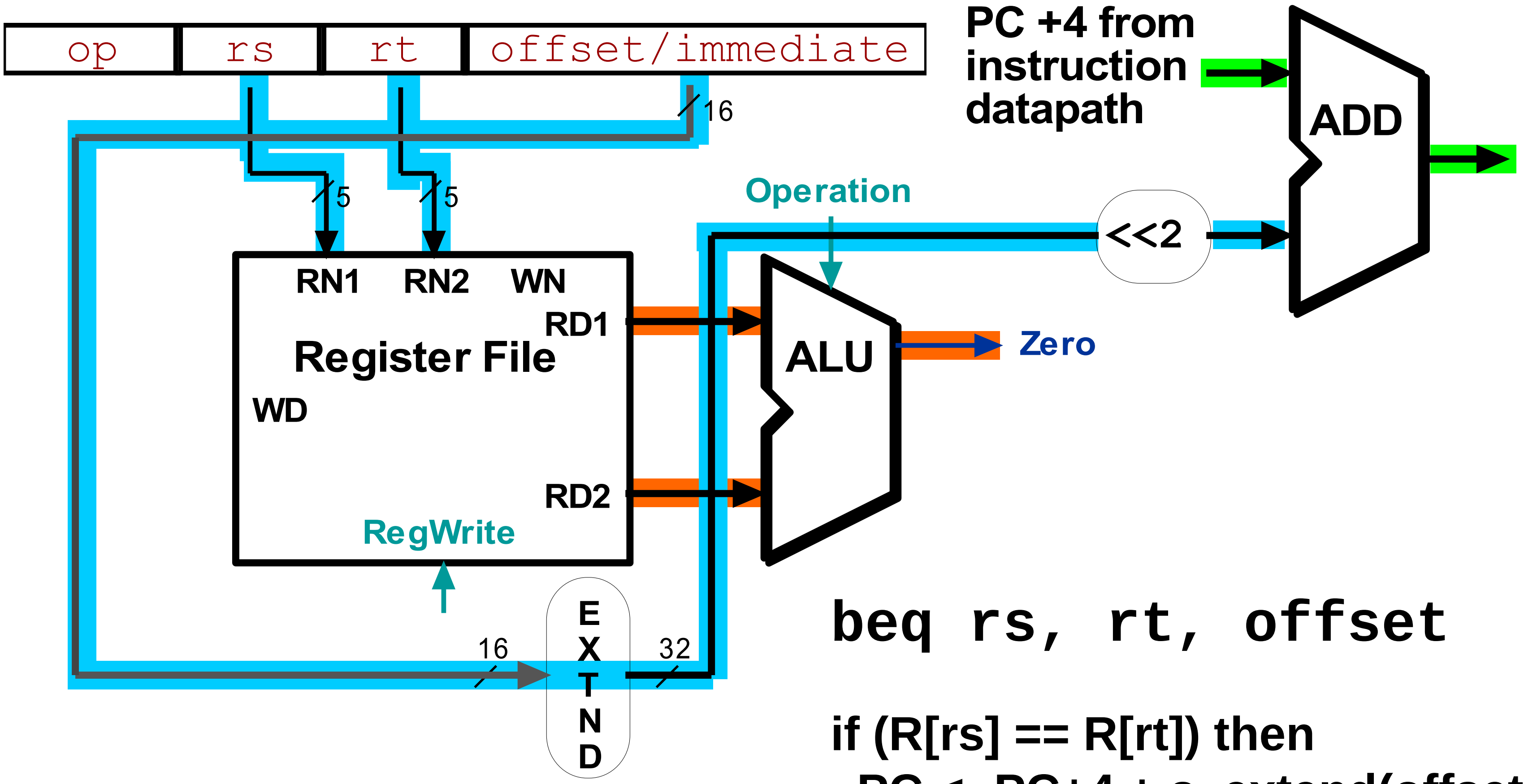
Datapath: Branch Instruction



No shift hardware required: simply connect wires from input to output, each shifted left 2 bits

Datapath

Animating the Branch Instruction Datapath

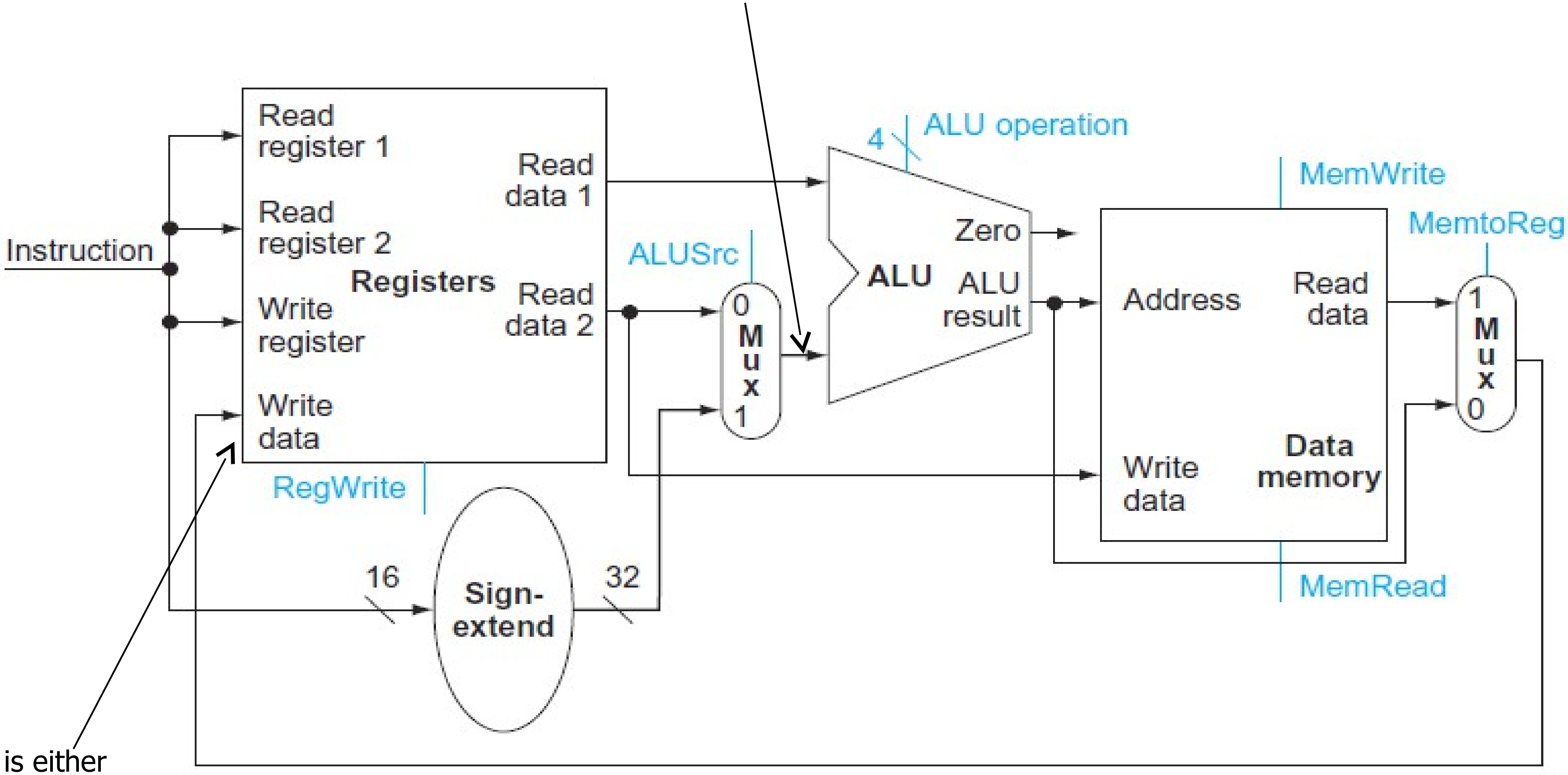


```
beq rs, rt, offset
```

if (R[rs] == R[rt]) then
  PC <- PC+4 + s_extend(offset<<2)

Combining the datapaths for R-type instructions and load/stores using two multiplexors

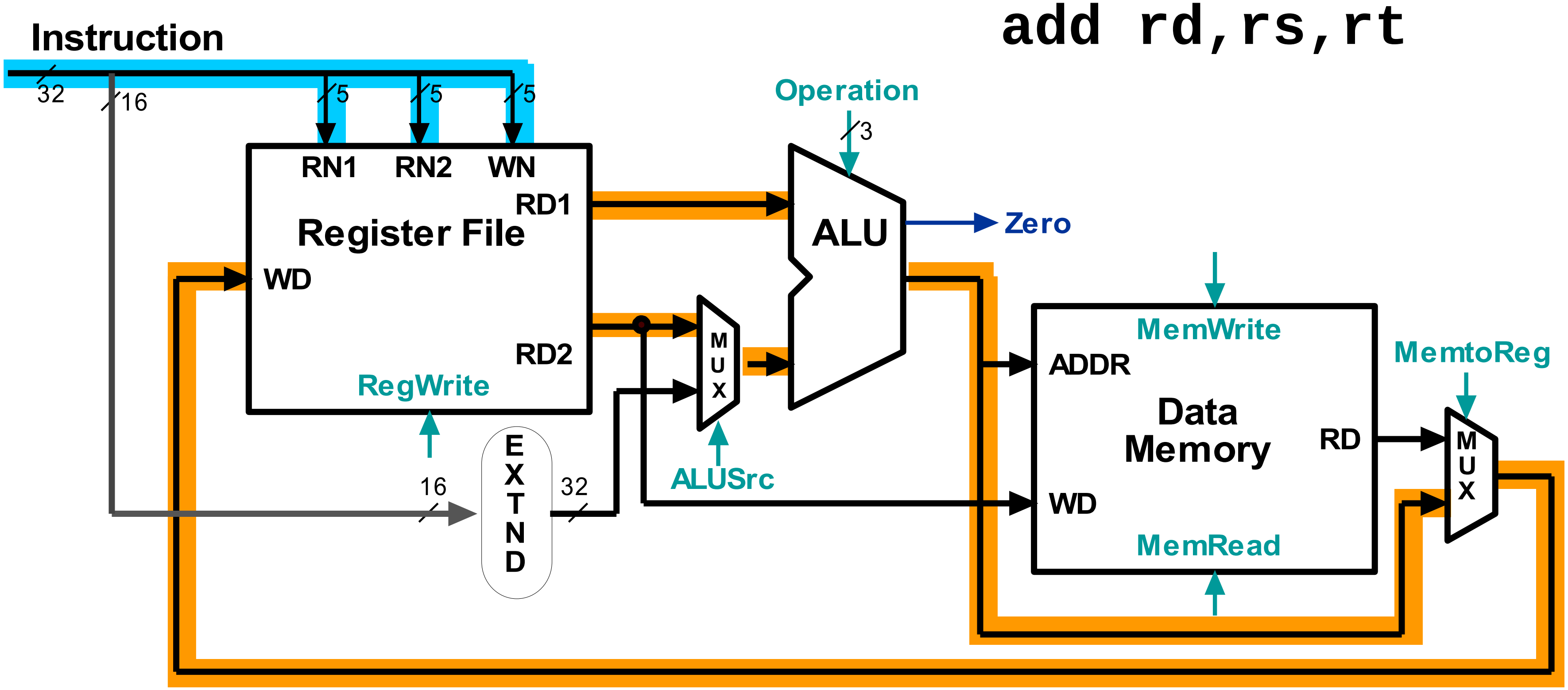
Input is either register (R-type) or sign-extended
lower half of instruction (load/store)



Data is either
from ALU (R-type)
or memory (load)

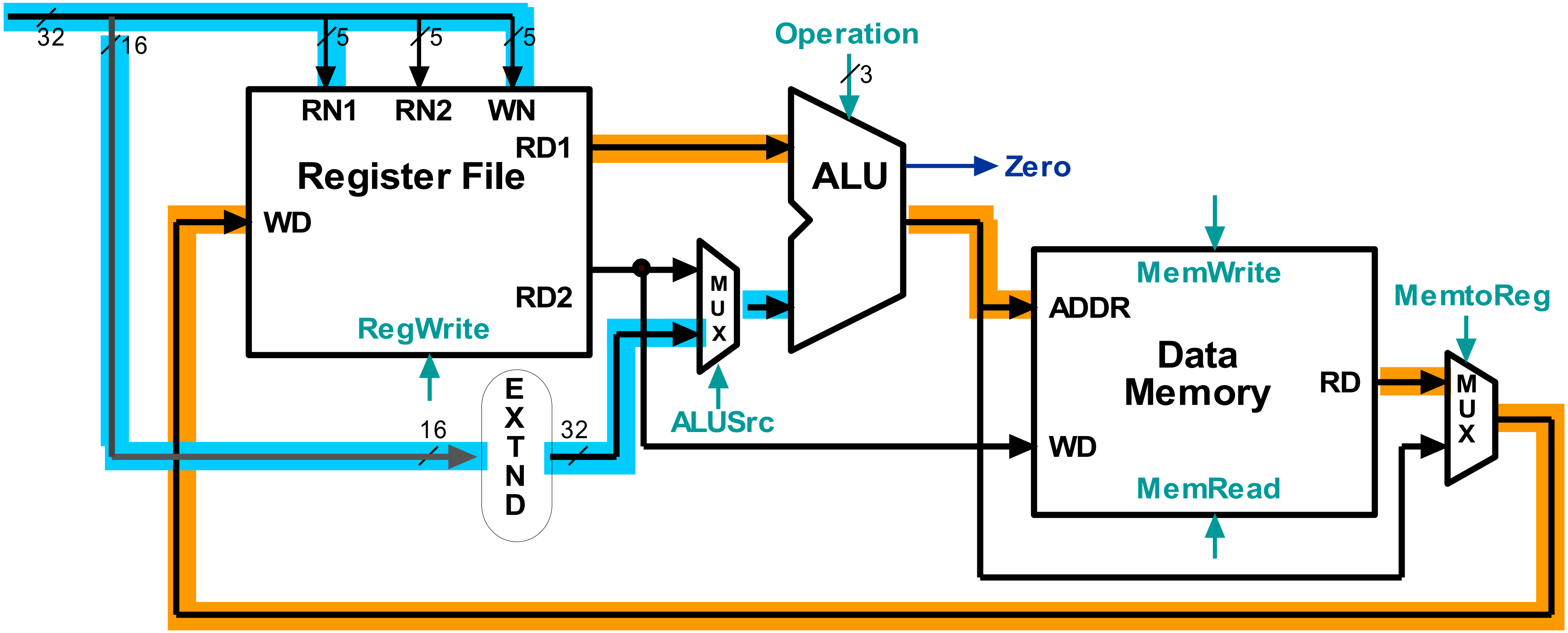
Animating the Datapath: R-type Instruction

`add rd,rs,rt`



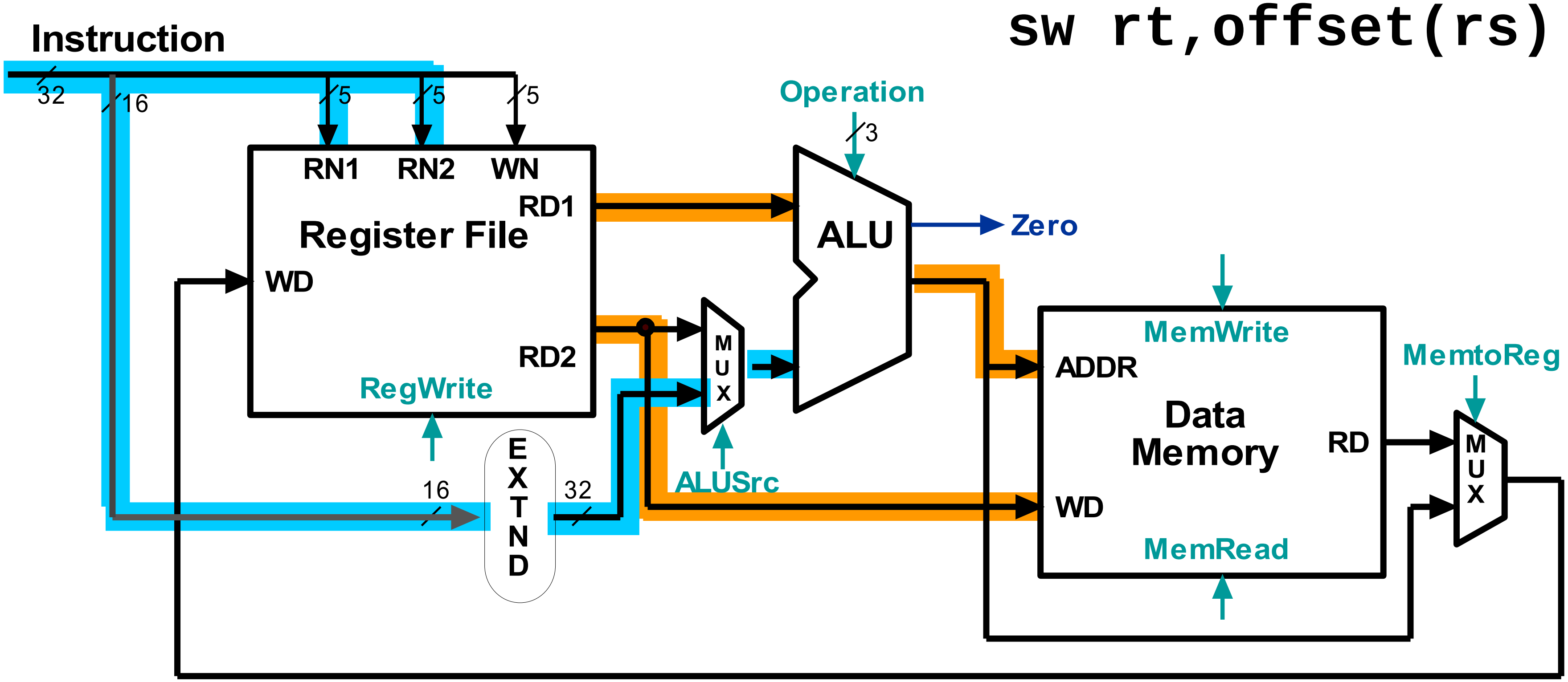
Animating the Datapath: Load Instruction

`lw rt,offset(rs)`

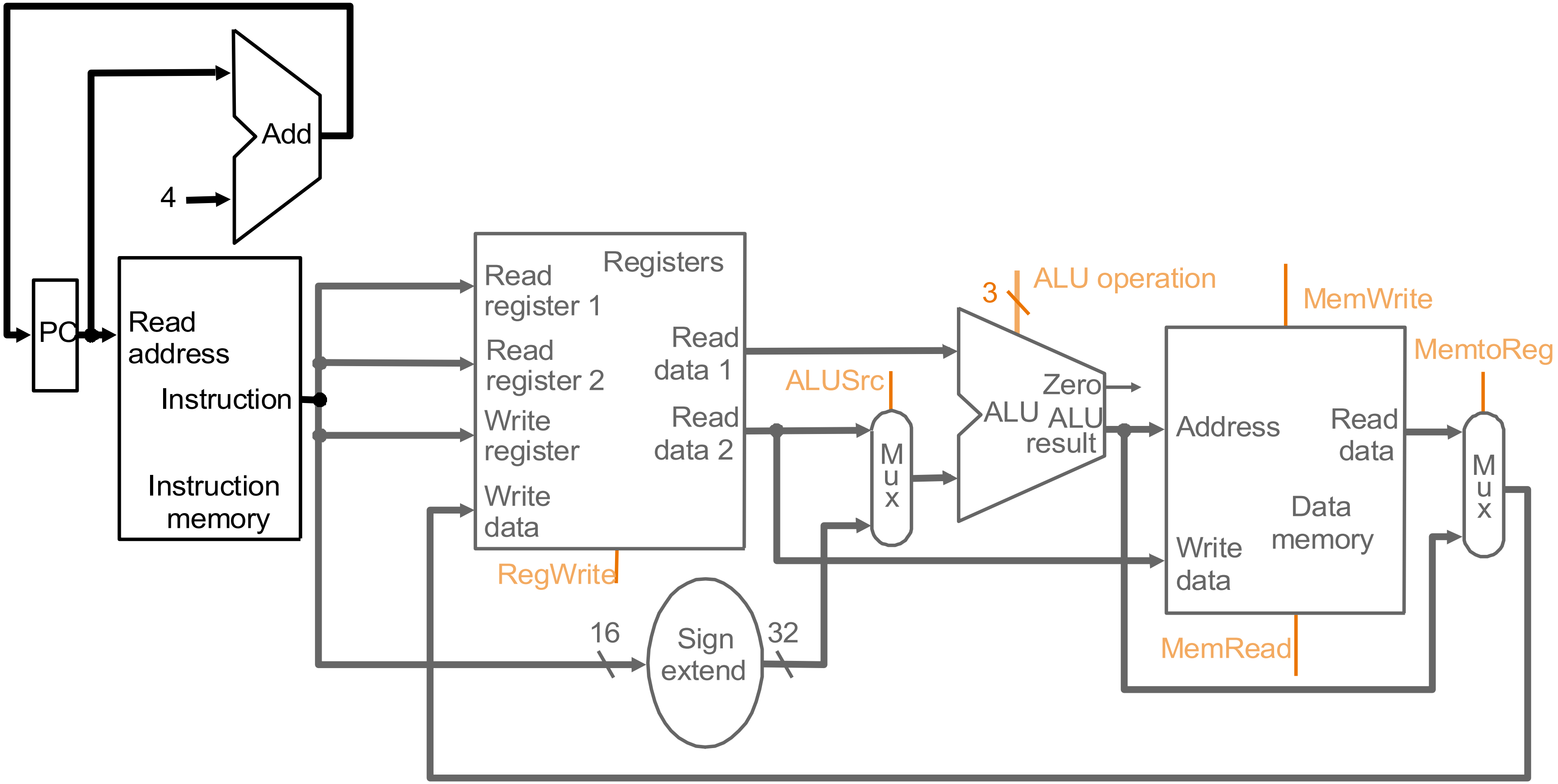


Animating the Datapath: Store Instruction

`sw rt,offset(rs)`



MIPS Datapath - Adding instruction fetch
(R-type & load/store & instruction fetch)



Branch taken & not taken

Branch taken

```
X = 1 ; Y = 2
If X==Y
then
    equal
else
    not equal
```

Assign PC:

PC = BTA

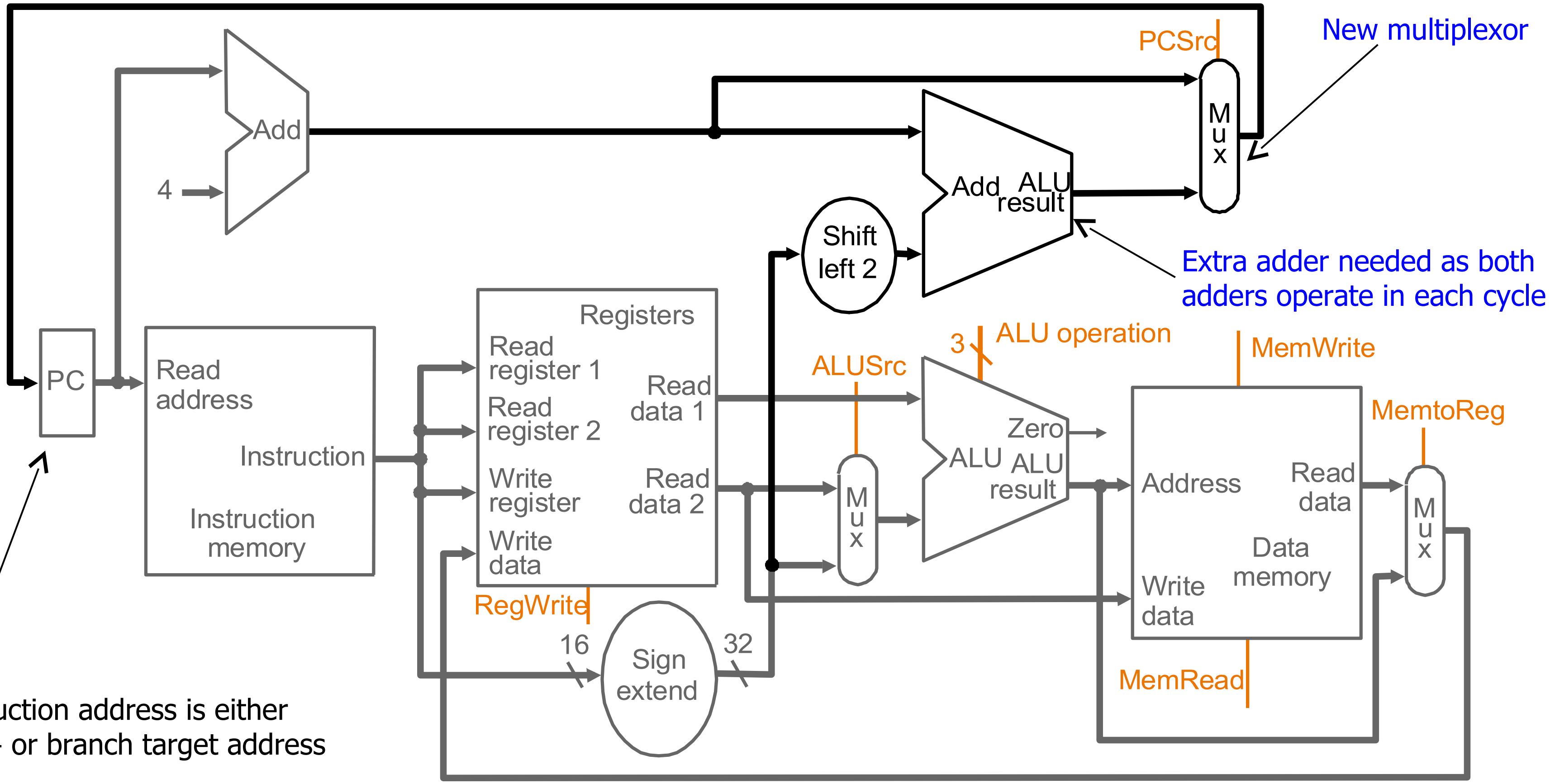
Branch not taken

```
X = 1 ; Y = 1
If X==Y
then
    equal
else
    not equal
```

Assign PC:

PC = PC+4

MIPS Datapath : Adding Branch capability (R-type, Load/Store, Instruction fetch, Branch)

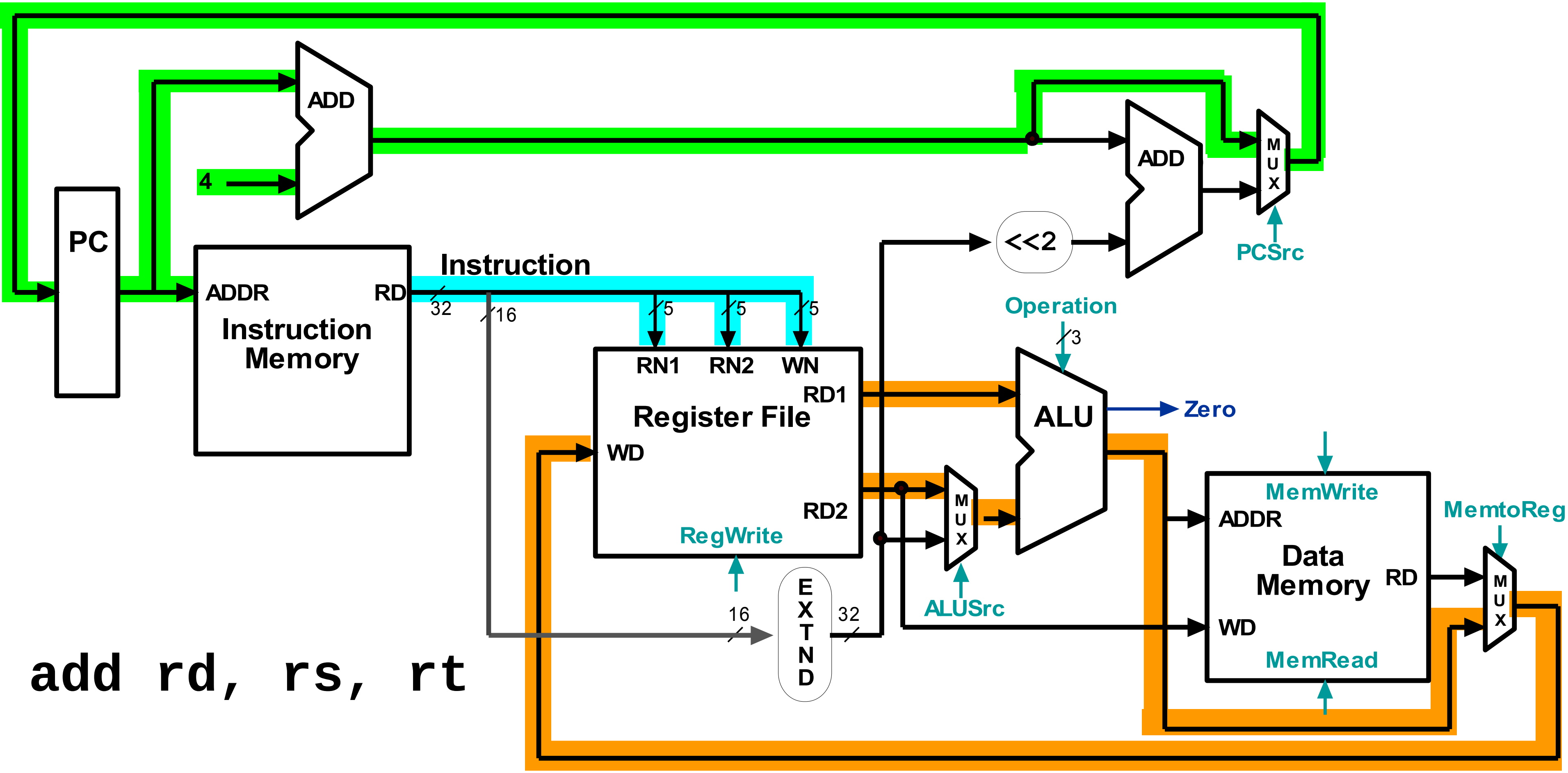


New multiplexor

Extra adder needed as both adders operate in each cycle

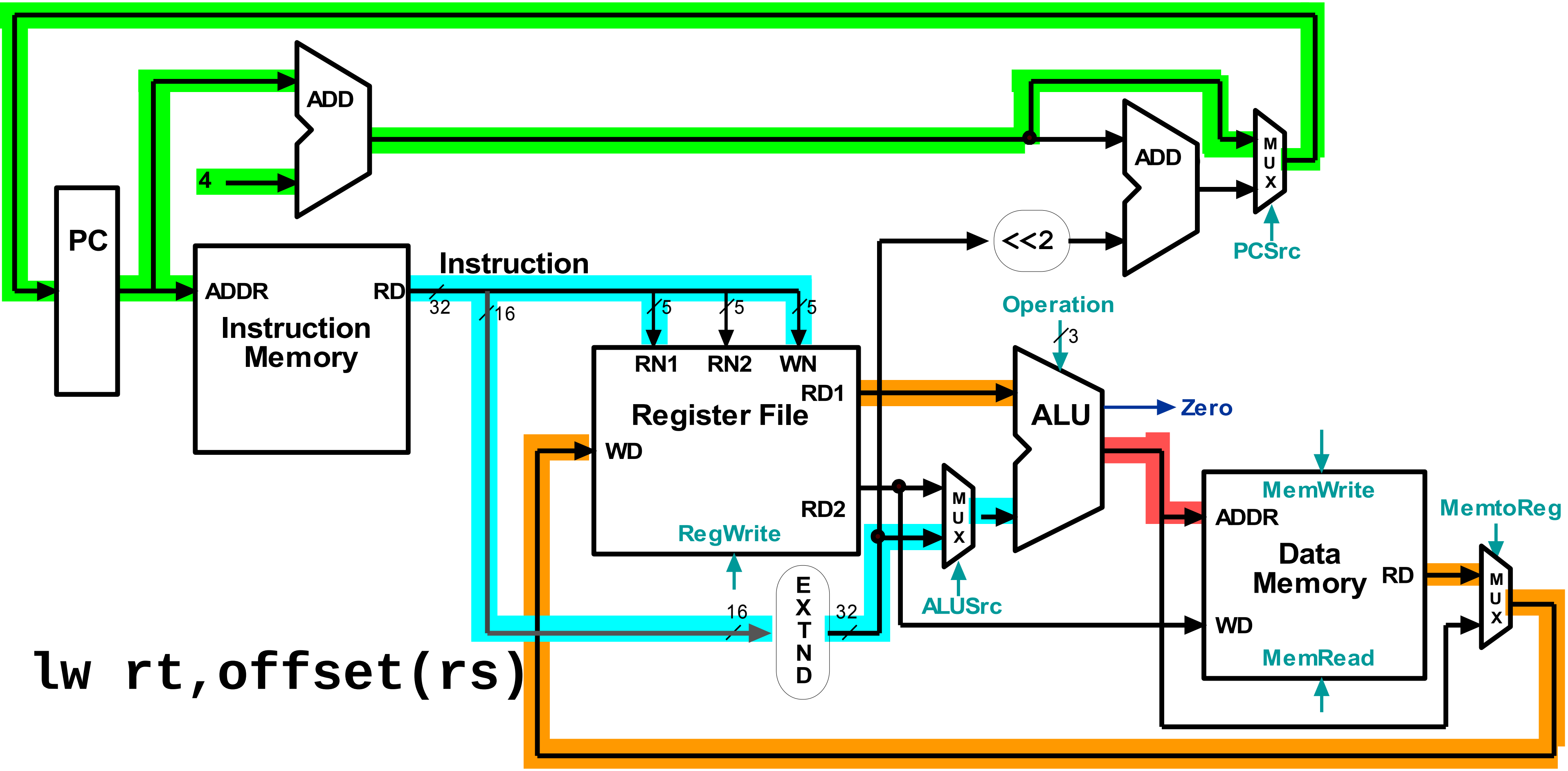
Instruction address is either PC+4 or branch target address

Datapath Executing `add`



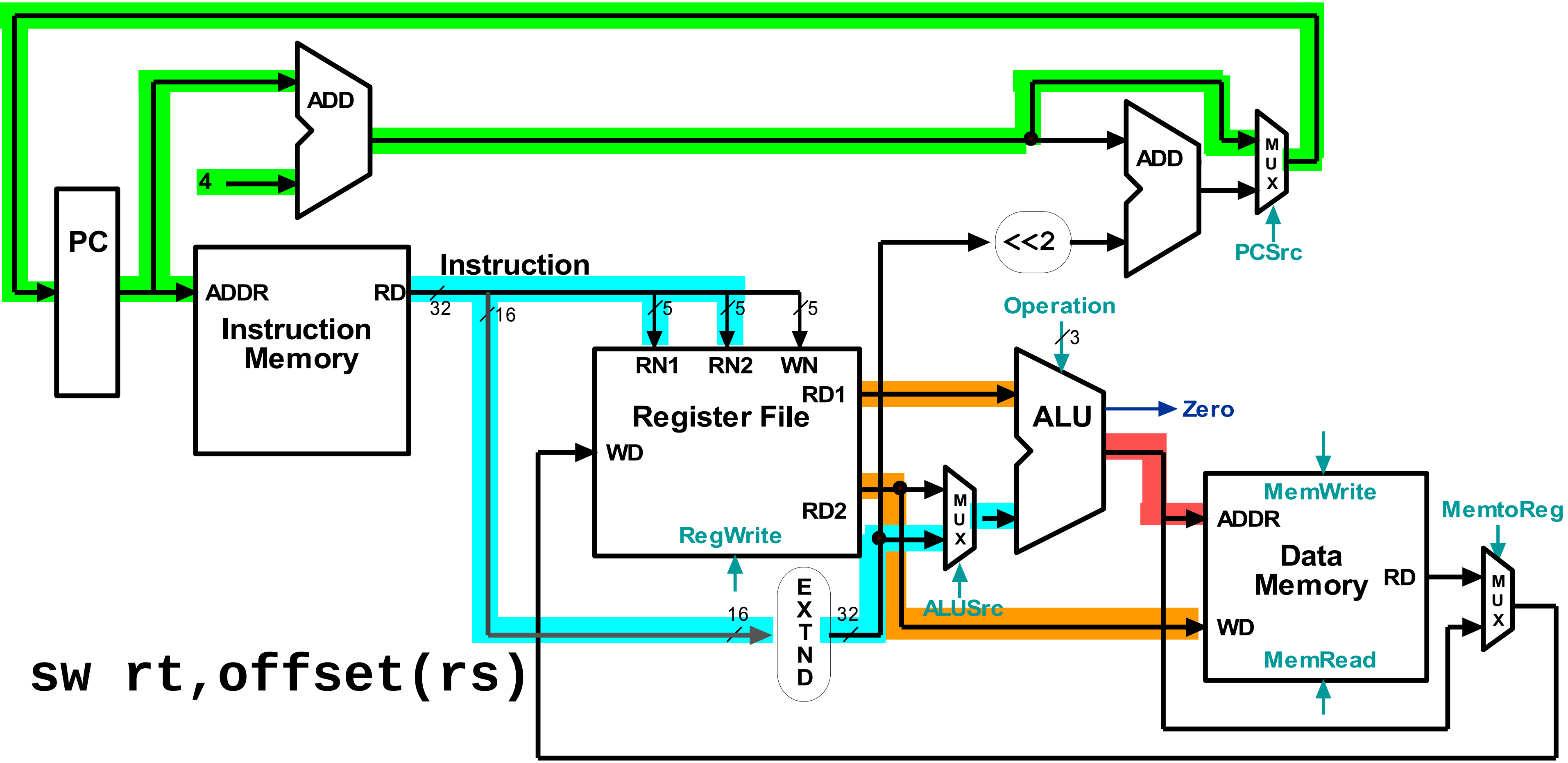
```
add rd, rs, rt
```

Datapath Executing `lw`



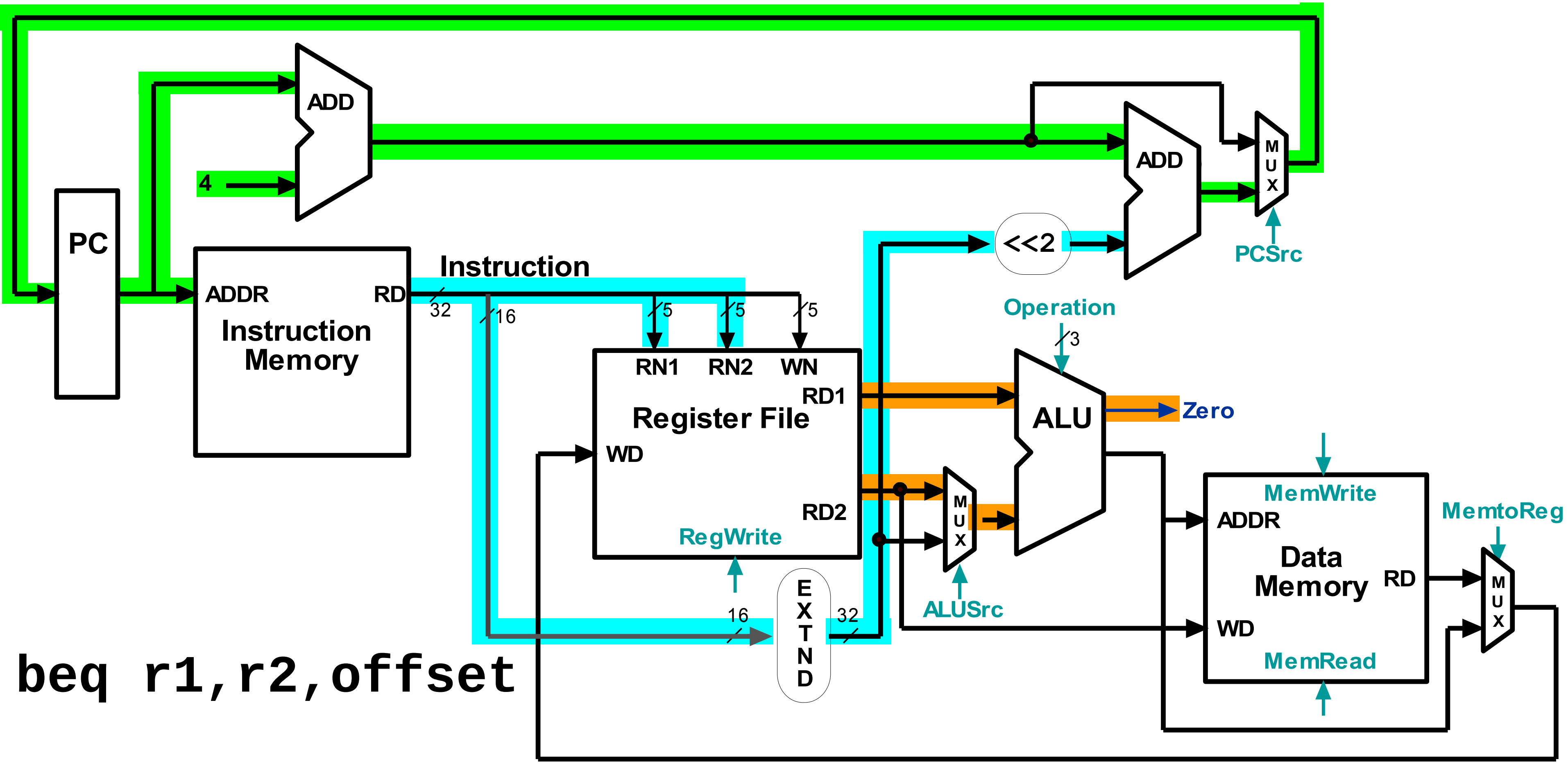
`lw rt,offset(rs)`

Datapath Executing `sw`



`sw rt,offset(rs)`

Datapath Executing `beq` (Branch Taken : PC = BTA)



`beq r1,r2,offset`

Control Implementation Scheme

Processor

- Two components of processor
  - Datapath
  - Control
- Control:
  - Sends control signals to all other units

Control

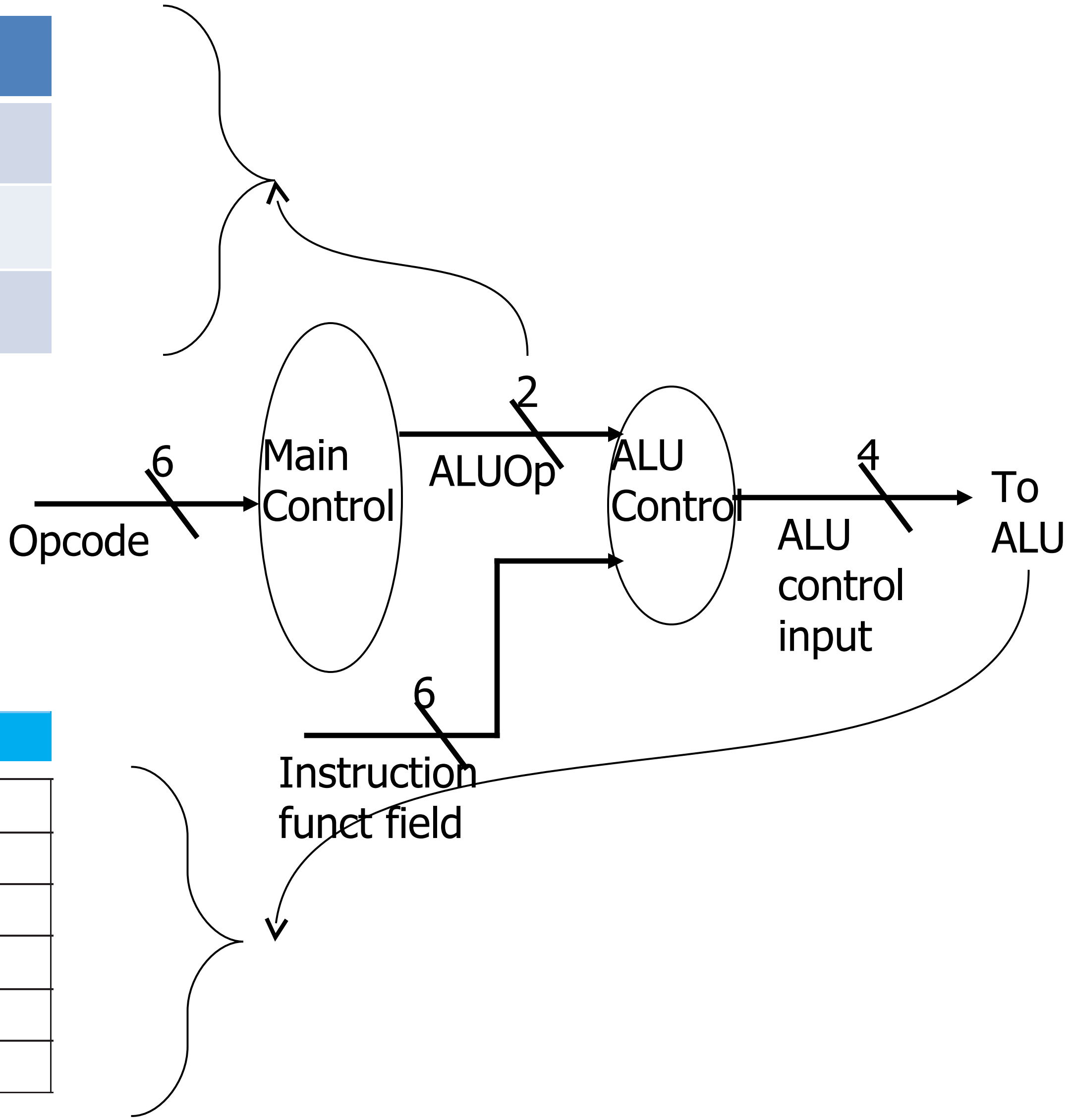
- **Input** to control unit
  - the instruction opcode bits

- Control unit generates **(Output)**
  - ALU control input
  - write enable (possibly, read enable also) signals for each storage element
  - selector controls for each multiplexor

ALU Control

ALUOp generation
by main control

Instruction	ALUOp
Load/Store	00
Branch	01
R-Type	10



Output of ALU control

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

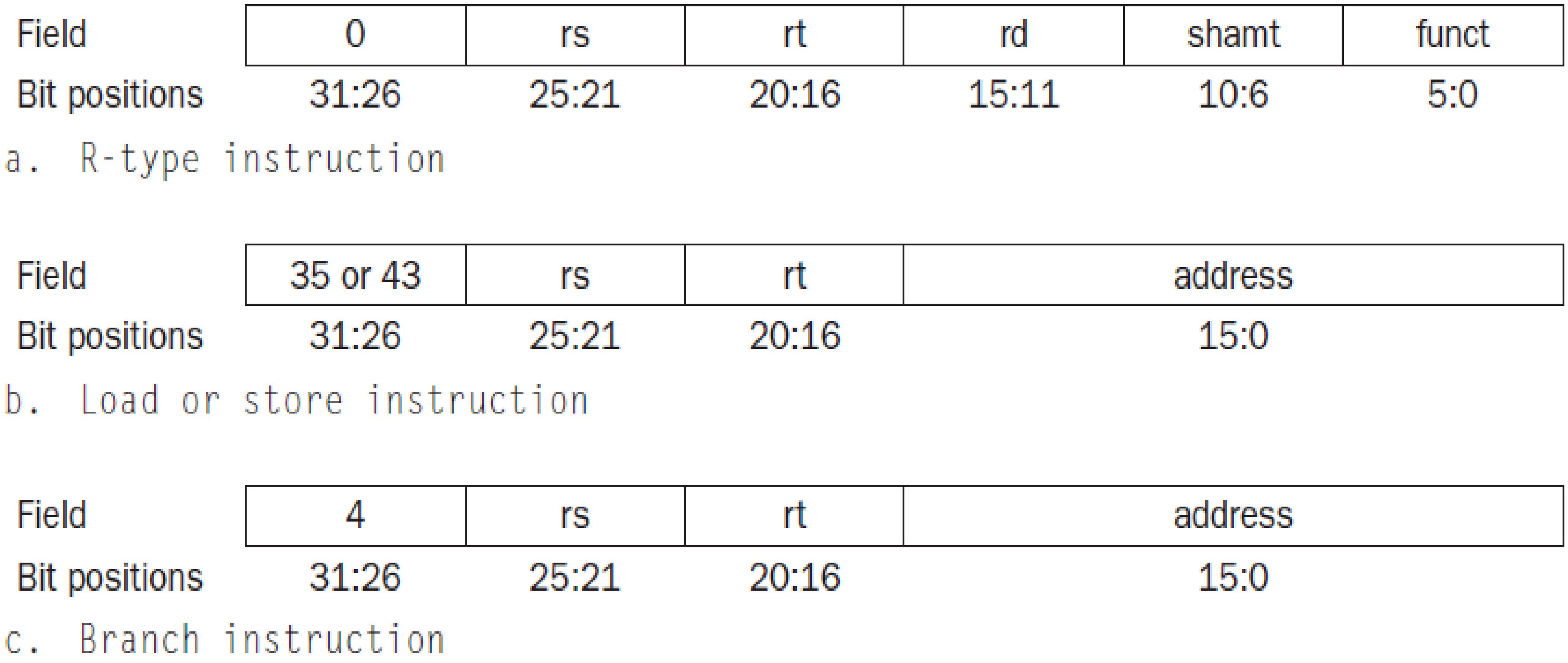
Setting ALU Control Bits

Instruction opcode	AluOp	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

MIPS instruction formats
(Observe the bit positions)



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

a. R-type instruction

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

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

Designing the Main Control

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

Load/store or branch	opcode	rs	rt	address
	31-26	25-21	20-16	15-0

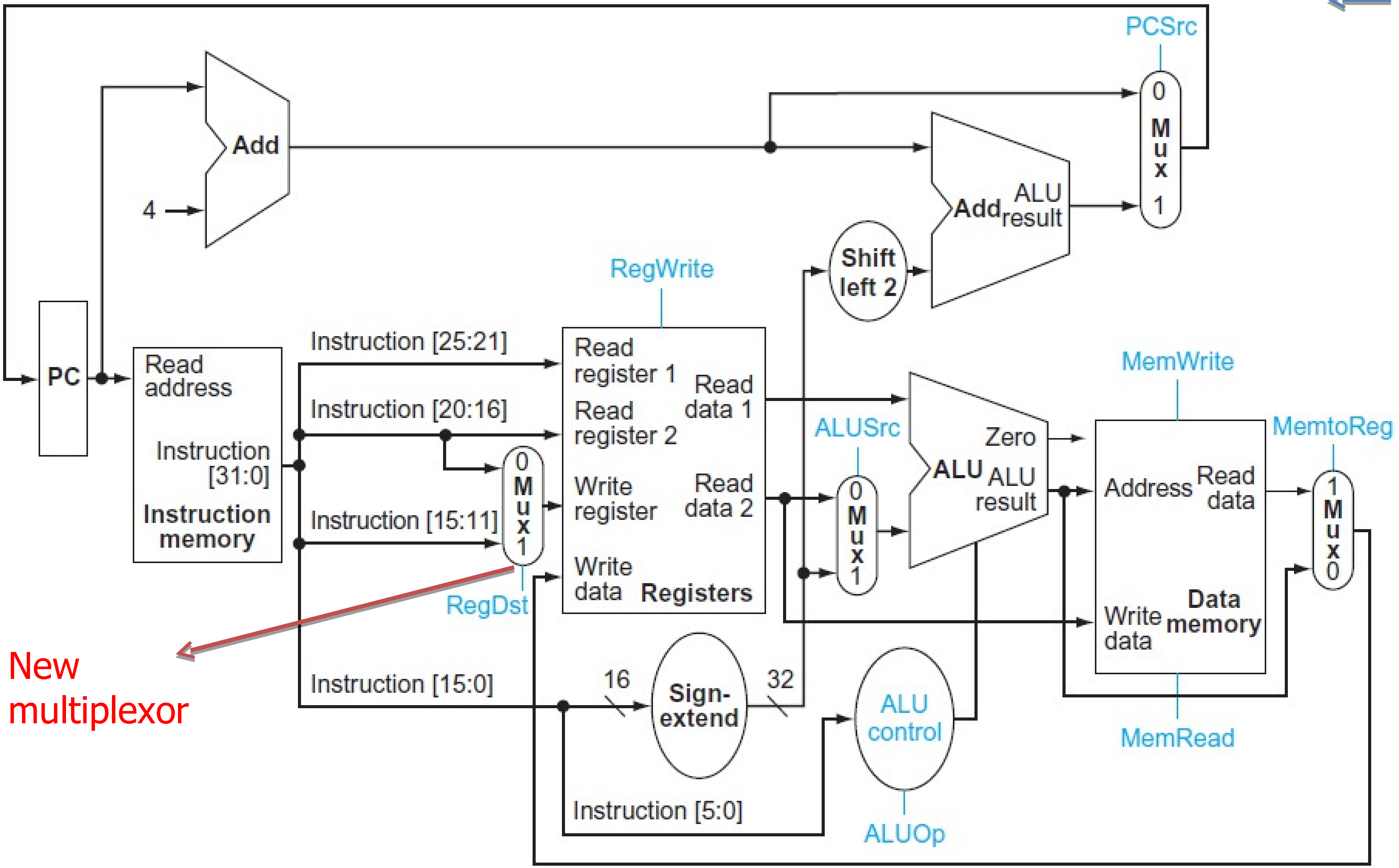
- Observations about MIPS instruction format
  - opcode is always in bits 31-26
  - two registers to be read are always rs (bits 25-21) and rt (bits 20-16)
  - base register for load/stores is always rs (bits 25-21)
  - 16-bit offset for branch equal and load/store is always bits 15-0
  - destination register for loads is in bits 20-16 (rt) while for R-type instructions it is in bits 15-11 (rd) (*will require multiplexor to select*)

Other Control signal values

- Asserted – value is 0
- Deasserted –value is 1

Data path with control lines

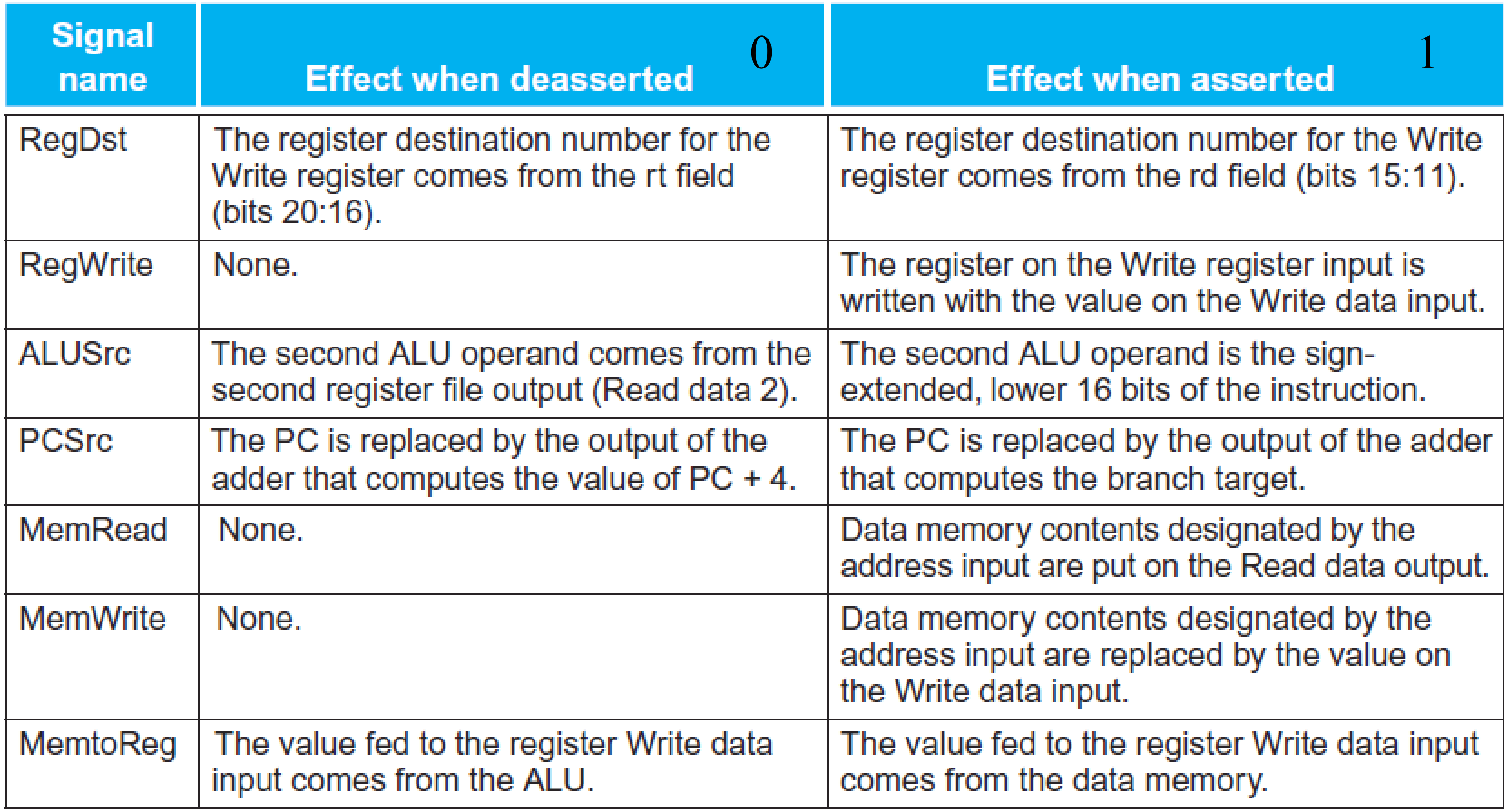
Control lines



New multiplexor

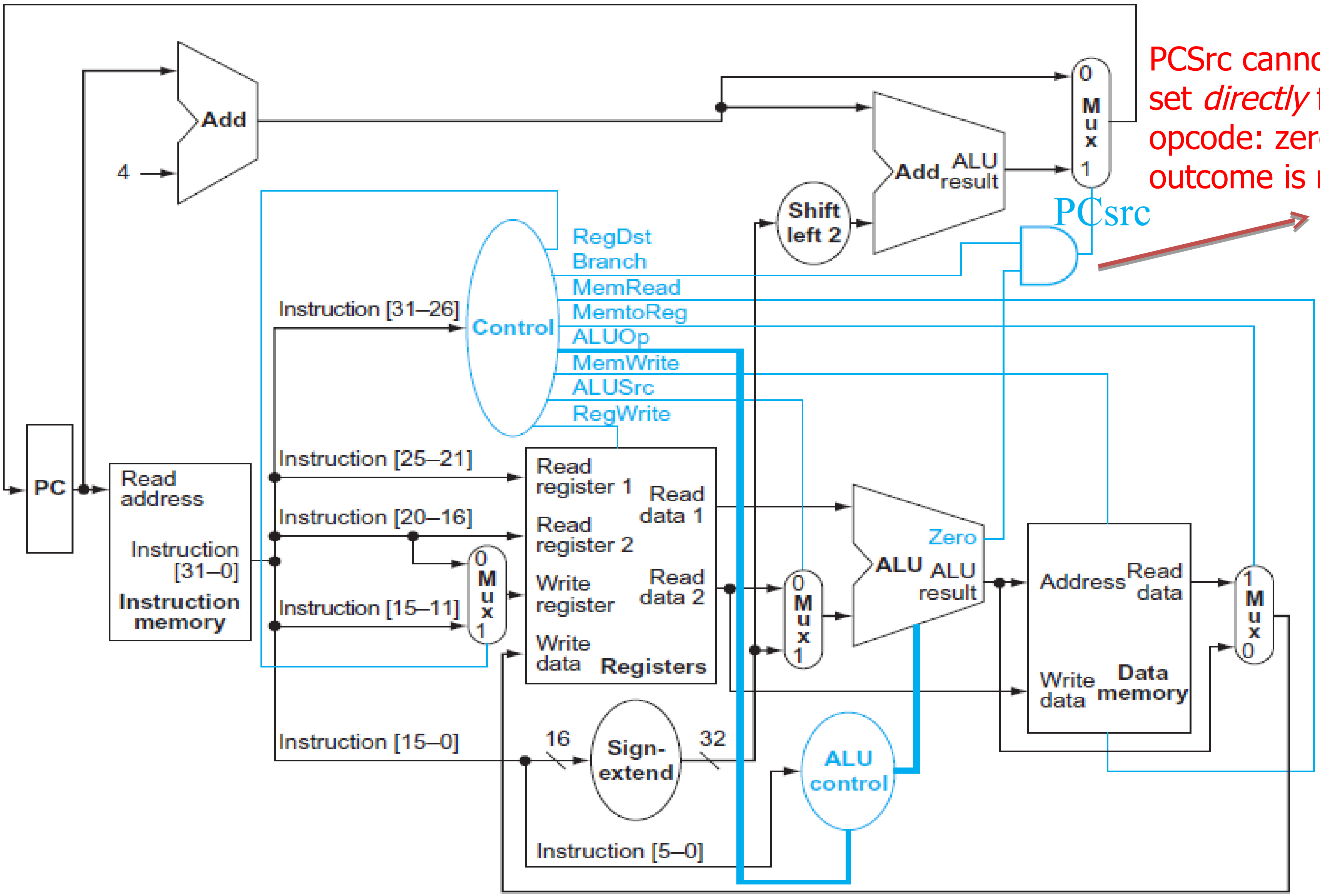
What are the functions of each control signals??

Effects of seven control signals



Signal name	Effect when deasserted 0	Effect when asserted 1
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.

Data path with control unit



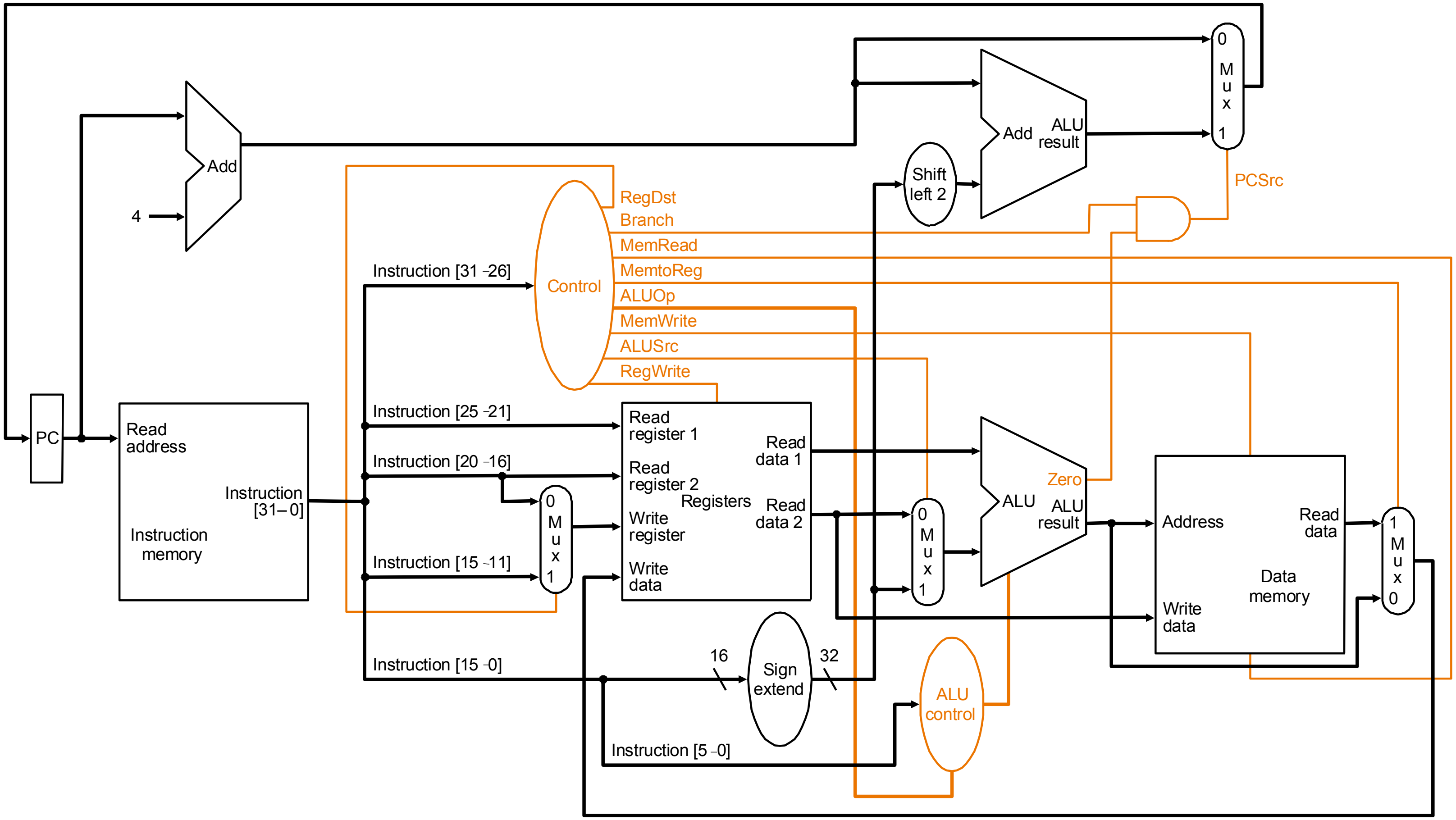
PCSrc cannot be set *directly* from the opcode: zero test outcome is required

Branch condition evaluation

instruction	Condition true	Condition false
beq	Zero =1	Zero = 0
Bnq	Zero = 1	Zero = 0

Zero	Branch instruction	Zero AND Branch
1	1	1
0	1	0
0	0	0
1	0	0

Datapath with Control II



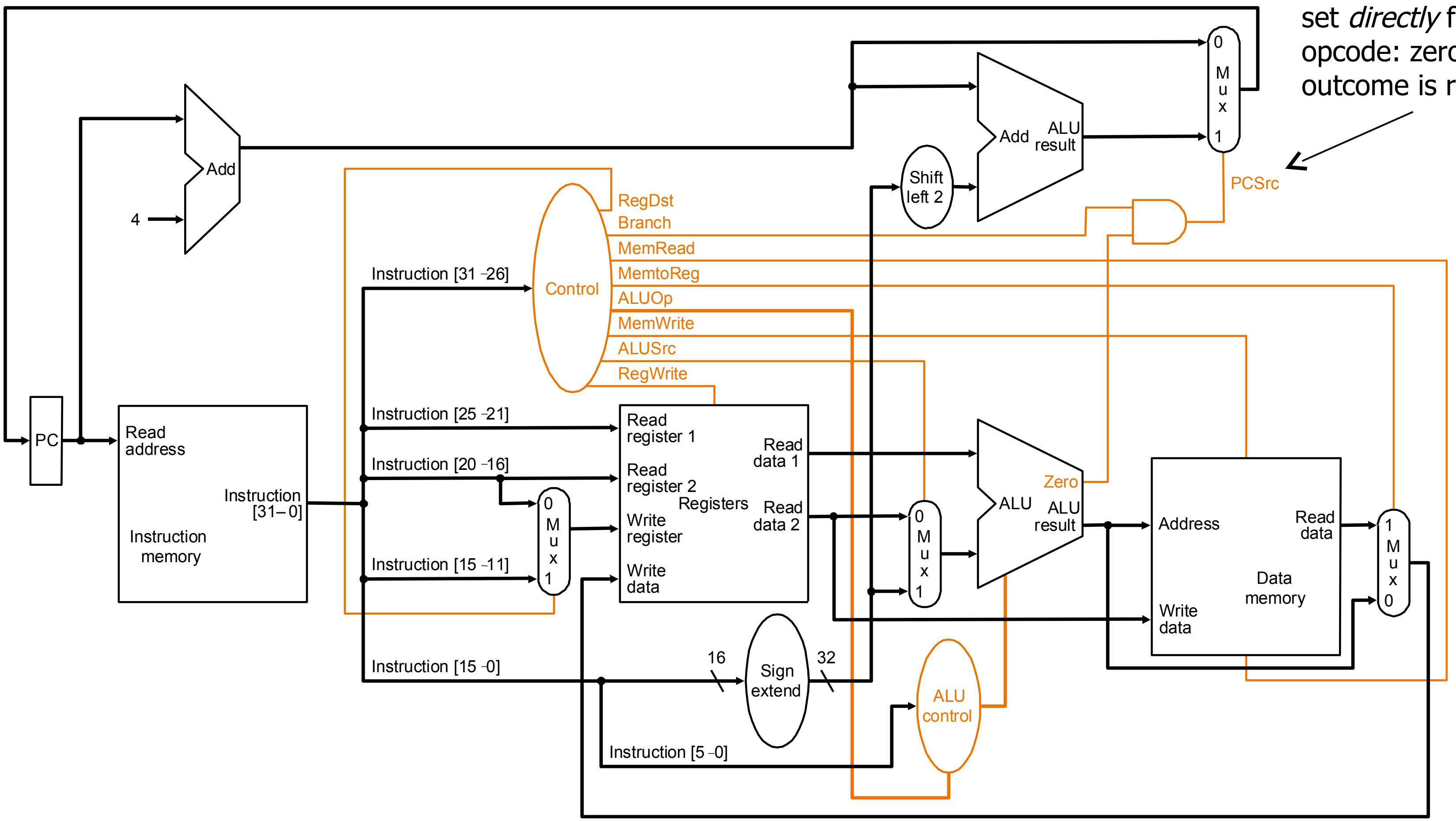
MIPS datapath with the control unit: input to control is the 6-bit instruction opcode field, output is seven 1-bit signals and the 2-bit ALUOp signal

Determining control signals for the MIPS datapath based on instruction opcode

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
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

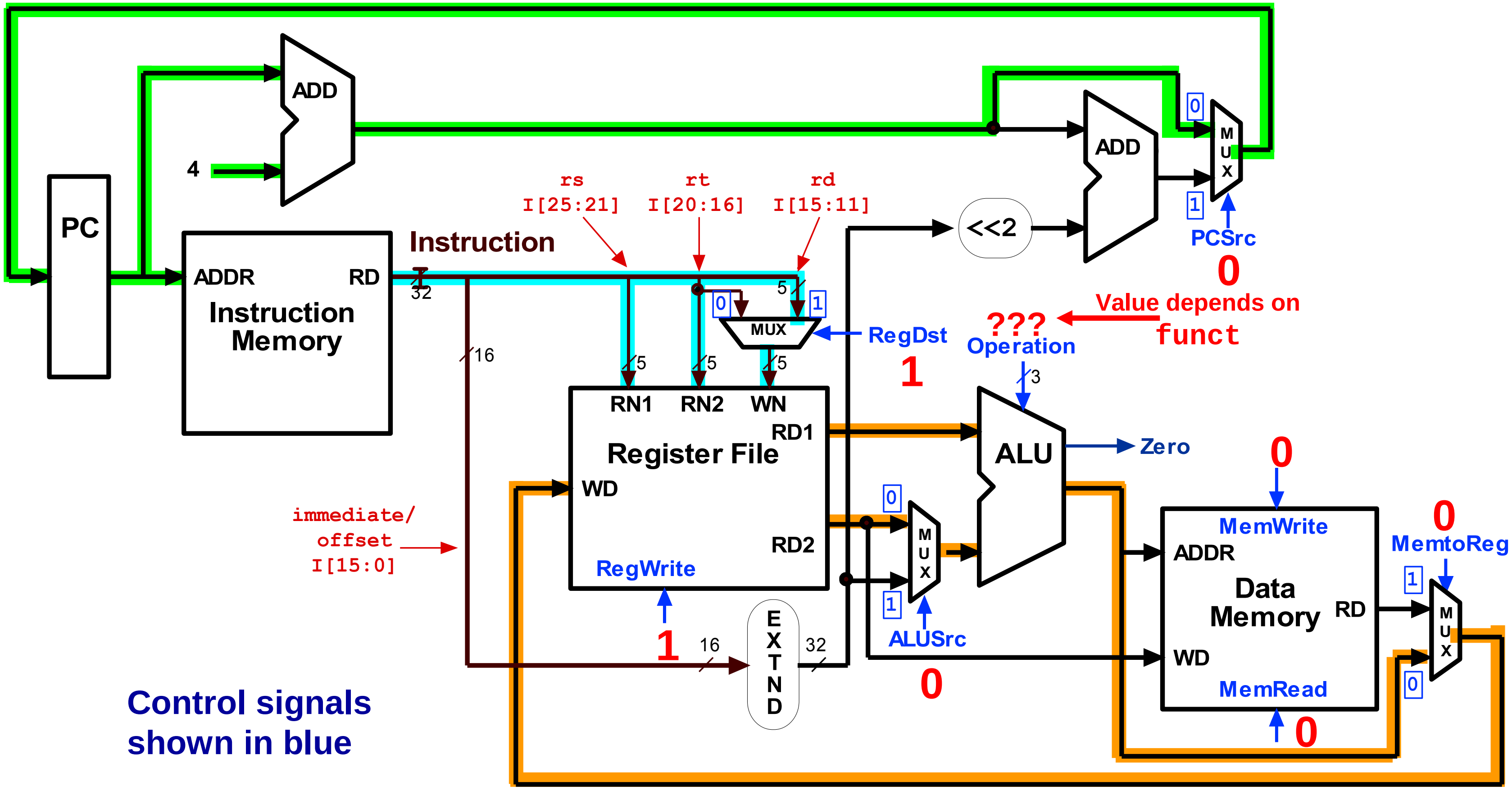
Control unit – 7 - 1-bit signals
1 - 2-bit signals

Datapath with Control II (contd)



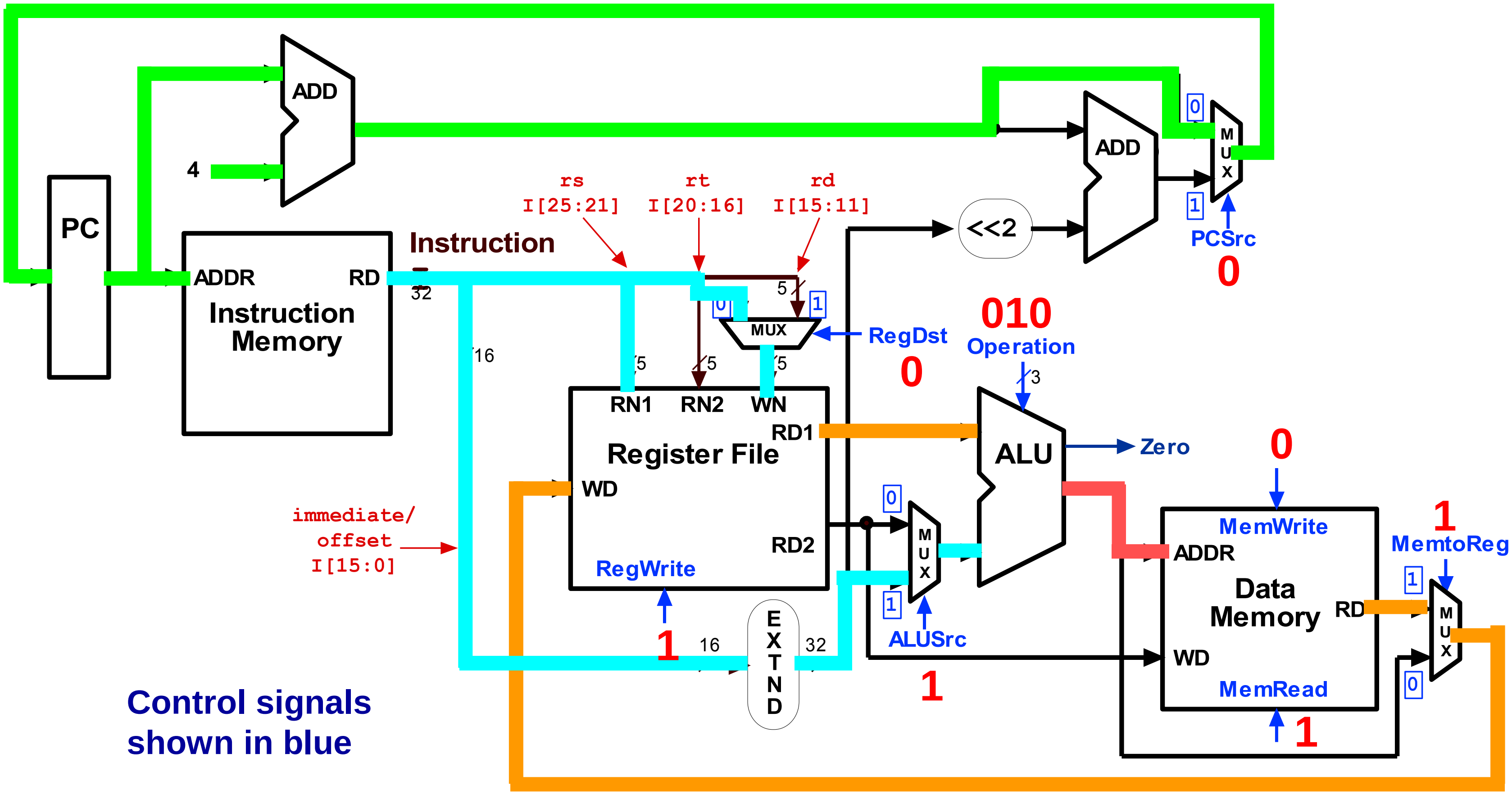
PCSrc cannot be set *directly* from the opcode: zero test outcome is required

Control Signals: R-Type Instruction



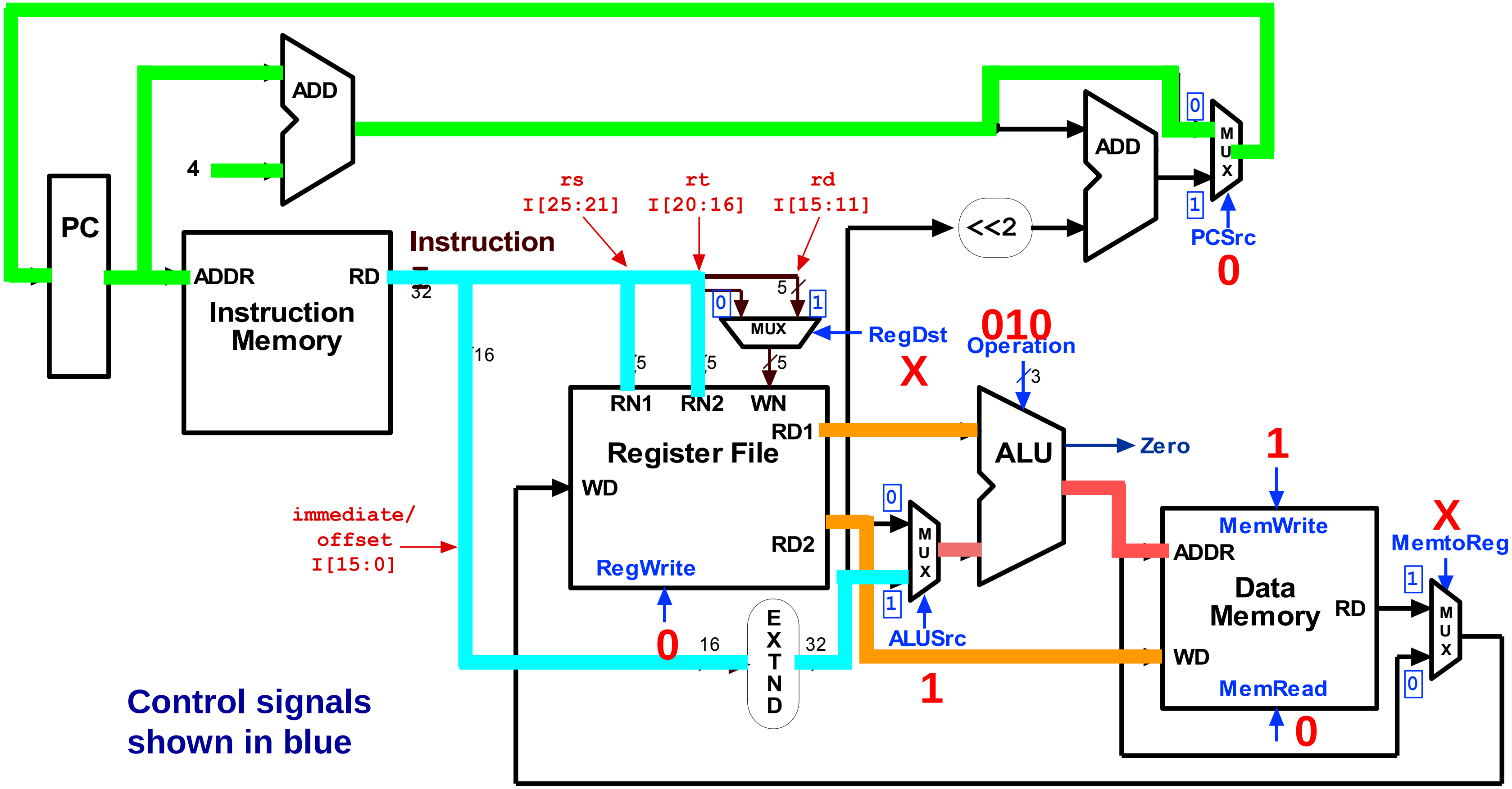
Control signals shown in blue

Control Signals: `lw` Instruction



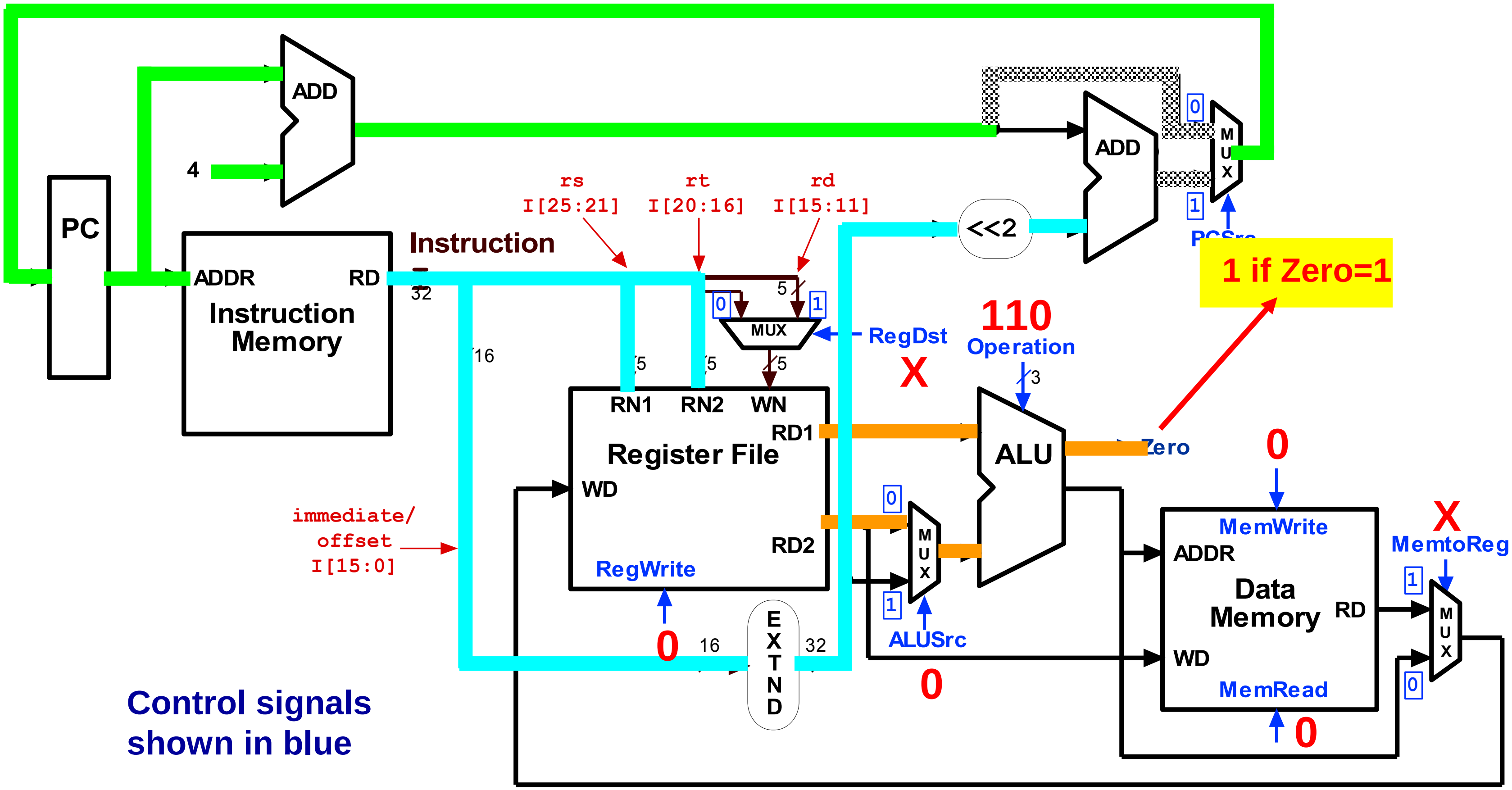
Control signals shown in blue

Control Signals: `sw` Instruction



Control signals shown in blue

Control Signals: `beq` Instruction



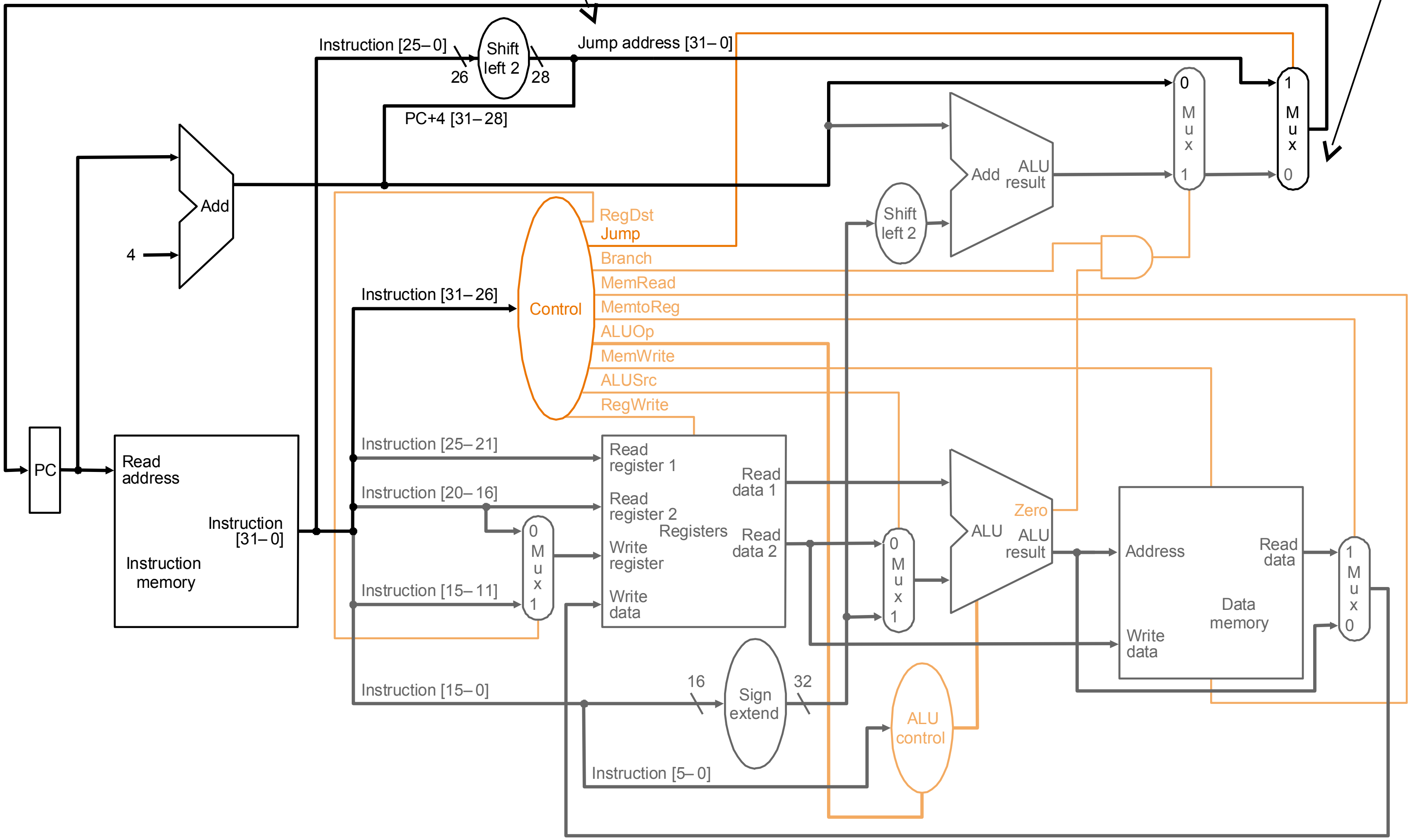
Control signals shown in blue

Datapath with Control III

Jump	opcode	address
	31-26	25-0

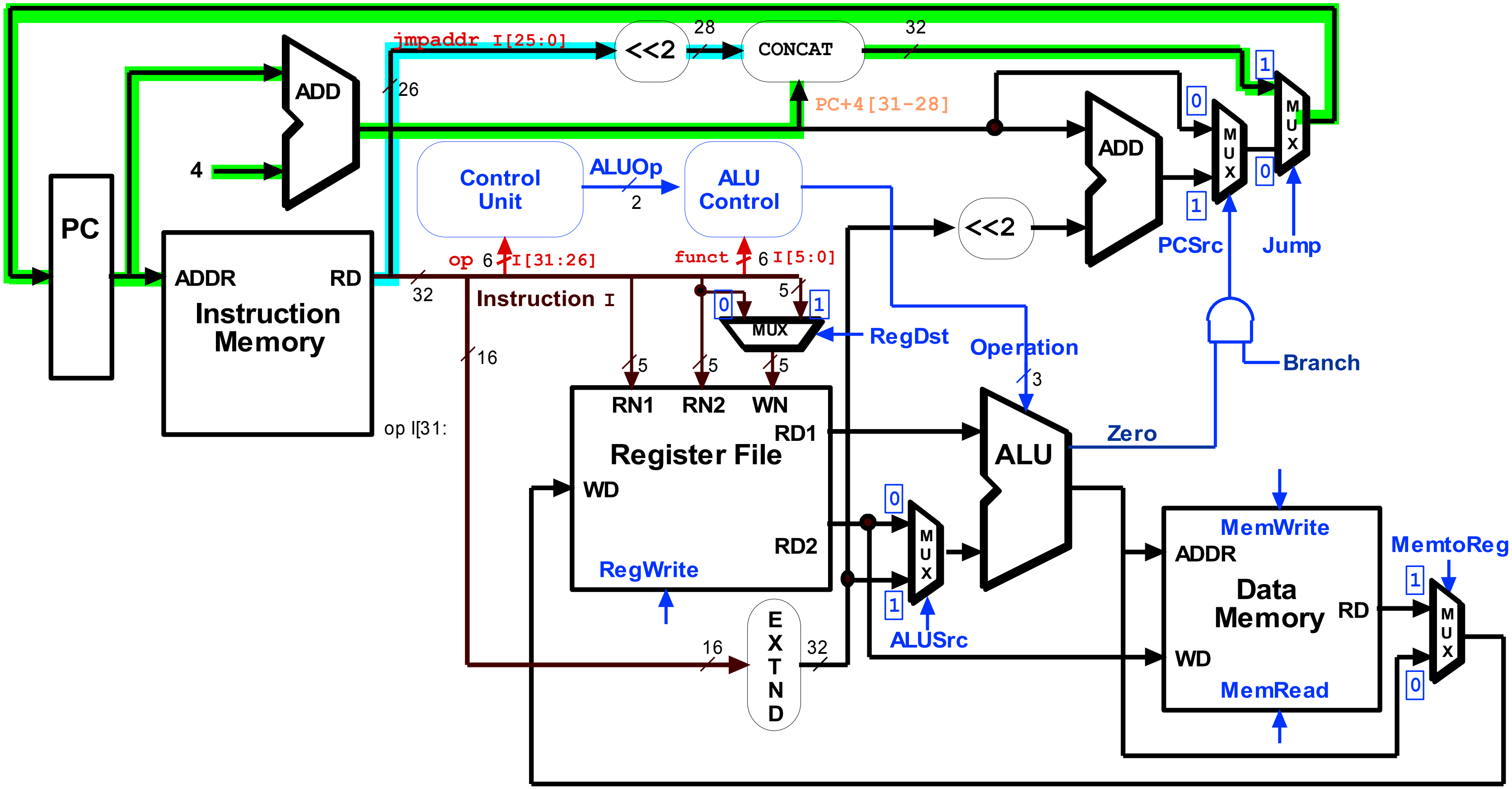
Composing jump target address

New multiplexor with additional control bit Jump

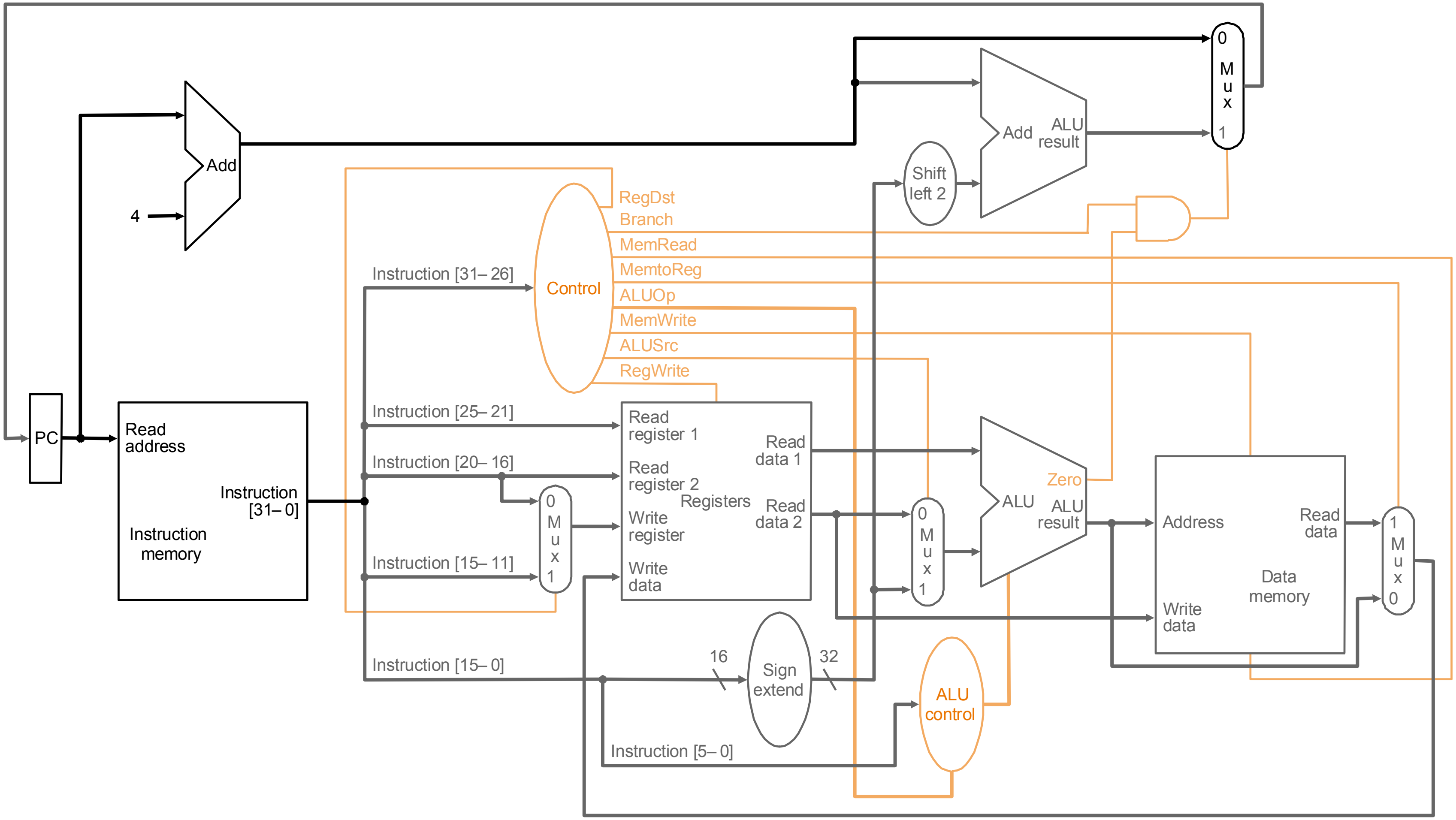


MIPS datapath extended to jumps: control unit generates new Jump control bit

Datapath Executing `j`

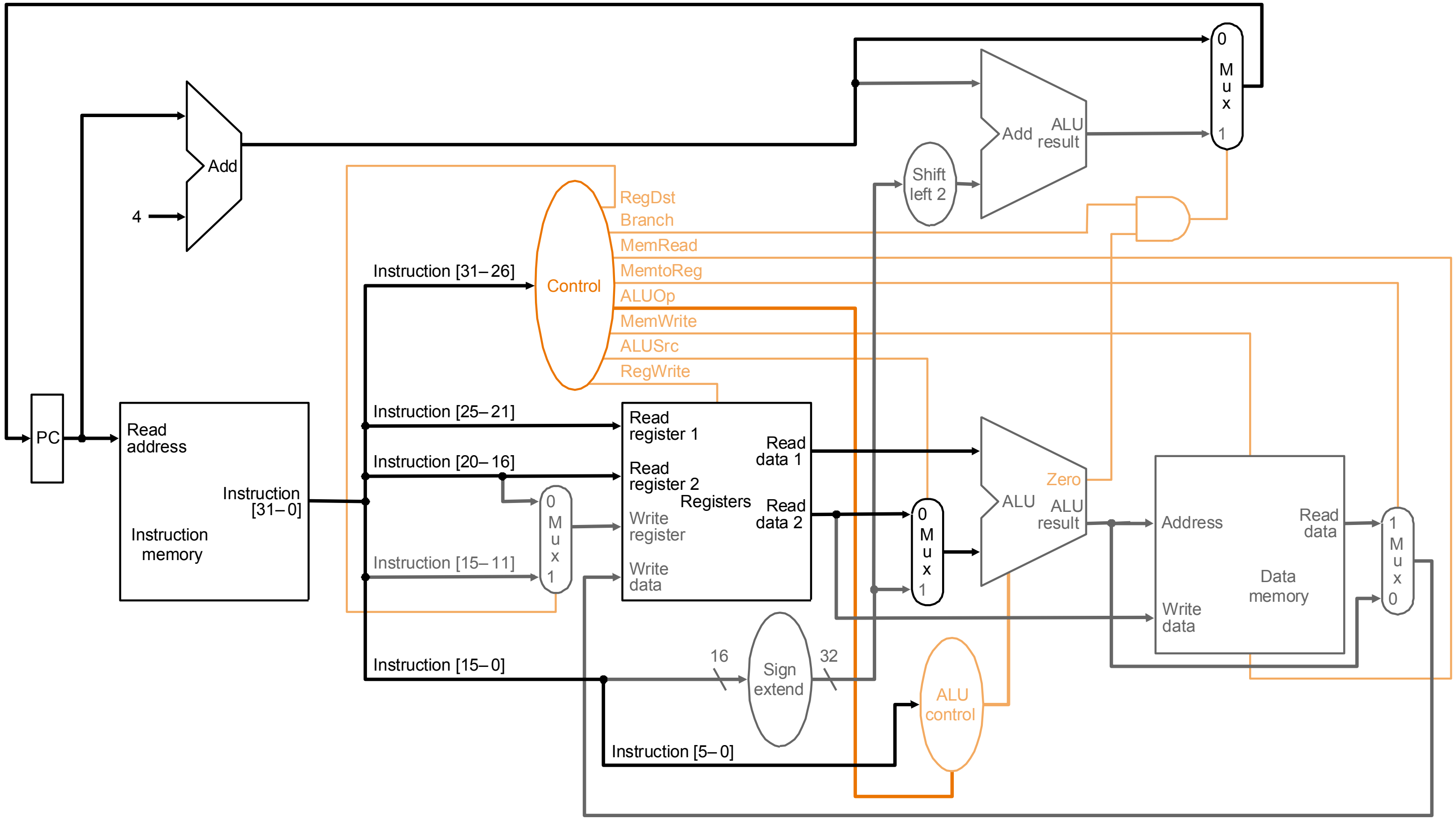


R-type Instruction: Step 1
add $t1, $t2, $t3 (active = bold)



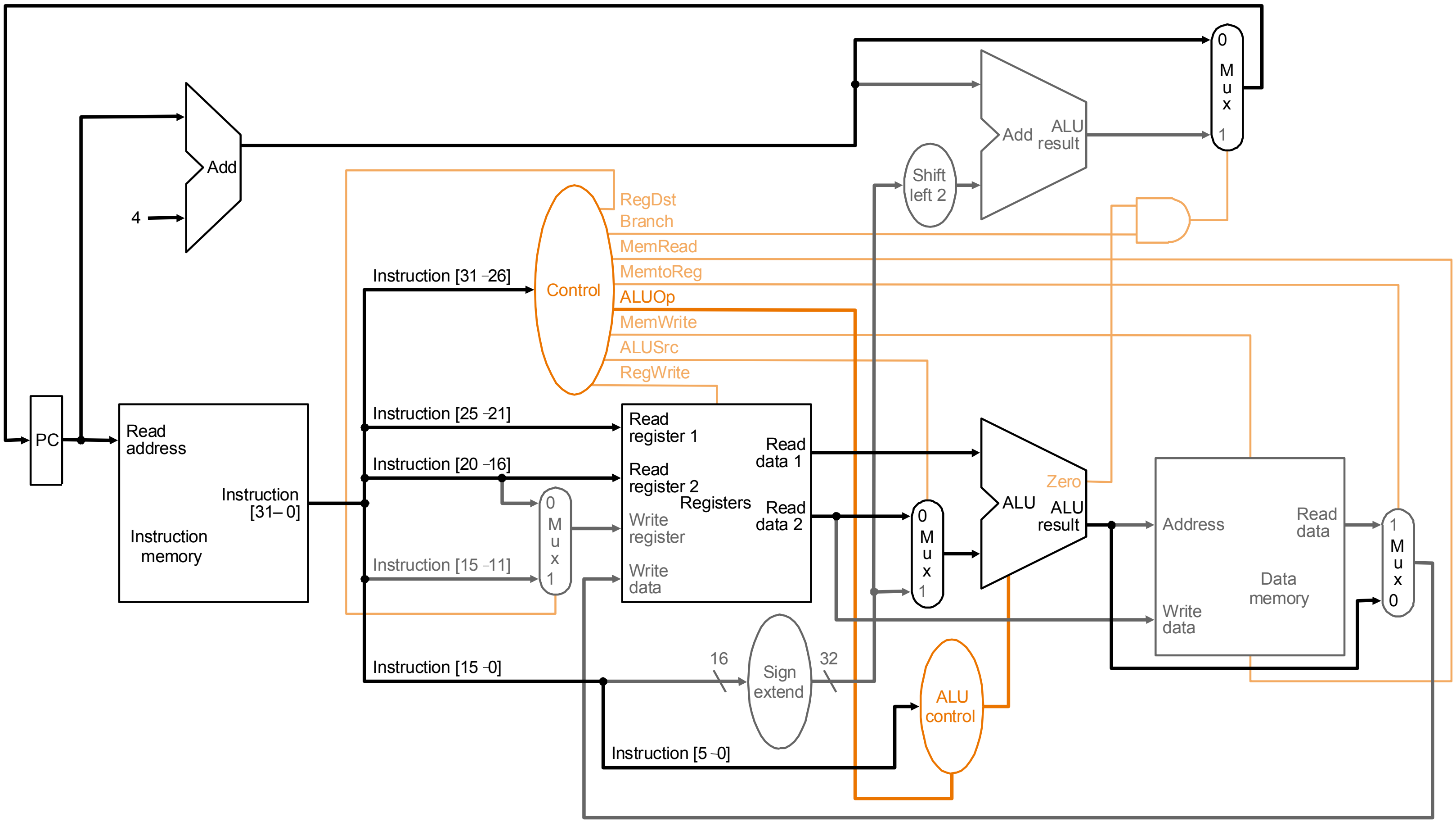
Fetch instruction and increment PC count

R-type Instruction: Step 2
add $t1, $t2, $t3 (active = bold)



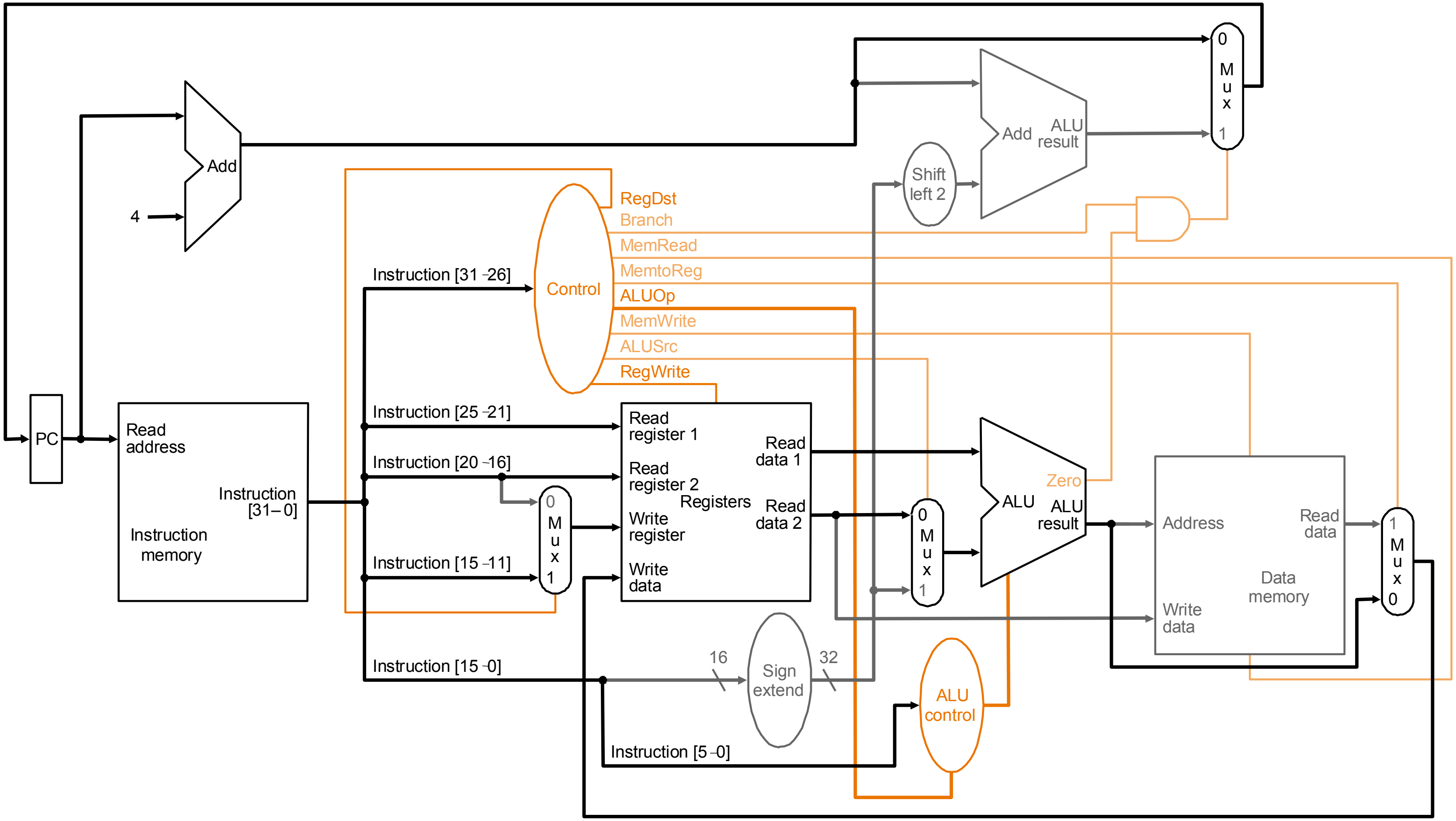
Read two source registers from the register file

R-type Instruction: Step 3
add $t1, $t2, $t3 (active = bold)



ALU operates on the two register operands

R-type Instruction: Step 4
add $t1, $t2, $t3 (active = bold)



Write result to register

Single-cycle Implementation Notes

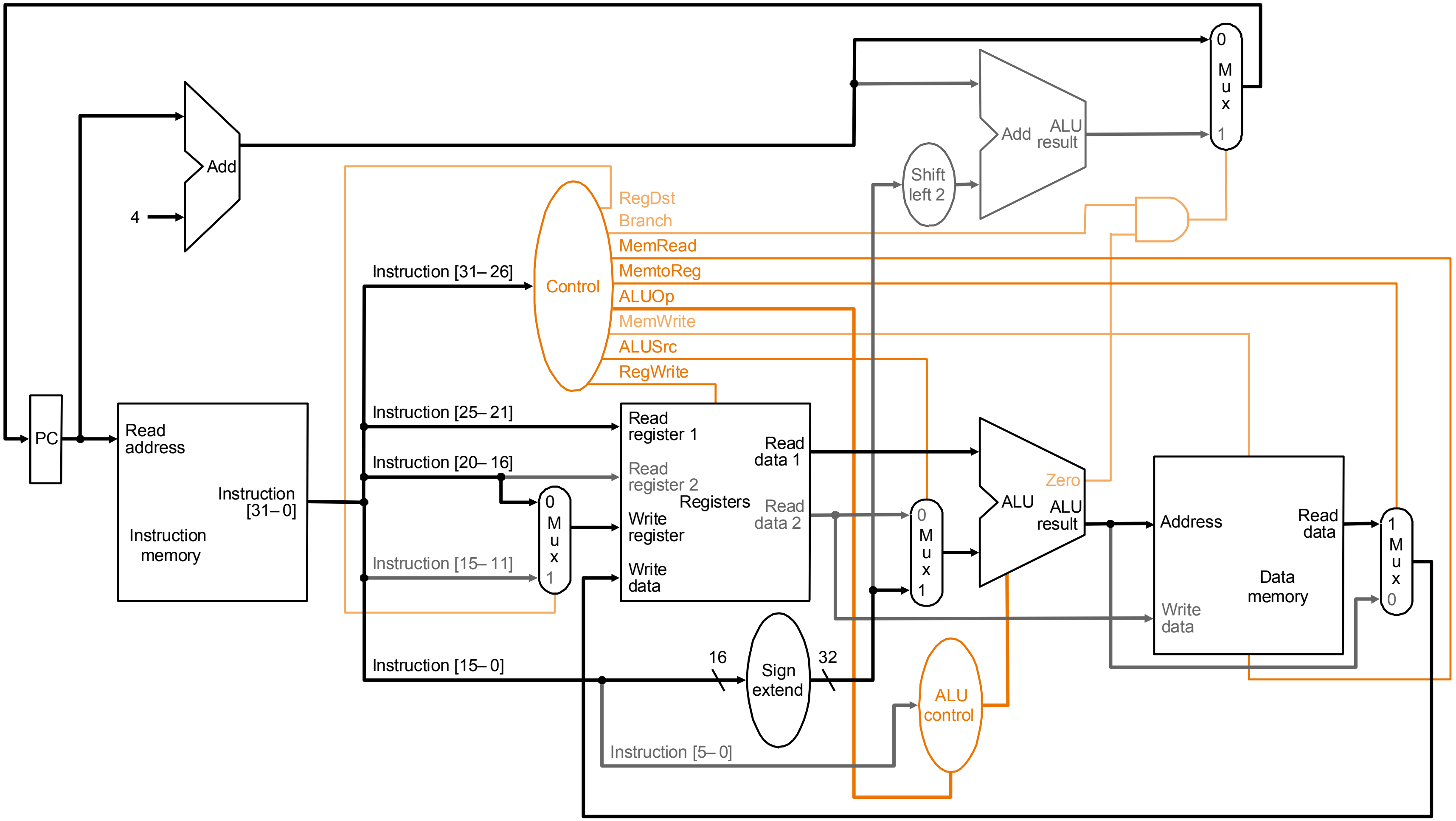
- *The steps are not really distinct* as each instruction completes in exactly one clock cycle – they simply indicate the sequence of data flowing through the datapath
- *The operation of the datapath during a cycle is purely combinational* – nothing is stored during a clock cycle
- Therefore, the machine is stable in a particular state at the start of a cycle and reaches a new stable state only at the end of the cycle
- *Very important for understanding single-cycle computing*:

See our simple Verilog single-cycle computer in the folder SimpleSingleCycleComputer in Verilog/Examples

Load Instruction Steps
lw $t1, offset($t2)

1. Fetch instruction and increment PC
2. Read base register from the register file: the base register ($t2) is given by bits 25-21 of the instruction
3. ALU computes sum of value read from the register file and the sign-extended lower 16 bits (offset) of the instruction
4. The sum from the ALU is used as the address for the data memory
5. The data from the memory unit is written into the register file: the destination register ($t1) is given by bits 20-16 of the instruction

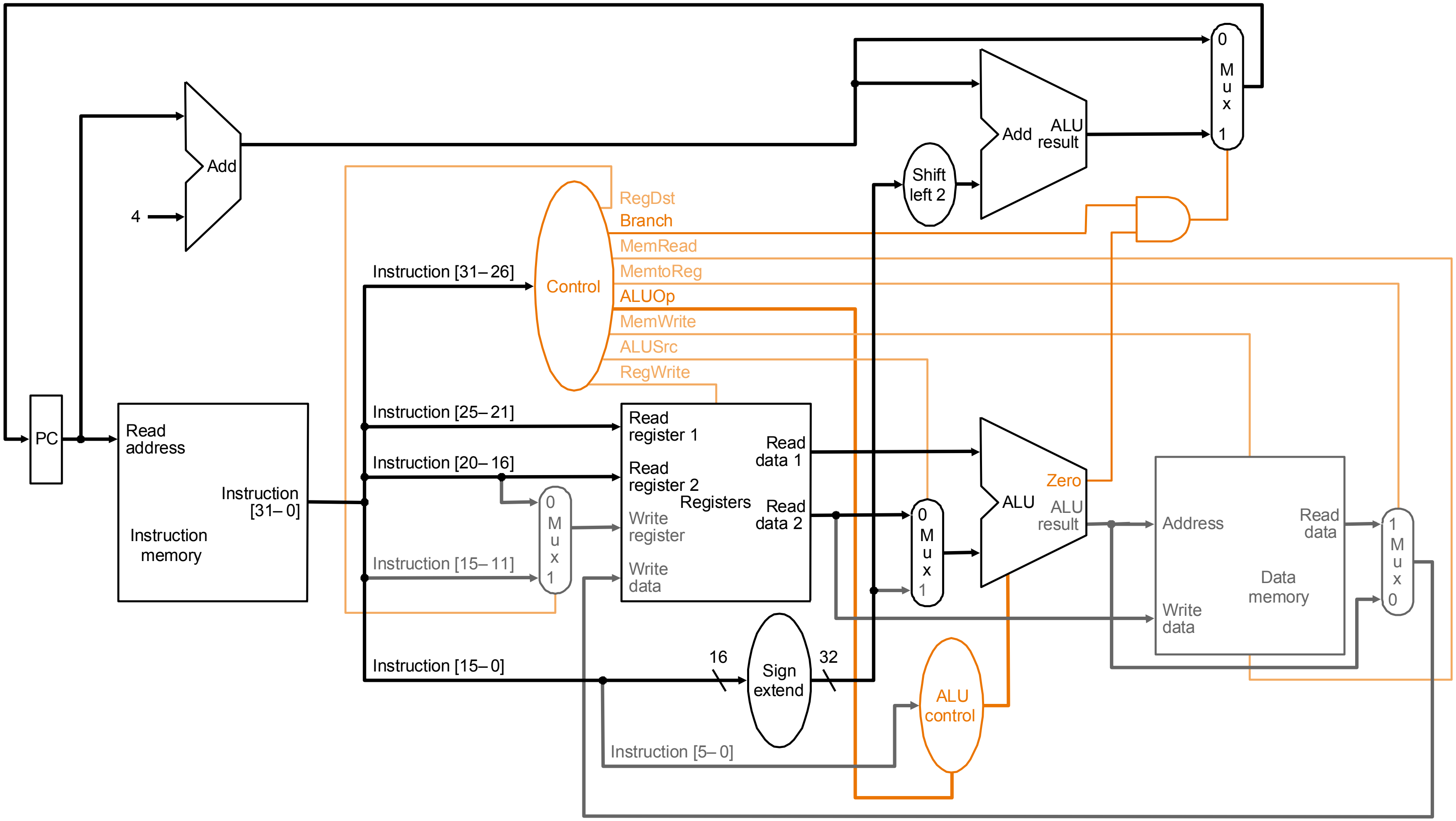
Load Instruction
lw $t1, offset($t2)



Branch Instruction Steps beq $t1, $t2, offset

1. Fetch instruction and increment PC
2. Read two register ($t1 and $t2) from the register file
3. ALU performs a subtract on the data values from the register file; the value of PC+4 is added to the sign-extended lower 16 bits (offset) of the instruction shifted left by two to give the branch target address
4. The Zero result from the ALU is used to decide which adder result (from step 1 or 3) to store in the PC

Branch Instruction
beq $t1, $t2, offset

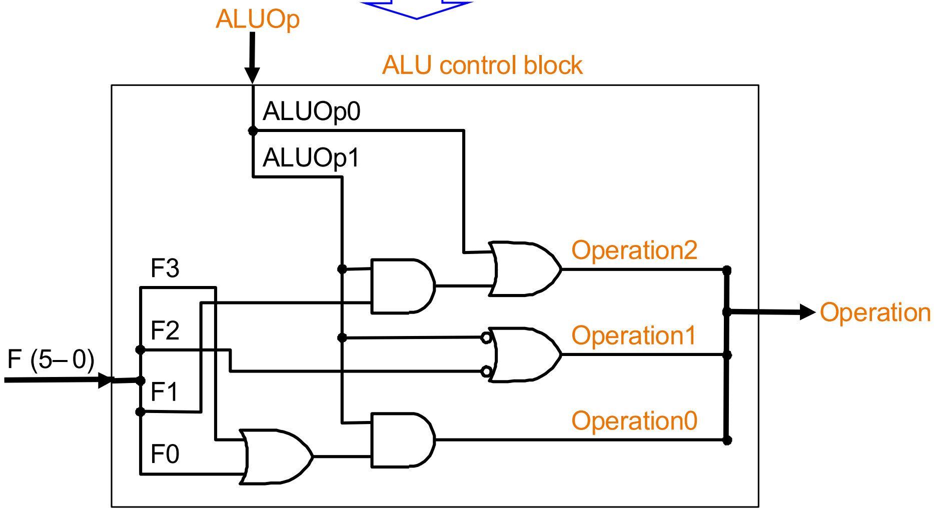


Implementation: ALU Control Block

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0 *	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

*Typo in text
Fig. 5.15: if it is X
then there is potential
conflict between
line 2 and lines 3-7!

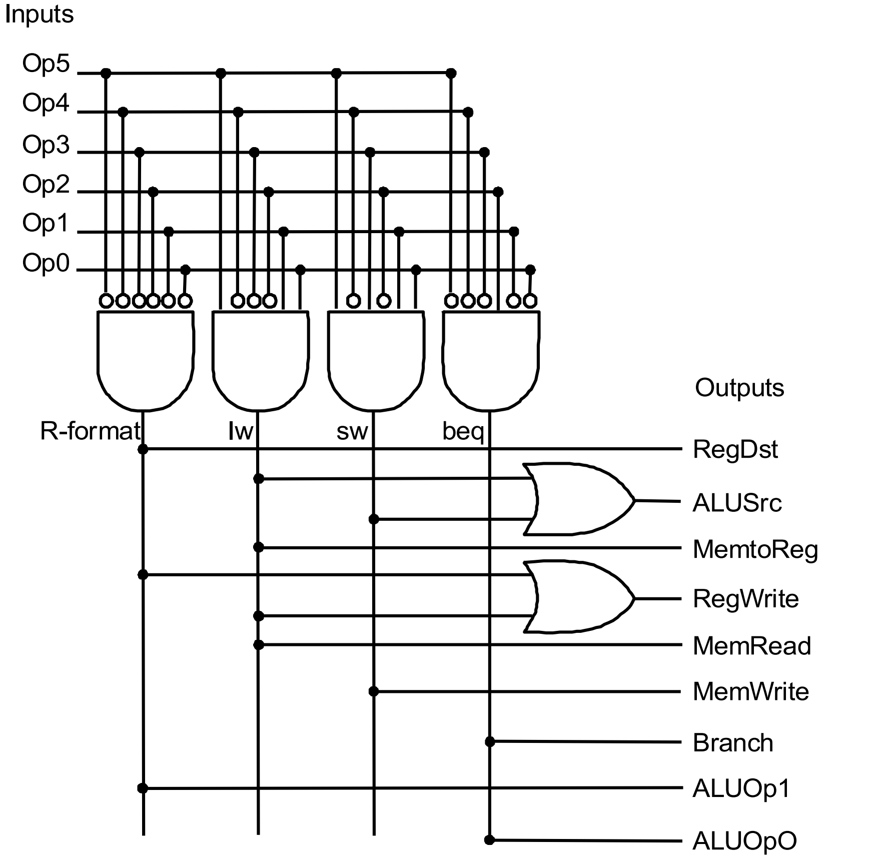


ALU control logic

Implementation: Main Control Block

	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
	ALUOP2	0	0	0	1

Truth table for main control signals



Main control PLA (programmable logic array): principle underlying PLAs is that any logical expression can be written as a sum-of-products

Single-Cycle Design Problems

- Assuming fixed-period clock every instruction datapath uses one clock cycle implies:
  - CPI = 1
  - cycle time determined by length of the longest instruction path (load)
    - but several instructions could run in a shorter clock cycle: *waste of time*
    - consider if we have more complicated instructions like floating point!
  - resources used more than once in the same cycle need to be duplicated
    - *waste of hardware and chip area*

Example: Fixed-period clock vs. variable-period clock in a single-cycle implementation

- Consider a machine with an additional floating point unit. Assume functional unit delays as follows
  - *memory*: 2 ns., *ALU* and *adders*: 2 ns., *FPU add*: 8 ns., *FPU multiply*: 16 ns., *register file access* (read or write): 1 ns.
  - *multiplexors*, *control unit*, *PC accesses*, *sign extension*, *wires*: no delay
- Assume instruction mix as follows
  - all loads take same time and comprise 31%
  - all stores take same time and comprise 21%
  - R-format instructions comprise 27%
  - branches comprise 5%
  - jumps comprise 2%
  - FP adds and subtracts take the same time and totally comprise 7%
  - FP multiplys and divides take the same time and totally comprise 7%
- *Compare the performance of (a) a single-cycle implementation using a fixed-period clock with (b) one using a variable-period clock where each instruction executes in one clock cycle that is only as long as it needs to be (not really practical but pretend it's possible!)*

Solution

Instruction class	Instr. mem.	Register read	ALU oper.	Data mem.	Register write	FPU add/ sub	FPU mul/ div	Total time ns.
Load word	2	1	2	2	1			8
Store word	2	1	2	2				7
R-format	2	1	2	0	1			6
Branch	2	1	2					5
Jump	2							2
FP mul/div	2	1			1		16	20
FP add/sub	2	1			1	8		12

- Clock period for fixed-period clock = longest instruction time = 20 ns.
- Average clock period for variable-period clock = 8 × 31% +

  7 × 21% + 6 × 27% + 5 × 5% + 2 × 2% + 20 × 7% + 12 × 7%

  = 7.0 ns.
- Therefore, performance$_{\text{var-period}}$/performance$_{\text{fixed-period}}$ = 20/7 = 2.9

Fixing the problem with single-cycle designs

- One solution: a variable-period clock with different cycle times for each instruction class
  - *unfeasible*, as implementing a variable-speed clock is technically difficult
- Another solution:
  - use a smaller cycle time…
  - …have different instructions take different numbers of cycles

    by breaking instructions into steps and fitting each step into one cycle
  - *feasible: multicyle approach*!

Summary

- Basic implementation of MIPS
- ALU control

Summary

- *Techniques described in this chapter to design datapaths and control are at the core of all modern computer architecture*
- Multicycle datapaths offer two great advantages over single-cycle
  - functional units can be reused within a single instruction if they are accessed in different cycles – reducing the need to replicate expensive logic
  - instructions with shorter execution paths can complete quicker by consuming fewer cycles
- Modern computers, in fact, take the multicycle paradigm to a higher level to achieve greater instruction throughput:
  - *pipelining* (next topic) where multiple instructions execute simultaneously by having cycles of different instructions overlap in the datapath
  - *the MIPS architecture was designed to be pipelined*

UNIT III PROCESSOR AND CONTROL UNIT

Pipelining – Pipelined datapath and control – Handling Data hazards & Control hazards – Exceptions.

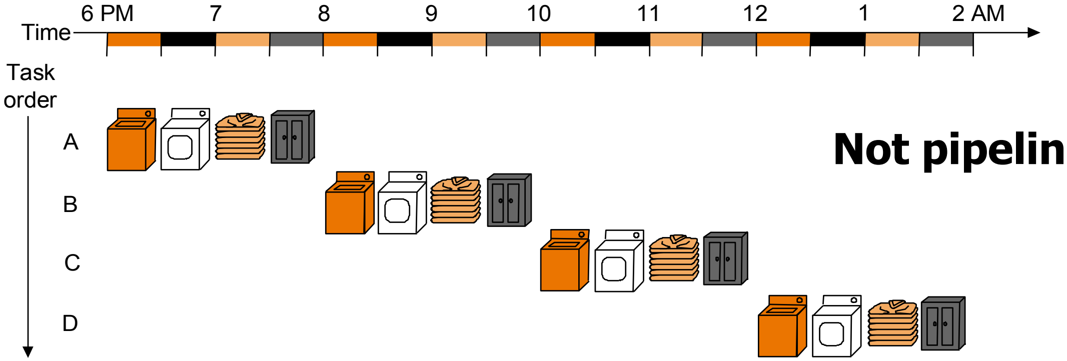
Reference:

- Chapter 4 – The Processor
- Book - David A. Patterson and John L. Hennessey, “Computer organization and design‟, Morgan Kauffman / Elsevier, Fifth edition, 2014.

Overview Of Pipeline

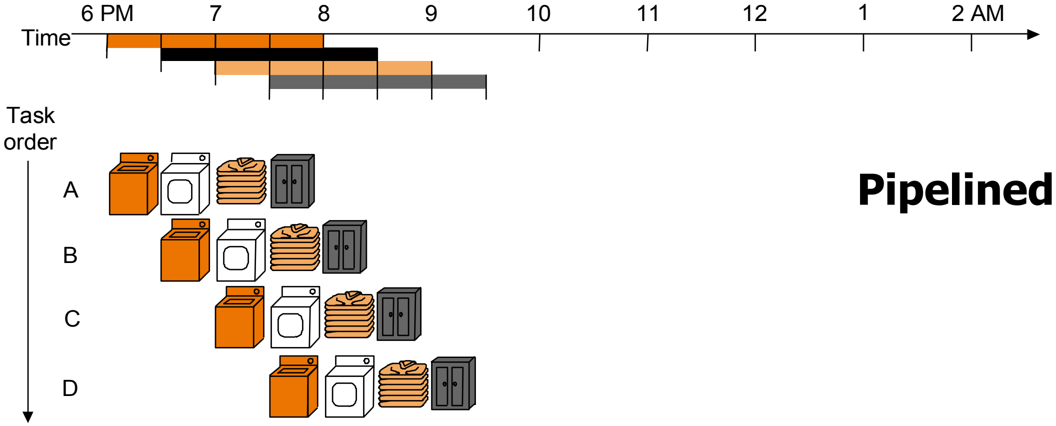
Pipelining

- Start work ASAP!! Do not waste time!



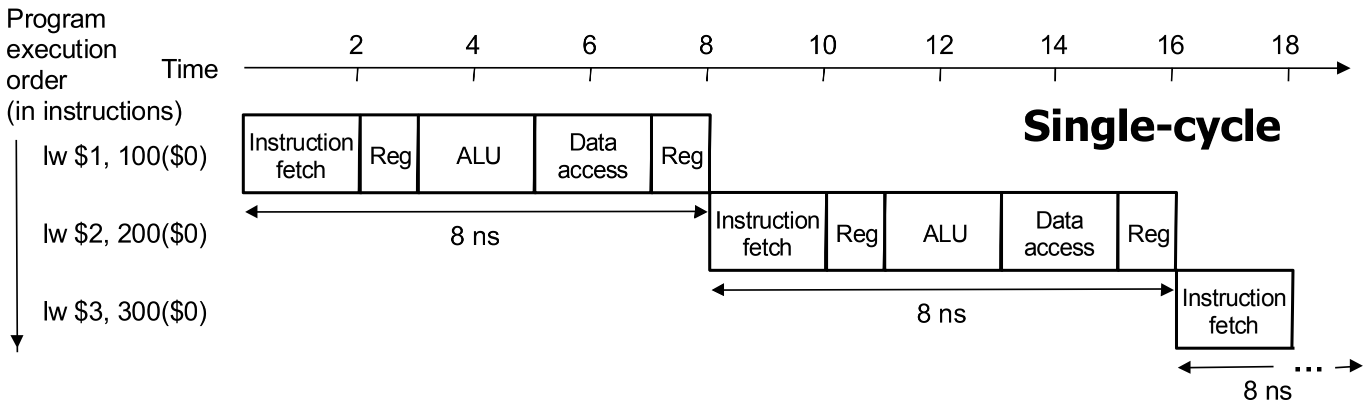
Not pipelined

Assume 30 min. each task – wash, dry, fold, store – and that separate tasks use separate hardware and so can be overlapped

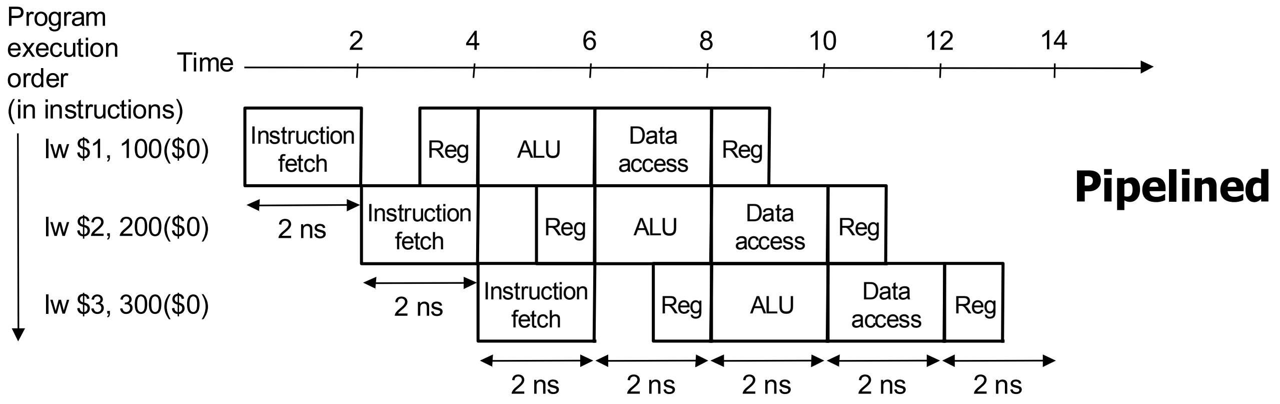


Pipelined

Pipelined vs. Single-Cycle Instruction Execution: the Plan



Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.



Computer Architecture
Pipeline

Pipelining: Keep in Mind

- Pipelining *does not reduce latency* of a single task, it *increases throughput* of entire workload
- Pipeline rate *limited by longest stage*
  - *potential* speedup = number pipe stages
  - *unbalanced lengths* of pipe stages reduces speedup
- Time to *fill* pipeline and time to *drain* it – when there is *slack* in the pipeline – reduces speedup

Example Problem

- *Problem: for the laundry fill in the following table when*
  1. *the stage lengths are 30, 30, 30 30 min., resp.*
  2. *the stage lengths are 20, 20, 60, 20 min., resp.*

Person	Unpipelined finish time	Pipeline 1 finish time	Ratio unpipelined to pipeline 1	Pipeline 2 finish time	Ratio unpiplelined to pipeline 2
1					
2					
3					
4					
n					

- *Come up with a formula for pipeline speed-up!*

Pipelining MIPS

- What makes it easy with MIPS?
  - all *instructions are same length*
    - so fetch and decode stages are similar for all instructions
  - just a *few instruction formats*
    - simplifies instruction decode and makes it possible in one stage
  - *memory operands appear only in load/stores*
    - so memory access can be deferred to exactly one later stage
  - *operands are aligned in memory*
    - one data transfer instruction requires one memory access stage

Pipelining MIPS

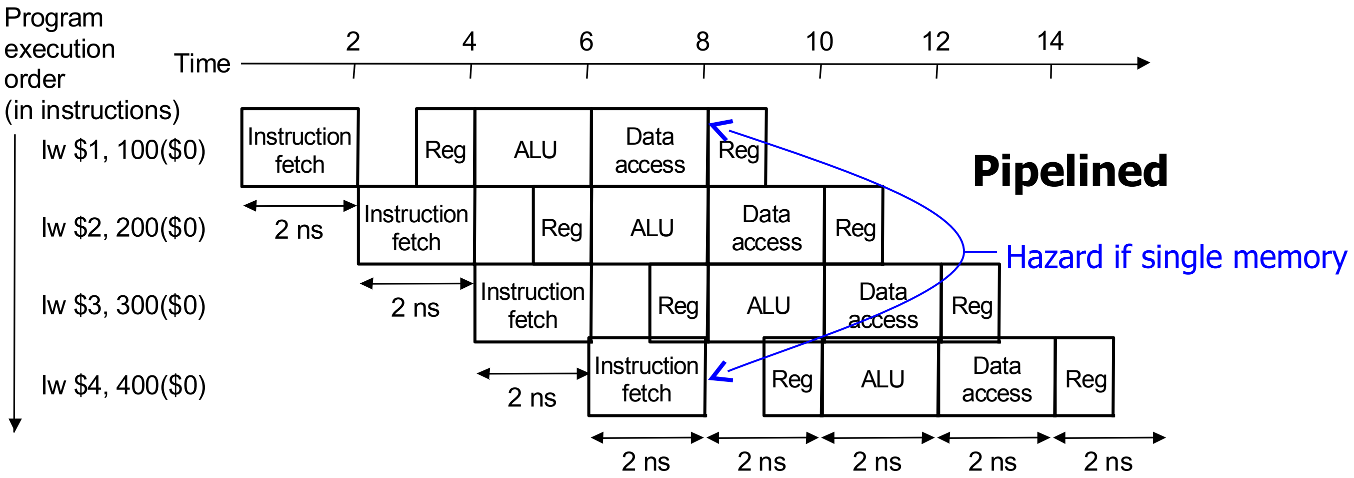
- What makes it hard?
  - *structural hazards*: different instructions, at different stages, in the pipeline want to use the same hardware resource
  - *control hazards*: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
  - *data hazards*: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline

- Before actually building the pipelined datapath and control we first briefly examine these potential hazards individually…

Pipeline Hazards

Structural Hazards

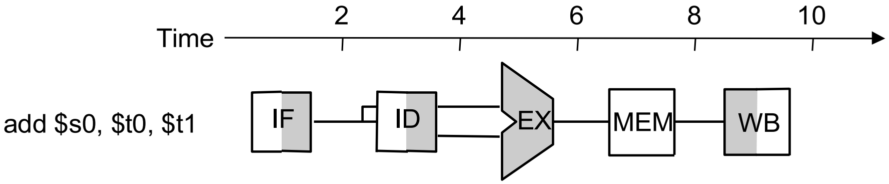
- *Structural hazard*: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth `lw` instructions



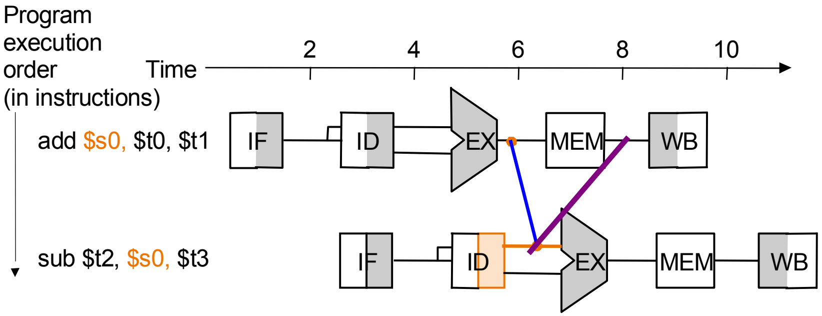
- *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

Data Hazards

- *Data hazard*: instruction needs data from the result of a previous instruction still executing in pipeline
- Solution *Forward* data if possible…



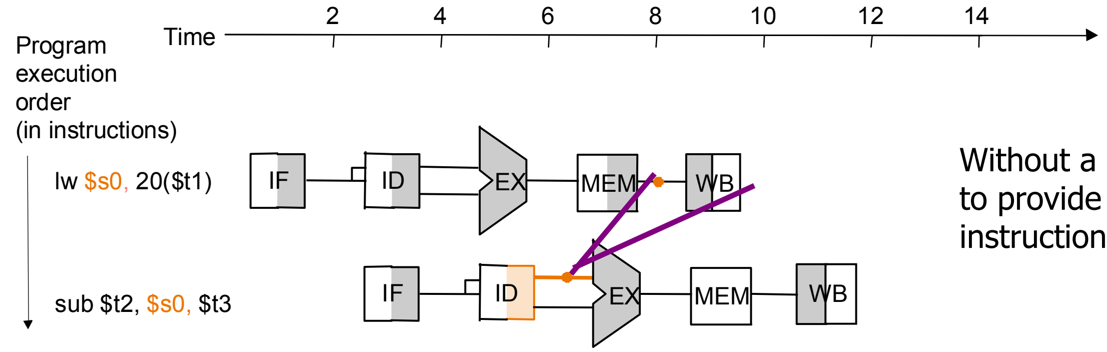
Instruction pipeline diagram: shade indicates use – left=write, right=read



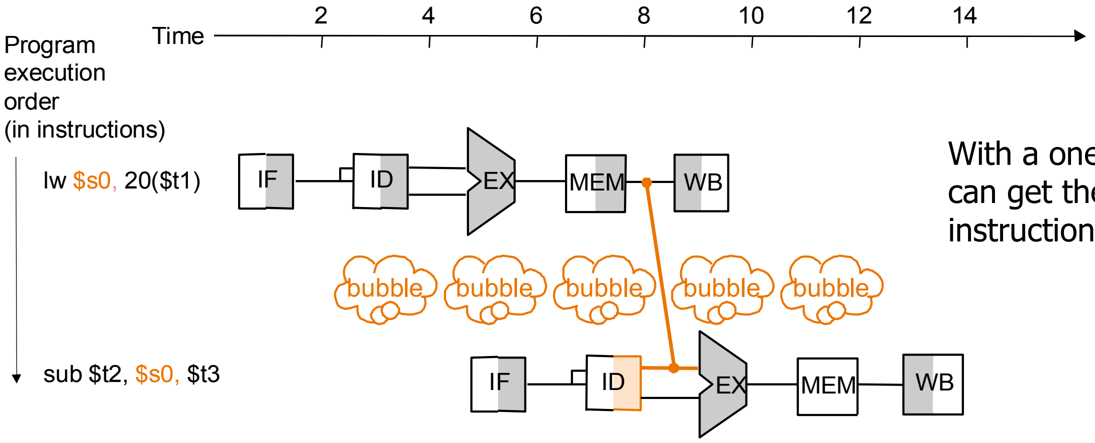
Without forwarding – blue line – data has to go back in time; with forwarding – red line – data is available in time

Data Hazards

- Forwarding may not be enough
  - e.g., if an R-type instruction following a load uses the result of the load – called *load-use data hazard*



Without a stall it is impossible to provide input to the `sub` instruction in time



With a one-stage stall, forwarding can get the data to the `sub` instruction in time

Reordering Code to Avoid Pipeline Stall (Software Solution)

- Example:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Data hazard

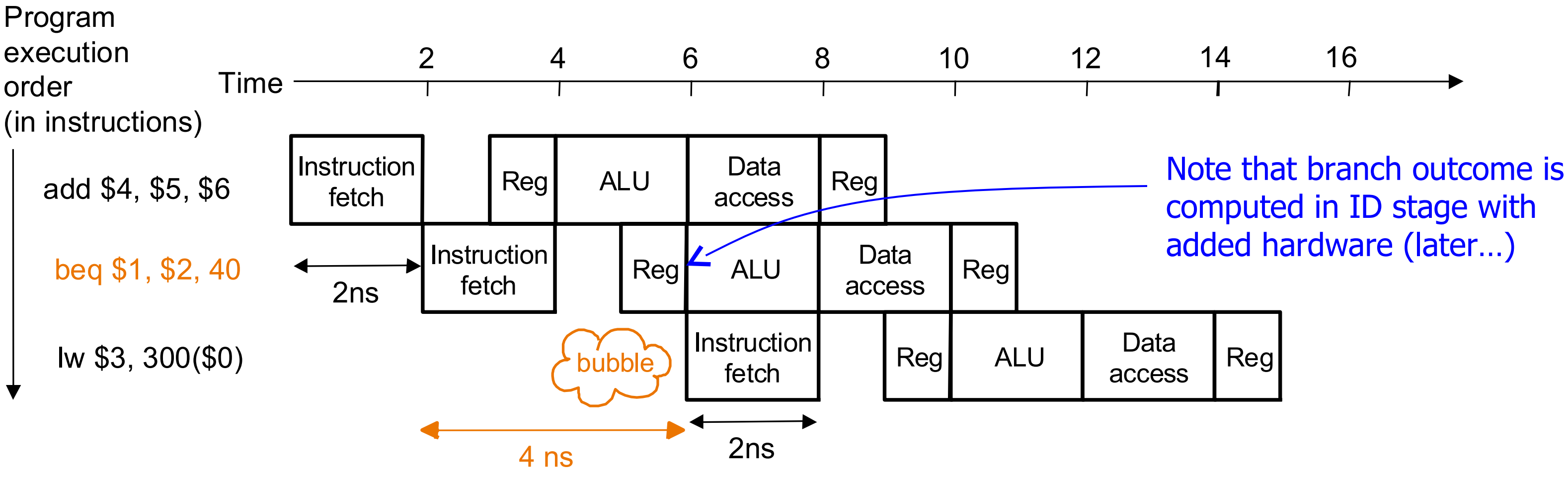
- Reordered code:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

Interchanged

Control Hazards

- *Control hazard*: need to make a decision based on the result of a previous instruction still executing in pipeline
- Solution 1 *Stall* the pipeline

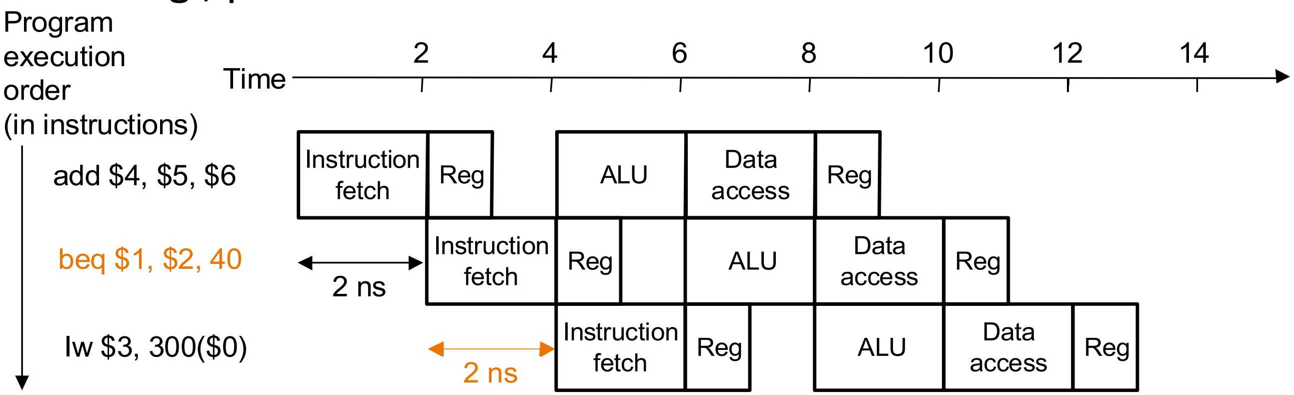


Note that branch outcome is computed in ID stage with added hardware (later…)

Pipeline stall

Control Hazards

- Solution 2 *Predict* branch outcome
  - e.g., predict *branch-not-taken* :



Program execution order (in instructions)

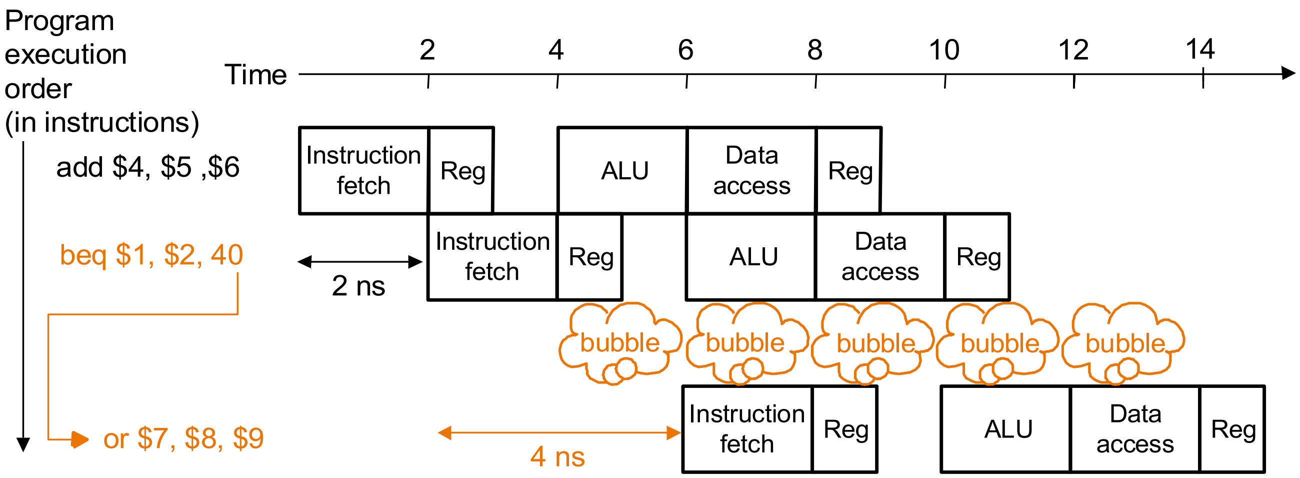
Time 2 4 6 8 10 12 14

add $4, $5, $6 — Instruction fetch | Reg | ALU | Data access | Reg

beq $1, $2, 40 — 2 ns — Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0) — 2 ns — Instruction fetch | Reg | ALU | Data access | Reg

Prediction success



Program execution order (in instructions)

Time 2 4 6 8 10 12 14

add $4, $5 ,$6 — Instruction fetch | Reg | ALU | Data access | Reg

beq $1, $2, 40 — 2 ns — Instruction fetch | Reg | ALU | Data access | Reg

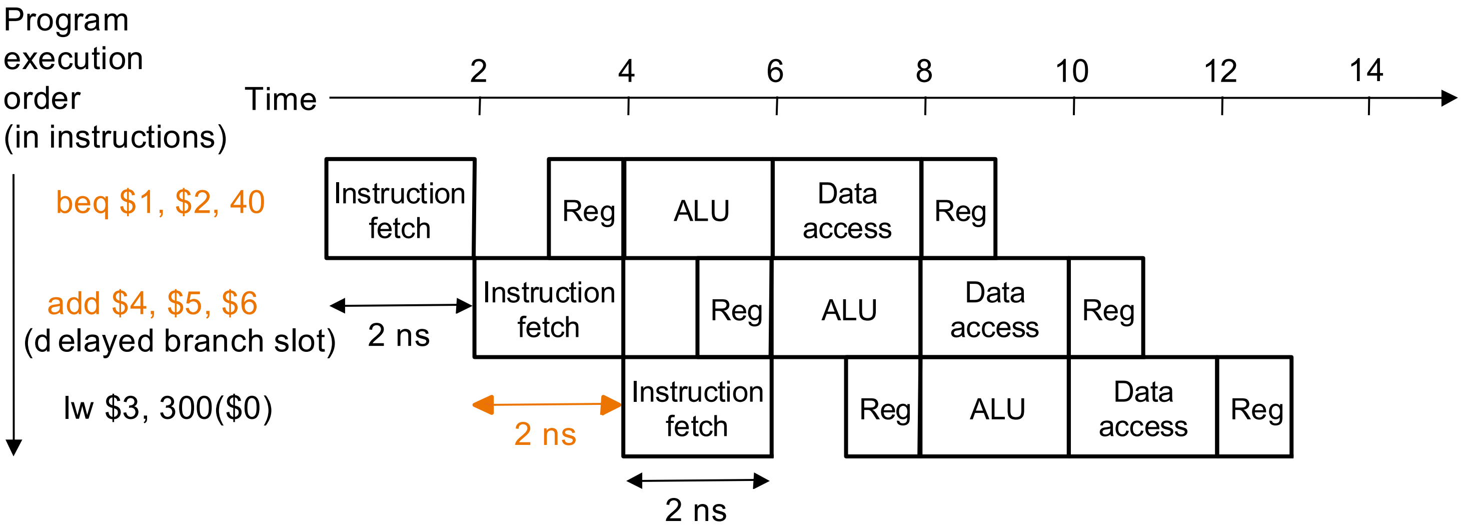
bubble bubble bubble bubble bubble

or $7, $8, $9 — 4 ns — Instruction fetch | Reg | ALU | Data access | Reg

Prediction failure: undo (=flush) `lw`

Control Hazards

- Solution 3 *Delayed branch:* always execute the sequentially next statement with the branch executing after one instruction delay – compiler’s job to find a statement that can be put in the slot that is independent of branch outcome
  - MIPS does this – but it is an option in SPIM (Simulator -> Settings)



Program execution order (in instructions)

Time

2 4 6 8 10 12 14

beq $1, $2, 40

Instruction fetch | Reg | ALU | Data access | Reg

add $4, $5, $6 (d elayed branch slot)

2 ns

Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0)

2 ns

Instruction fetch | Reg | ALU | Data access | Reg

2 ns

Delayed branch `beq` is followed by `add` that is independent of branch outcome

Dynamic Branch Prediction

- Prediction of branches at runtime using runtime information.
- Ex: Restaurant
- Two schemes:
    - 1-bit scheme
    - 2-bit scheme

1-bit scheme

- **branch prediction buffer -** Also called **branch history table.**
- A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.
- The memory contains a bit that says whether the branch was recently taken or not

Branch History Table

	Bit indicating branch taken(1) or not taken(0)
1000	0
1012	1
2046	0
2116	1

```
1000 Beq $s2,$s1, exit
1012 Beq $s2,$s1, for
2046 Beq $s2,$s1, else
```

Assume 0 – bit indicating branch not taken
1 – bit indicating branch taken

Disadvantage of 1-bit scheme

- What if the branch is taken and not taken the alternatively?
  - Then the prediction table will be wrong always.
  - (ie) prediction will be always a **failure**

Disadvantage example

```
1000 i=1;
1004 while(i<=9){
1008            i++;
        }
```

	Bit indicating branch taken(1)or not taken(0)
1004	1

Initially – bit is set to 1

1st iteration – misprediction(since branch is not taken)

Next 8 iteration – prediction is success

10th - exit iteration – misprediction (since branch is taken)

Disadvantage:

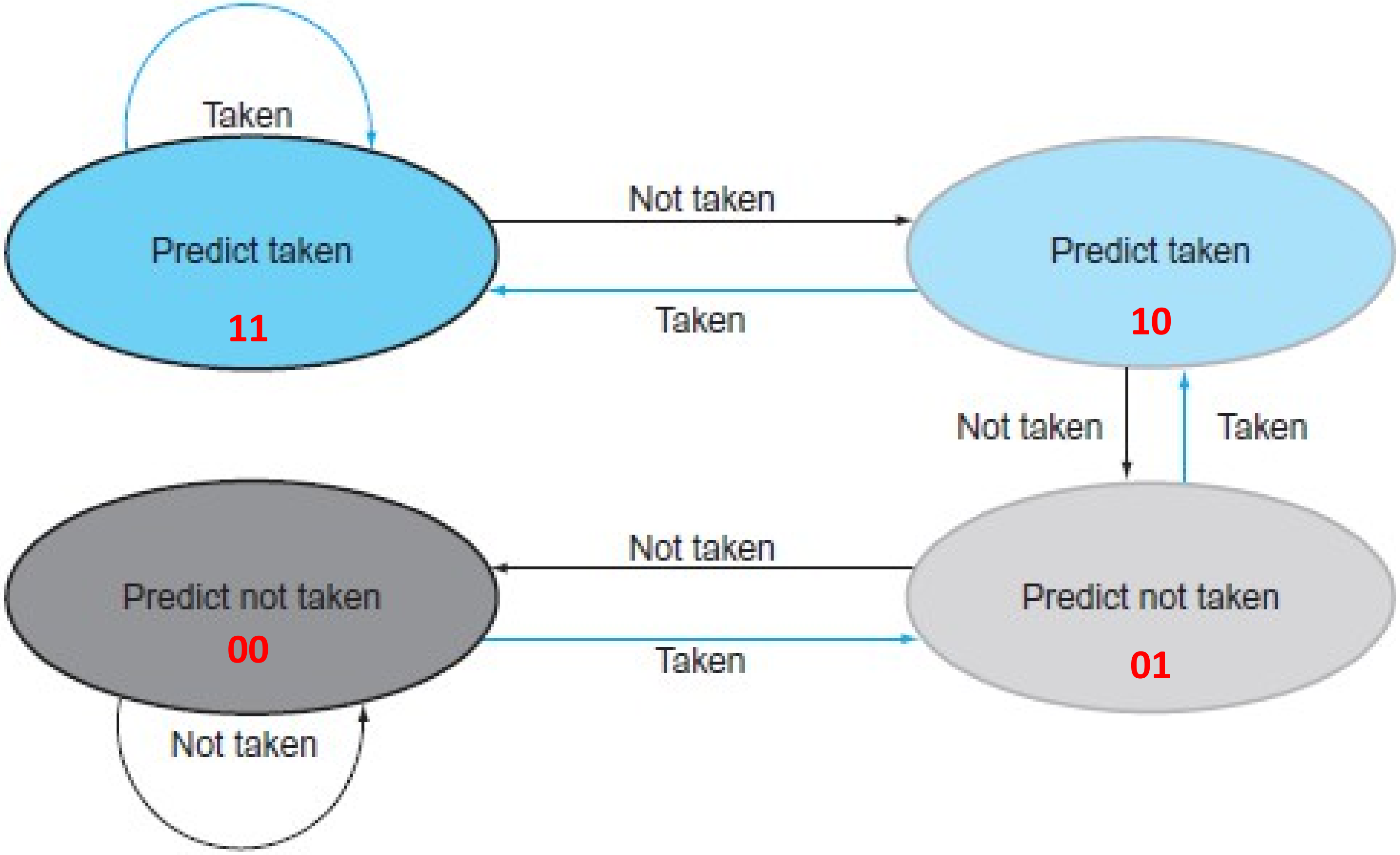
2 Mispredictions out of 10. so 80% accuracy is achieved

2-bit scheme

- A prediction must be wrong twice before the prediction is changed.
- A branch prediction buffer has 2 bits to store the history.

Bits indicating branch taken or not taken	Meaning
00	Strongly Branch not taken
01	Weakly branch not taken
10	Weakly branch taken
11	Strongly branch taken

Finite state diagram



Exception

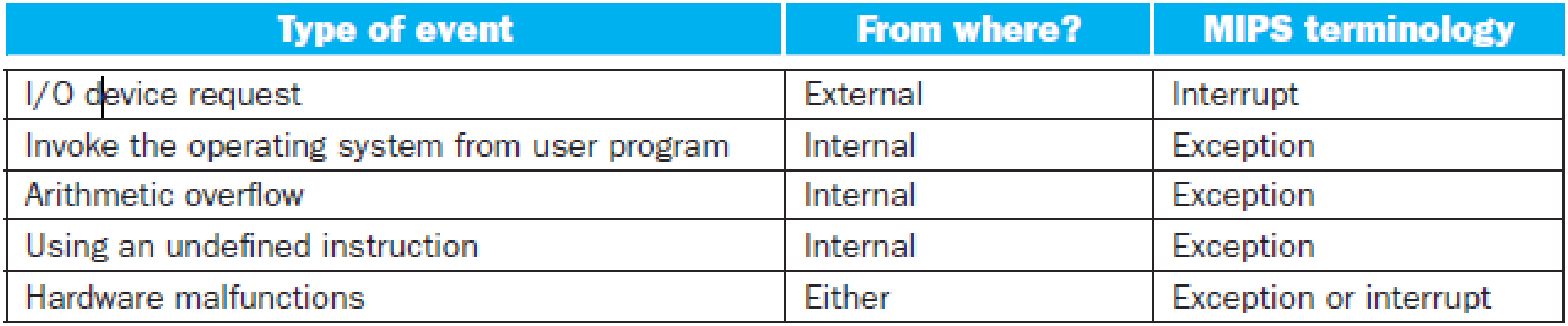
Exception

- **Exception –** (interrupt) An unscheduled event that disrupts program execution; used to detect overflow.
- **Interrupt -** An exception that comes from outside of the processor. (Some architectures use the term *interrupt for all* exceptions.)
- **vectored interrupt –** An interrupt for which the address to which control is transferred is determined by the cause of the exception.

Difference between exception and interrupt

Exception	Interrupt
Unscheduled event	Unscheduled event
Invoked Internal (within processor)	Invoked External (outside processor)
Ex: Arithmetic overflow	Ex: IO device request

Exception



Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

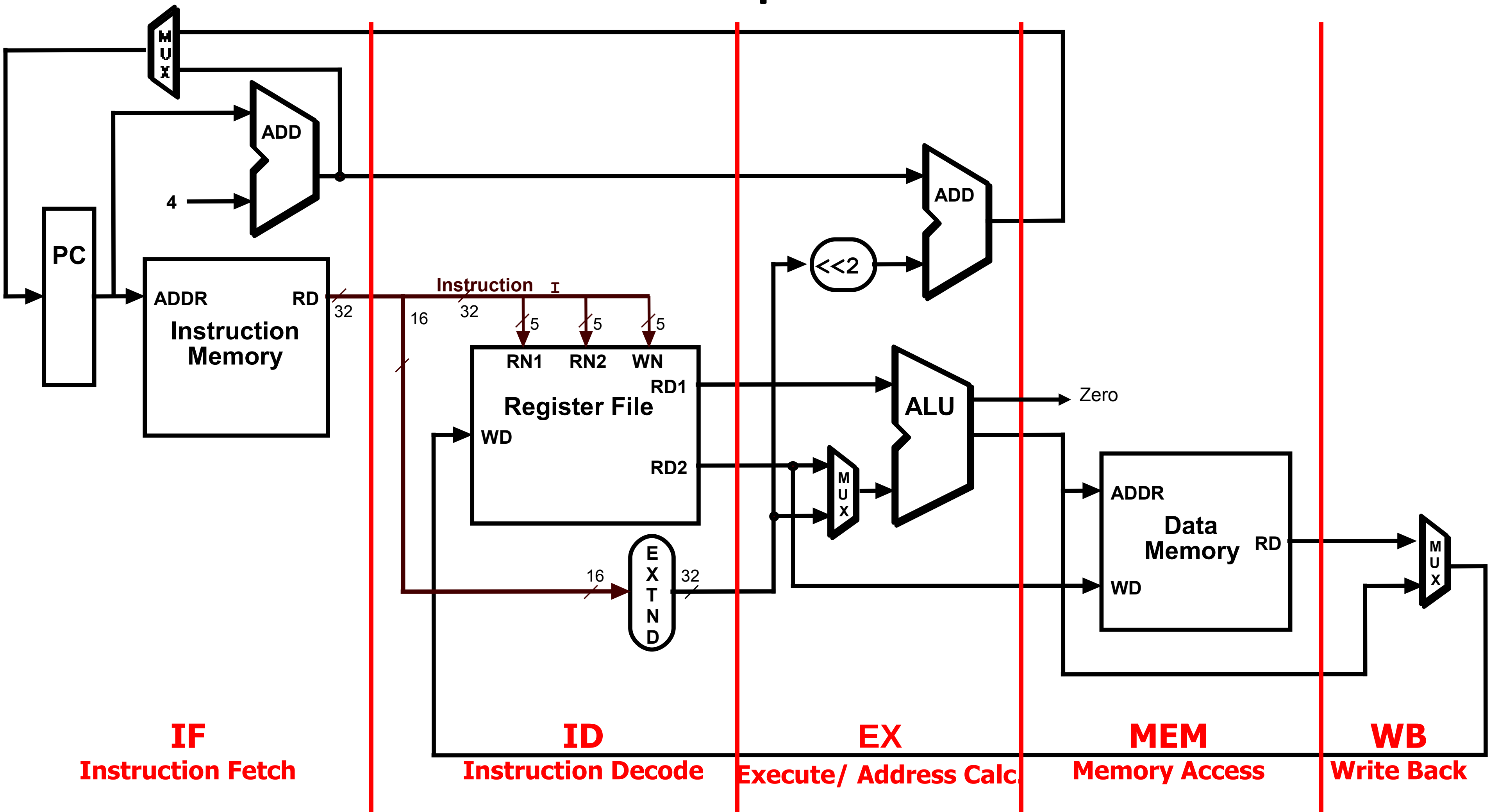
Pipelined datapath and control

Pipelined Datapath

- We now move to actually building a pipelined datapath
- First recall the 5 steps in instruction execution
  1. Instruction Fetch & PC Increment (IF)
  2. Instruction Decode and Register Read (ID)
  3. Execution or calculate address (EX)
  4. Memory access (MEM)
  5. Write result into register (WB)
- Review: single-cycle processor
  - all 5 steps done in a single clock cycle
  - dedicated hardware required for each step

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*

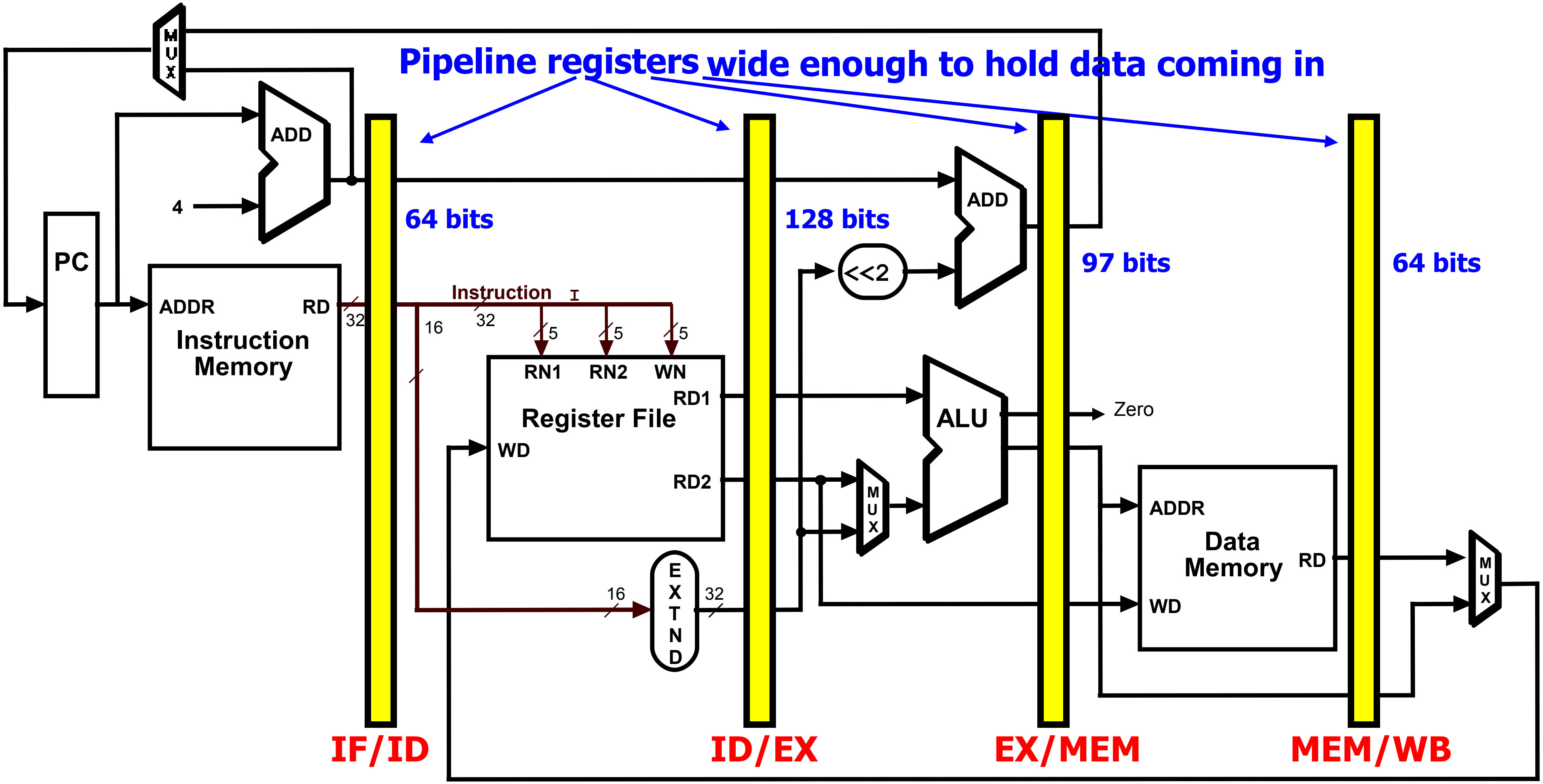
Review - Single-Cycle Datapath "Steps"



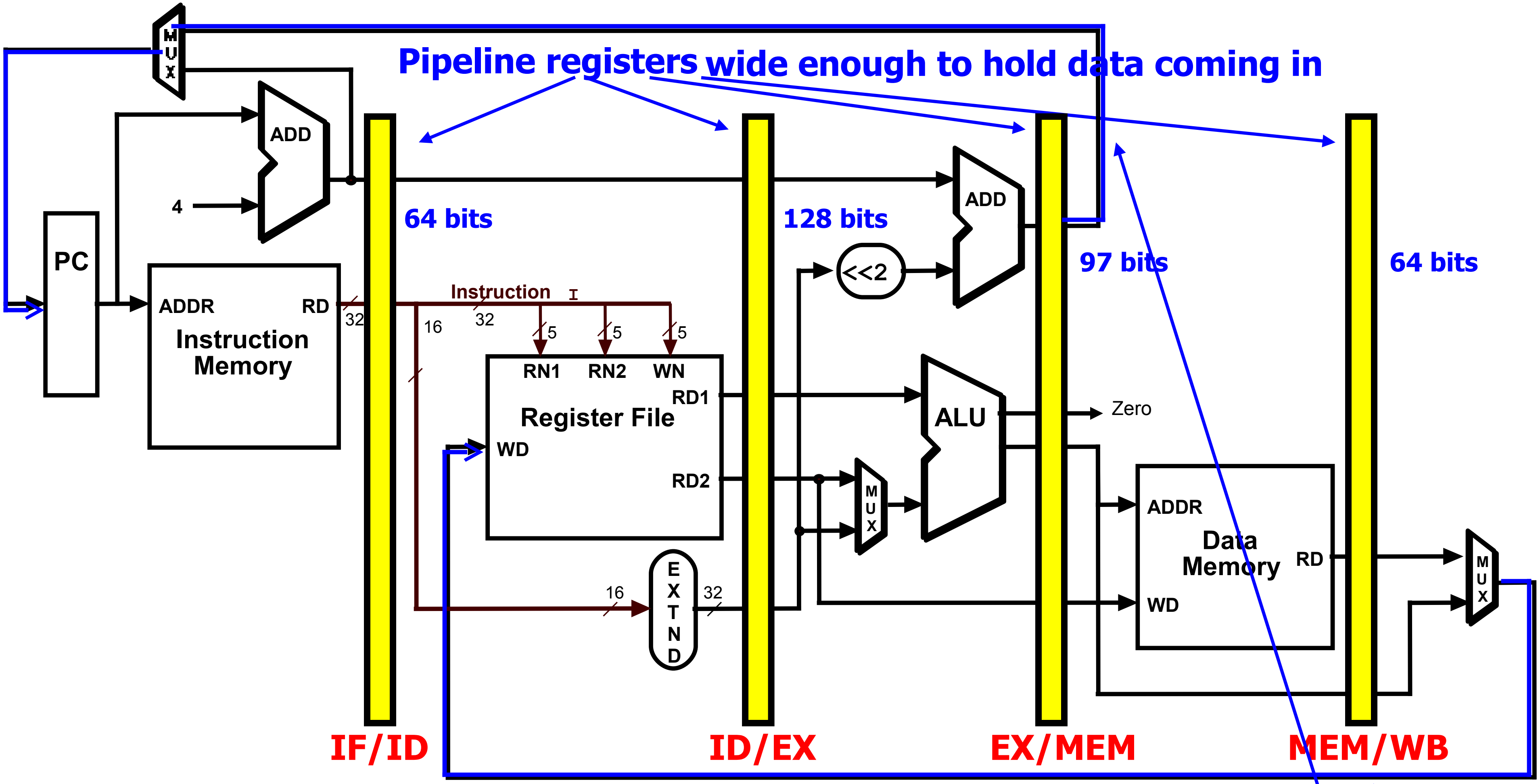
Pipelined Datapath – Key Idea

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*
    - Answer: *We may be able to start executing a new instruction at each clock cycle* - pipelining
- …but we shall need *extra* registers to hold data between cycles – *pipeline registers*

Pipelined Datapath



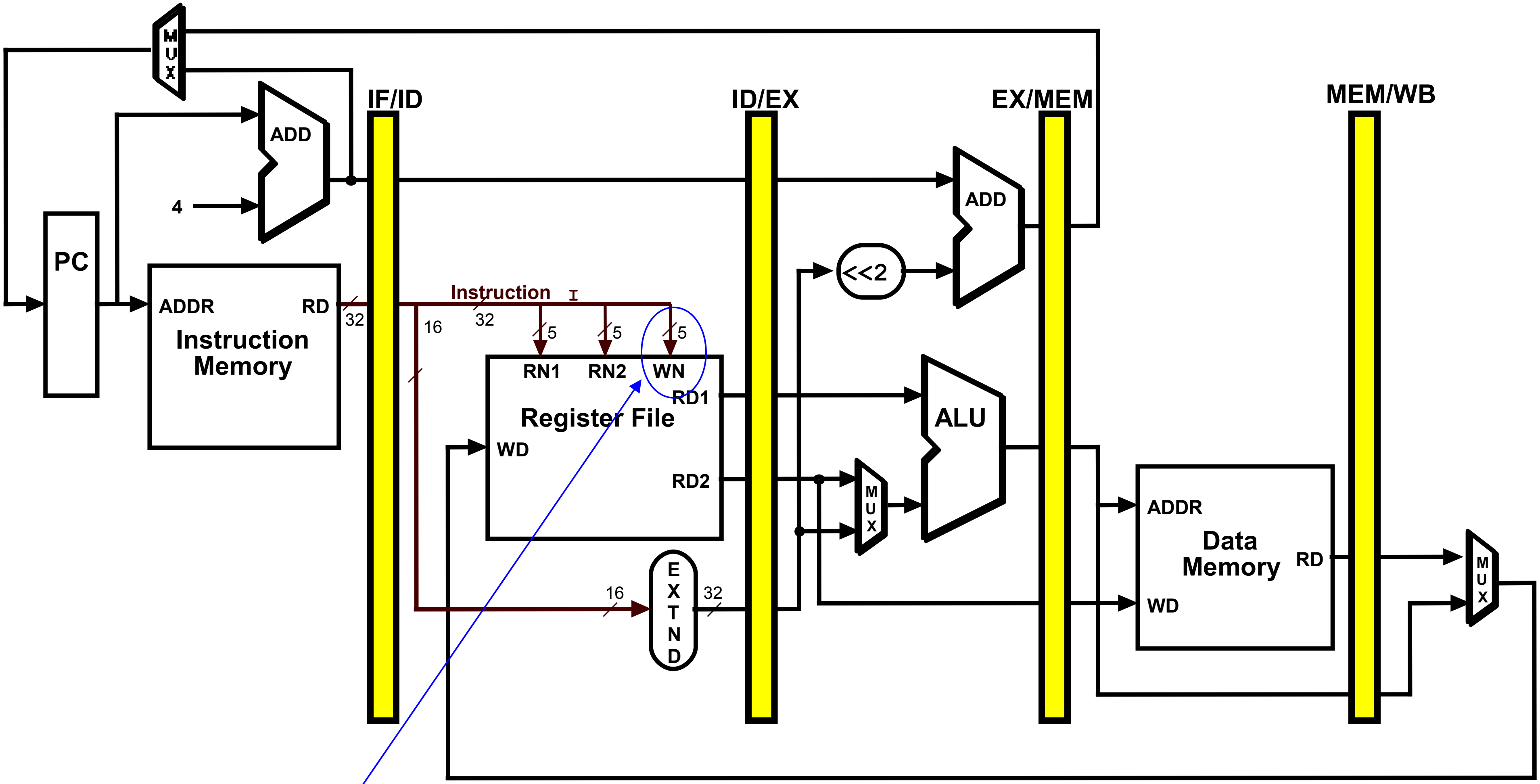
Pipelined Datapath



Pipeline registers wide enough to hold data coming in

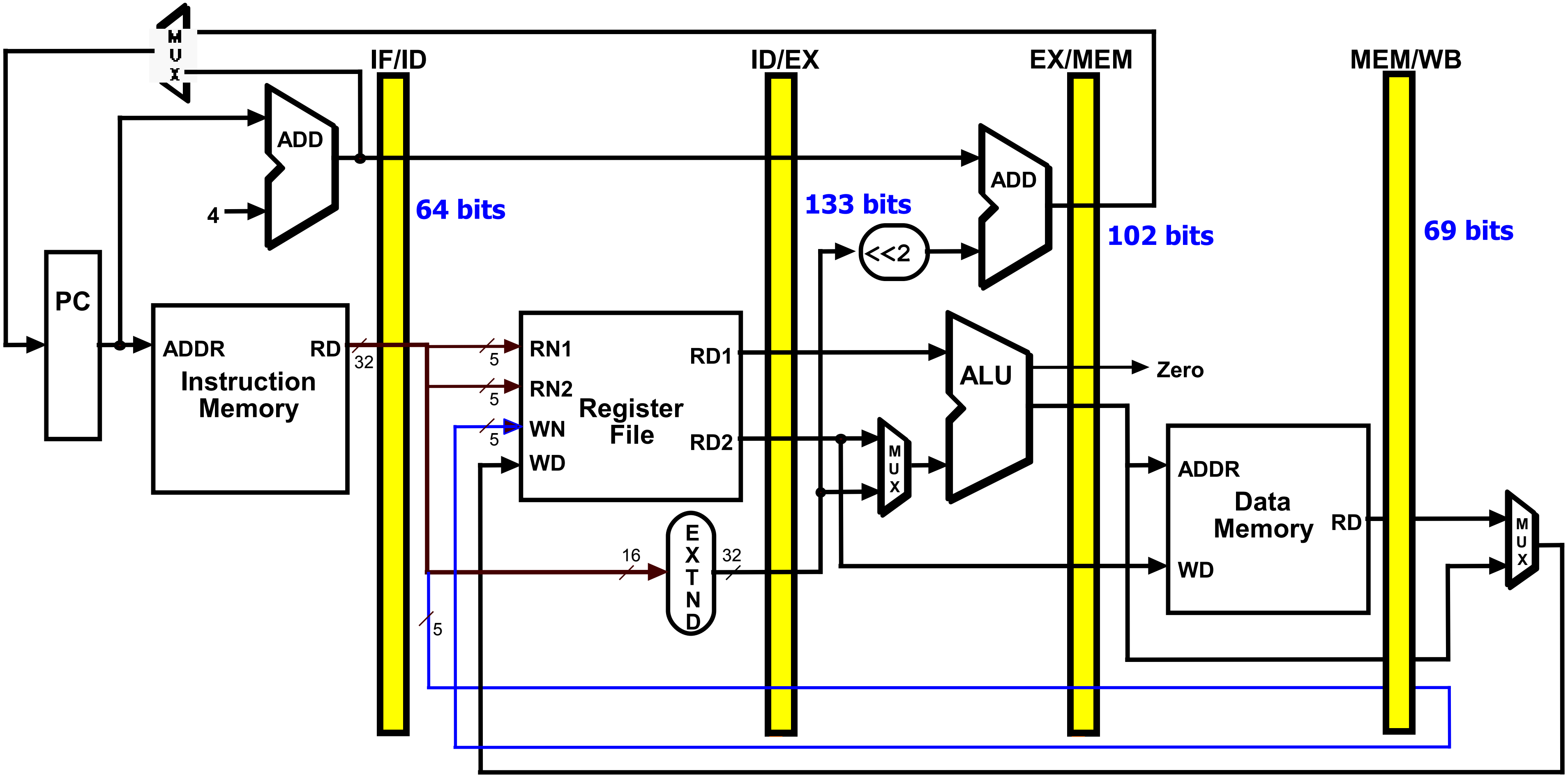
Only data flowing right to left may cause hazard…, why?

Bug in the Datapath



Write register number comes from another *later* instruction!

Corrected Datapath



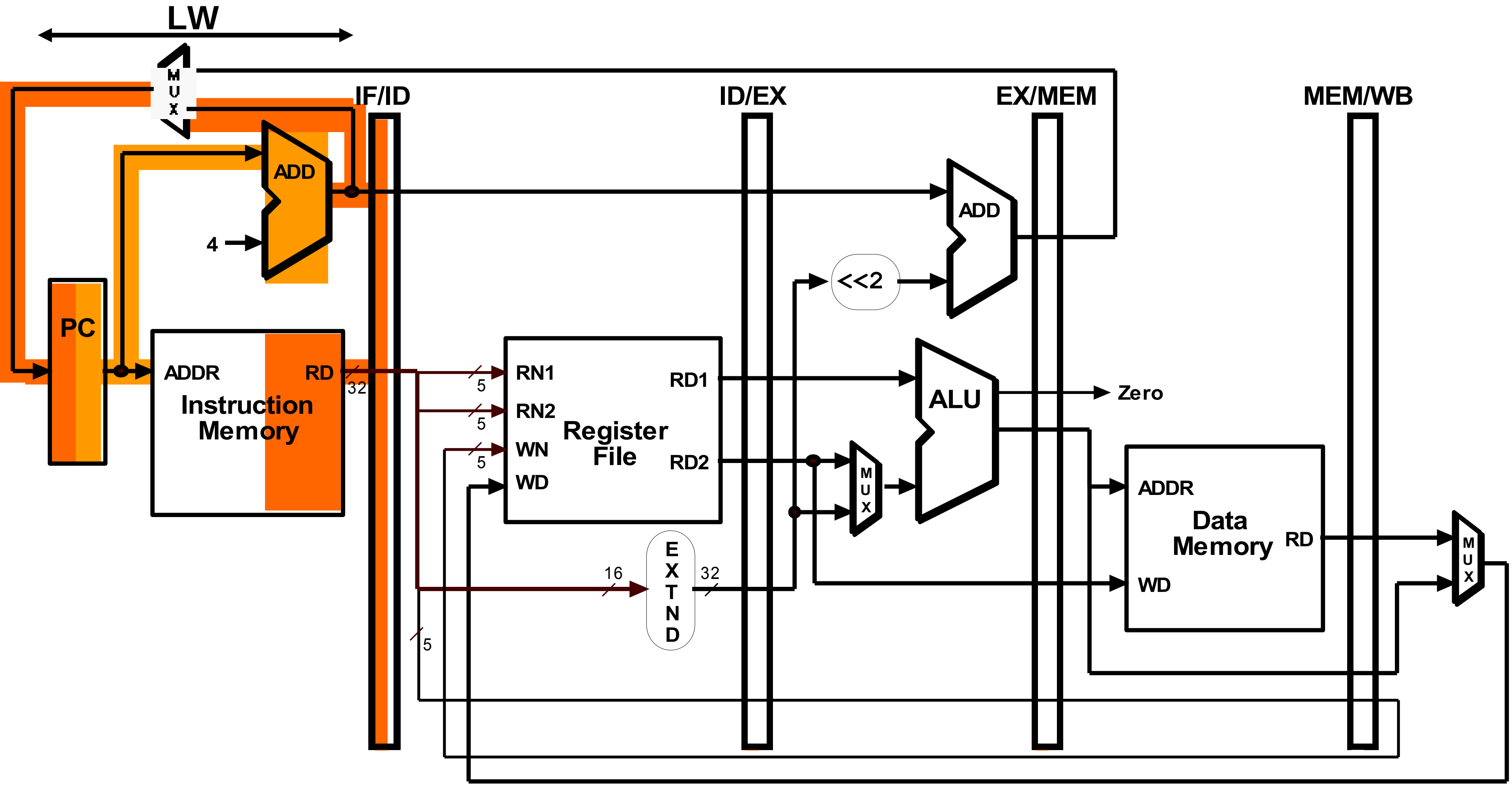
Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

Pipelined Example

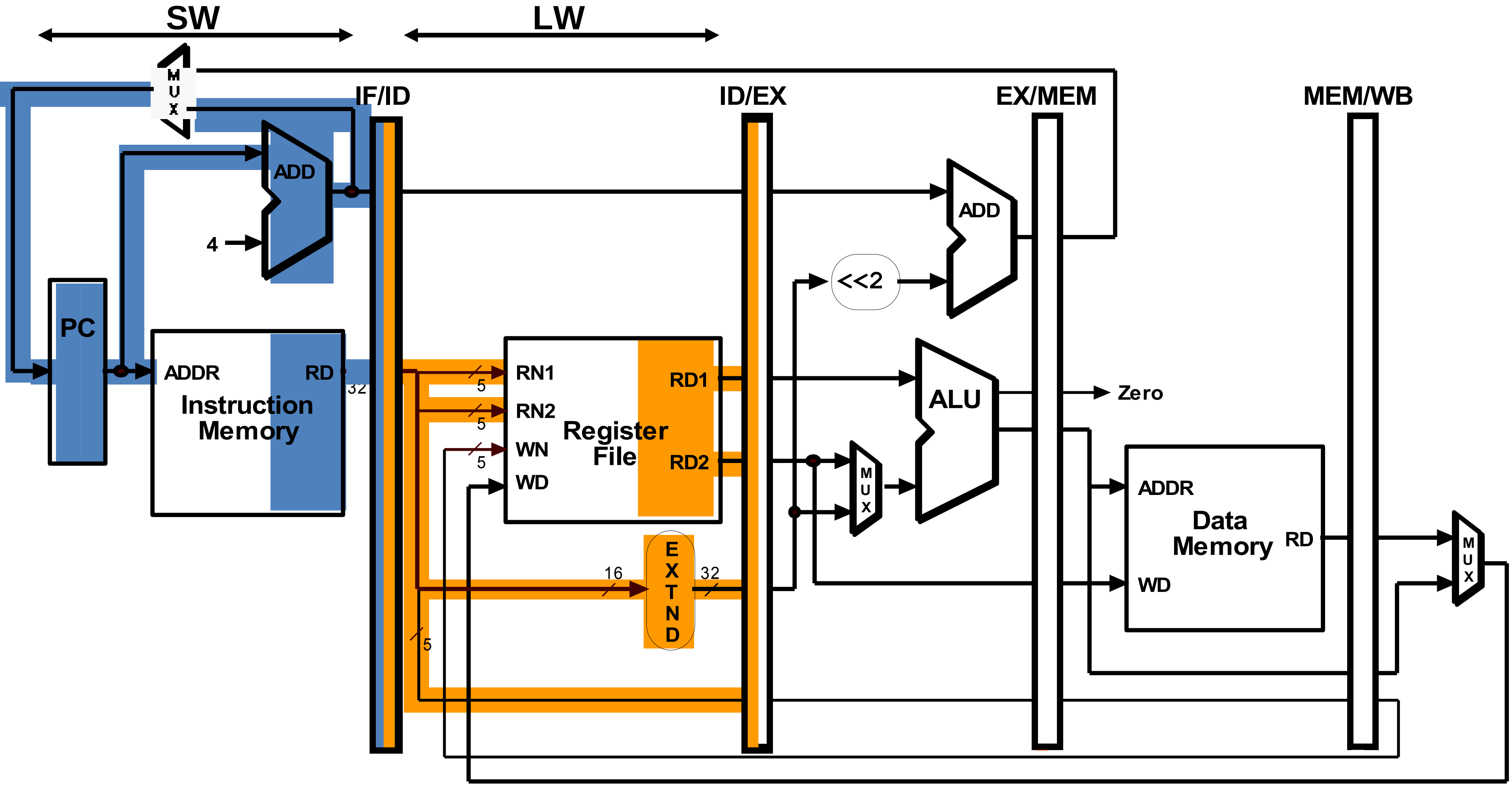
- Consider the following instruction sequence:

```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

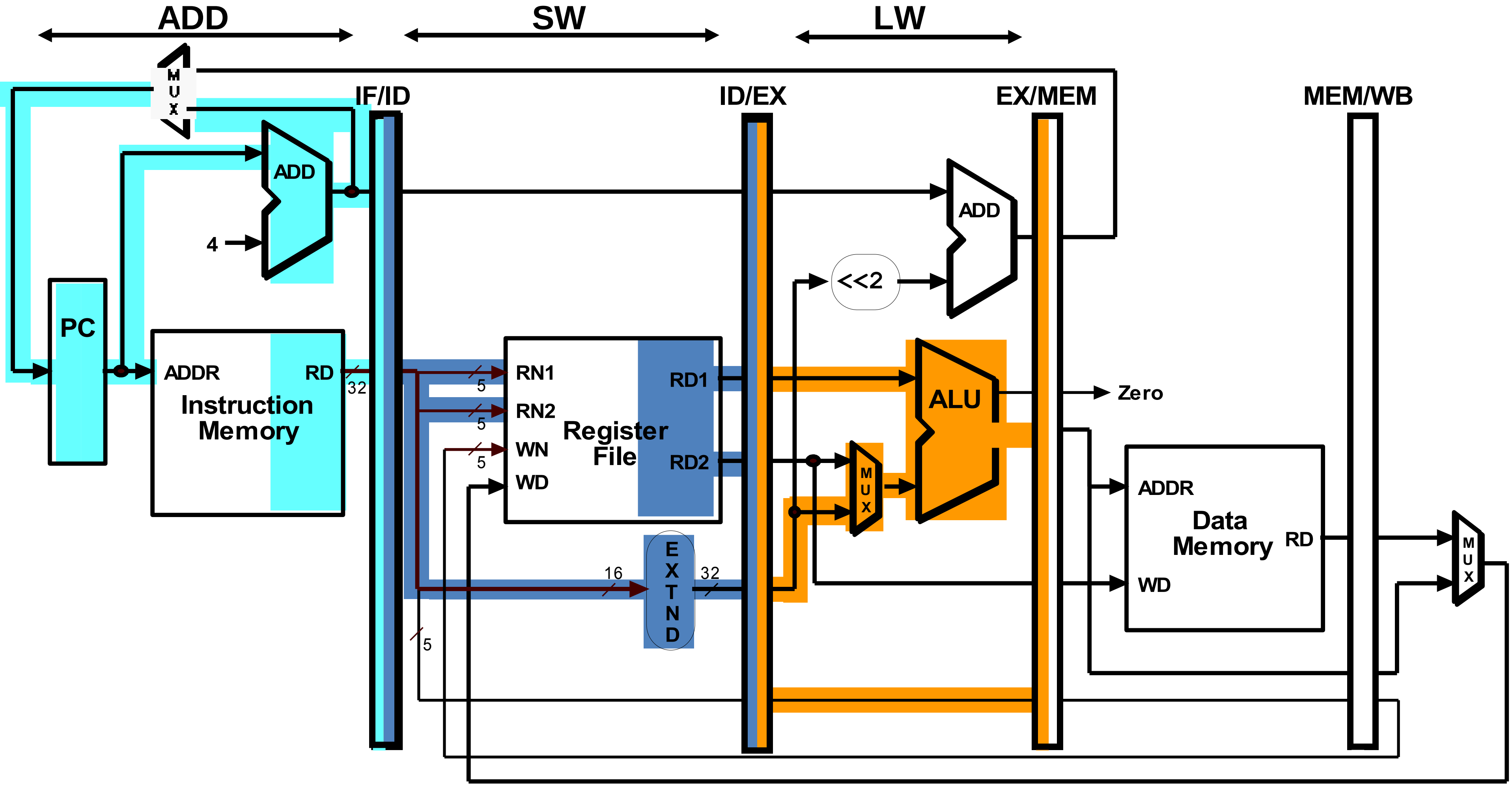
Single-Clock-Cycle Diagram: Clock Cycle 1



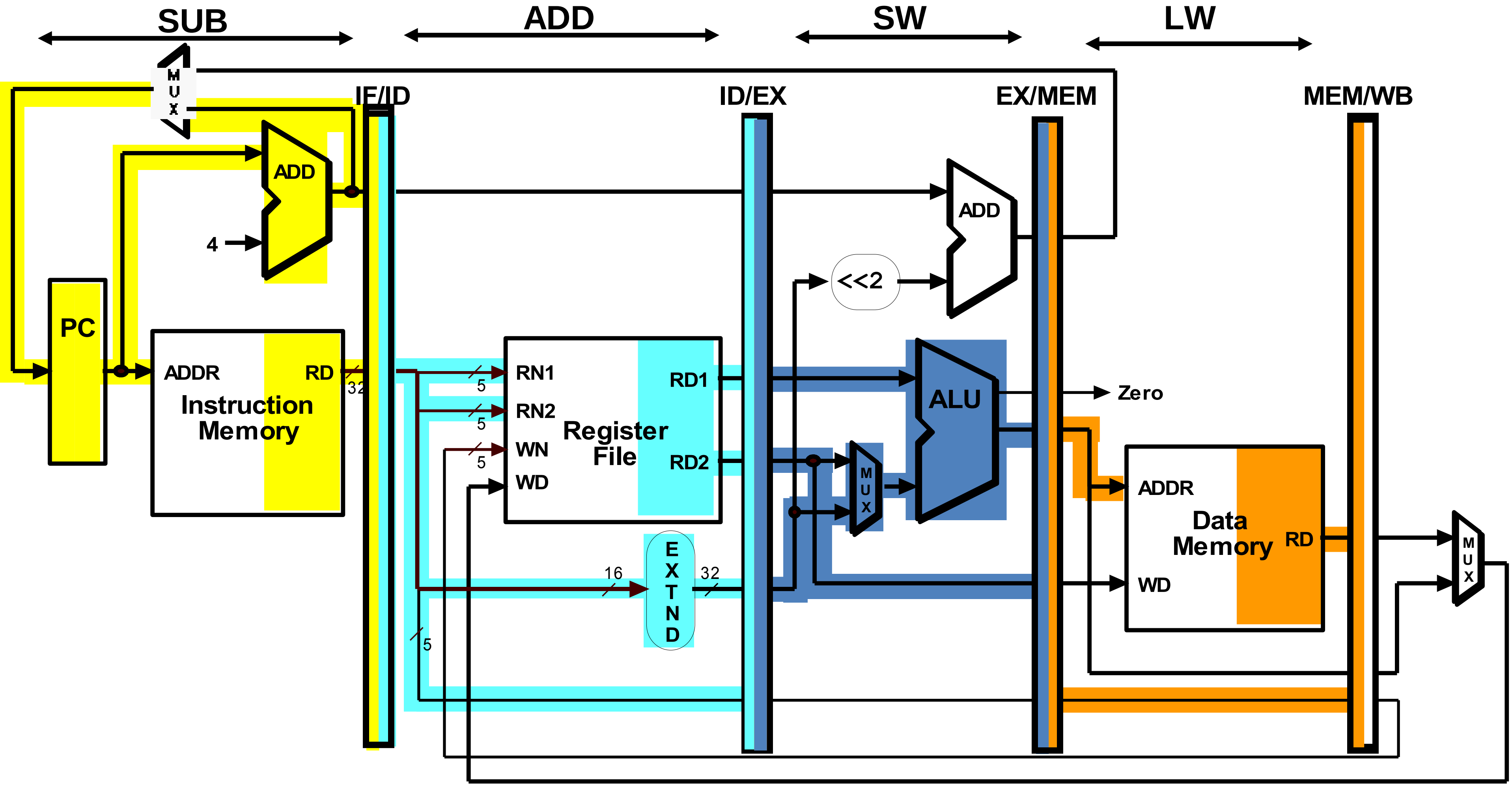
Single-Clock-Cycle Diagram: Clock Cycle 2



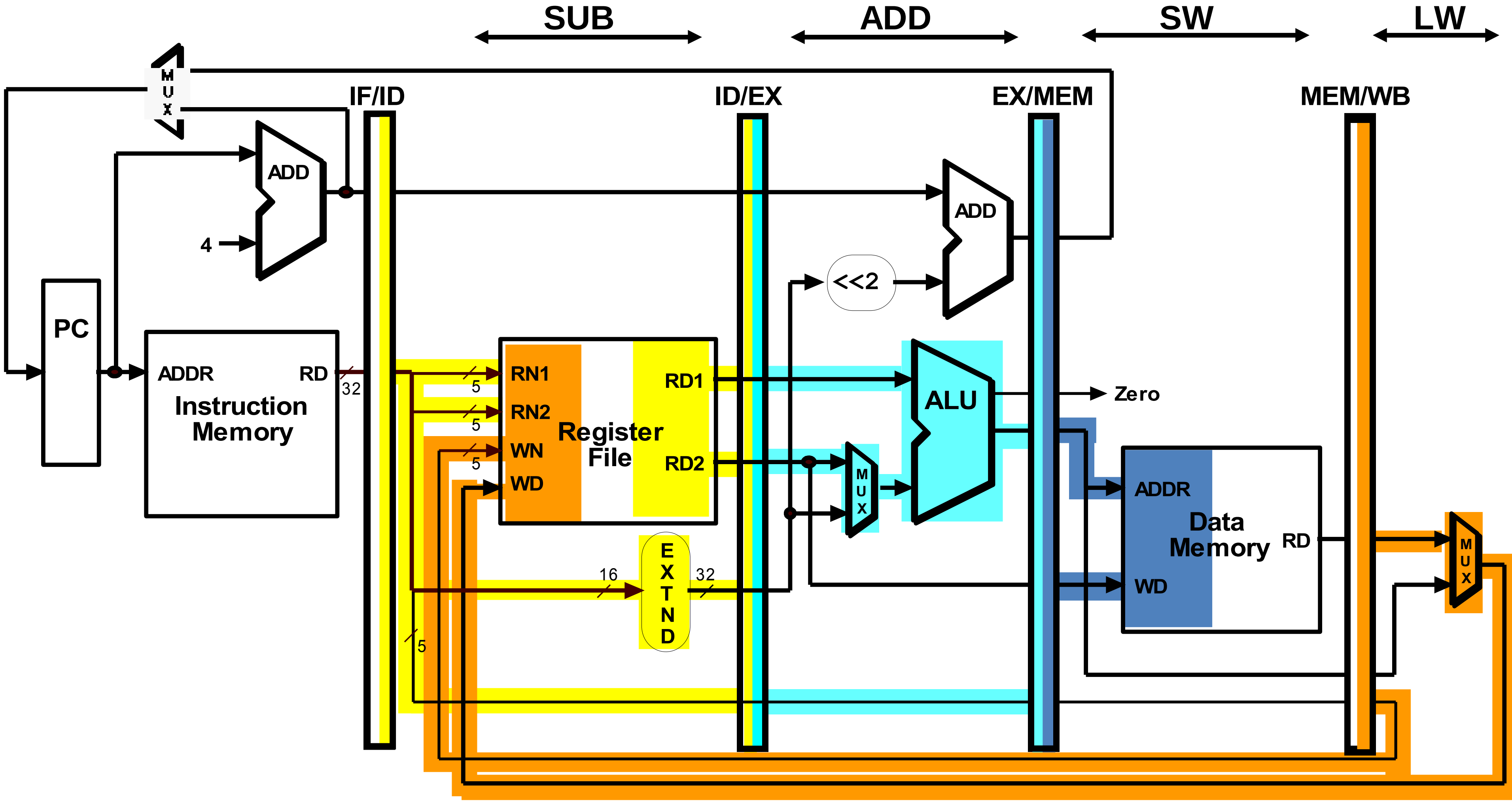
Single-Clock-Cycle Diagram: Clock Cycle 3



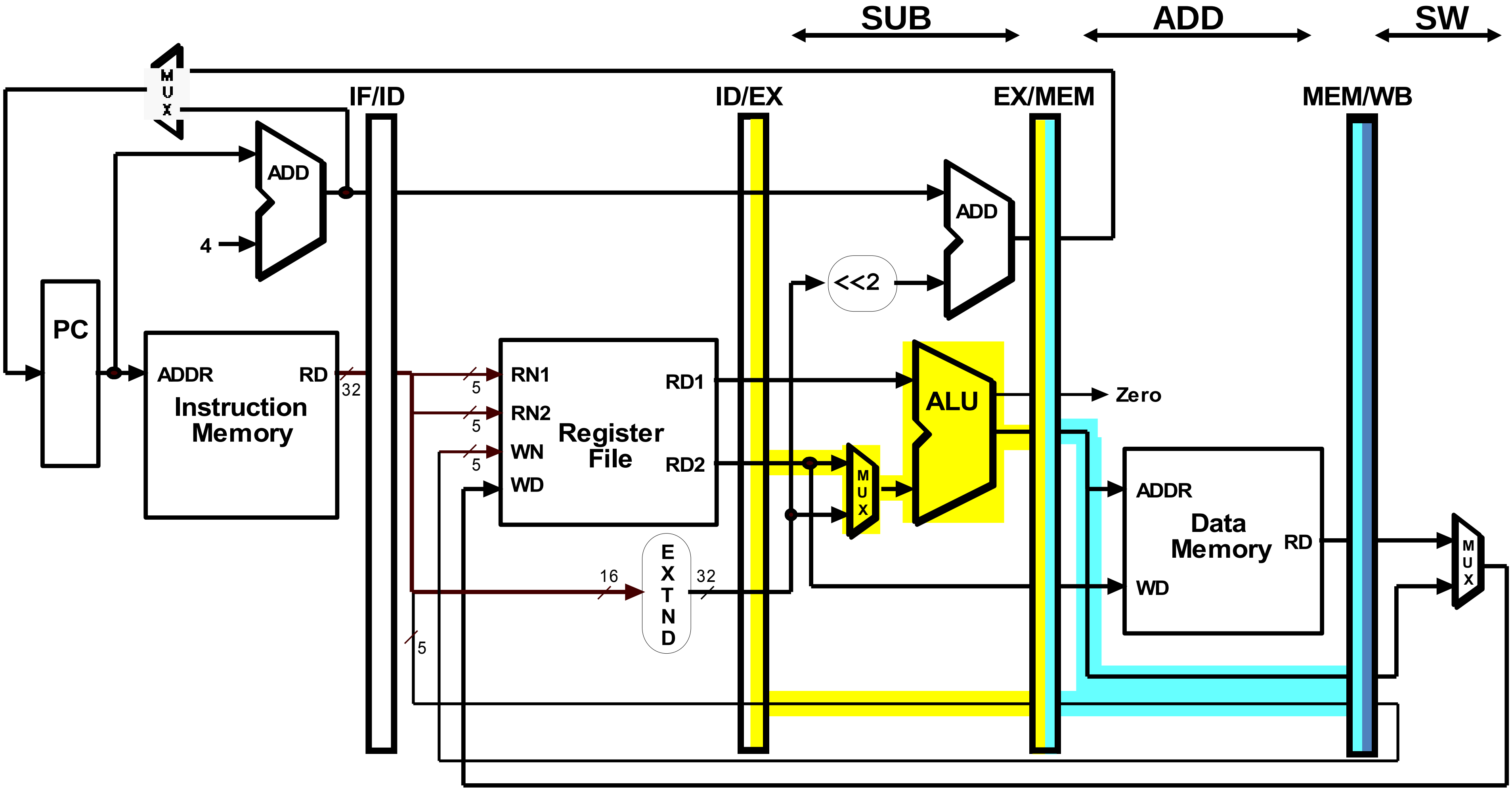
Single-Clock-Cycle Diagram: Clock Cycle 4



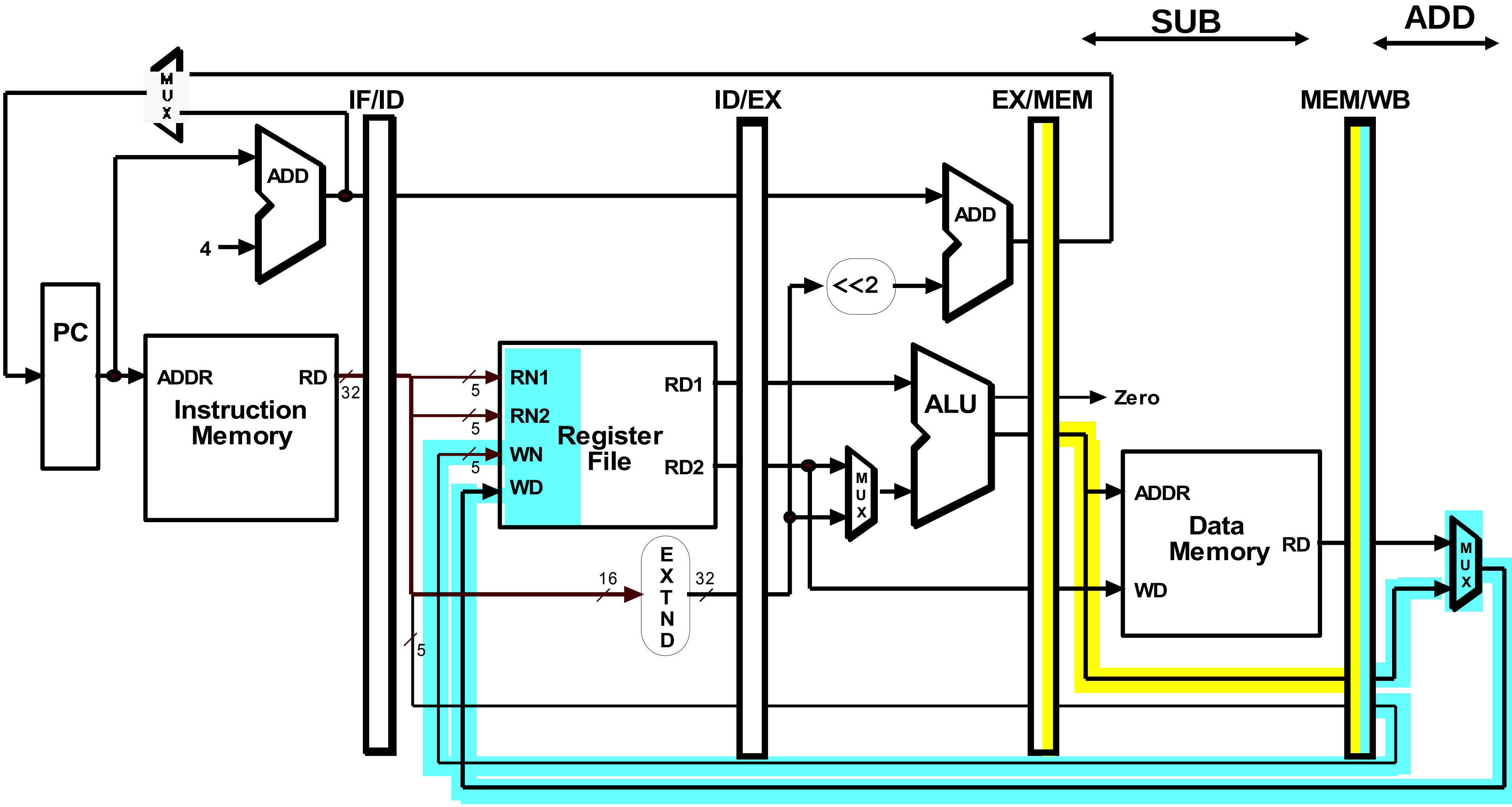
Single-Clock-Cycle Diagram: Clock Cycle 5



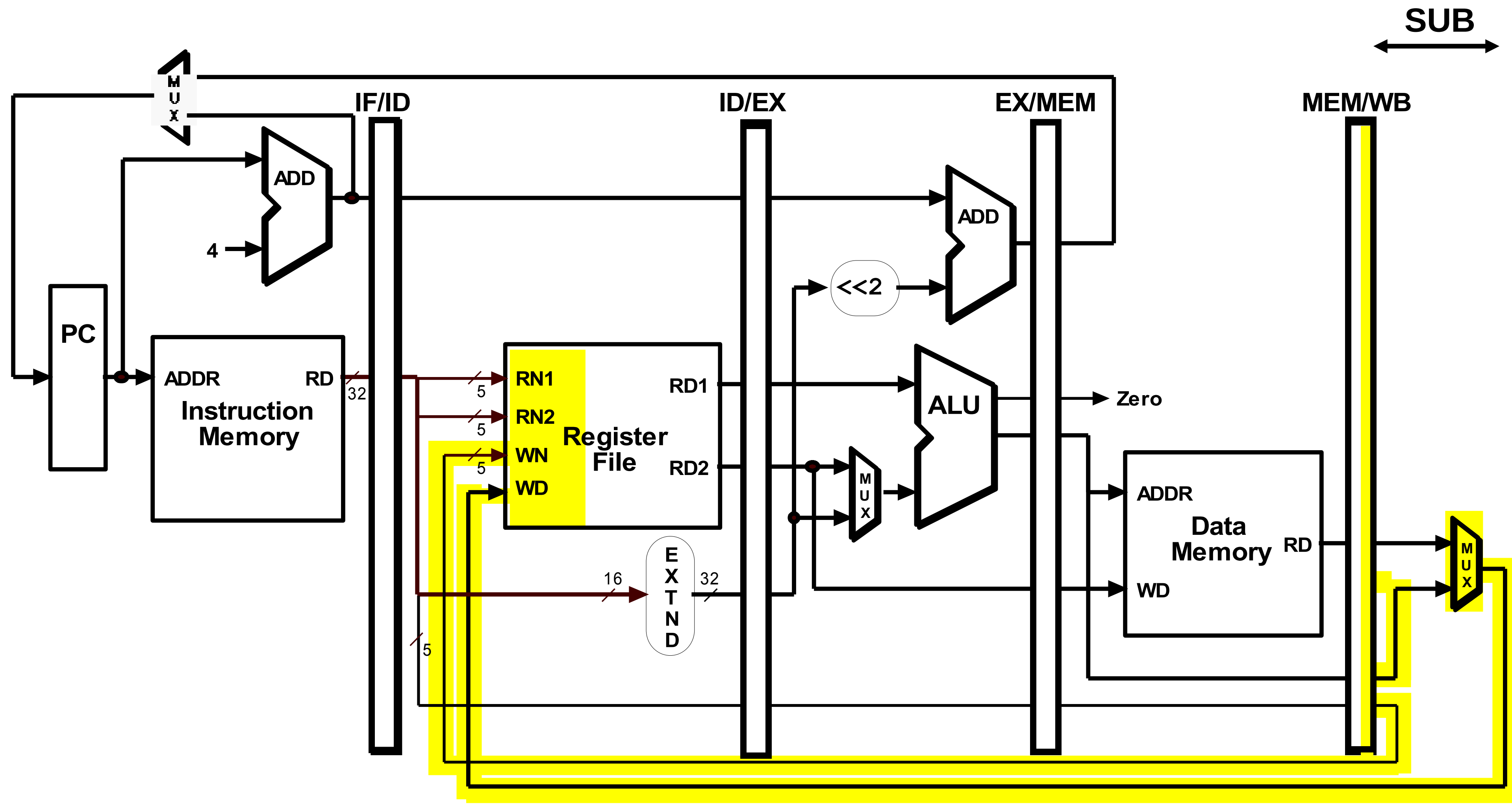
Single-Clock-Cycle Diagram: Clock Cycle 6



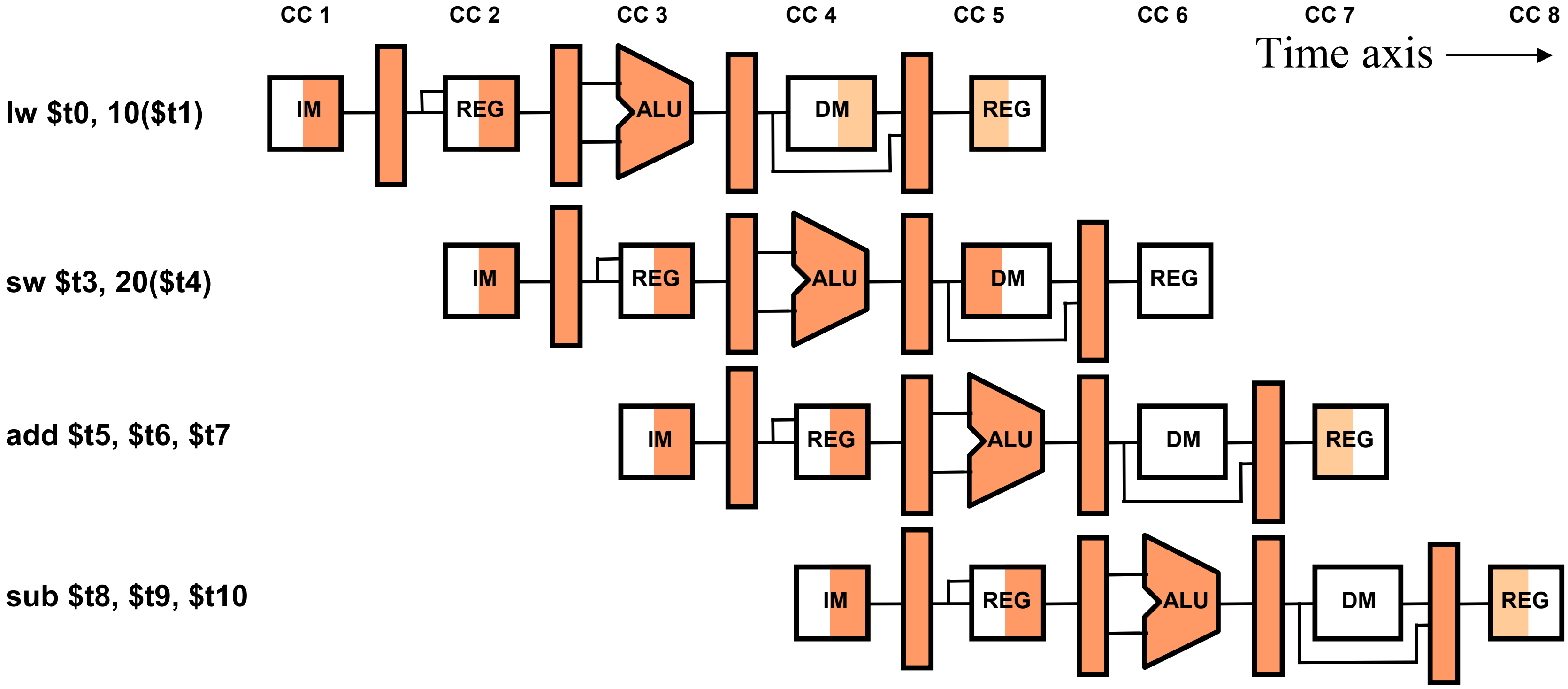
Single-Clock-Cycle Diagram: Clock Cycle 7



Single-Clock-Cycle Diagram: Clock Cycle 8



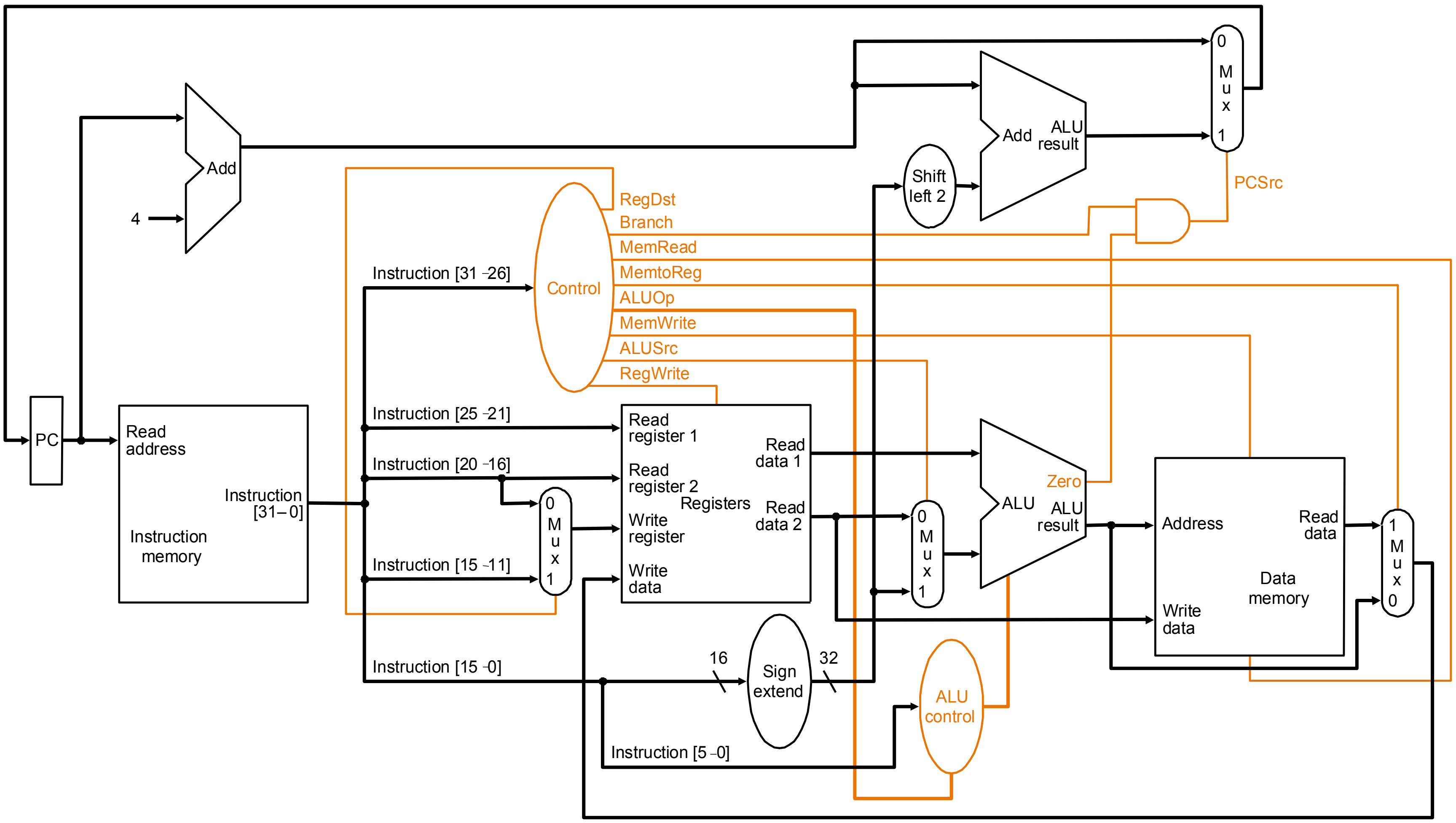
Alternative View – Multiple-Clock-Cycle Diagram



Notes

- One significant difference in the execution of an R-type instruction between multicycle and pipelined implementations:
  - register write-back for the R-type instruction is the 5th (the last write-back) pipeline stage vs. the 4th stage for the multicycle implementation. *Why?*
  - think of *structural hazards* when writing to the register file…
- Worth repeating: the *essential difference* between the pipeline and multicycle implementations is the insertion of pipeline registers to *decouple the 5 stages*
- The CPI of an *ideal pipeline* (no stalls) is 1. *Why?*
- The RaVi Architecture Visualization Project of Dortmund U. has pipeline simulations – see link in our Additional Resources page
- As we develop control for the pipeline keep in mind that the text *does not consider* `jump` – should not be too hard to implement!

Recall Single-Cycle Control – the Datapath



Recall Single-Cycle – ALU Control

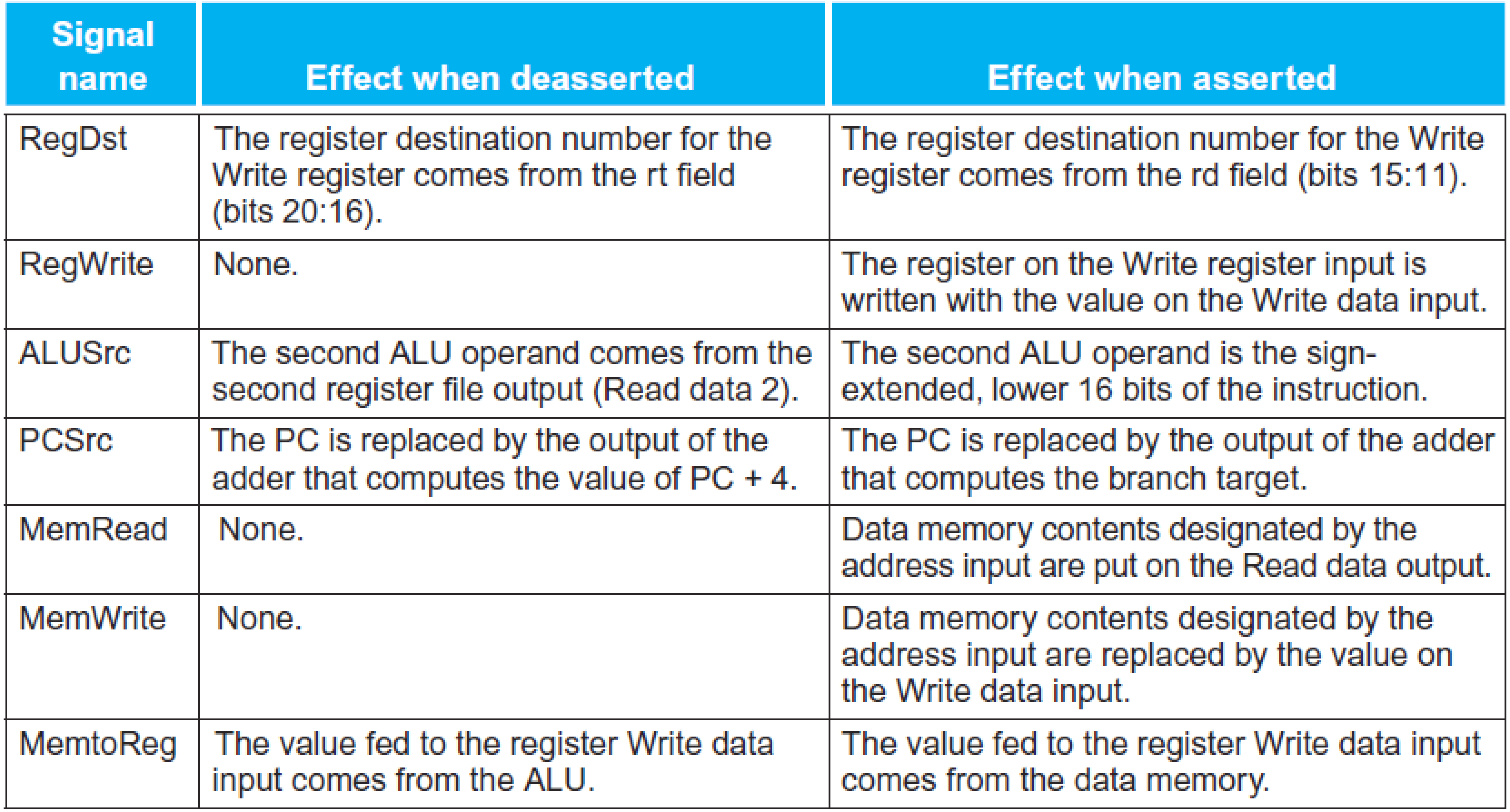
Instruction	AluOp	Instruction	Funct Field	Desired	ALU control
opcode		operation		ALU action	input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

ALUOp		**Funct field**						**Operation**
ALUOp1	**ALUOp0**	**F5**	**F4**	**F3**	**F2**	**F1**	**F0**	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

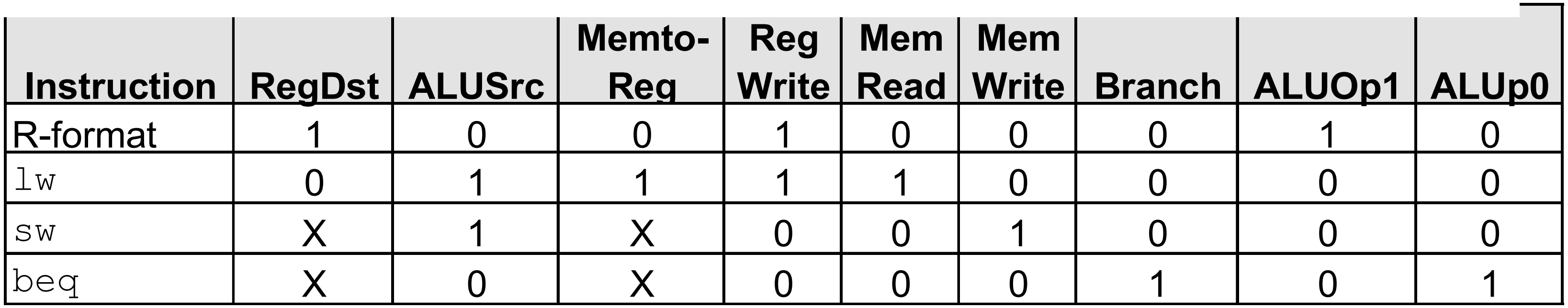
Recall Single-Cycle – Control Signals

Effect of control bits



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.

Determining control bits



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
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

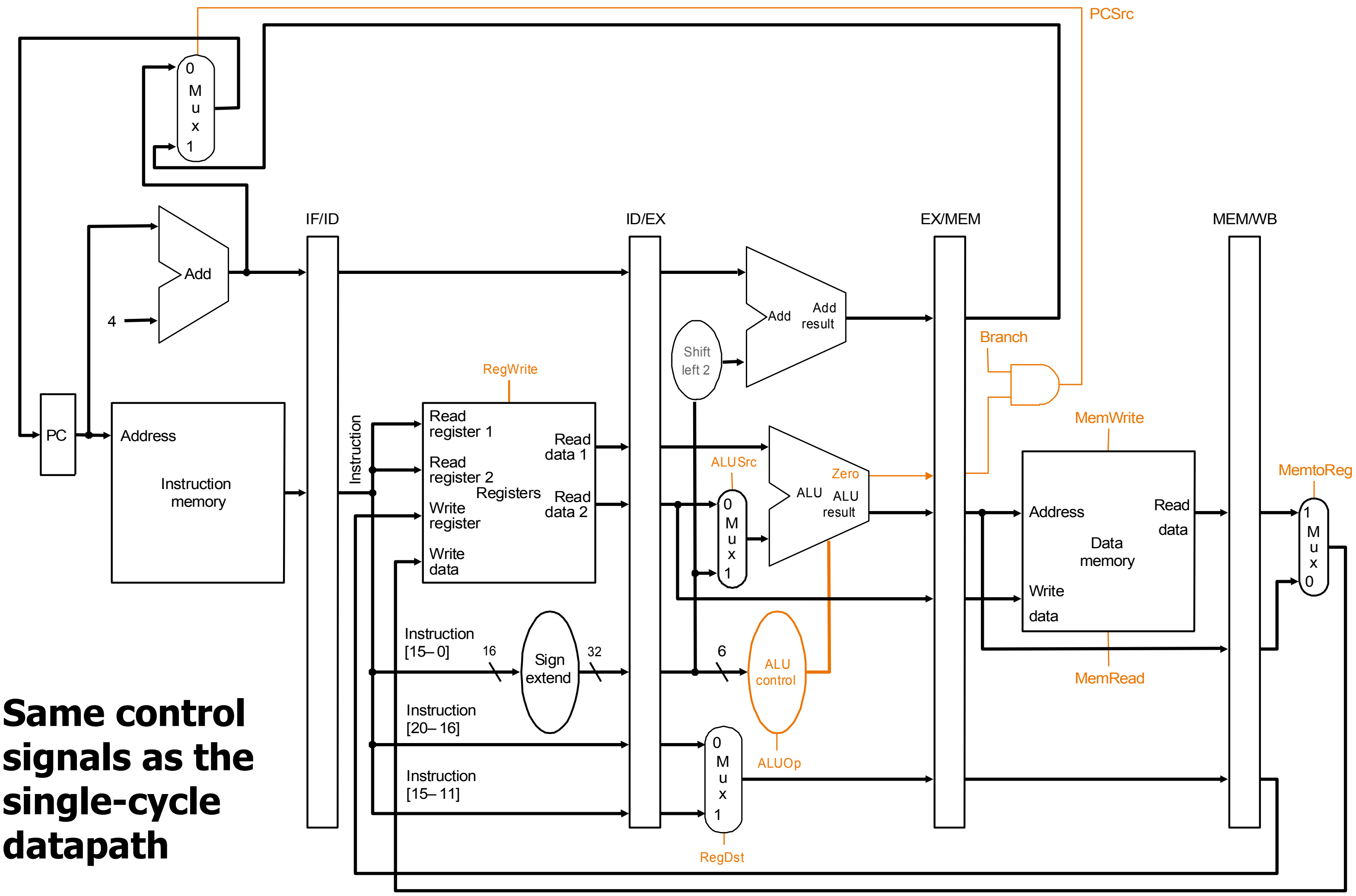
Pipeline Control

- Initial design – *motivated by single-cycle datapath control* – use the *same* control signals
- Observe:
  - No separate write signal for the PC as it is written every cycle
  - No separate write signals for the pipeline registers as they are written every cycle

  Will be modified by hazard detection unit!!

  - *No separate read signal for instruction memory* as it is read every clock cycle
  - *No separate read signal for register file* as it is read every clock cycle
- Need to *set control signals during each pipeline stage*
- Since control signals are associated with components active during a single pipeline stage, can *group control lines into five groups according to pipeline stage*

Pipelined Datapath with Control I



Same control signals as the single-cycle datapath

Pipeline Control Signals

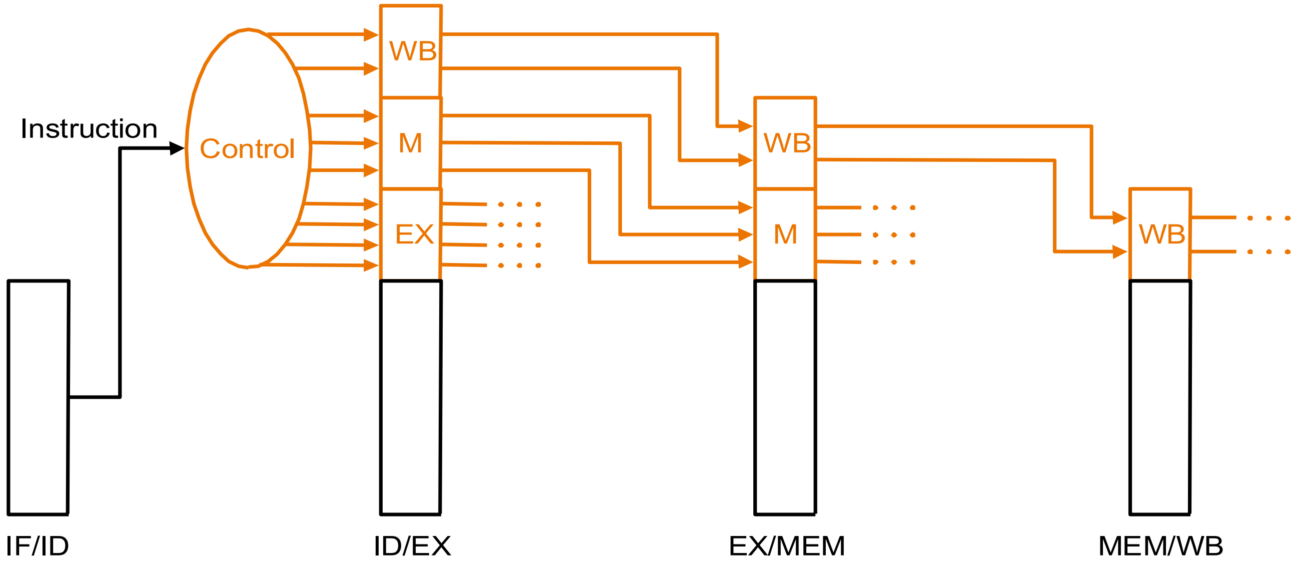
- There are five stages in the pipeline
  - *instruction fetch / PC increment*
  - *instruction decode / register fetch*
  - *execution / address calculation*
  - *memory access*
  - *write back*

Nothing to control as instruction memory read and PC write are always enabled

	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
Instruction	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

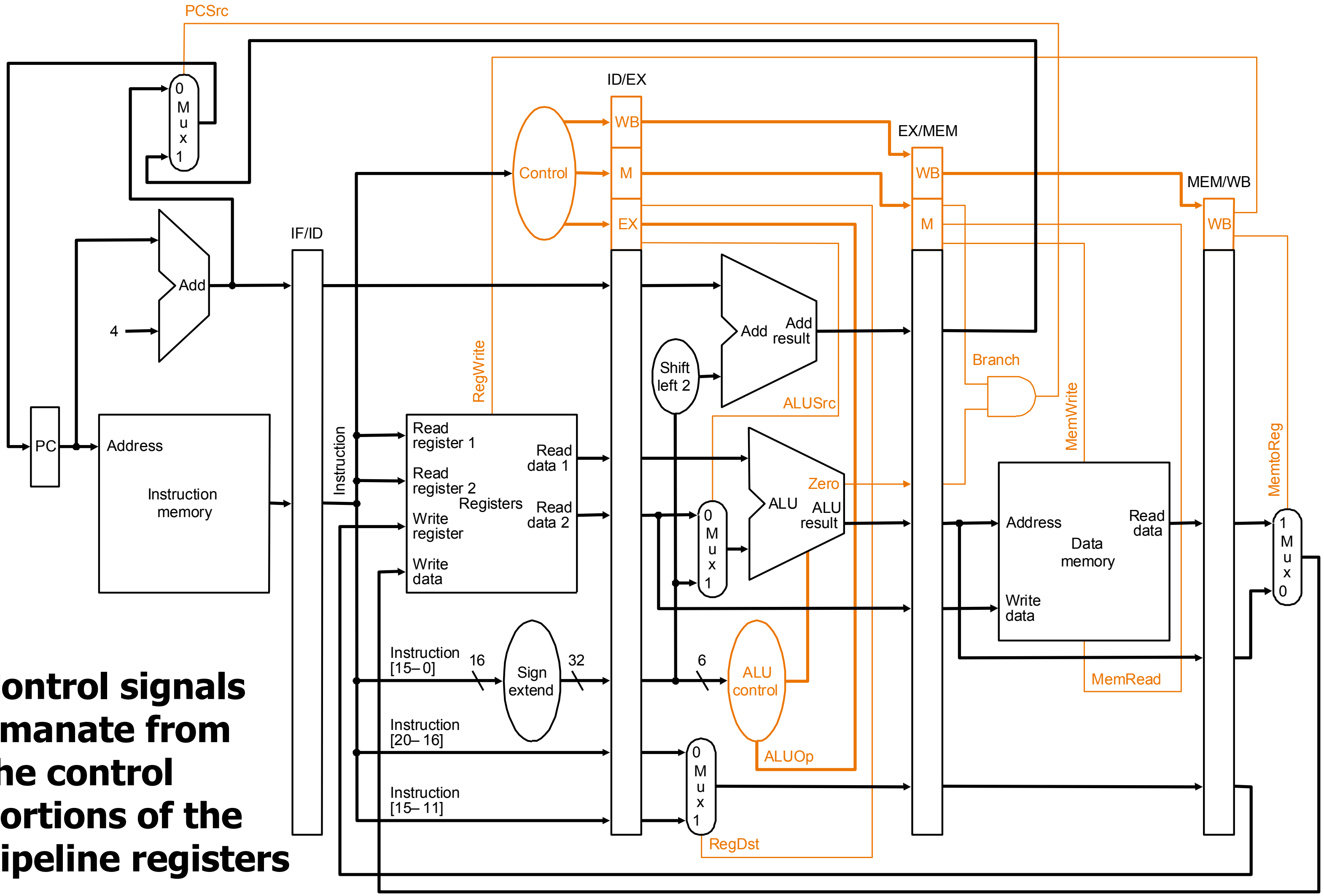
Pipeline Control Implementation

- *Pass control signals along just like the data* – extend each pipeline register to hold needed control bits for succeeding stages



- *Note*: The 6-bit *funct field* of the instruction required in the EX stage to generate ALU control can be retrieved as the 6 least significant bits of the immediate field which is sign-extended and passed from the IF/ID register to the ID/EX register

Pipelined Datapath with Control II



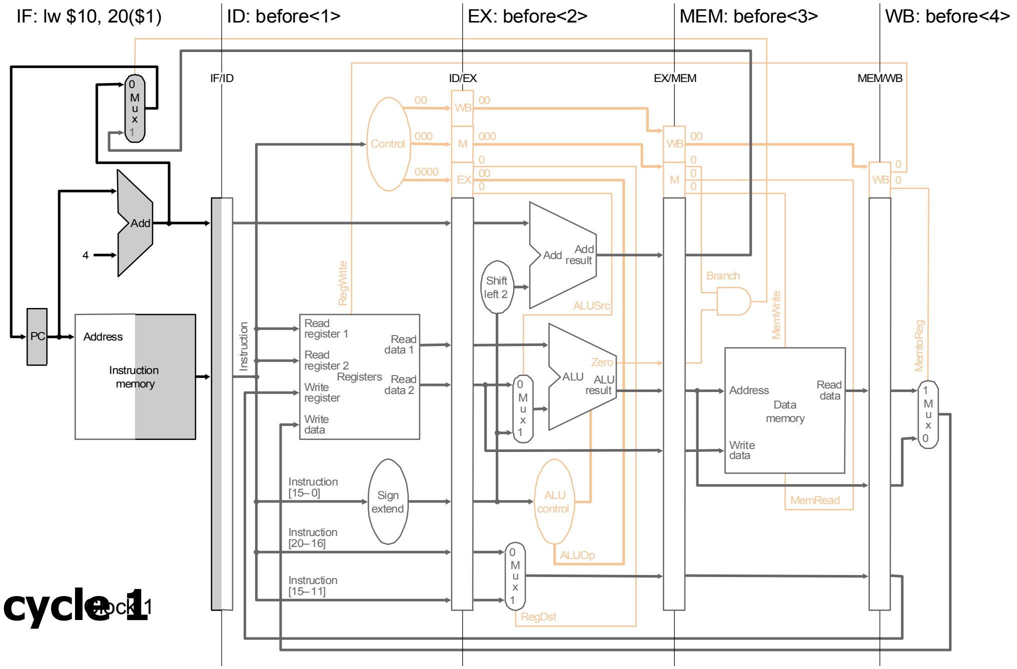
Control signals emanate from the control portions of the pipeline registers

Pipelined Execution and Control

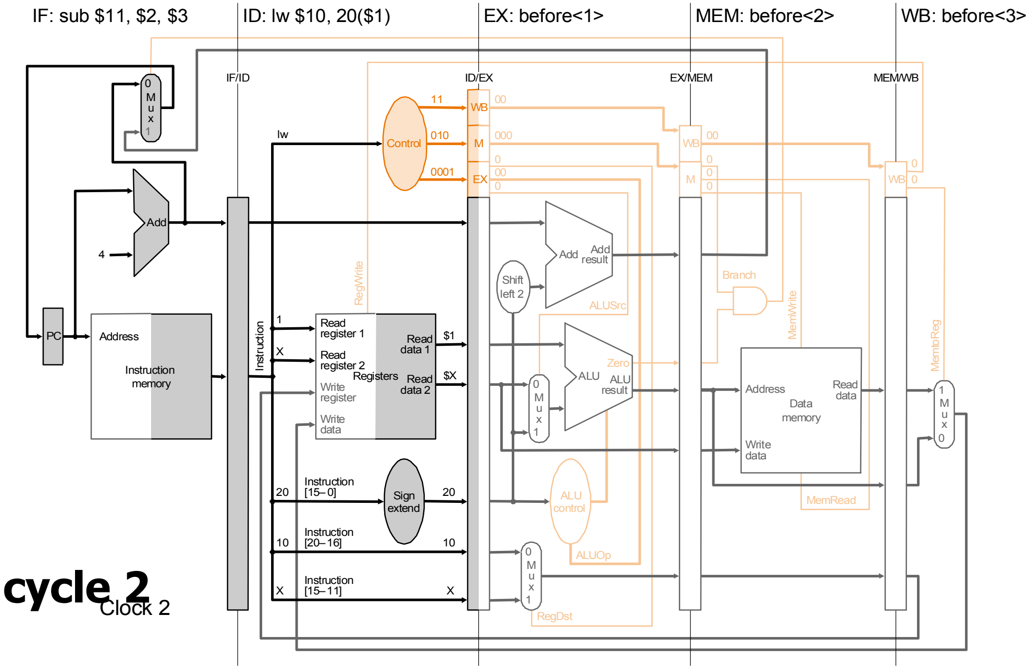
Instruction sequence:

```
lw   $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $7
or   $13, $6, $7
add  $14, $8, $9
```

Label "before<i>" means i th instruction before `lw`



Clock cycle 1

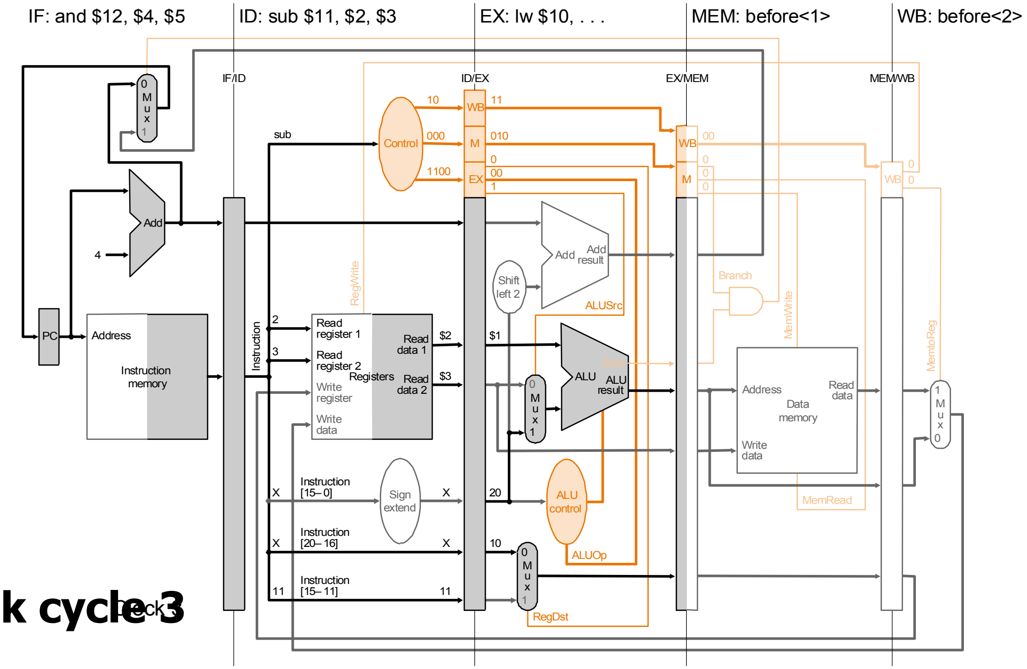


Clock cycle 2

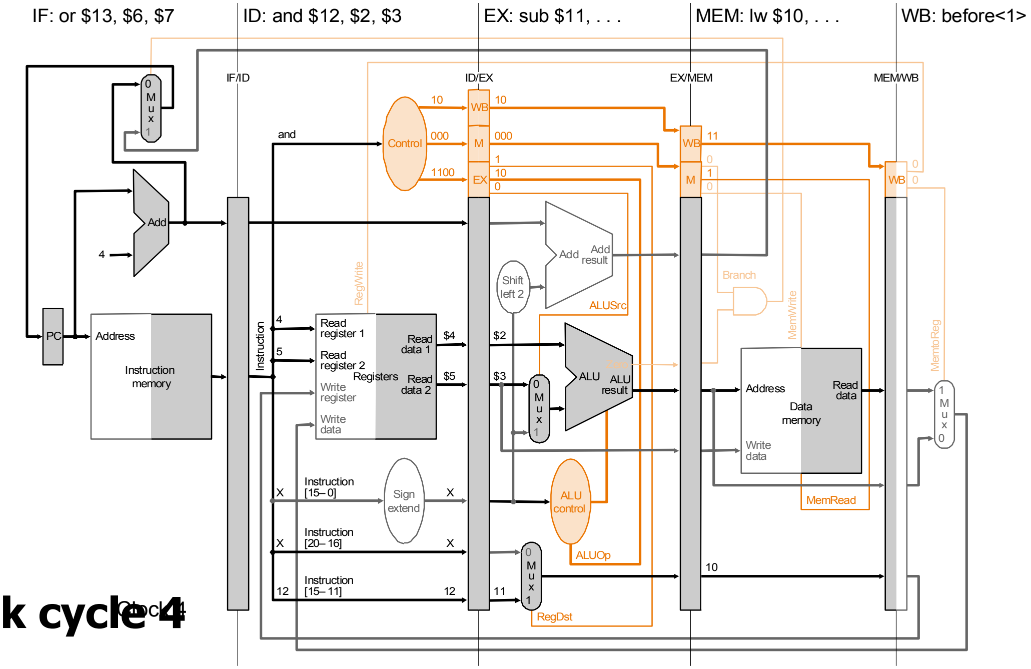
Pipelined Execution and Control

Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```



Clock cycle 3



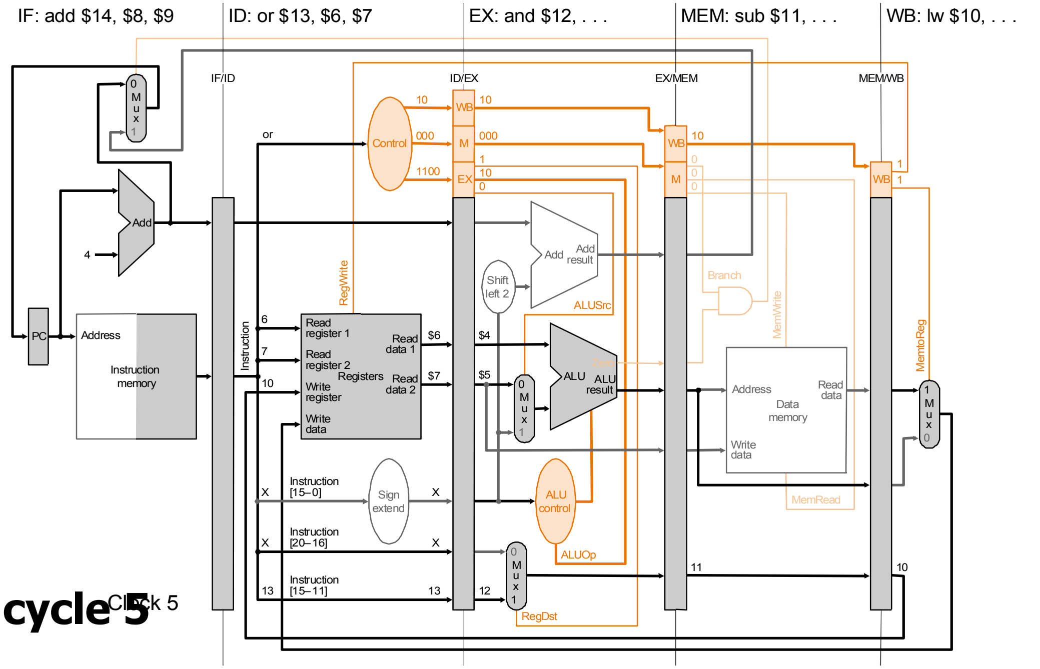
Clock cycle 4

Pipelined Execution and Control

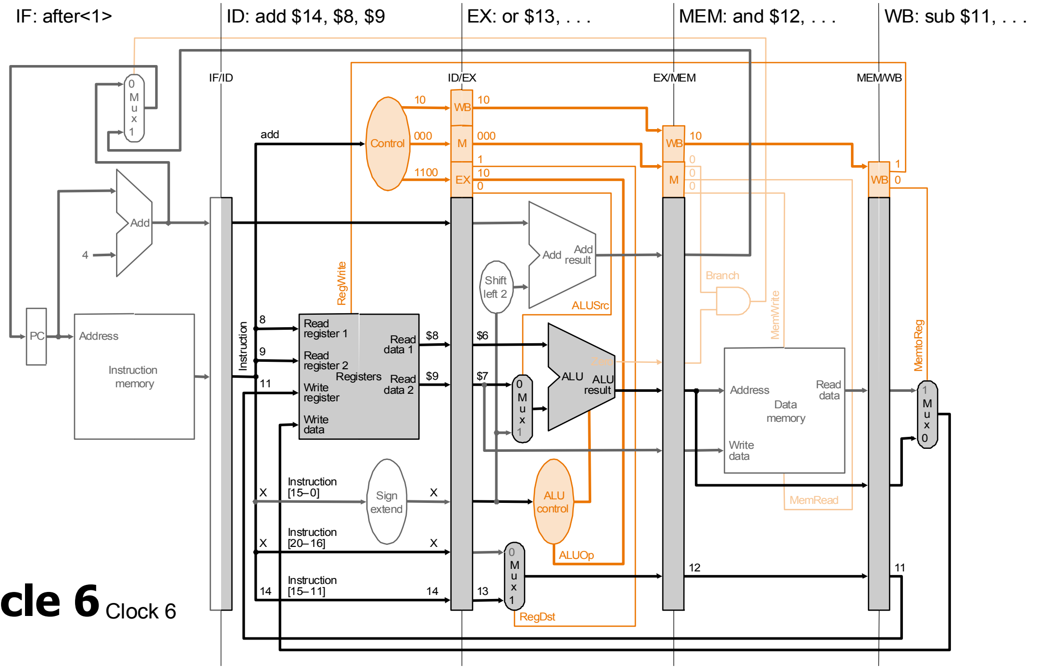
Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

Label “after<i>” means i th instruction after `add`



Clock cycle 5

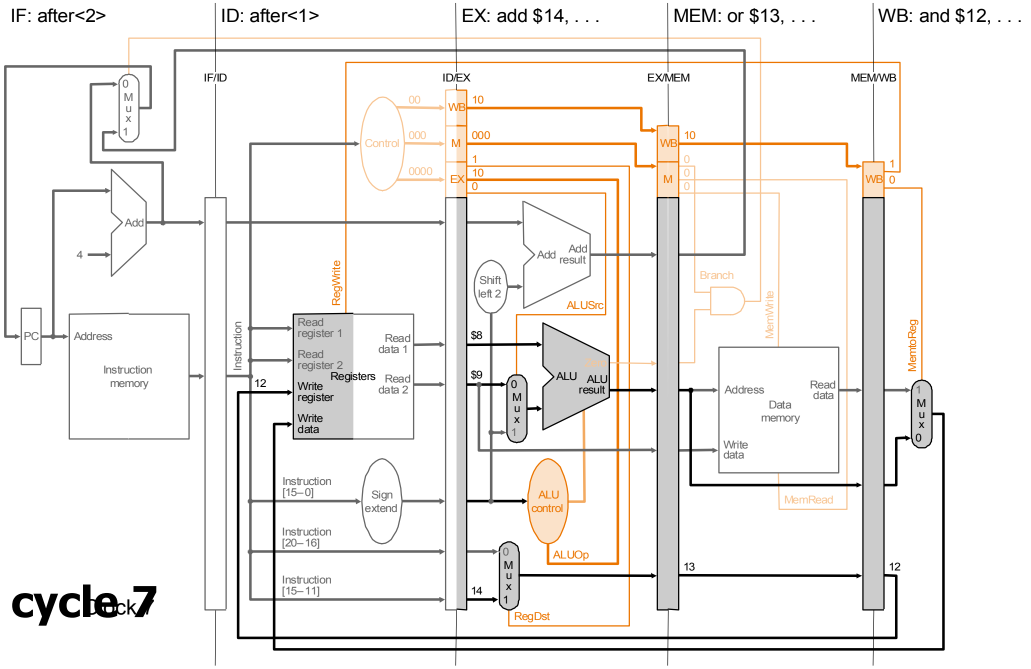


Clock cycle 6

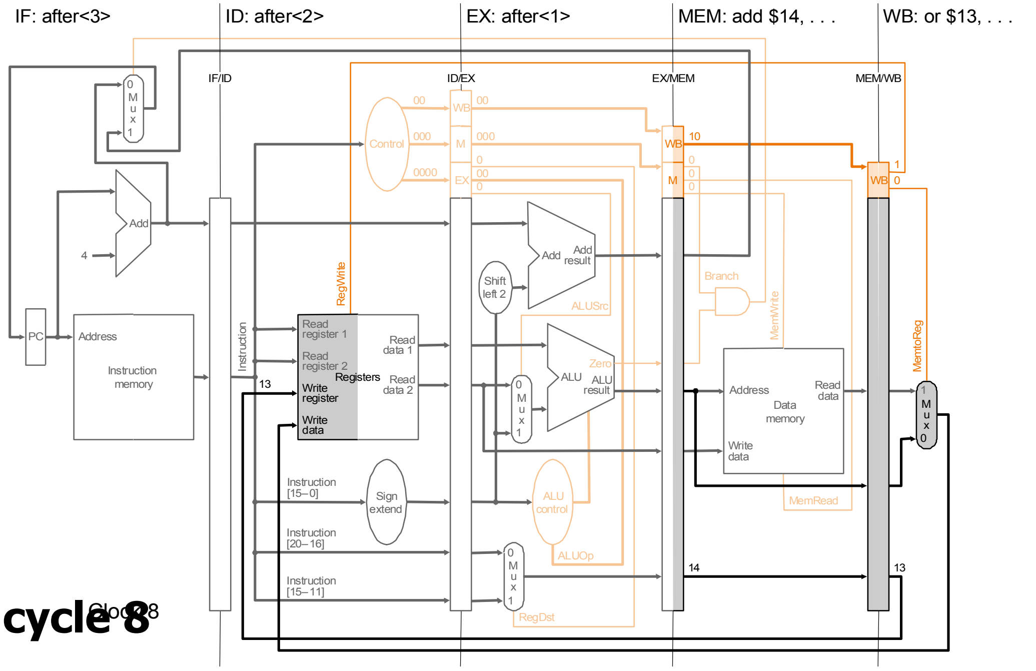
Pipelined Execution and Control

Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```



Clock cycle 7

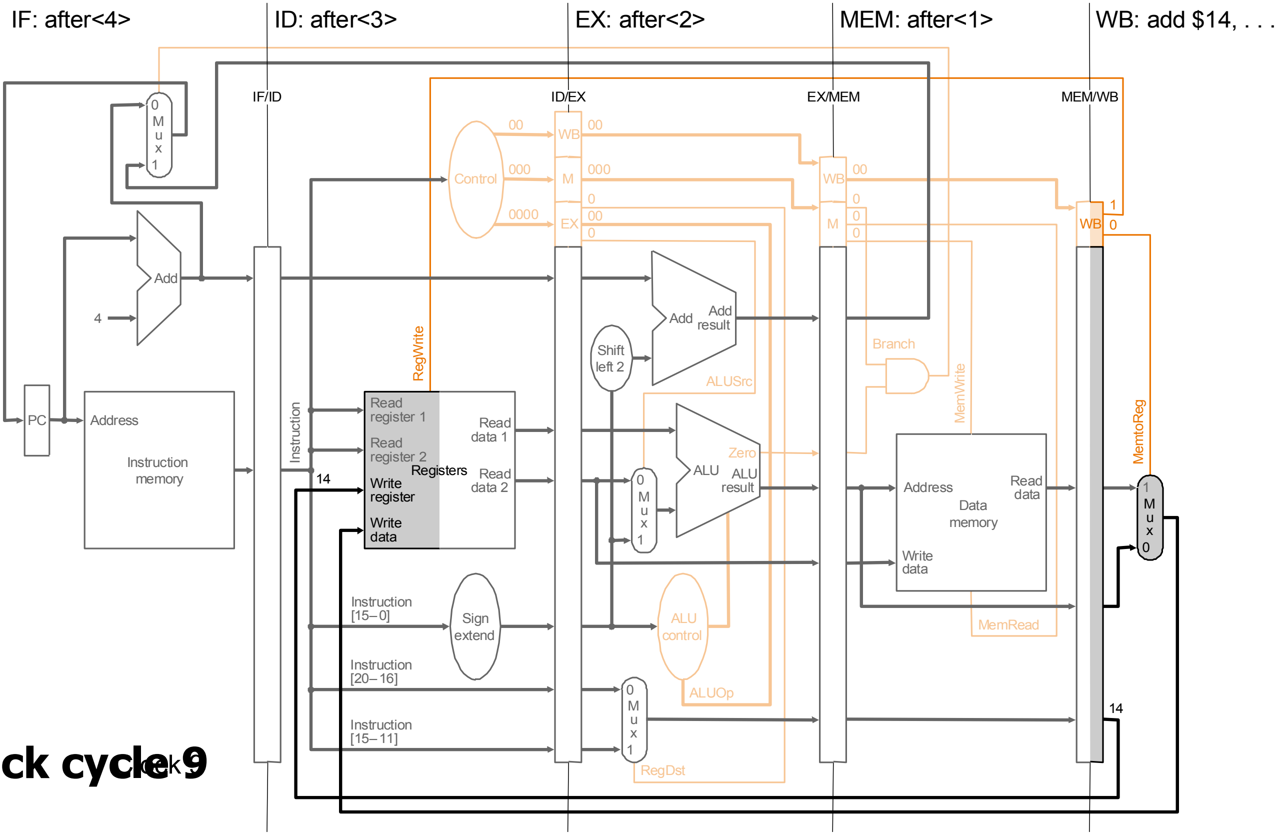


Clock cycle 8

Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```



Clock cycle 9

Data Hazards

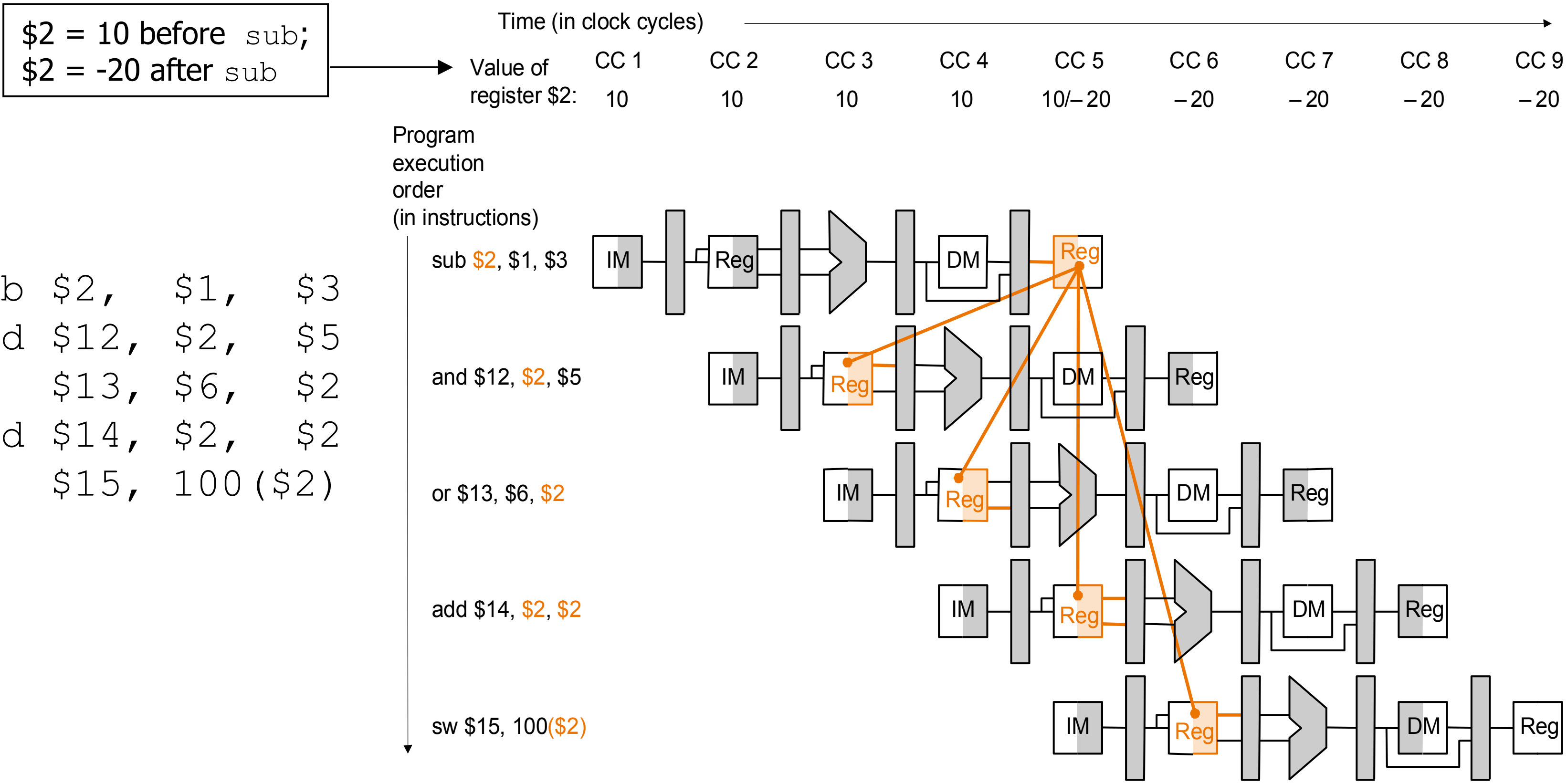
Revisiting Hazards

- So far our datapath and control have ignored hazards
- We shall revisit *data hazards* and *control hazards* and enhance our datapath and control to handle them in *hardware*…

Data Hazards and Forwarding

- Problem with starting an instruction before previous are finished:
  - data dependencies that go backward in time – called *data hazards*

```
sub $2,  $1,  $3
and $12, $2,  $5
or  $13, $6,  $2
add $14, $2,  $2
sw  $15, 100($2)
```



Software Solution

- Have compiler guarantee *never* any data hazards!
  - by *rearranging instructions to insert independent instructions between instructions* that would otherwise have a data hazard between them,
  - or, if such rearrangement is not possible, *insert* `nop`s

```
sub     $2,  $1, $3
lw      $10, 40($3)
slt     $5, $6, $7
and     $12, $2, $5
or      $13, $6, $2
add     $14, $2, $2
sw      $15, 100($2)
```

or

```
sub     $2,  $1, $3
nop
nop
and     $12, $2, $5
or      $13, $6, $2
add     $14, $2, $2
sw      $15, 100($2)
```

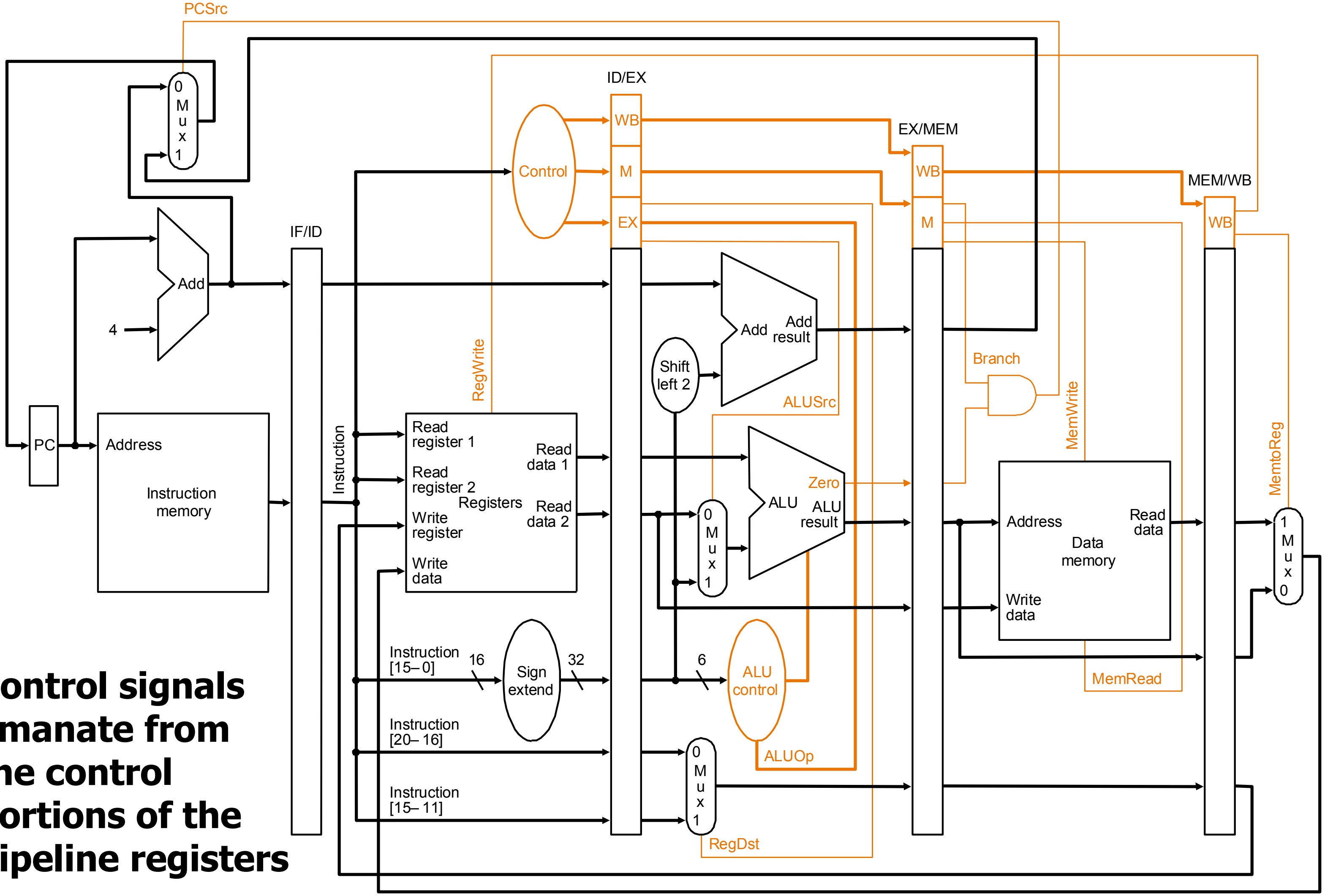
- Such compiler solutions may not always be possible, and `nop`s slow the machine down

MIPS: `nop` = "no operation" = 00…0 (32bits) = `sll $0, $0, 0`

Hardware Solution: Forwarding

- Idea: *use intermediate data*, do not wait for result to be finally written to the destination register. Two steps:
  1. *Detect* data hazard
  2. *Forward* intermediate data to resolve hazard

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

Hazard Detection

- Hazard conditions:

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

  - Eg., in the earlier example, first hazard between `sub $2, $1, $3` and `and $12, $2, $5` is detected when the `and` is in EX stage and the `sub` is in MEM stage because
    - `EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)`

- Whether to forward also depends on:
  - *if the later instruction is going to write a register* – if *not*, no need to forward, even if there is register number match as in conditions above
  - *if the destination register of the later instruction is $0* – in which case there is no need to forward value ($0 is always 0 and never overwritten)

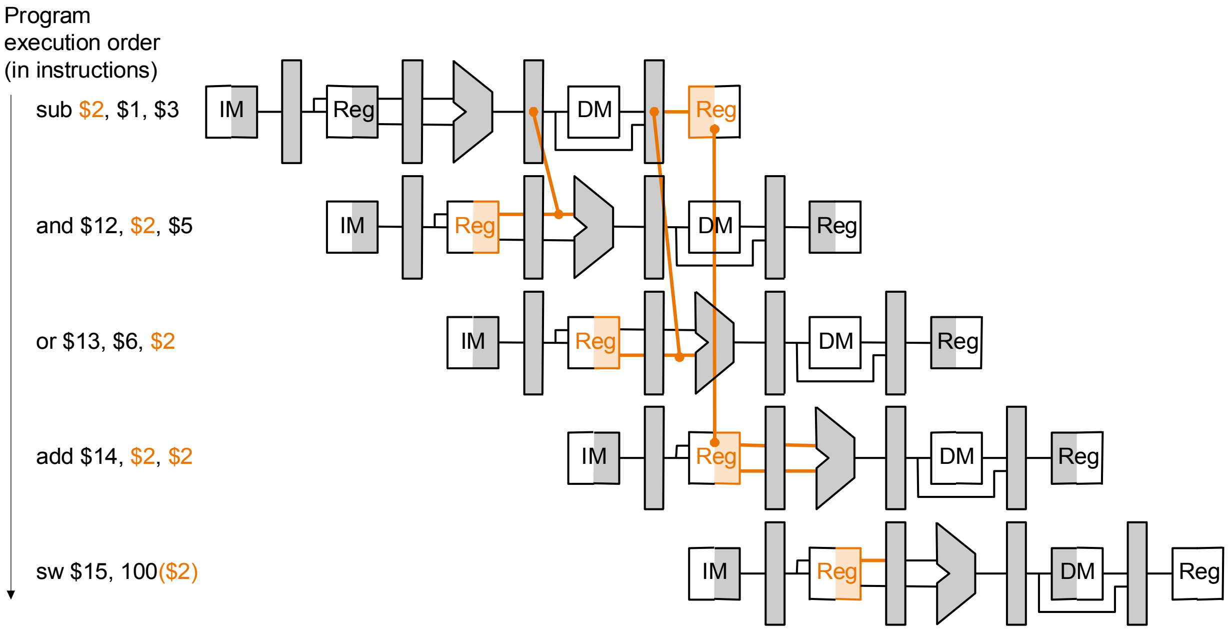
Data Forwarding

- Plan:
  - *allow inputs to the ALU not just from ID/EX, but also later pipeline registers*, and
  - *use multiplexors and control signals to choose appropriate inputs* to ALU

```
sub $2,  $1,  $3
and $12, $2,  $5
or  $13, $6,  $2
add $14, $2,  $2
sw  $15, 100($2)
```

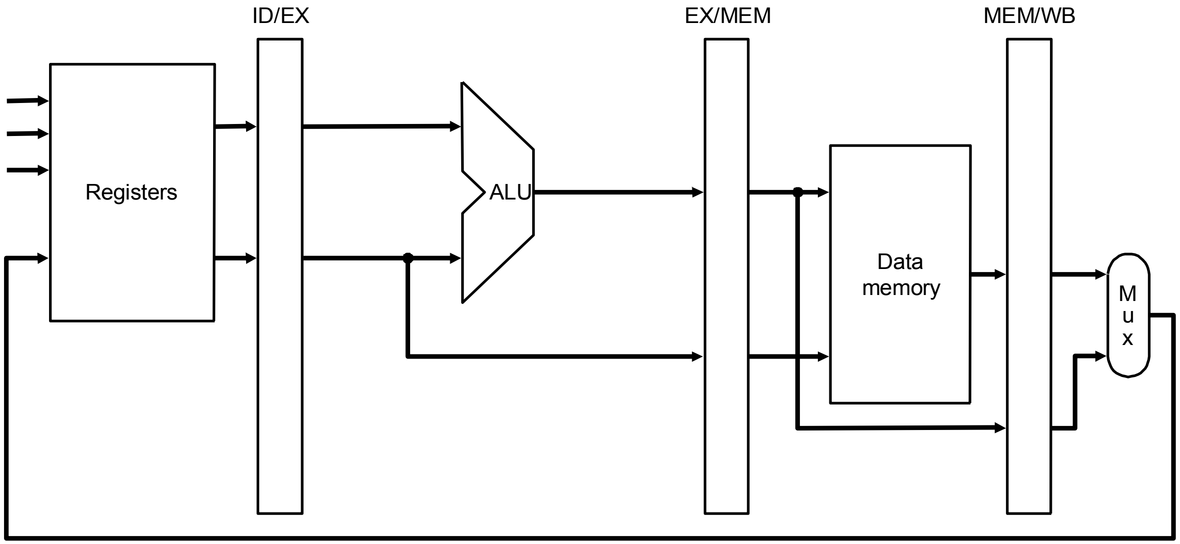
Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X

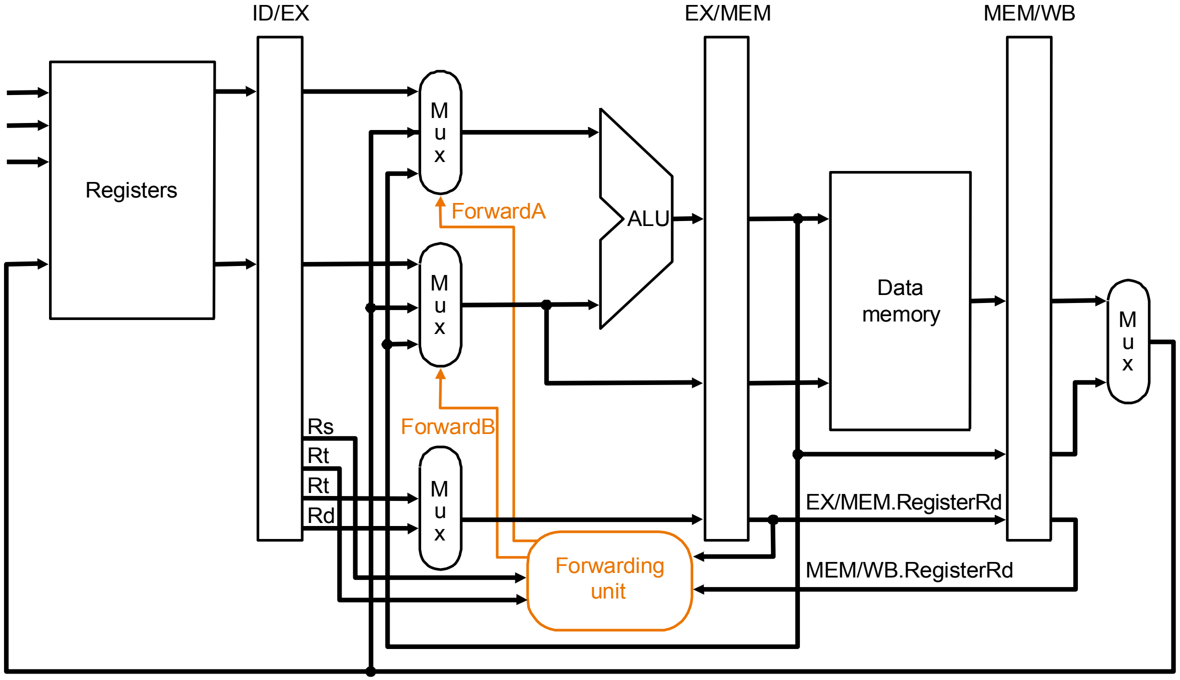


Dependencies between pipelines move forward in time

Forwarding Hardware



a. No forwarding **Datapath before adding forwarding hardware**



b. With forwarding **Datapath after adding forwarding hardware**

Forwarding Hardware: Multiplexor Control

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from prior ALU result
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result
ForwardB = 00	ID/EX	The second ALU operand comes from the register file
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from prior ALU result
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result

Depending on the selection in the rightmost multiplexor
(see datapath with control diagram)

Data Hazard: Detection and Forwarding

- Forwarding unit determines multiplexor control according to the following rules:

1. **EX hazard**

```
if (     EX/MEM.RegWrite                              // if there is a write…
    and ( EX/MEM.RegisterRd $\neq$ 0 )                   // to a non-$0 register…
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRs ) )  // which matches, then…
ForwardA = 10

if (     EX/MEM.RegWrite                              // if there is a write…
    and ( EX/MEM.RegisterRd $\neq$ 0 )                   // to a non-$0 register…
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRt ) )  // which matches, then…
 ForwardB = 10
```

Data Hazard: Detection and Forwarding

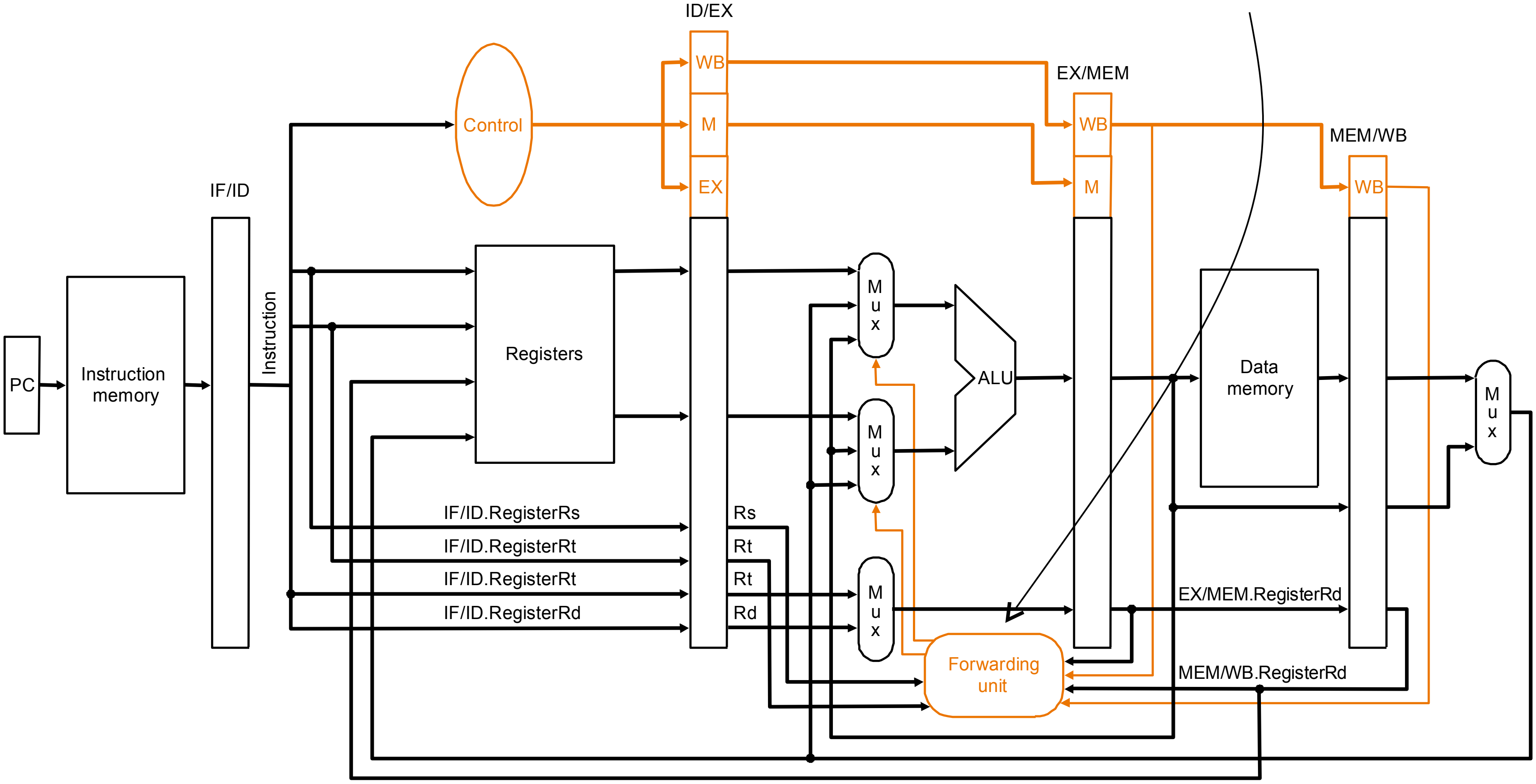
2. MEM hazard

```
if (        MEM/WB.RegWrite                                  // if there is a write…
   and ( MEM/WB.RegisterRd ≠ 0 )                             // to a non-$0 register…
   and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRs )              // and not already a register match
                                                             // with earlier pipeline register…
   and ( MEM/WB.RegisterRd = ID/EX.RegisterRs ) )            // but match with later pipeline register, then…

ForwardA = 01

if (        MEM/WB.RegWrite                                  // if there is a write…
   and ( MEM/WB.RegisterRd ≠ 0 )                             // to a non-$0 register…
   and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRt )              // and not already a register match
                                                             // with earlier pipeline register…
   and ( MEM/WB.RegisterRd = ID/EX.RegisterRt ) )            // but match with later pipeline register, then…

ForwardB = 01
```

This check is necessary, e.g., for sequences such as add $1, $1, $2; add $1, $1, $3; add $1, $1, $4; (array summing?), where an earlier pipeline (EX/MEM) register has more recent data

Forwarding Hardware with Control

Called forwarding unit, not hazard detection unit, because once data is forwarded there is no hazard!



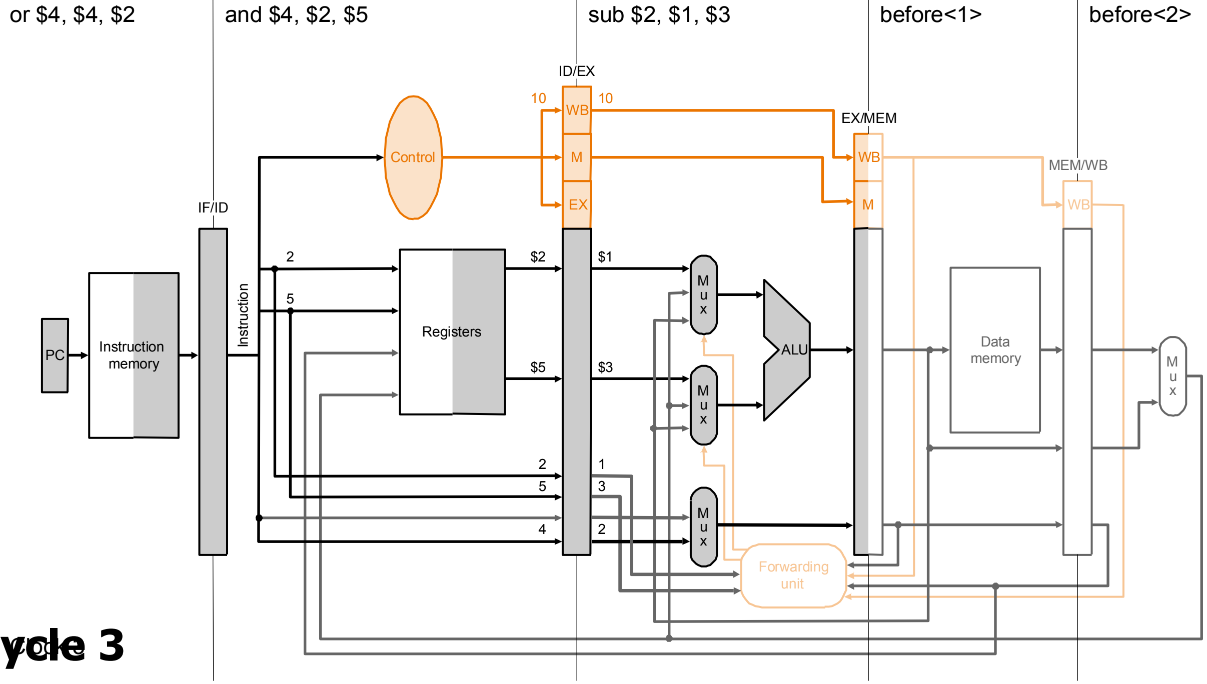
Datapath with forwarding hardware and control wires – certain details, e.g., branching hardware, are omitted to simplify the drawing

Note: so far we have only handled forwarding to R-type instructions…!

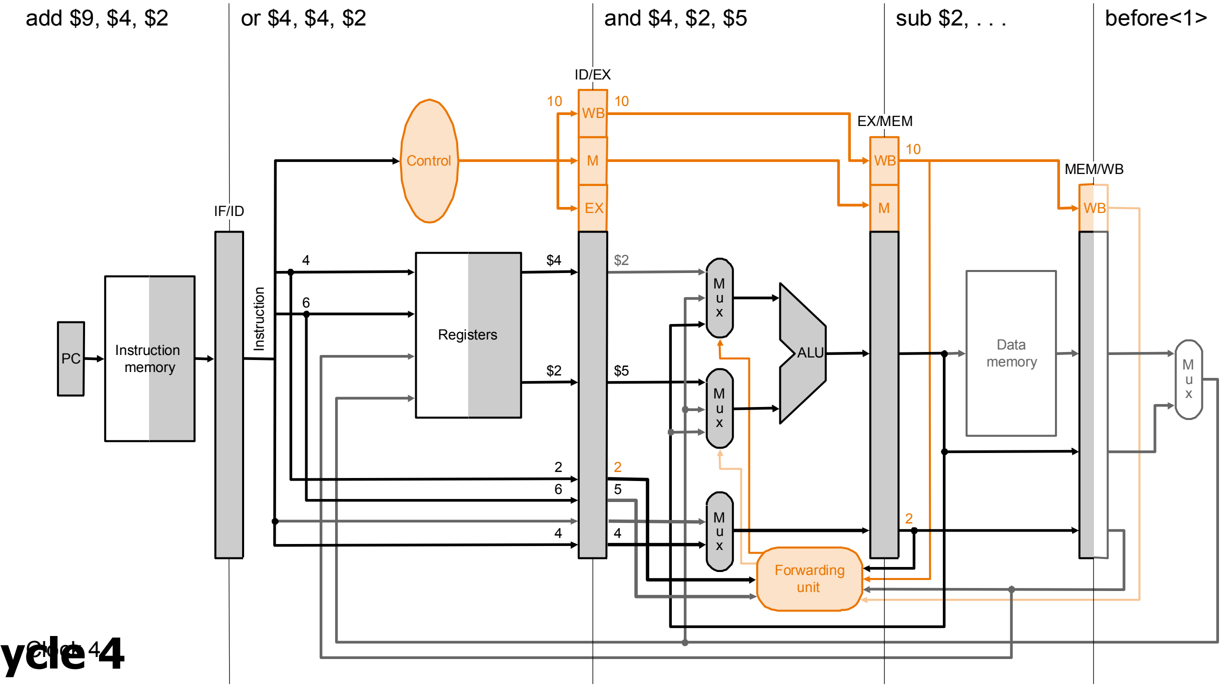
Forwarding

Execution example:

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 3

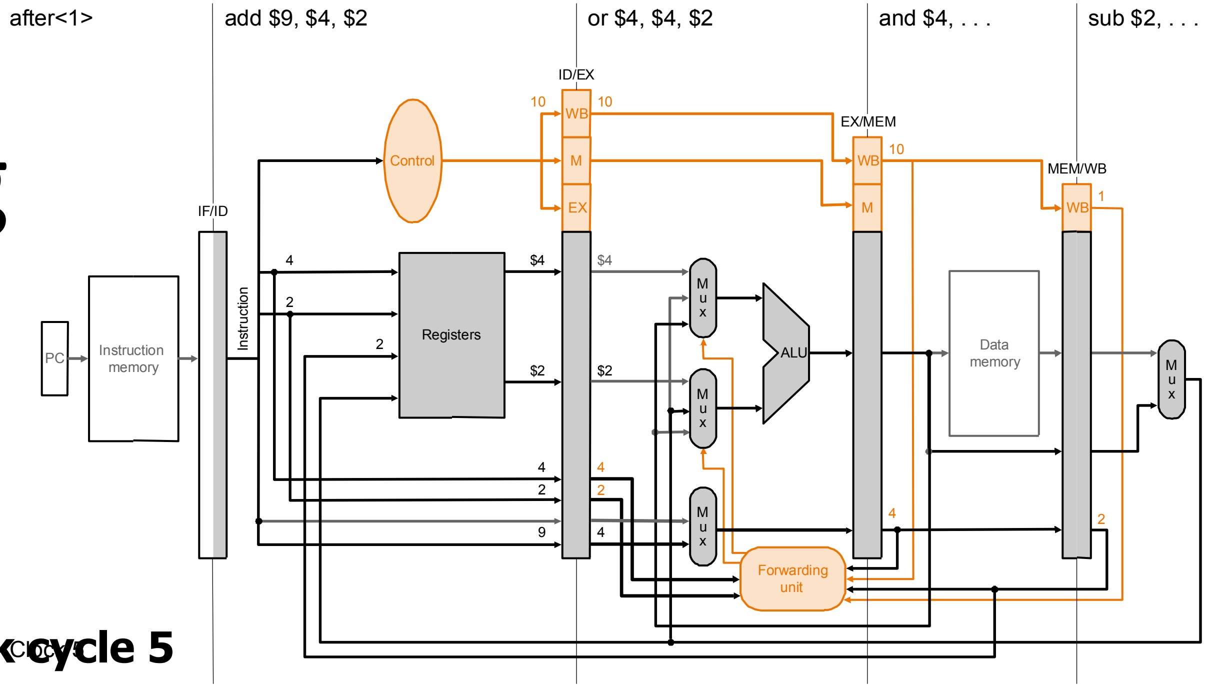


Clock cycle 4

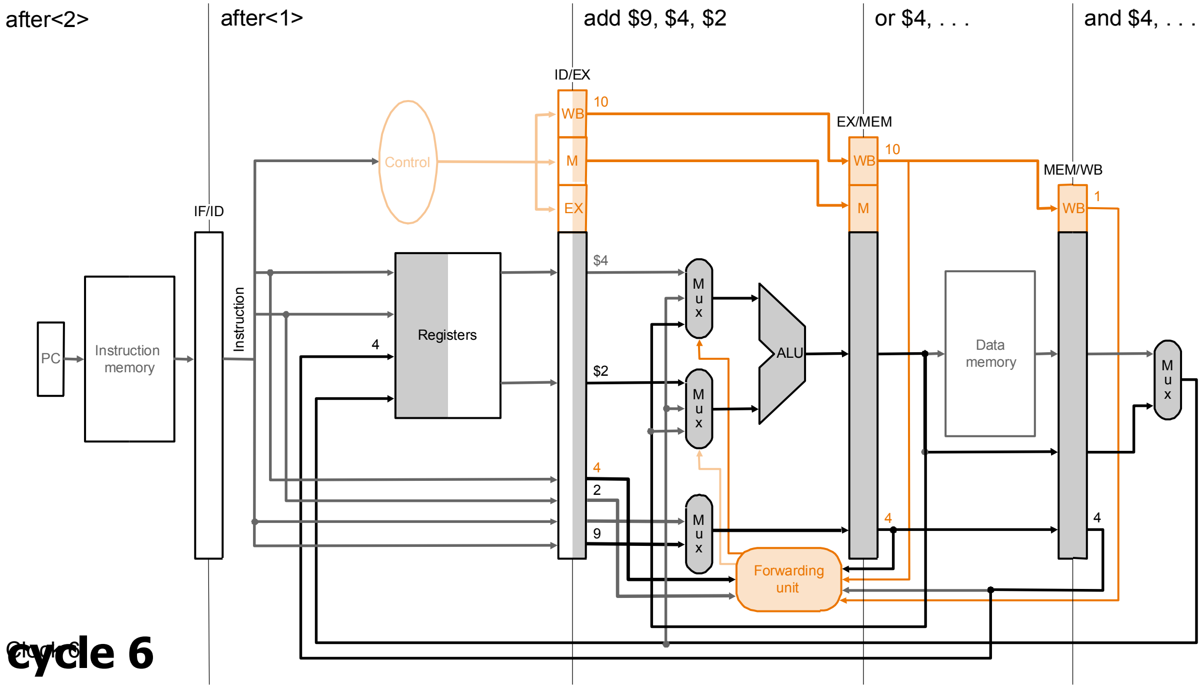
Forwarding

Execution example (cont.):

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 5



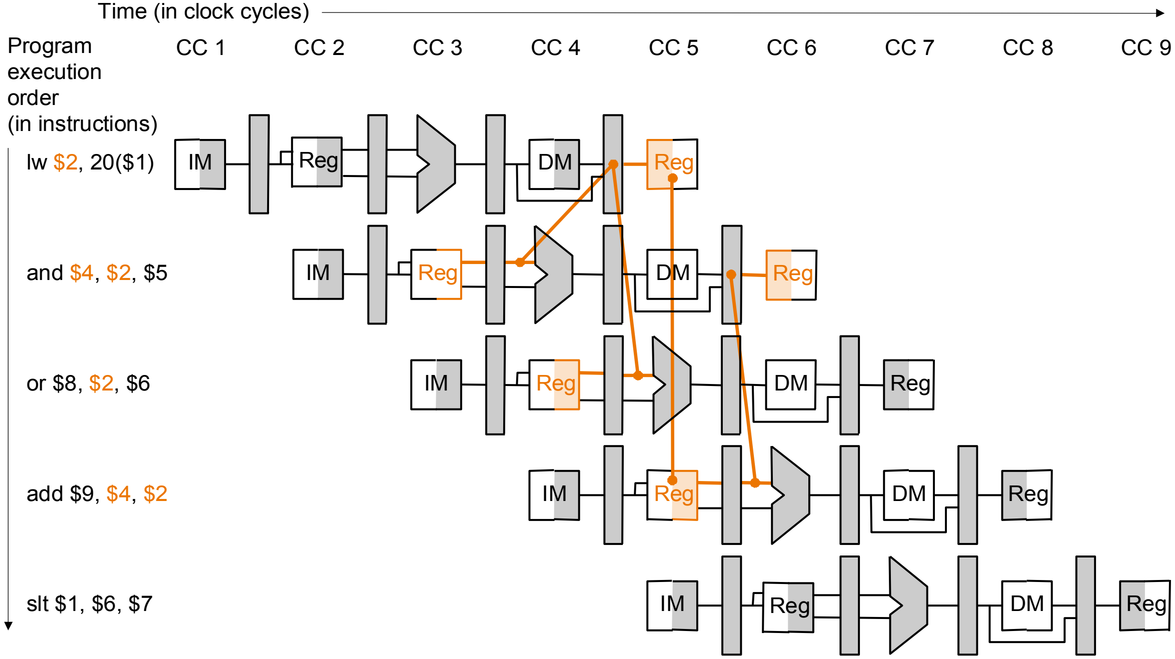
Clock cycle 6

Data Hazards and Stalls

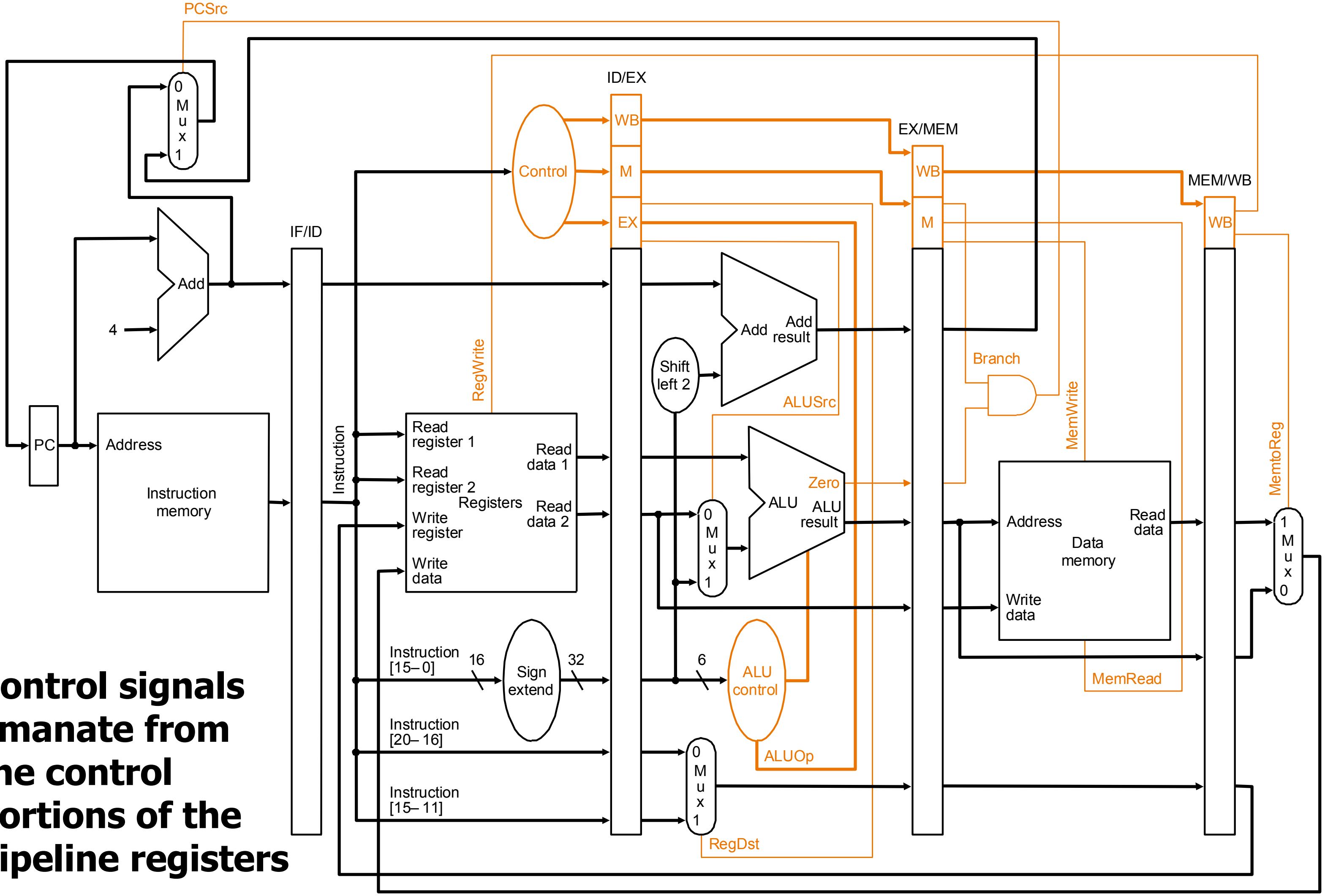
- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register
  - therefore, we need a *hazard detection unit* to *stall* the pipeline after the load instruction

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
        -
```

As even a pipeline dependency goes backward in time forwarding will not solve the hazard



Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

Hazard Detection Logic to Stall

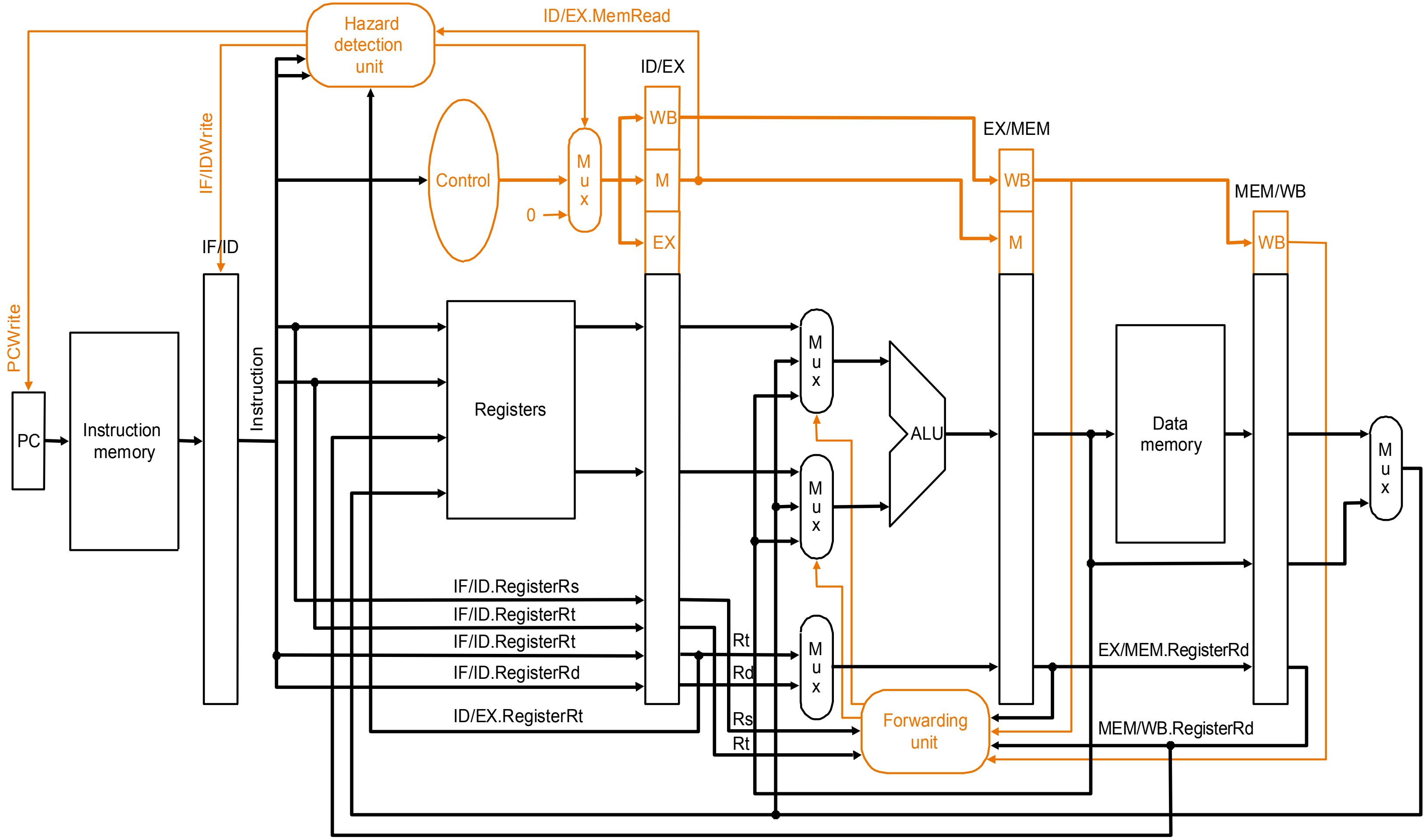
- Hazard detection unit implements the following check if to stall

```
if ( ID/EX.MemRead  // if the instruction in the EX stage is a load…
  and ( ( ID/EX.RegisterRt = IF/ID.RegisterRs )    // and the destination register
  or  ( ID/EX.RegisterRt = IF/ID.RegisterRt ) ) )   // matches either source register of the
                                                   //instruction in the ID stage, then… stall the pipeline
```

Mechanics of Stalling

- If the check to stall verifies, then the *pipeline needs to stall only 1 clock cycle* after the load as after that the forwarding unit can resolve the dependency
- What the hardware does to stall the pipeline 1 cycle:
  - *does not let the IF/ID register change* (*disable write*!) – this will cause the instruction in the ID stage to repeat, i.e., *stall*
  - therefore, the instruction, just behind, in the IF stage must be stalled as well – so hardware *does not let the PC change* (*disable write*!) – this will cause the instruction in the IF stage to repeat, i.e., *stall*
  - *changes all the EX, MEM and WB control fields in the ID/EX pipeline register to 0*, so effectively the instruction just behind the load becomes a `nop` – a *bubble* is said to have been inserted into the pipeline
    - note that we cannot turn that instruction into an `nop` by 0ing all the bits in the instruction itself – recall `nop` = 00…0 (32 bits) – because it has already been decoded and control signals generated

Hazard Detection Unit

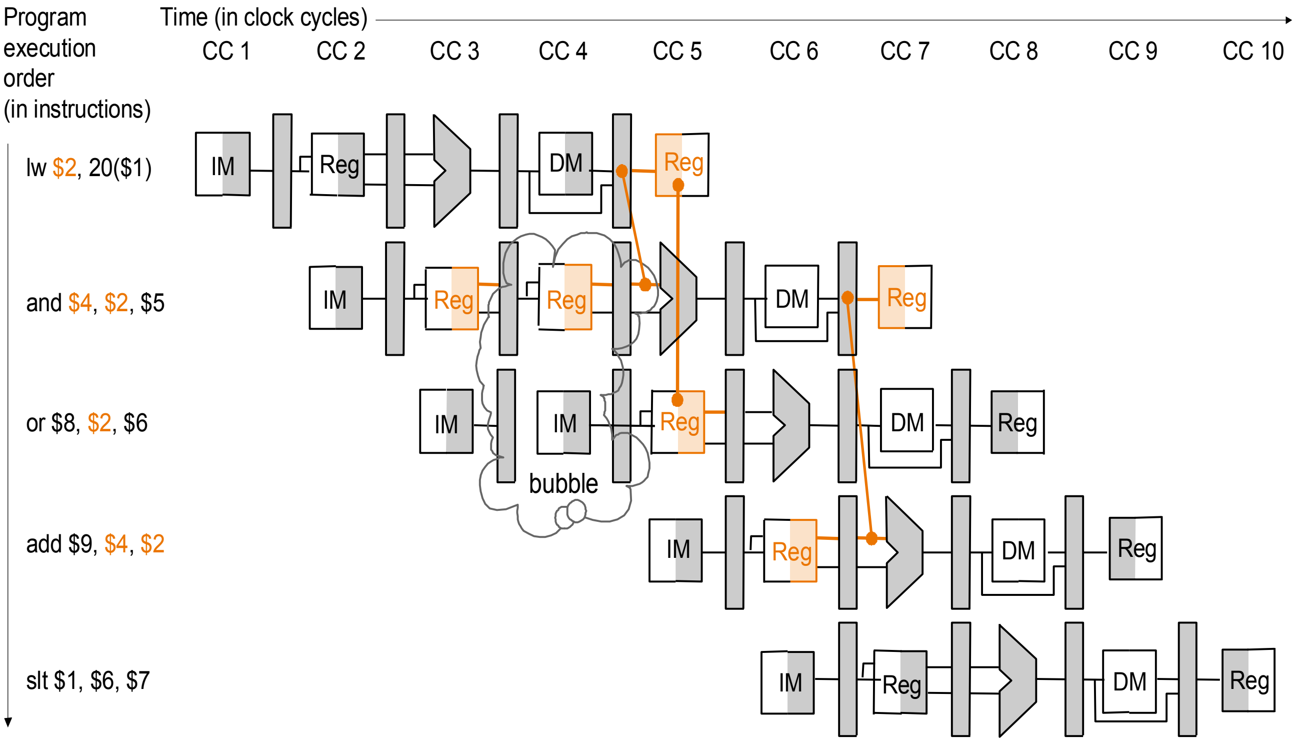


Datapath with forwarding hardware, the hazard detection unit and controls wires – certain details, e.g., branching hardware are omitted to simplify the drawing

Stalling Resolves a Hazard

- Same instruction sequence as before for which forwarding by itself could not resolve the hazard:

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
```

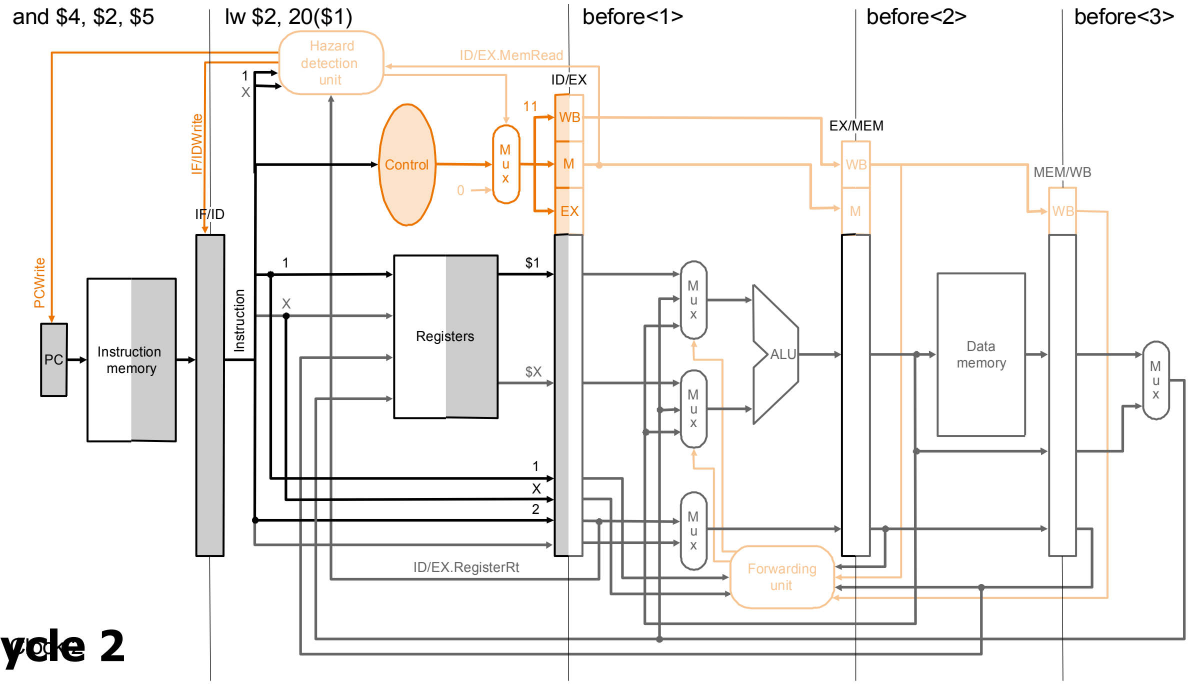


Hazard detection unit inserts a 1-cycle bubble in the pipeline, after which all pipeline register dependencies go forward so then the forwarding unit can handle them and there are no more hazards

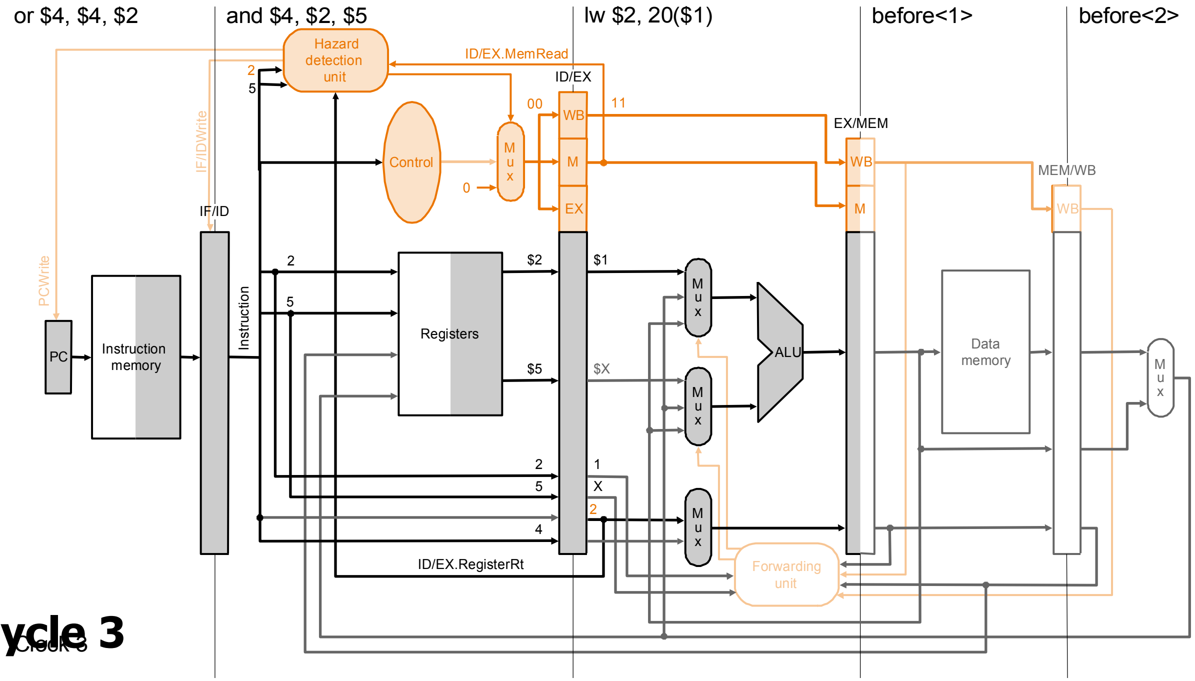
Stalling

- Execution example:

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 2

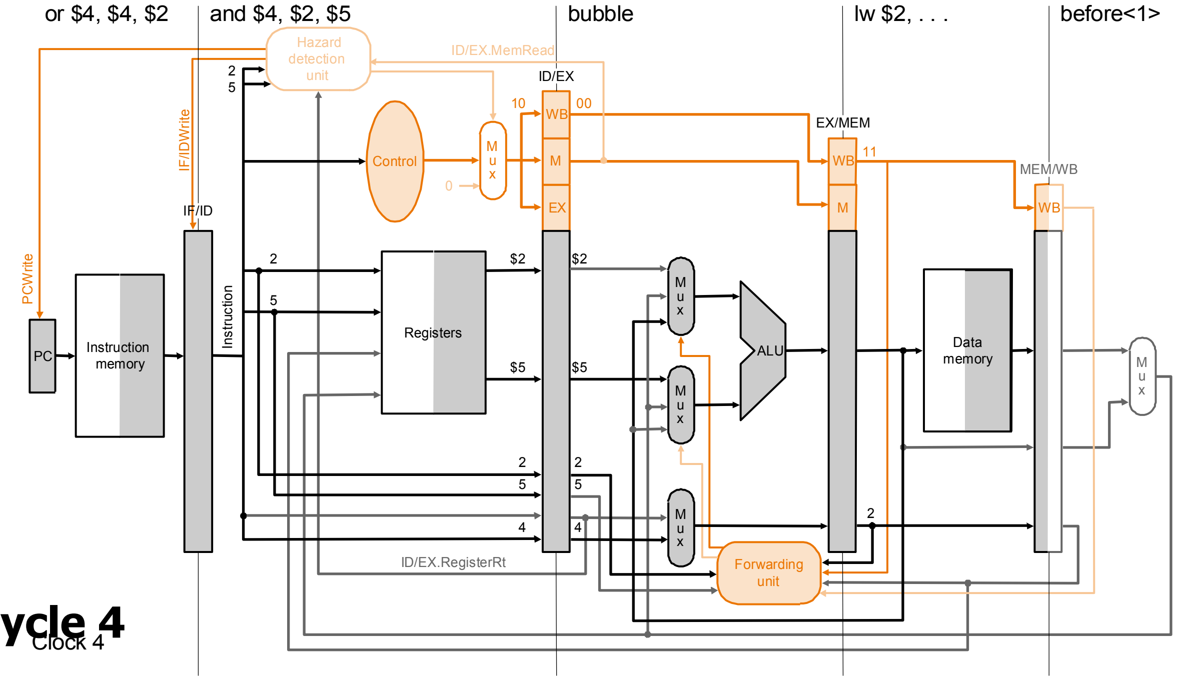


Clock cycle 3

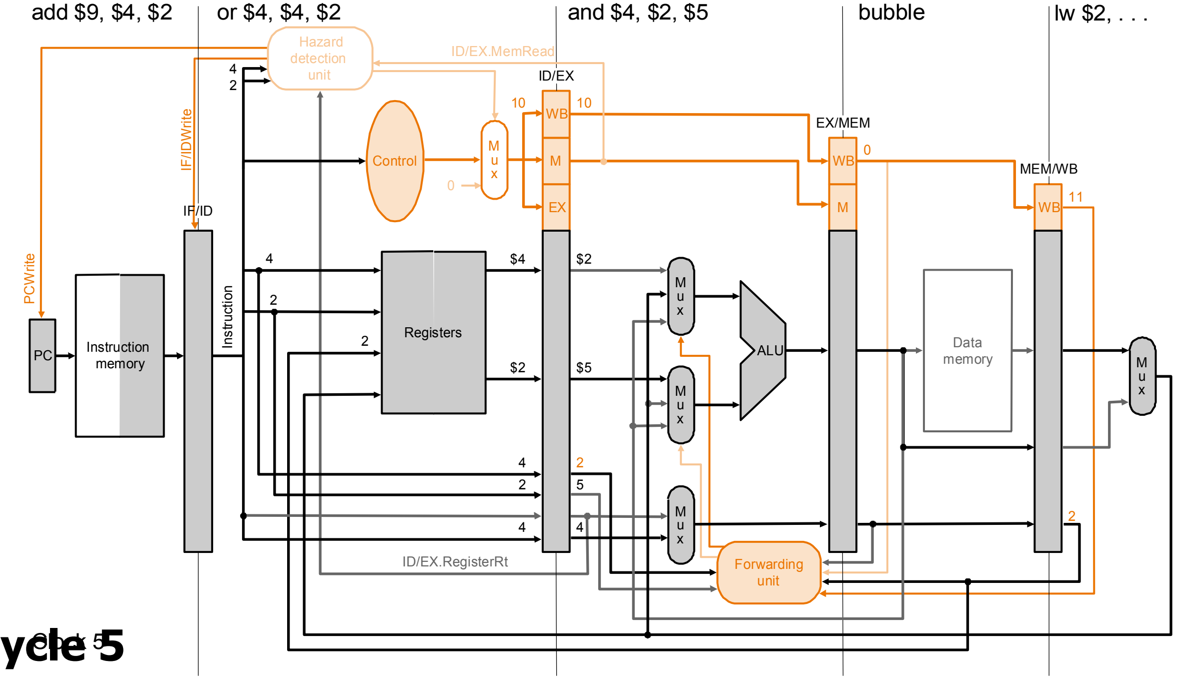
Stalling

- Execution example (cont.):

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 4

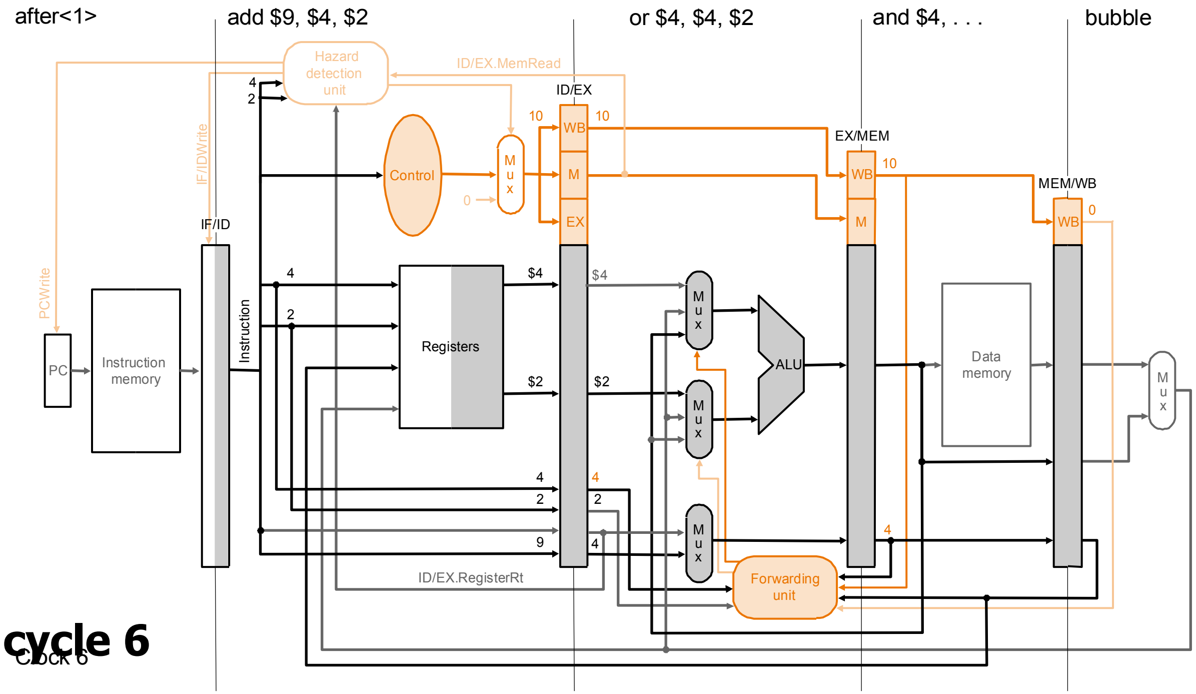


Clock cycle 5

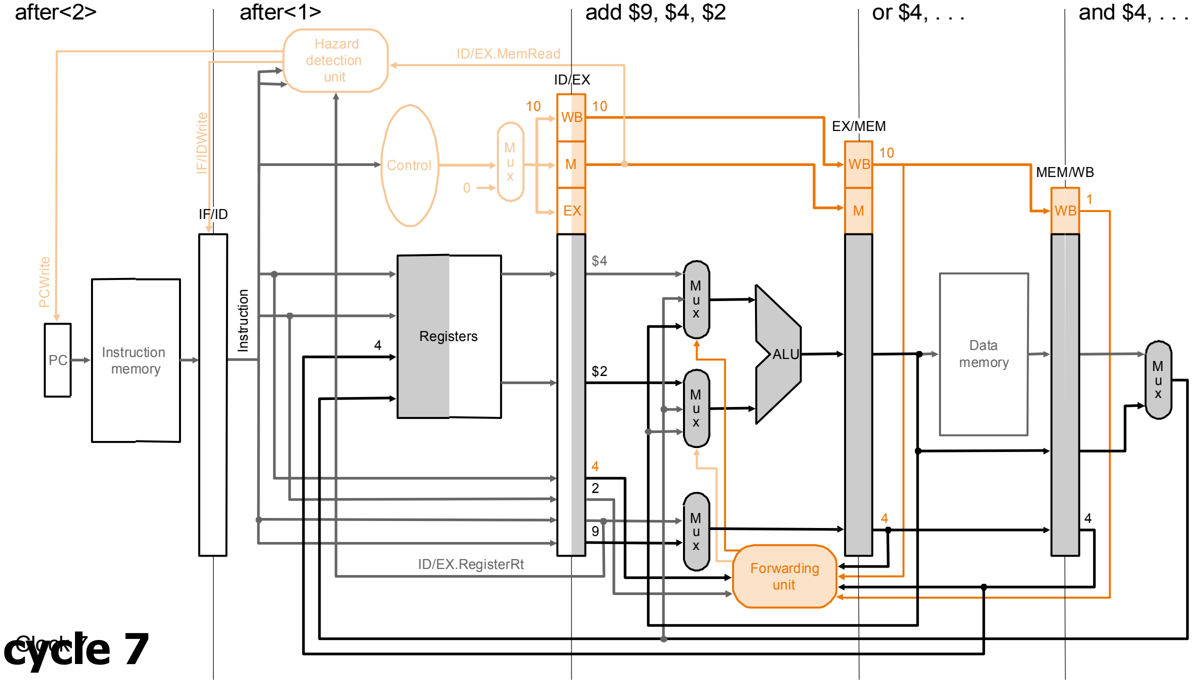
Stalling

- Execution example (contd).

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 6



Clock cycle 7

Control Hazards

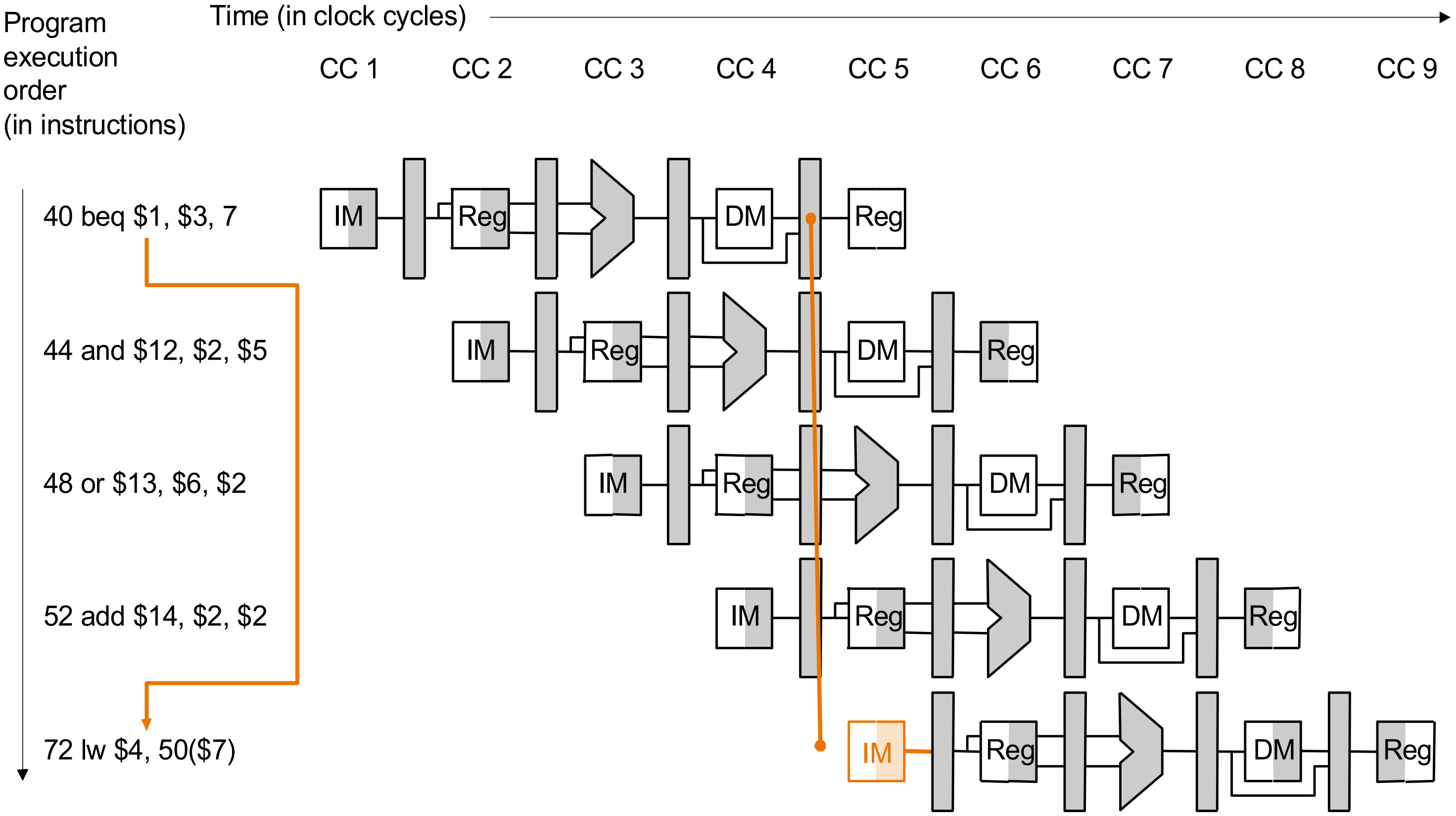
Control (or Branch) Hazards

- Problem with branches in the pipeline we have so far is that the *branch decision is not made till the MEM stage* – so *what instructions, if at all, should we insert into the pipeline following the branch instructions*?

- Possible solution: *stall* the pipeline till branch decision is known
  - not efficient, slow the pipeline significantly!

- Another solution: *predict* the branch outcome
  - e.g., always predict *branch-not-taken – continue with next sequential instructions*
  - if the prediction is wrong have to *flush* the pipeline behind the branch – discard instructions already fetched or decoded – and *continue execution at the branch target*

Predicting Branch-not-taken: Misprediction delay



Program execution order (in instructions)

Time (in clock cycles)

CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9

40 beq $1, $3, 7

44 and $12, $2, $5

48 or $13, $6, $2

52 add $14, $2, $2

72 lw $4, 50($7)

The outcome of branch taken (prediction wrong) is decided only when `beq` is in the MEM stage, so the following three sequential instructions already in the pipeline have to be flushed and execution resumes at `lw`

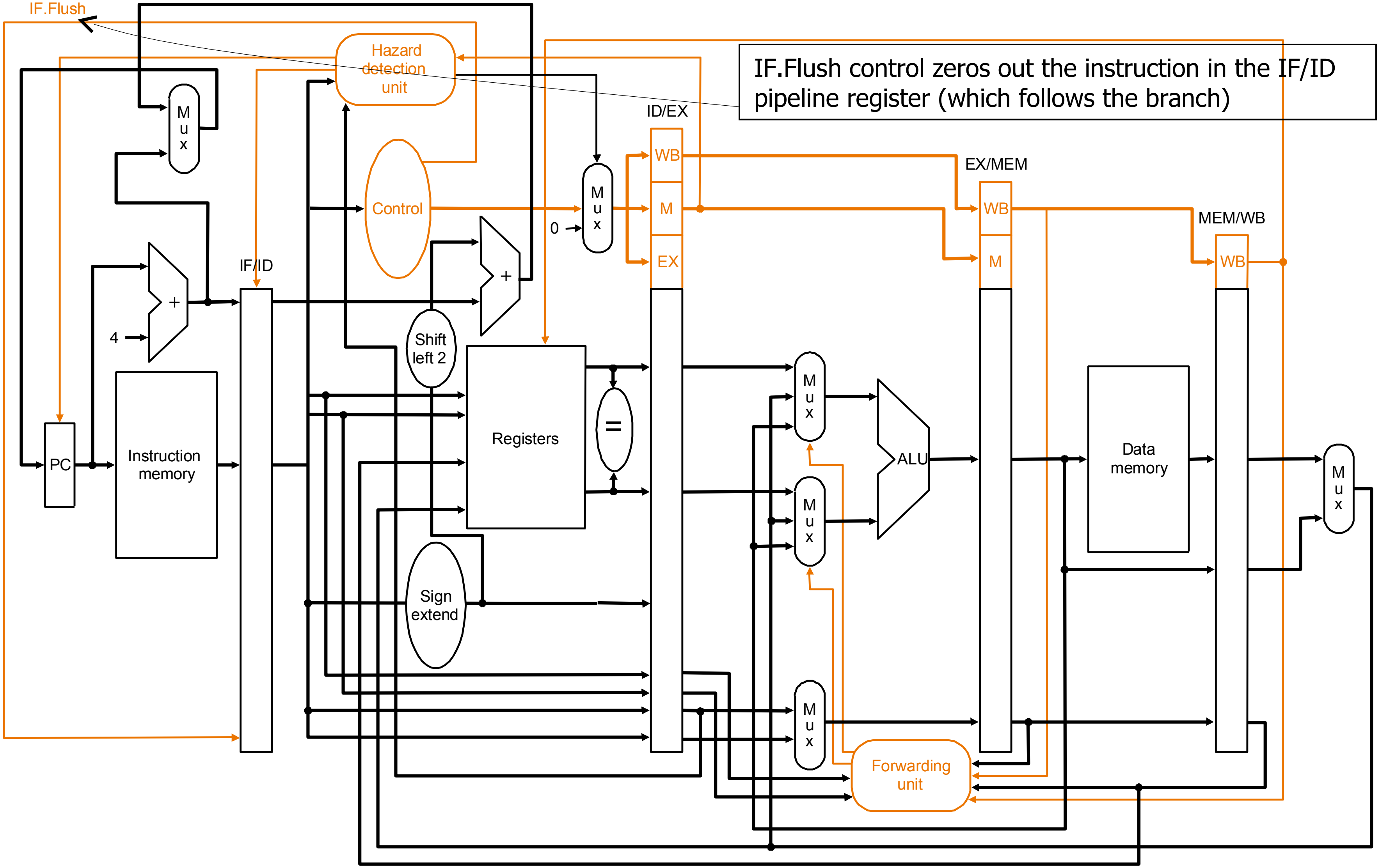
Optimizing the Pipeline to Reduce Branch Delay

- *Move the branch decision* from the MEM stage (as in our current pipeline) *earlier to the ID stage*
  - *calculating the branch target address* involves moving the branch adder from the MEM stage to the ID stage – inputs to this adder, the PC value and the immediate fields are already available in the IF/ID pipeline register
  - *calculating the branch decision* is efficiently done, e.g., for equality test, by XORing respective bits and then ORing all the results and inverting, rather than using the ALU to subtract and then test for zero (when there is a carry delay)
    - with the more efficient equality test we can put it in the ID stage without significantly lengthening this stage – remember an objective of pipeline design is to keep pipeline stages balanced
  - *we must correspondingly make additions to the forwarding and hazard detection units* to forward to or stall the branch at the ID stage in case the branch decision depends on an earlier result

Flushing on Misprediction

- Same strategy as for stalling on load-use data hazard…
- Zero out all the control values (or the instruction itself) in pipeline registers for the instructions following the branch that are already in the pipeline – effectively turning them into `nop`s – so they are flushed
  - in the optimized pipeline, with branch decision made in the ID stage, we have to flush only one instruction in the IF stage – the branch delay penalty is then only one clock cycle

Optimized Datapath for Branch



IF.Flush control zeros out the instruction in the IF/ID pipeline register (which follows the branch)

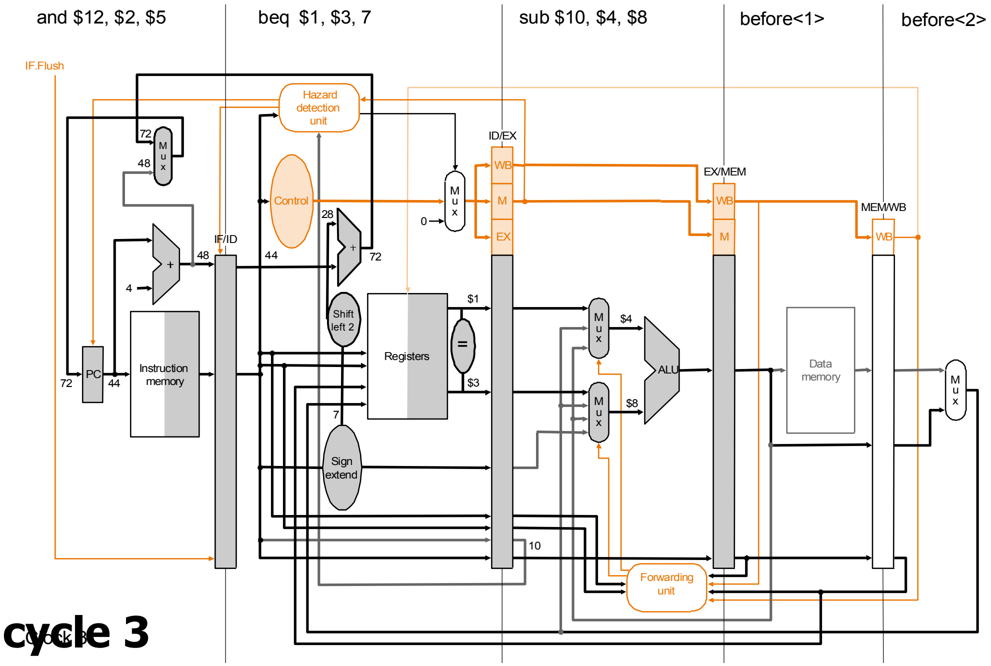
Branch decision is moved from the MEM stage to the ID stage – simplified drawing not showing enhancements to the forwarding and hazard detection units

Pipelined Branch

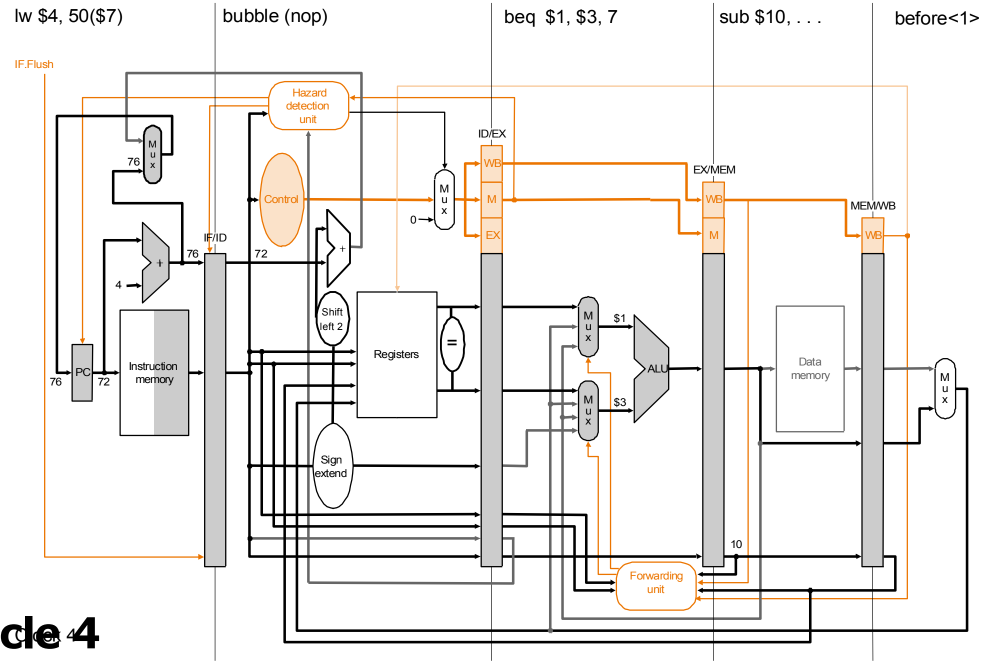
- Execution example:

```
36 sub $10, $4, $8
40 beq $1,  $3,  7
44 and $12  $2, $5
48 or  $13  $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
…
72 lw  $4,  50($7)
```

Optimized pipeline with only one bubble as a result of the taken branch



Clock cycle 3



Clock cycle 4

Simple Example: Comparing Performance

- *Compare performance for single-cycle, multicycle, and pipelined datapaths using the gcc instruction mix*
  - assume 2 ns for memory access, 2 ns for ALU operation, 1 ns for register read or write
  - assume gcc instruction mix 23% loads, 13% stores, 19% branches, 2% jumps, 43% ALU
  - for pipelined execution assume
    - 50% of the loads are followed immediately by an instruction that uses the result of the load
    - 25% of branches are mispredicted
    - branch delay on misprediction is 1 clock cycle
    - jumps always incur 1 clock cycle delay so their average time is 2 clock cycles

Simple Example: Comparing Performance

- *Single-cycle* (p. 373): average instruction time 8 ns
- *Multicycle* (p. 397): average instruction time 8.04 ns
- *Pipelined*:
  - loads use 1 cc (clock cycle) when no load-use dependency and 2 cc when there is dependency – given 50% of loads are followed by dependency the average cc per load is 1.5
  - stores use 1 cc each
  - branches use 1 cc when predicted correctly and 2 cc when not – given 25% misprediction average cc per branch is 1.25
  - jumps use 2 cc each
  - ALU instructions use 1 cc each
  - therefore, average CPI is

  1.5 × 23% + 1 × 13% + 1.25 × 19% + 2 × 2% + 1 × 43% = 1.18

  - therefore, average instruction time is 1.18 × 2 = 2.36 ns

Dynamic Branch Prediction

- Prediction of branches at runtime using runtime information.
- Ex: Restaurant
- Two schemes:
  - 1-bit scheme
  - 2-bit scheme

1-bit scheme

- **branch prediction buffer -** Also called **branch history table.**
- A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.
- The memory contains a bit that says whether the branch was recently taken or not

Branch History Table

	Bit indicating branch taken(1) or not taken(0)
1000	0
1012	1
2046	0
2116	1

```
1000 Beq $s2,$s1, exit
1012 Beq $s2,$s1, for
2046 Beq $s2,$s1, else
```

Assume 0 – bit indicating branch not taken
            1 – bit indicating branch taken

Disadvantage of 1-bit scheme

- What if the branch is taken and not taken the alternatively?
  - Then the prediction table will be wrong always.
  - (ie) prediction will be always a **failure**

Disadvantage example

```
1000 i=1;
1004 while(i<=9){
1008          i++;
      }
```

	Bit indicating branch taken(1)or not taken(0)
1004	1

Initially – bit is set to 1

1st iteration – misprediction(since branch is not taken)

Next 8 iteration – prediction is success

10th - exit iteration – misprediction (since branch is taken)

Disadvantage:

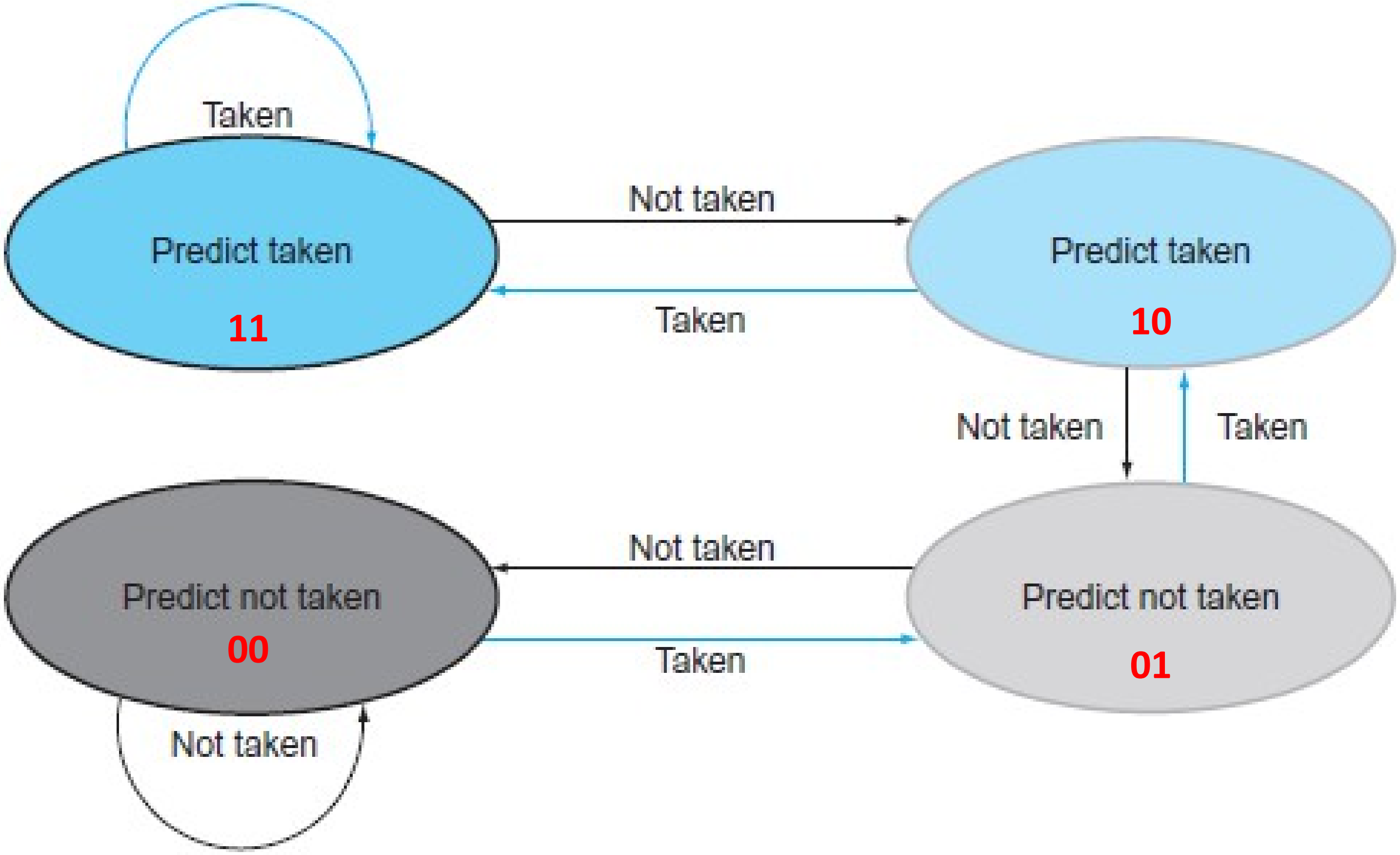
2 Mispredictions out of 10. so 80% accuracy is achieved

2-bit scheme

- A prediction must be wrong twice before the prediction is changed.
- A branch prediction buffer has 2 bits to store the history.

Bits indicating branch taken or not taken	Meaning
00	Strongly Branch not taken
01	Weakly branch not taken
10	Weakly branch taken
11	Strongly branch taken

Finite state diagram



Exception

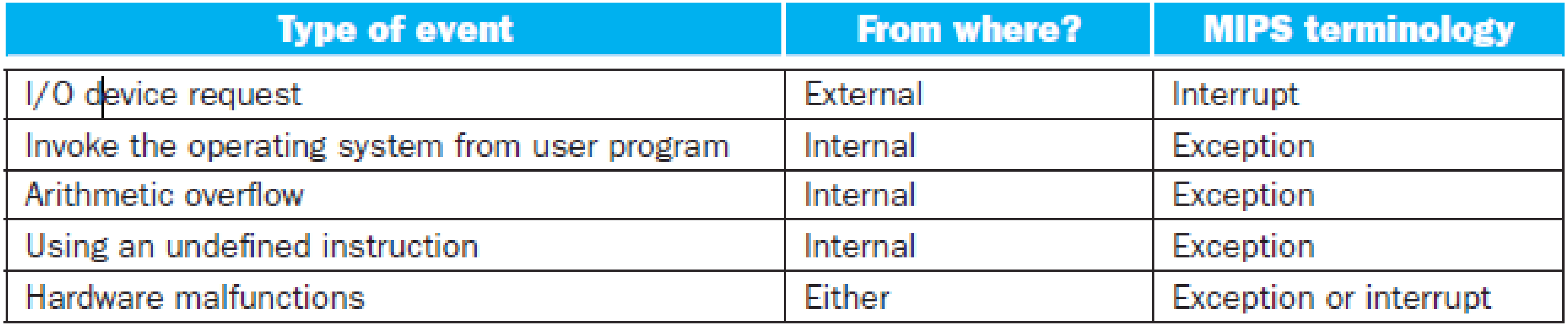
Exception

- **Exception –** (interrupt) An unscheduled event that disrupts program execution; used to detect overflow.
- **Interrupt -** An exception that comes from outside of the processor. (Some architectures use the term *interrupt for all* exceptions.)
- **vectored interrupt –** An interrupt for which the address to which control is transferred is determined by the cause of the exception.

Difference between exception and interrupt

Exception	Interrupt
Unscheduled event	Unscheduled event
Invoked Internal (within processor)	Invoked External (outside processor)
Ex: Arithmetic overflow	Ex: IO device request

Exception



Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

What happens in a processor when exception occur?

1. Processor save the address of the offending instruction in the *exception program counter (EPC)*
2. Determine the cause of exception
3. Transfer control to the operating system at some specified address(based on the cause of exception)

How to find the cause of exception

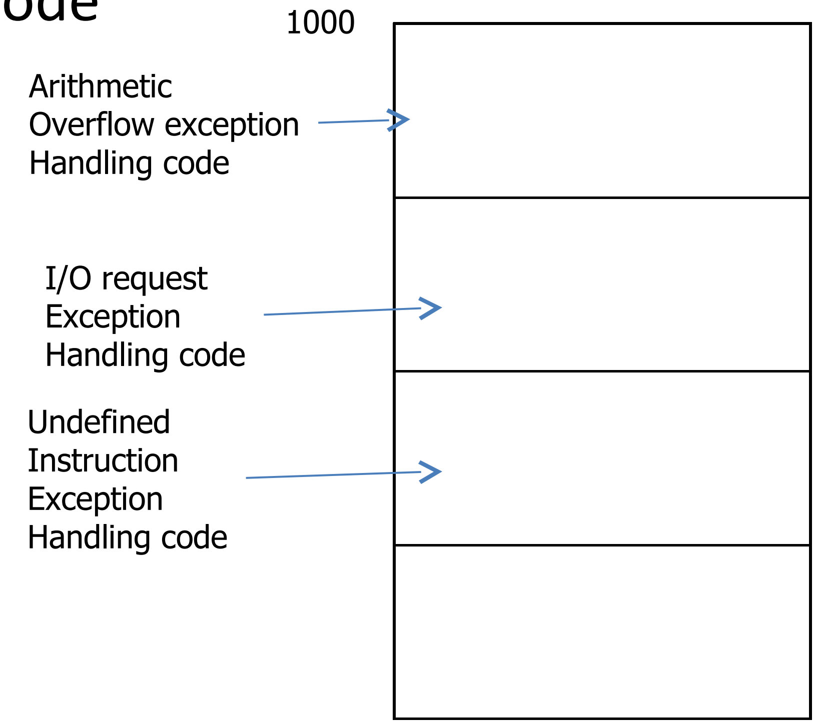
- Two methods to detect the cause of exception:
  - Use cause register
  - Vectored interrupts

Cause register

- Cause register – stores the reason for exception
- Ex: arithmetic overflow
- I/O device request

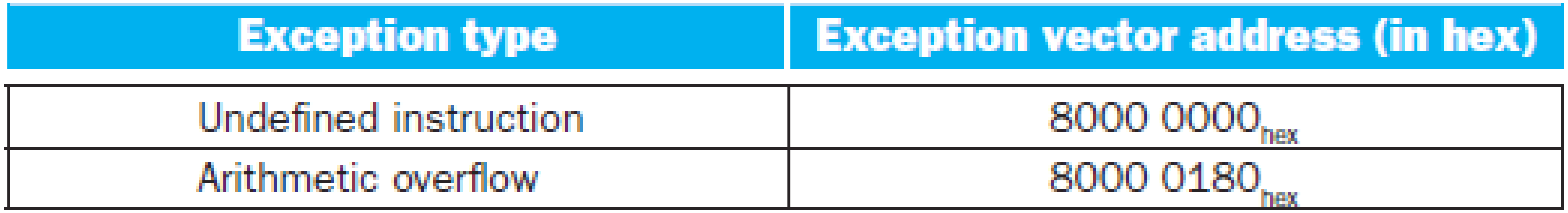
Exception handling using cause register

1. EPC = address of offending instruction
2. Cause register = cause of exception
3. Single entry point for exception handling code. (say 1000)
   - Then decode the cause register to move to the specific exception handling code


1000
Arithmetic Overflow exception Handling code
I/O request Exception Handling code
Undefined Instruction Exception Handling code

2. Vectored Interrupt

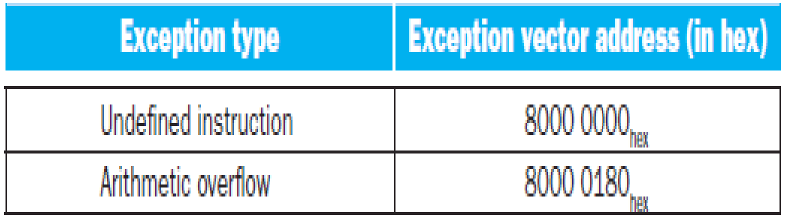
- An interrupt for which the address to which control is transferred is determined by the cause of the exception.



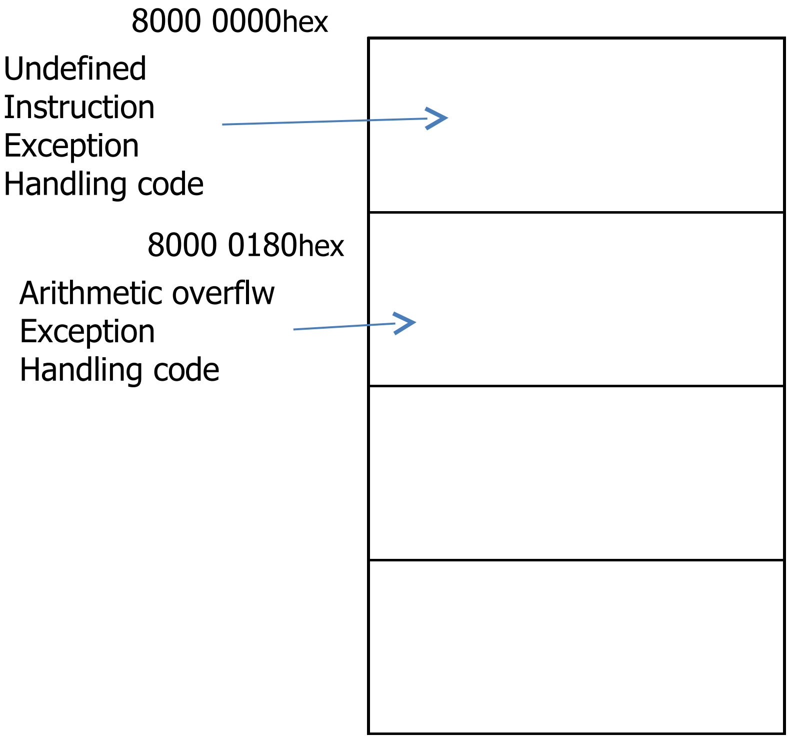
Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

Exception handling using Vectored Interrupt

1. EPC = address of offending instruction
2. Refer the vectored interrupt table to find the address of the specific exception handling code.



Exception type	Exception vector address (in hex)
Undefined instruction	8000 0000_{hex}
Arithmetic overflow	8000 0180_{hex}



8000 0000hex

Undefined
Instruction
Exception
Handling code

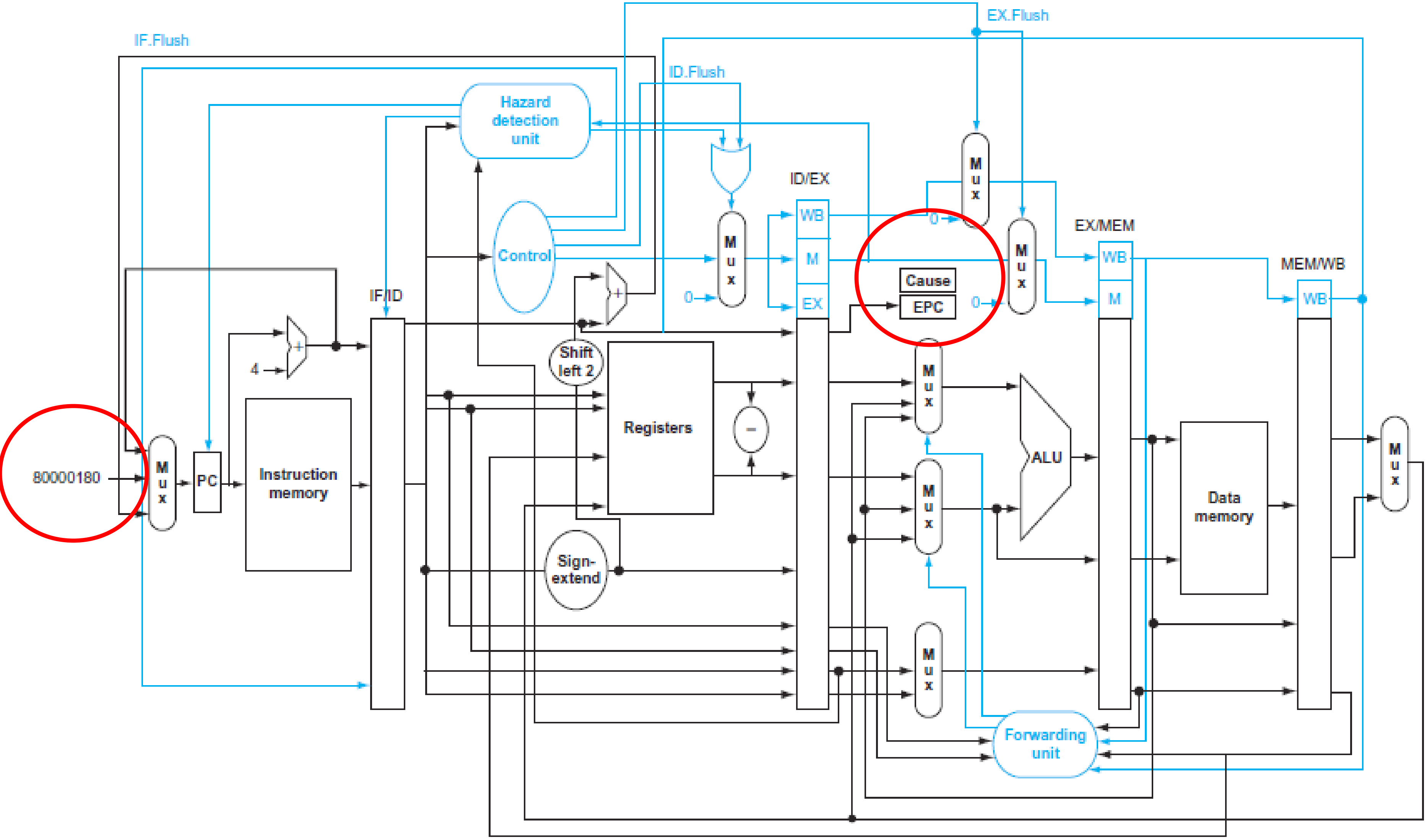
8000 0180hex

Arithmetic overflw
Exception
Handling code

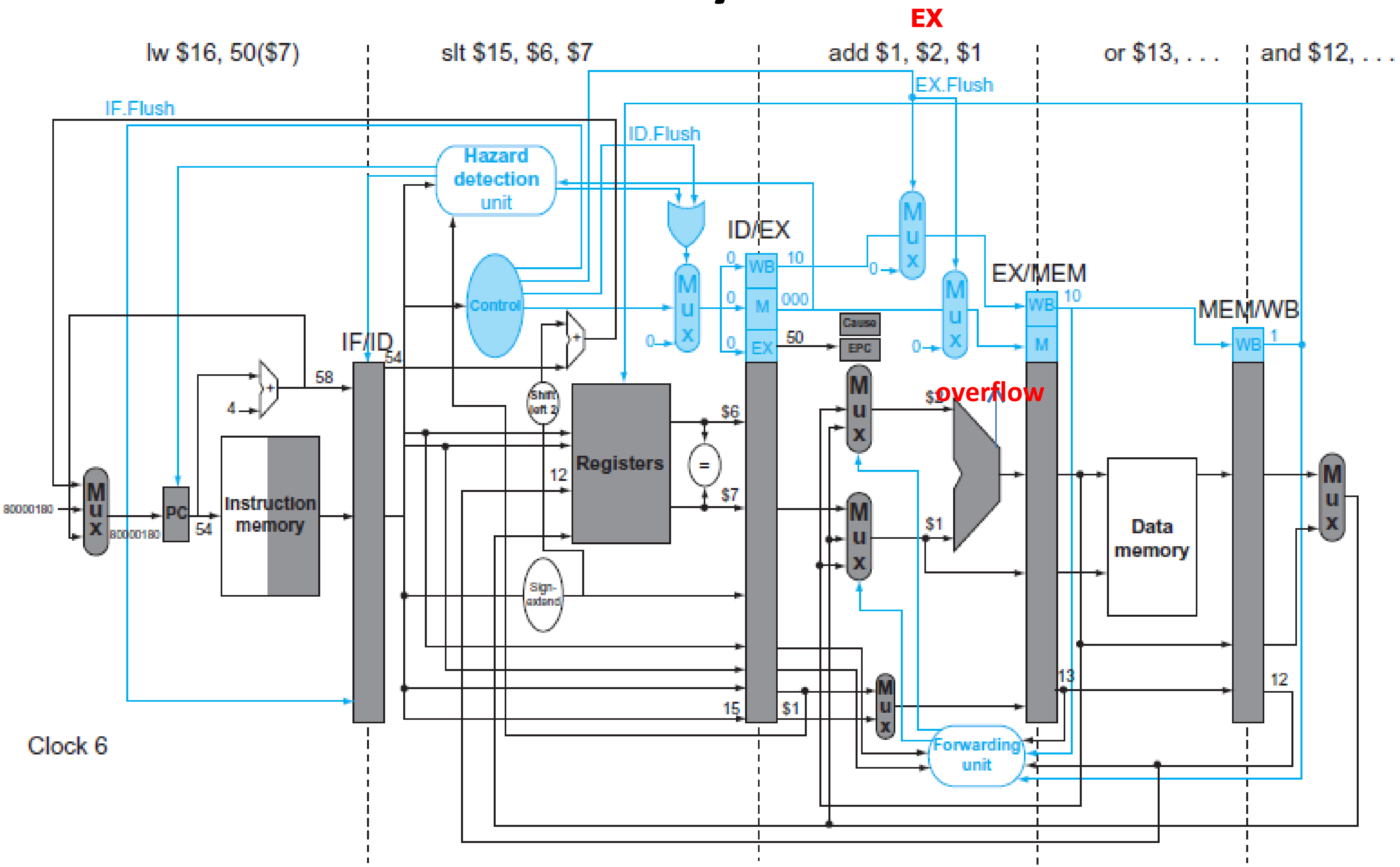
Implementing the exception system – using cause register

- Elements added to implement exception system:
  - Two registers
    - Cause register
    - EPC register (Exception Program Counter)
  - Single entry point – exception handling code starting address

Exception - Datapath

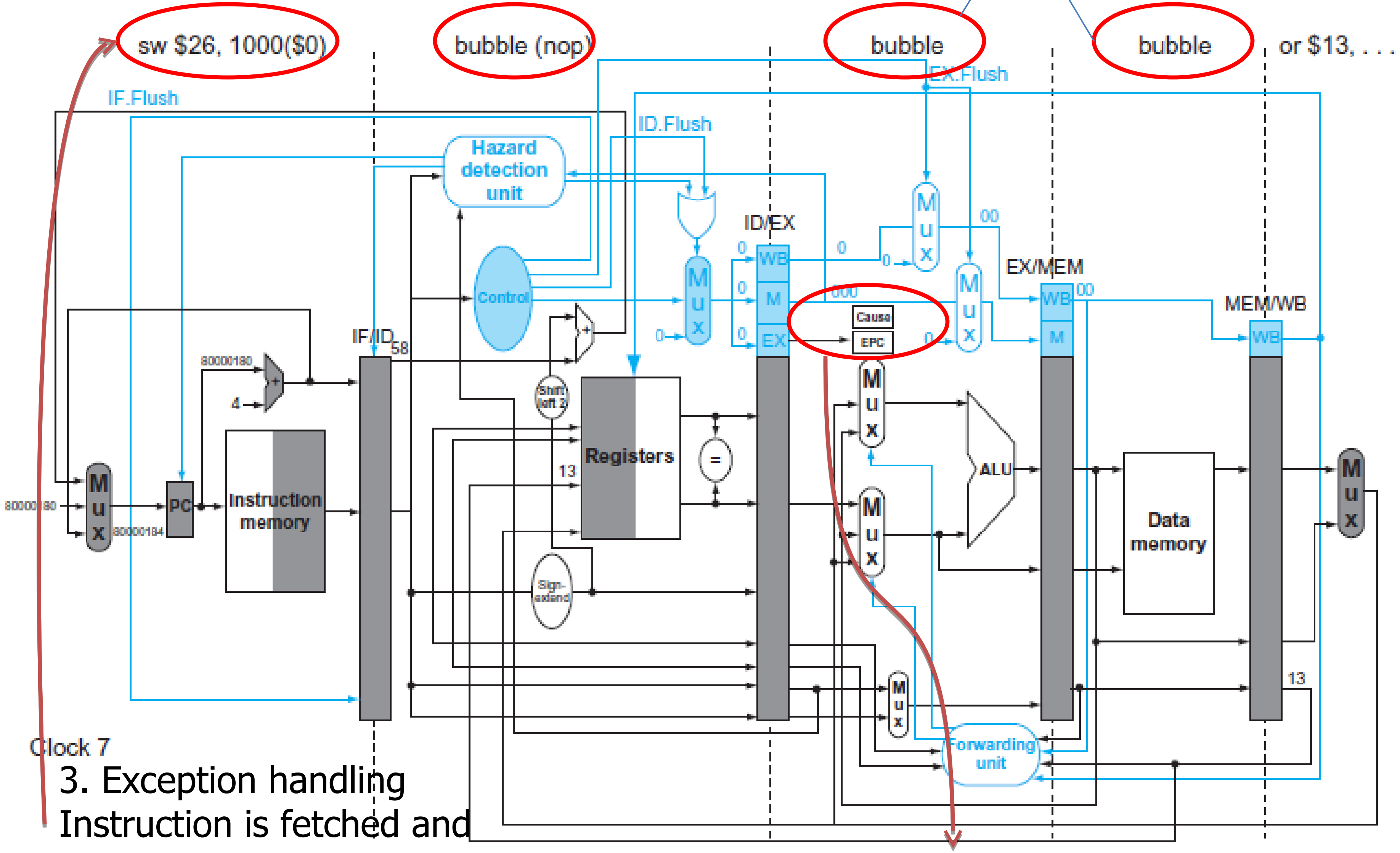


Arithmetic overflow exception detected clock cycle 6



Exception handling of Arthimetic Overflow – clock cycle 7

2. All the instruction after The offending instruction are flushed



3. Exception handling Instruction is fetched and started to execute

1. Stored EPC and cause register

Summary

- Pipeline
- Pipeline Hazards
- Pipeline Datapath and Control
- Data and Control Hazards

UNIT IV MEMORY AND I/O SYSTEMS

Memory hierarchy - Memory technologies – Cache basics – Measuring and improving cache performance – Virtual memory, TLBs

Reference:

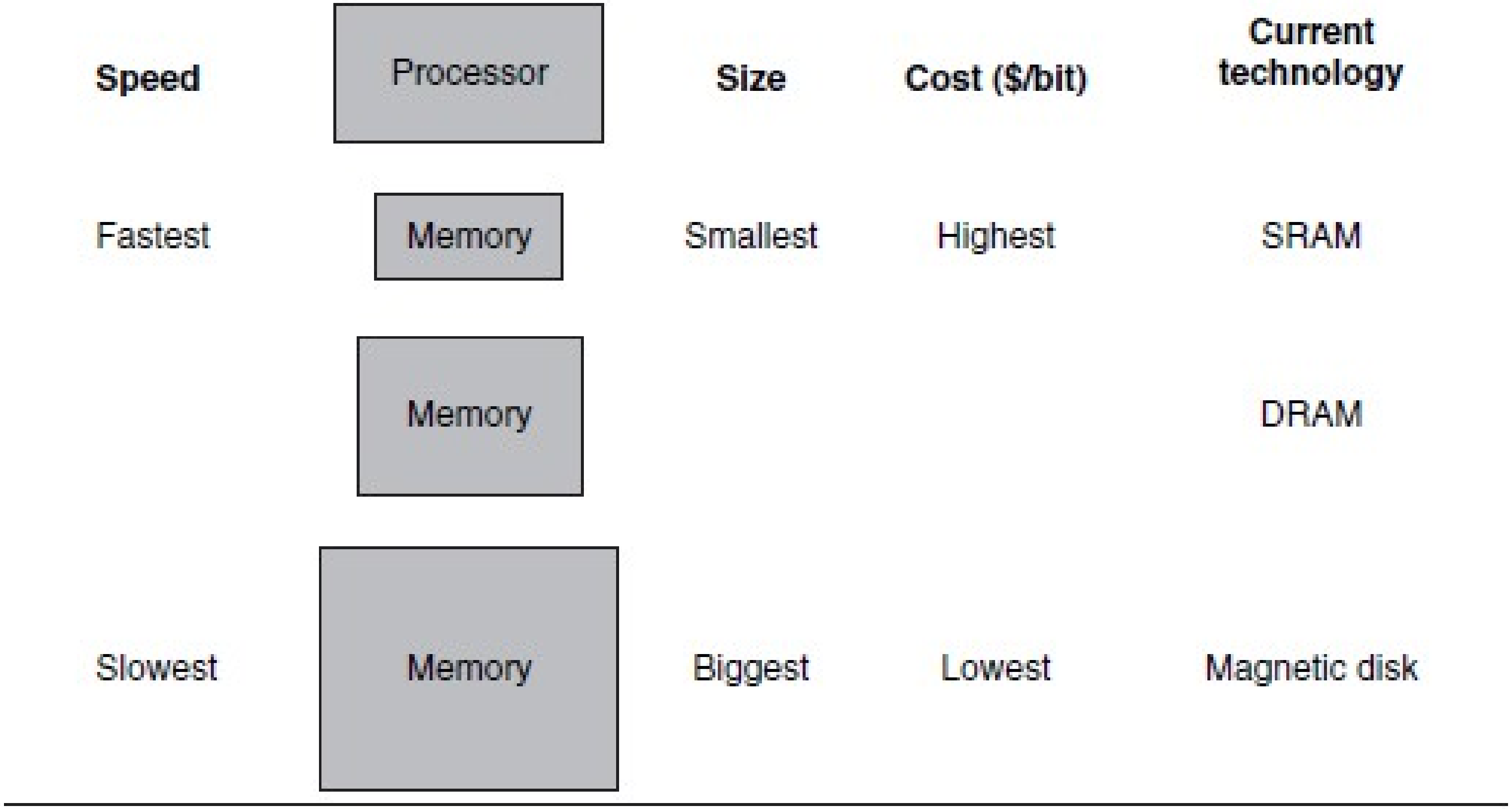
- Chapter 5 – Large and Fast: Exploiting Memory Hierarchy
- Book - David A. Patterson and John L. Hennessey, “Computer organization and design‟, Morgan Kauffman / Elsevier, Fifth edition, 2014.

Memory Hierarchy

Principle of Locality

- Temporal locality - (locality in time):
  - ✓ if an item is referenced, it will tend to be referenced again soon.
  - Spatial locality - (locality in space):
  - ✓ if an item is referenced, items whose addresses are close by will tend to be referenced soon

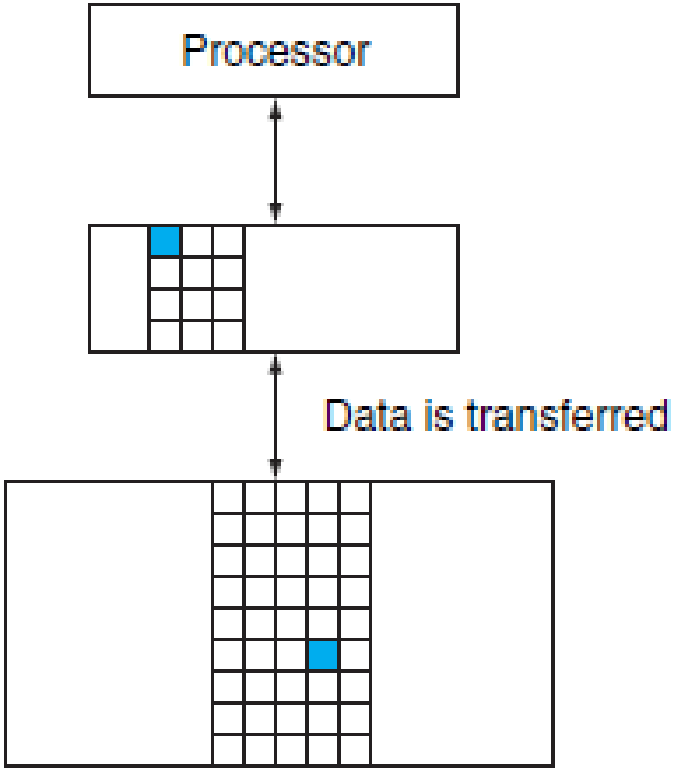
Memory Hierarchy



Speed	Processor	Size	Cost ($/bit)	Current technology
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic disk

Hit and Miss

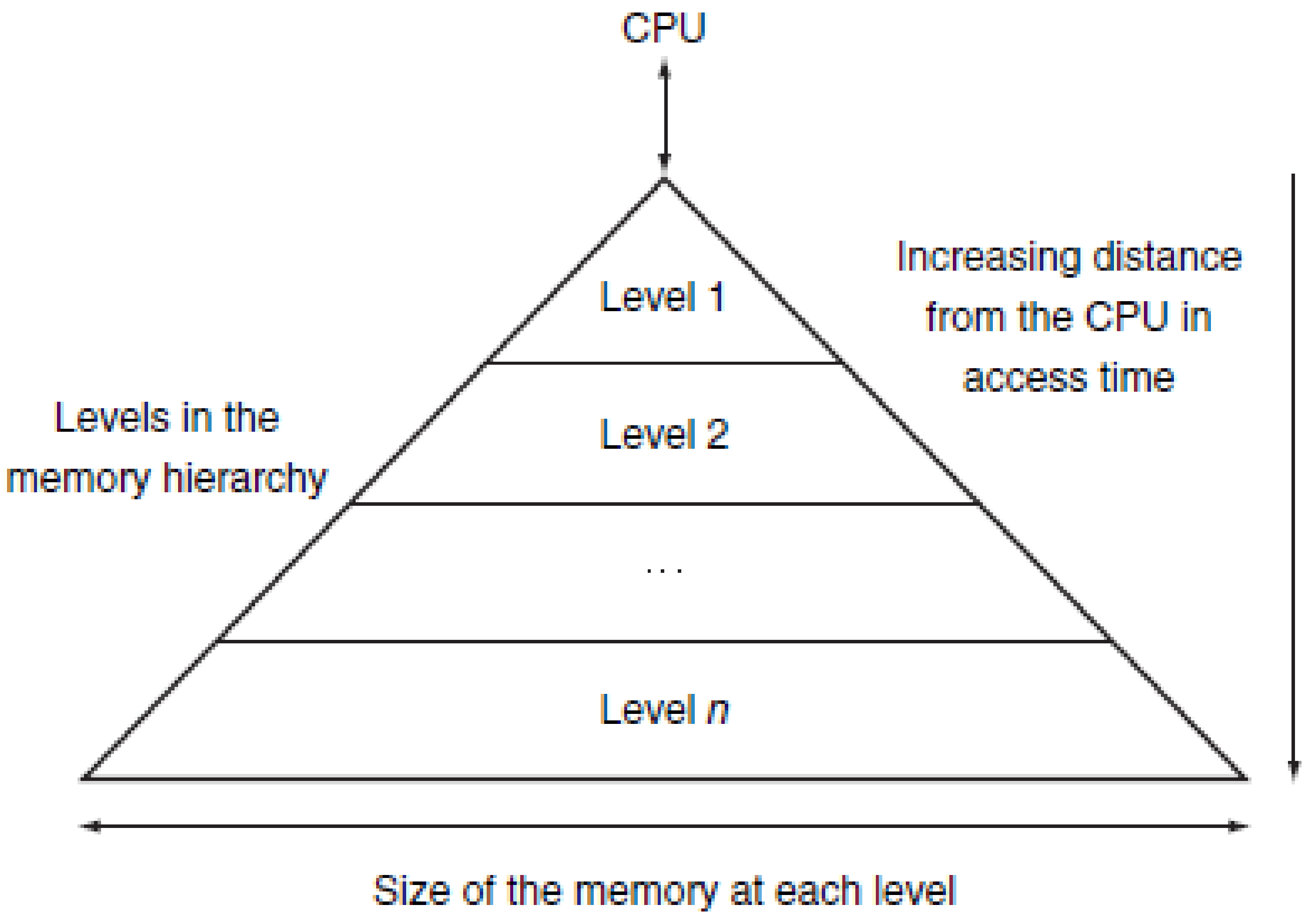
- *two adjacent* levels – called, *upper* (closer to CPU) and *lower* (farther from CPU)
- Terminology:
  - *block*: minimum unit of data to move between levels
  - *hit*: data requested is in upper level
  - *miss*: data requested is not in upper level



Memory Access

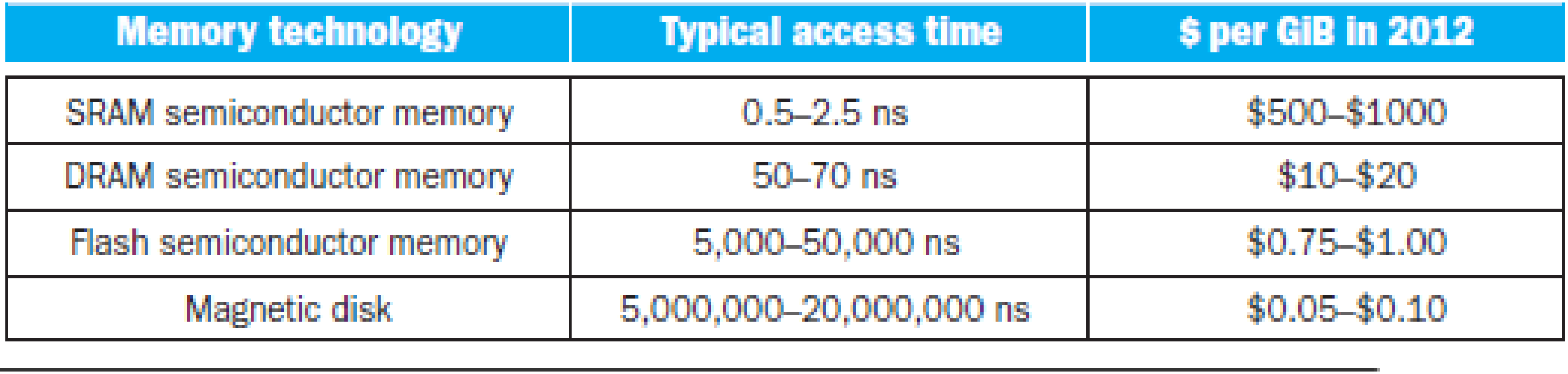
- **hit rate** – Hit memory access / Total memory access
- **miss rate** – Miss memory access / Total memory access
- **Total memory access** = Hit memory access + Miss memory access
- **Miss rate** = 1 – hit rate
- **hit time –**time to determine if the access is indeed a hit + time to access and deliver the data from the upper level to the CPU
- ***miss penalty*****:** time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU

Memory hierarchy


CPU
Level 1
Level 2
. . .
Level *n*
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

Memory Technology

- SRAM
- DRAM
- Flash memory
- Disk memory



Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

SRAM

- Static random access memory
- Volatile – loses data when there is no power
- Used to make caches
- Very fast
- Very costly

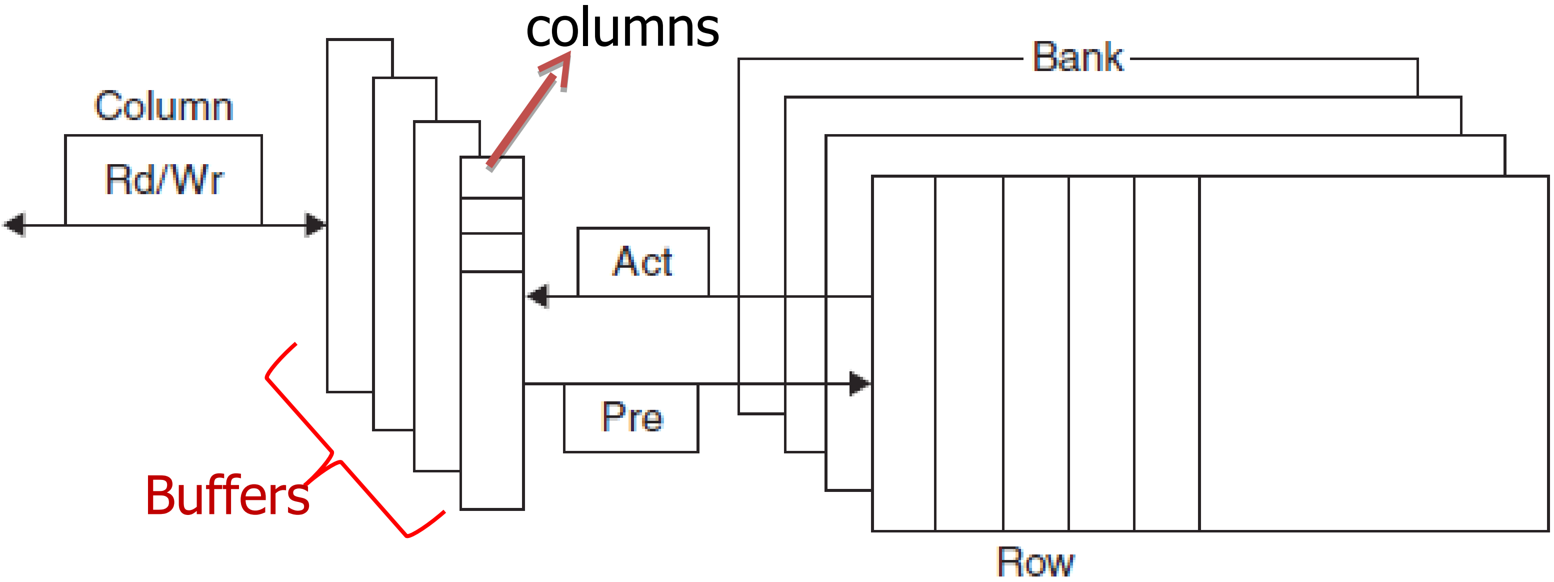
DRAM

- Dynamic random access memory
- Volatile – loses data when there is no power
- Used to make main memory
- Slower than SRAM
- Cheaper than SRAM

Difference between SRAM and DRAM

S.No	SRAM	DRAM
1	Static Random Access Memory	Dynamic Random Access Memory
2	Volatile	Volatile
3	Faster	Slower
4	Costlier	Cheaper than SRAM
5	Less denser	Denser than SRAM
6	Used as Cache memory	Used as Main memory

DRAM Internal Organisation



Each DRAM – has 4 banks
Each bank – has many rows
Commands :

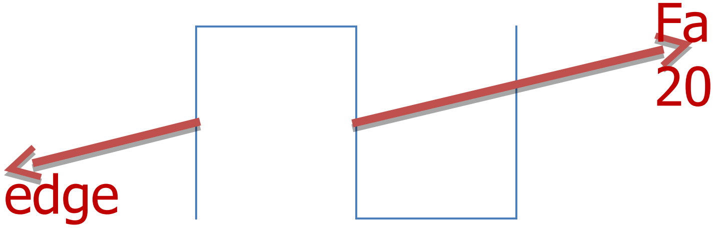
Pre – Precharge command – opens/close a bank
Act – Activate command – transfer row from bank to buffer

Each buffer – has many columns
Command:

Rd – Read command – read data from buffer column
Wr – Write command – write data to buffer column

DDR - DRAM

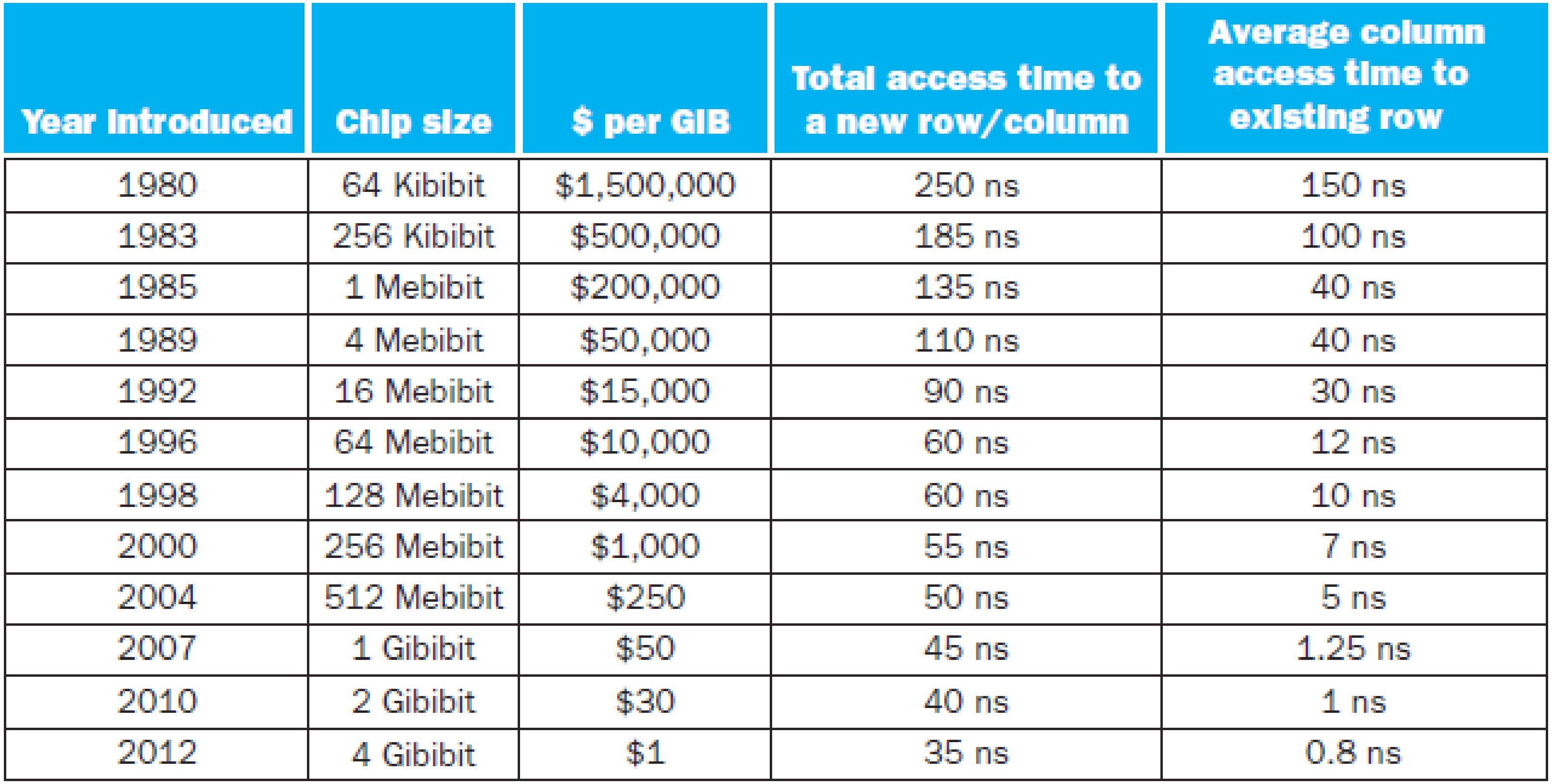
- DDR – Double Data Rate
  - Where data is transferred during both rising and falling edge of the clock. (20 + 20 = 40Mbps)



Falling edge
20Mbps

Rising edge
20Mbps

DRAM Growth

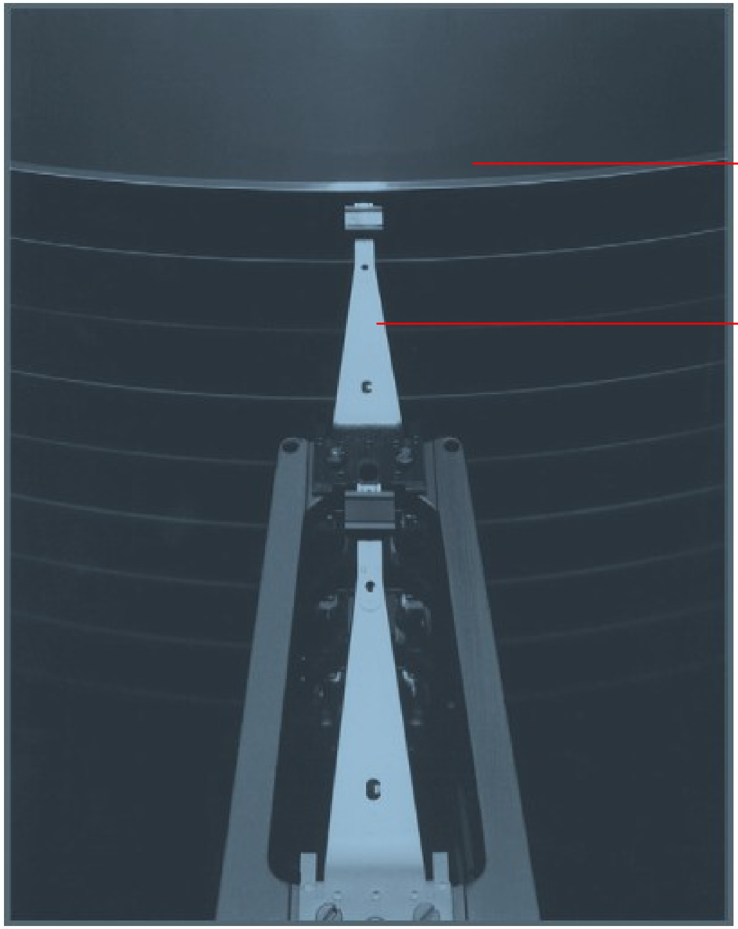


Year Introduced	Chip size	$ per GIB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

Flash memory

- EEPROM - Electrically erasable programmable read-only memory
- **Disadvantage:** Writes can wear out flash memory bits. (10,000 writes)
- **Solution: (wear levelling)**
  - Controller - spread the writes by remapping blocks that have been written many times to less trodden blocks.

Disk Memory



Platter

Read Write Head

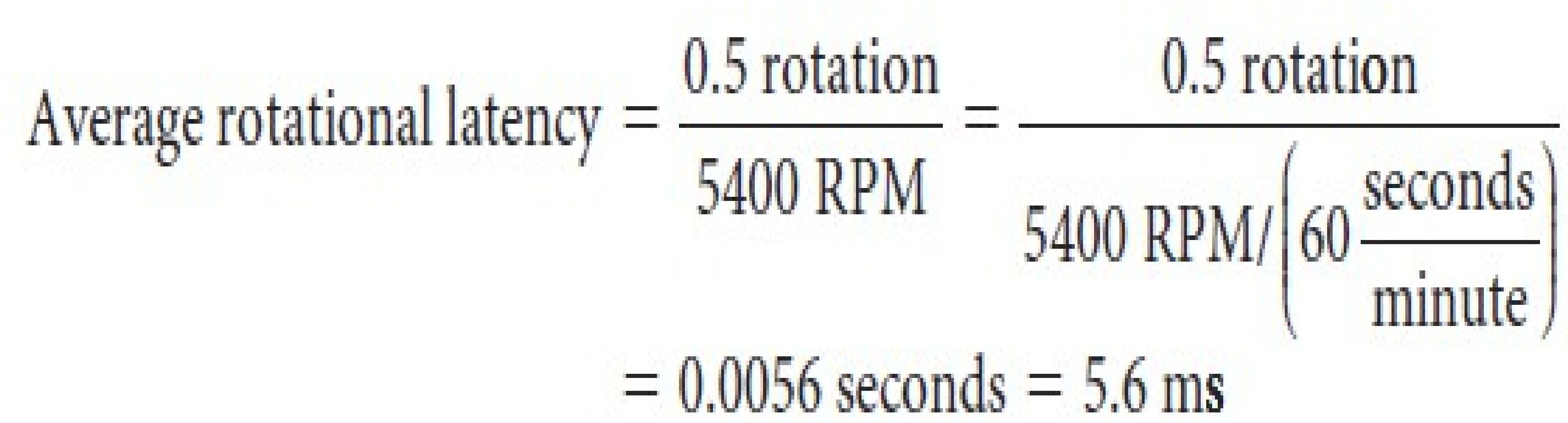
Disk Memory(2)

- **Read-write head** - To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a *read-write head is located just above* each surface.
- **Track -** One of thousands of concentric circles that makes up the surface of a magnetic disk.
- **Sector -** One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.
- **seek -** The process of positioning a read/write head over the proper track on a disk

3 computational times

- **Seek time -** the time to move the head to the desired track.
- **Rotational latency** – Also called rotational delay. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.
- **Transfer time** - is the time to transfer a block of bits.
  - The transfer time is a function of the sector size, the rotation speed, and the recording density of a track.
  - Transfer rates in 2012 were between 100 and 200 MB/sec.

Problem



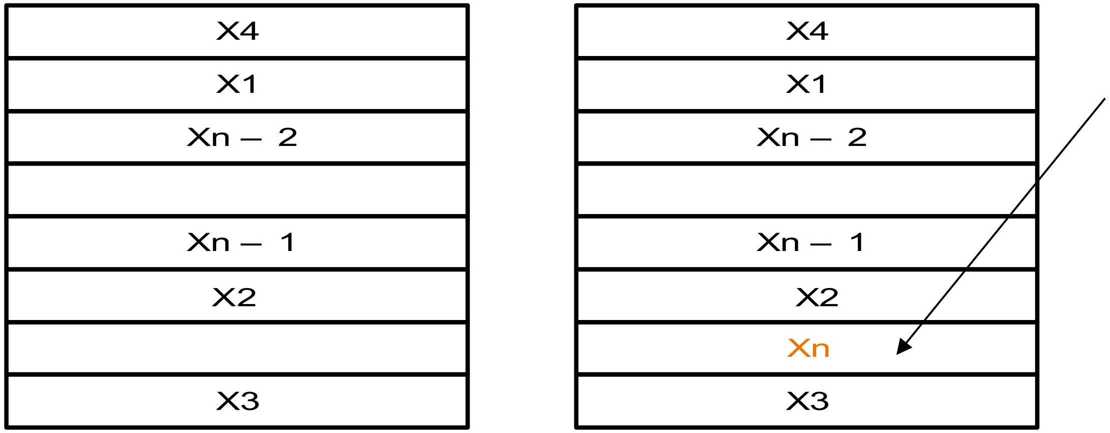
$$\text{Average rotational latency} = \frac{0.5\ \text{rotation}}{5400\ \text{RPM}} = \frac{0.5\ \text{rotation}}{5400\ \text{RPM}/\left(60\,\dfrac{\text{seconds}}{\text{minute}}\right)}$$

$$= 0.0056\ \text{seconds} = 5.6\ \text{ms}$$

Cache Basics

Caches

- By simple example
  - assume block size = one word of data



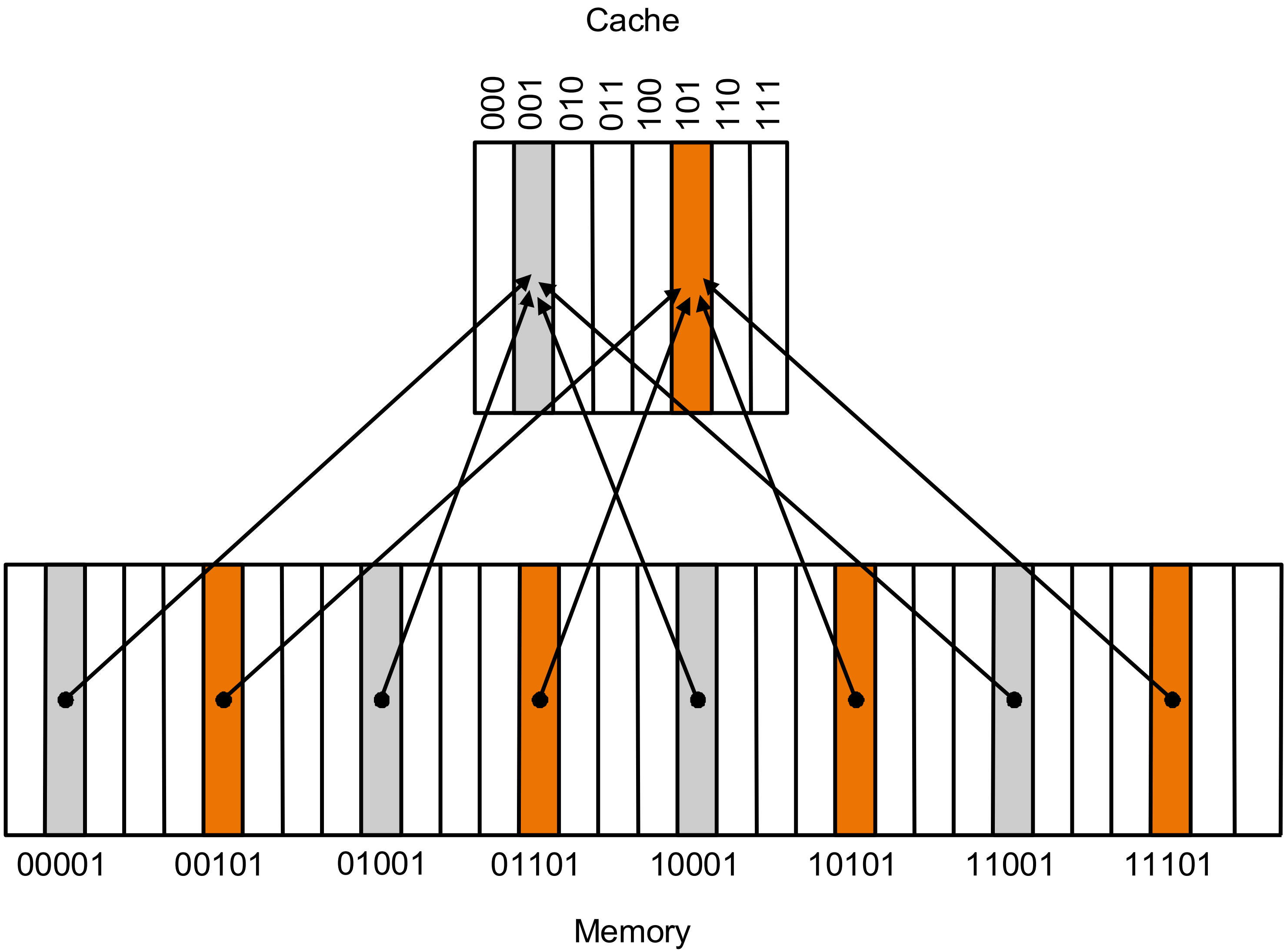
a. Before the reference to Xn

b. After the reference to Xn

Reference to X_n causes miss so it is fetched from memory

- Issues:
  - *how do we know if a data item is in the cache?*
  - *if it is, how do we find it?*
  - *if not, what do we do?*
- Solution depends on *cache addressing scheme*…

Direct Mapped Cache



Direct Mapped Cache

- Addressing scheme in *direct mapped* cache:
  - cache block address = memory block address *mod* (no of blocks in cache)
  - 3 fields
    - Index
    - Tag
    - valid

Accessing Cache

- Example:

(0) Initial state:

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

(1) Address referred 10110 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(2) Address referred 11010 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(3) Address referred 10110 (*hit*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

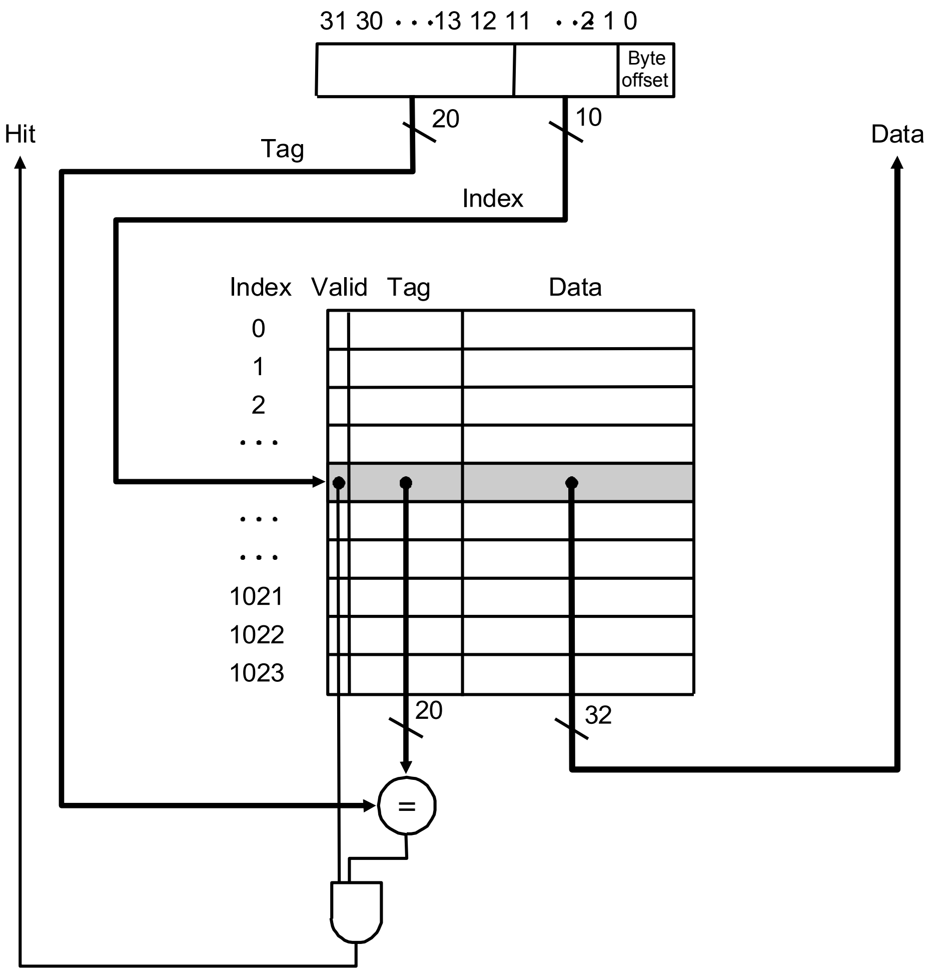
to CPU

(4) Address referred 10010 (*miss*): replace

Index	V	Tag	Data
000	N		
001	N		
010	Y	10	Mem(10010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

Direct Mapped Cache

Address showing bit positions



Cache with 1024 1-word blocks:
byte offset **(least 2 significant bits)**
is ignored and
next 10 bits used to index into cache

What kind of locality are we taking advantage of?

Formula

- The size of the block above was one word
- For the following situation:
- 32-bit addresses
- A direct-mapped cache
- The cache size is 2^n *blocks, so n bits are used for the index*
- The block size is 2^m *words (2^{m+2} bytes), so m bits are used for the word within* the block, and two bits are used for the byte part of the address
- **size of the tag field = 32 - (*n + m + 2)***
- The total number of bits in a direct-mapped cache
- **For one block = *block size(data) + tag size + valid field size***
- The total number of bits in a direct-mapped cache is
- **2^n ** (block size + tag size + valid field size)***

Example Problem

- *How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word block size, assuming a 32-bit address?*

- Cache data = 16 KB = 2^{14} bytes = 2^{12} words =

  = 2^{12} words / 4 words = 2^{10} blocks
- Each cache entry size = block data bits + tag bits + valid bit

  = (4*32) + (32 – 10 – 2 - 2) + 1 = 128 + 18 + 1 = 147 bits
- Therefore, Entire cache size = 2^{10} × 147 bits = 2^{10} × (147) bits = 147 Kbits
  - Total cache size -> 147Kbits / 8 = 18.4 Kbytes
  - Actual cache size(only data) -> 16Kbytes
  - total cache size/actual cache data = 18.4/16 = 1.15
  - So 15% increase in the cache size to include tag and valid bits.

Example problem

- Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?
- Use the formula:

  (Block address) modulo (Number of blocks in the cache)

  - Block address = byte address/ bytes per block
  - = 1200 / 16 = 75
  - 75 modulo 64 = 11th block in cache

Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 1-word block size, assuming a 32-bit address?*

- Cache data = 128 KB = 2^{17} bytes = 2^{15} words = 2^{15} blocks
- Cache entry size = block data bits + tag bits + valid bit

  = 32 + (32 – 15 – 2) + 1 = 48 bits
- Therefore, cache size = $2^{15} \times 48$ bits = $2^{15} \times (1.5 \times 32)$ bits = 1.5×2^{20} bits = 1.5 Mbits
  - data bits in cache = 128 KB × 8 = 1 Mbits
  - total cache size/actual cache data = 1.5

Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 4-word block size, assuming a 32-bit address?*

- Cache size = 128 KB = 2^{17} bytes = 2^{15} words = 2^{13} blocks
- Cache entry size = block data bits + tag bits + valid bit

  = 128 + (32 – 13 – 2 – 2) + 1 = 144 bits
- Therefore, cache size = $2^{13} \times 144$ bits = $2^{13} \times (1.25 \times 128)$ bits = 1.25×2^{20} bits = 1.25 Mbits
  - data bits in cache = 128 KB × 8 = 1 Mbits
  - total cache size/actual cache data = 1.25

Cache Read Hit/Miss

- *Cache read hit*: no action needed
- *Instruction cache read miss*:
  1. *Send original PC value* (*current PC – 4*, as PC has already been incremented in first step of instruction cycle) to memory
  2. Instruct main memory to perform read and wait for memory to complete access – *stall* on read
  3. After read completes *write cache* entry
  4. *Restart* instruction execution at first step to refetch instruction
- *Data cache read miss*:
  - Similar to instruction cache miss
  - To reduce data miss penalty allow processor to execute instructions while waiting for the read to complete *until* the word is required – *stall on use* (why won't this work for instruction misses?)

Cache Write Hit/Miss

- *Write-through* scheme
  - on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
  - on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
  - Write-through is slow because of always required memory write
    - performance is improved with a *write buffer* where words are stored while waiting to be written to memory – processor can continue execution until write buffer is full
    - when a word in the write buffer completes writing into main that buffer slot is freed and becomes available for future writes
    - DEC 3100 write buffer has 4 words
- *Write-back* scheme
  - write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
  - more efficient than write-through, more complex to implement

Measuring and Improving cache performance

Measuring Cache Performance

- Simplified model assuming equal read and write miss penalties:
  - **CPU time = (execution cycles + memory stall cycles) × cycle time**
  - Memory stall cycles = reads stall cycles + write stall cycles
  - Read stall cycles = reads/program + read miss rate + read miss penalty
  - Write stall cycles = writes/program + write miss rate + write miss penalty

  - **memory stall cycles = memory accesses/program × miss rate × miss penalty**
  - **memory stall cycles = instructions/program × miss/instruction × miss penalty**

Where, memory accesses = read memory access + write memory access

miss rate = read miss rate + write miss rate

miss penalty = read miss penalty = write miss penalty

Example Problems

- Assume for a given machine and program:
  - instruction cache miss rate 2%
  - data cache miss rate 4%
  - miss penalty always 100 cycles
  - CPI of 2 without memory stalls
  - frequency of load/stores 36% of instructions

1. *How much faster is a machine with a perfect cache that never misses?*
2. *What happens if we speed up the machine by reducing its CPI to 1 without changing the clock rate?*

Solution - 1

- Assume instruction count = I
- **memory stall cycles = instructions/program × miss/instruction × miss penalty**
- Instruction miss cycles = I × 2% × 100 = 2.0 × I
- Data miss cycles = I × 36% × 4% × 100 = 1.44 × I
- **total memory-stall cycles = Instruction miss cycles + Data miss cycles** = 2 × I + 1.44 × I = 3.44 × I
- **CPI $_{\textbf{memory stalls}}$ = Basic CPI + memory stall CPI**
- = 2 + 3.44 = 5.44
- **CPU time = IC x CPI x clock cycles**
- **CPU time with stalls / CPU time with perfect cache**

  = I x 5.44xclock cycles/ I x 2 x clock cycles = 5.44 / 2 = 2.72
- Performance with a perfect cache is better by a factor of 2.72

Solution - 2

- CPI without stall = 1
- **CPI with stall = Basic CPI + memory stall CPI**
- = 1 + 3.44 = 4.44
- **CPU time with stalls / CPU time with perfect cache**

  = CPI with stall / CPI without stall

  = 4.44/1
- Performance with a perfect cache is better by a factor of 4.44
- Conclusion: with higher CPI cache misses "hurt more" than with lower CPI

Average Memory Access Time

- AMAT = Time for a hit + Miss rate x Miss penalty

AMAT – Problem

- Given:
  - Clock cycle time = 1ns
  - Miss penalty = 20 clock cycles
  - Miss rate = 0.05 misses per instruction
  - Cache access(hit) time = 1 clock cycle

Solution

- AMAT = Time for a hit + Miss rate x Miss penalty

  = 1 + 0.05 x 20

  = 1 + 1

  = 2 clock cycles

Or clock cycle time = 2 x 1ns = 2ns

Cache Read Hit/Miss

- *Cache read hit*: no action needed
- *Instruction cache read miss*:

1. *Send original PC value* to memory
2. Instruct main memory to perform read and wait for memory to complete access – *stall* on read
3. After read completes *write cache* entry
4. *Restart* instruction execution at first step to refetch instruction

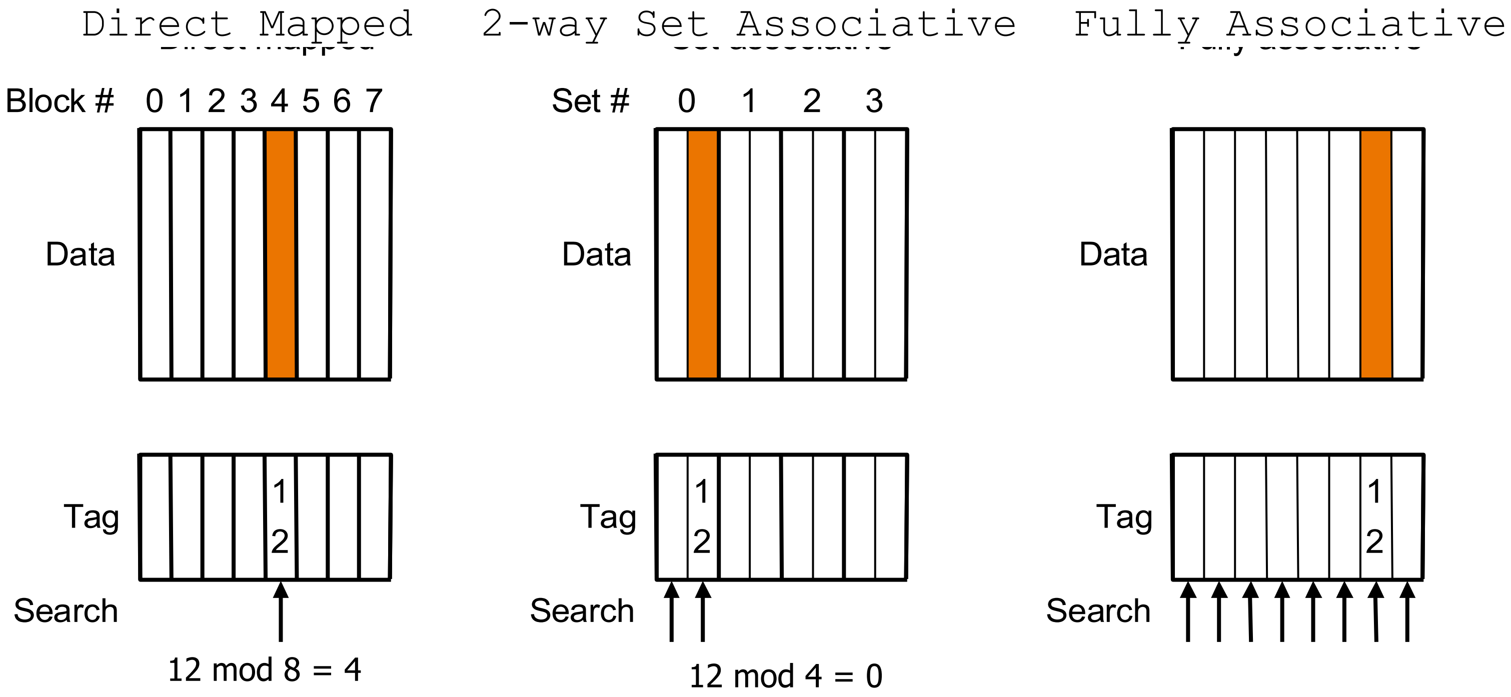
Cache Write Hit/Miss

- ***Write-through*** **scheme – write data both in cache and main memory**
  - on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
  - on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
  - Write-through is **slow** because of always required memory write
    - performance is improved with a ***write buffer***
- ***Write-back*** **scheme**
  - write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
  - more efficient than write-through, more complex to implement

Decreasing Miss Rates with Associative Block Placment

- *Direct mapped*: one *unique* cache location for each memory block
  - cache block address = memory block address *mod* cache size
- *Fully associative*: each memory block can locate *anywhere* in cache
  - *all* cache entries are searched (in parallel) to locate block
- *Set associative*: each memory block can place in a *unique set* of cache locations – if the set is of size n it is n-way set-associative
  - cache set address = memory block address *mod* number of sets in cache
  - all cache entries in the corresponding set are searched (*in parallel*) to locate block
- Increasing degree of associativity
  - *reduces miss rate*
  - *increases hit time* because of the parallel search and then fetch

Decreasing Miss Rates with Associative Block Placment

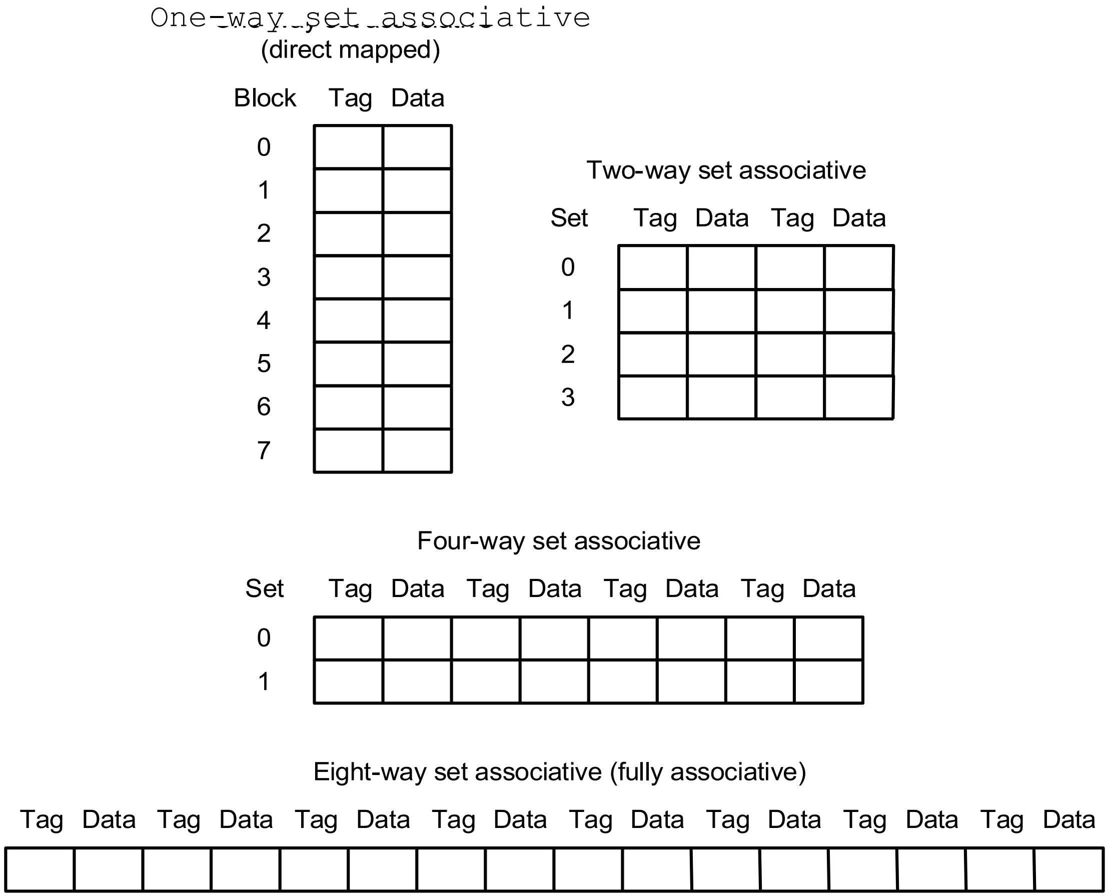


Location of a memory block with address 12 in a cache with 8 blocks with different degrees of associativity

Adv and Disadv of mapping

- Direct mapped :
  - Adv – easy to search
  - Disadv – only one block for multiple data
- Set-Associativity mapped :
  - Adv – more than one block for multiple data
  - Disadv – search more than one block
- Associativity mapped :
  - Adv – all blocks available for data
  - Disadv – search all the blocks

Decreasing Miss Rates with Associative Block Placment



One-way set associative
(direct mapped)

Block Tag Data

0
1
2
3
4
5
6
7

Two-way set associative

Set Tag Data Tag Data

0
1
2
3

Four-way set associative

Set Tag Data Tag Data Tag Data Tag Data

0
1

Eight-way set associative (fully associative)

Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data

Configurations of an 8-block cache with different degrees of associativity

Example Problems

- *Find the number of misses for a cache with four 1-word blocks given the following sequence of memory block accesses:*

  0, 8, 0, 6, 8,

  for each of the following cache configurations

  1. *direct mapped*
  2. *2-way set associative (use LRU replacement policy)*
  3. *fully associative*

- Note about LRU replacement
  - in a 2-way set associative cache LRU replacement can be implemented with one bit at each set whose value indicates the mostly recently referenced block

Solution

- 1 (direct-mapped)

Block address	Cache block
0	0 (= 0 *mod* 4)
6	2 (= 6 *mod* 4)
8	0 (= 8 *mod* 4)

Block address translation in direct-mapped cache

Address of memory	Hit or	Contents of cache blocks after reference			
block accessed	**miss**	**0**	**1**	**2**	**3**
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Cache contents after each reference – red indicates new entry added

- 5 misses

Solution (cont.)

- 2 (two-way set-associative)

Block address	Cache set
0	0 (= 0 *mod* 2)
6	0 (= 6 *mod* 2)
8	0 (= 8 *mod* 2)

Block address translation in a two-way set-associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Cache contents after each reference – red indicates new entry added

- Four misses

Solution (cont.)

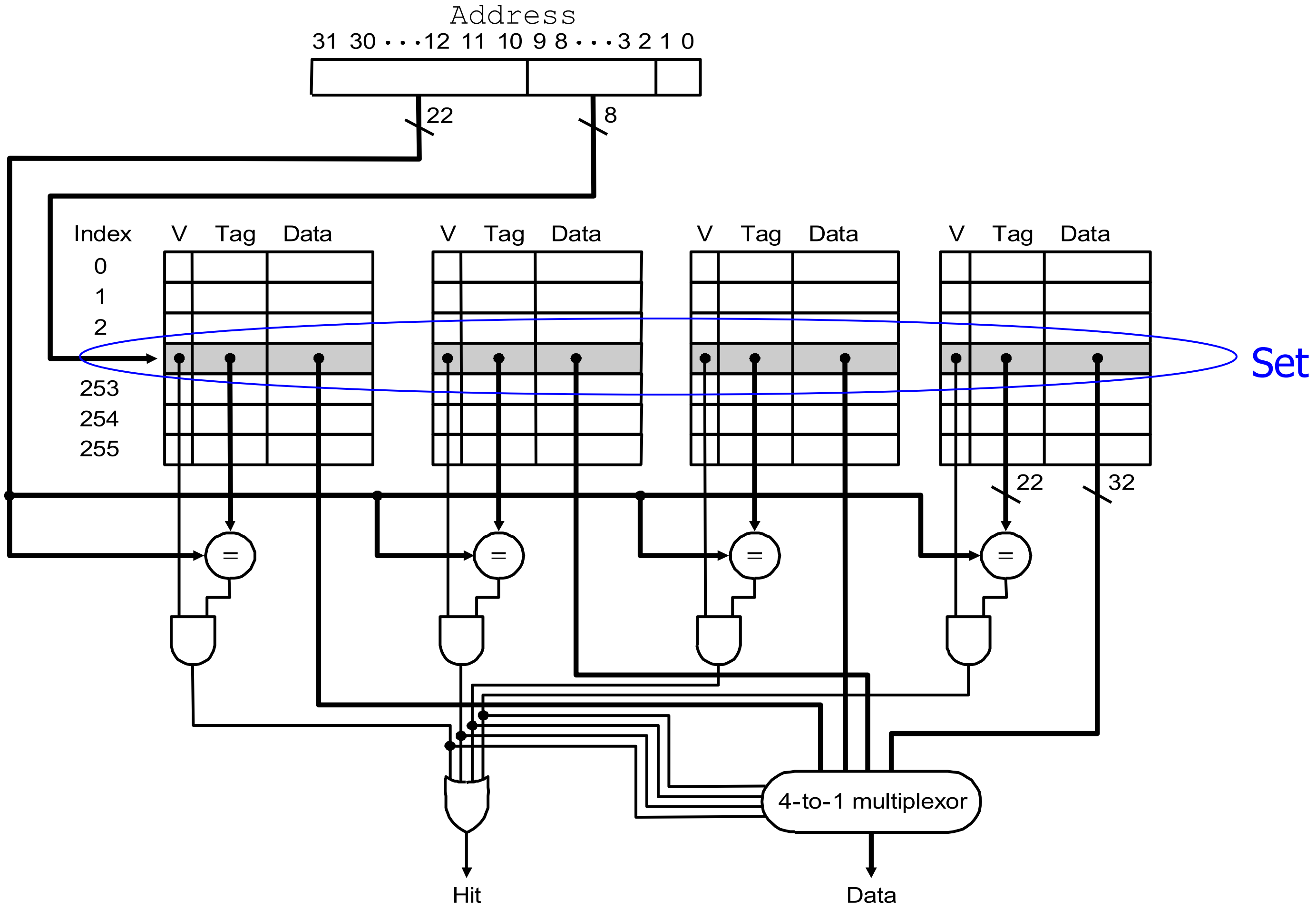
- 3 (fully associative)

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Cache contents after each reference – red indicates new entry added

- 3 misses

Implementation of a 4-way - Set-Associative Cache



4-way set-associative cache with 4 comparators and one 4-to-1 multiplexor: size of cache is 1K blocks = 256 sets * 4-block set size

Decreasing Miss Penalty with Multilevel Caches

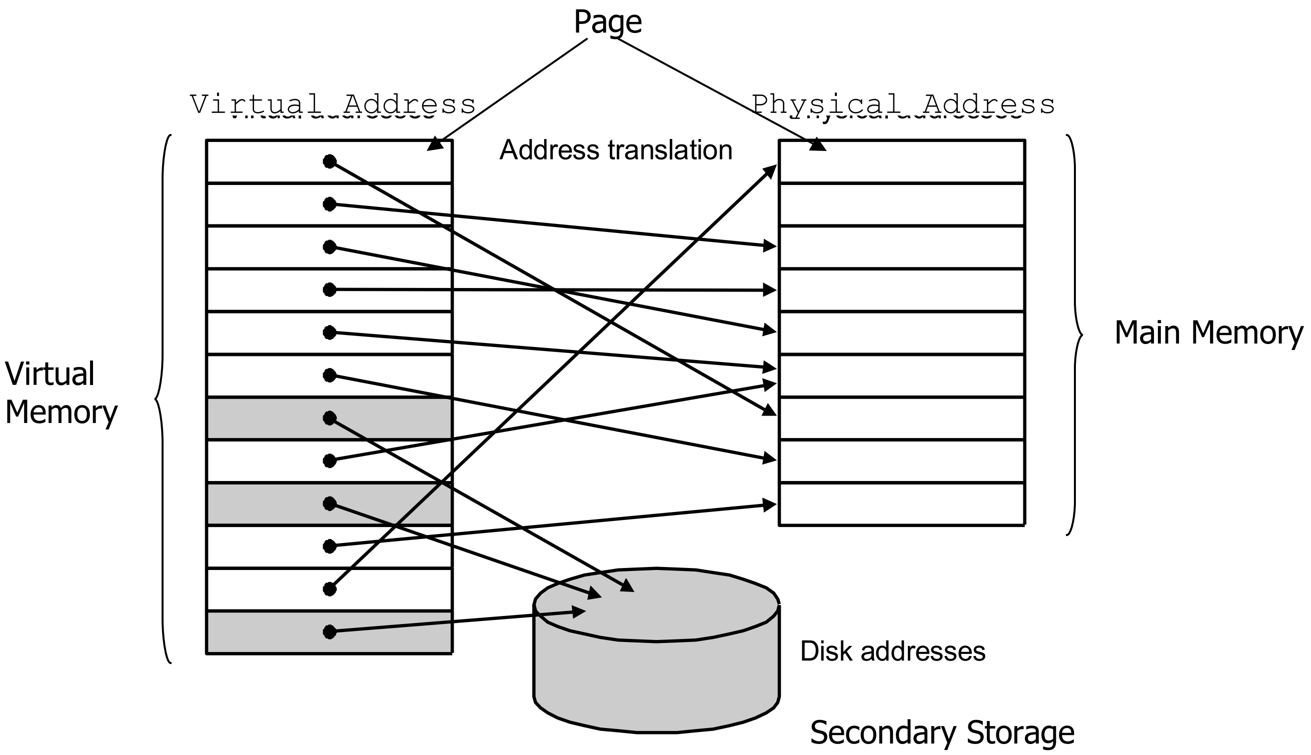
- Add a *second-level* cache
  - primary cache is on the same chip as the processor
  - use SRAMs to add a second-level cache, sometimes off-chip, *between main memory and the first-level cache*
  - if miss occurs in primary cache second-level cache is accessed
  - if data is found in second-level cache miss penalty is access time of second-level cache which is much less than main memory access time
  - if miss occurs again at second-level then main memory access is required and large miss penalty is incurred

Virtual Memory

Virtual Memory

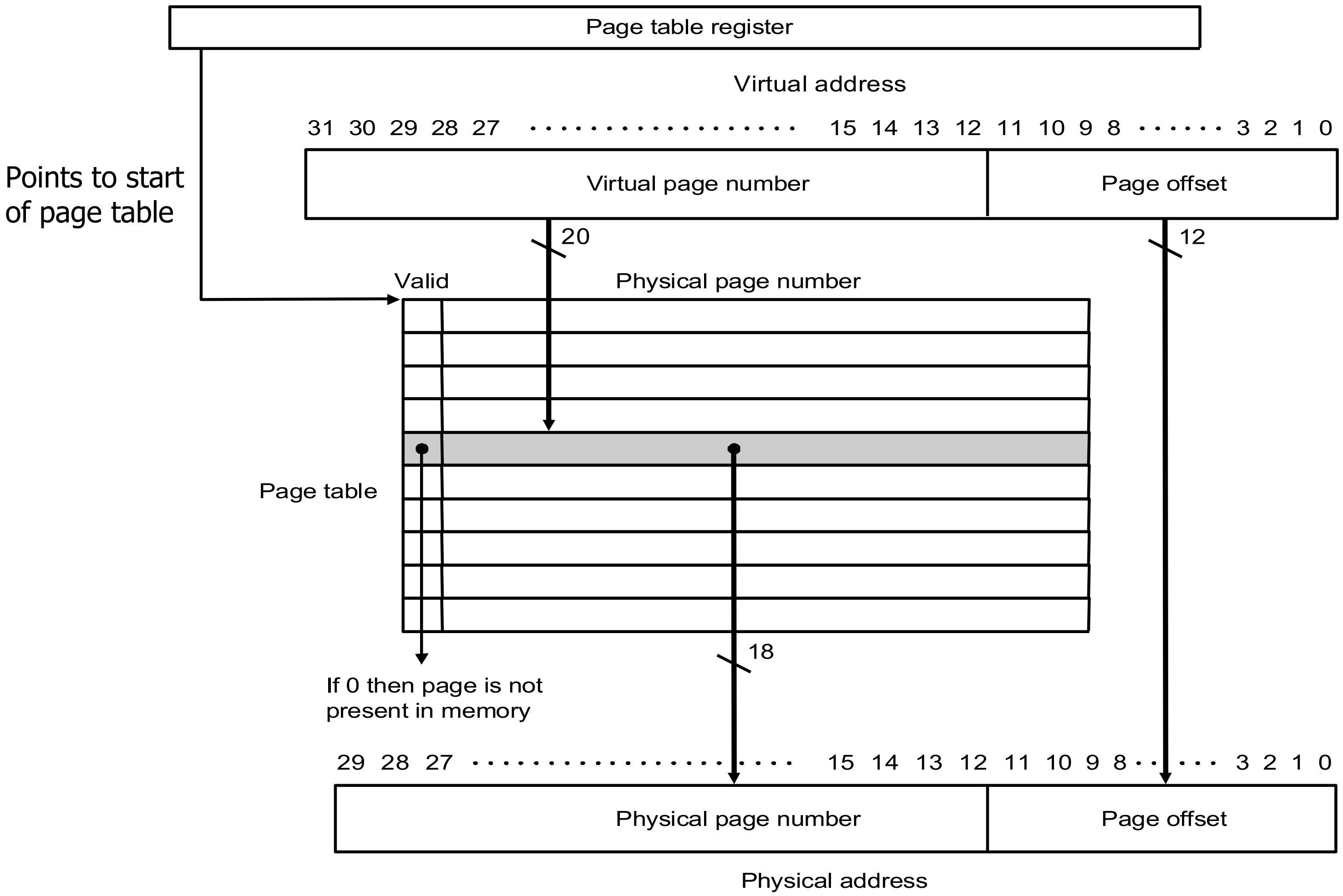
- Motivation: main memory acts as *cache for secondary storage*, e.g., magnetic disk
- **main memory size ≤ disk size ≤ virtual address space size**
- ***Page table*** transparently converts a virtual memory address to a physical memory address,
  - *if the data is already in main*; *if not*, it issues call to OS to fetch the data from disk into main
- Virtual memory is organized in fixed-size (power of 2, typically at least 4 KB) blocks, called ***pages***. Physical memory is also considered a collection of pages of the same size.
  - the unit of data transfer between disk and physical memory is a page

Virtual Memory



Mapping of pages from a virtual address to a physical address or disk address

Page Table Implements Virtual to Physical Address Translation



Page table: page size 4 KB, virtual address space 4 GB, physical memory 1 GB

Example Problem

- Assume:
  - 32-bit virtual address
  - 4 KB page size
  - 4 bytes per page table entry

- *What is the total page table size is we want to be able to access all of the virtual memory?*

Solution

- No. of page table entries = address space size / page size

  = $2^{32} / 2^{12} = 2^{20}$
- Size of page table = No. of entries × entry size

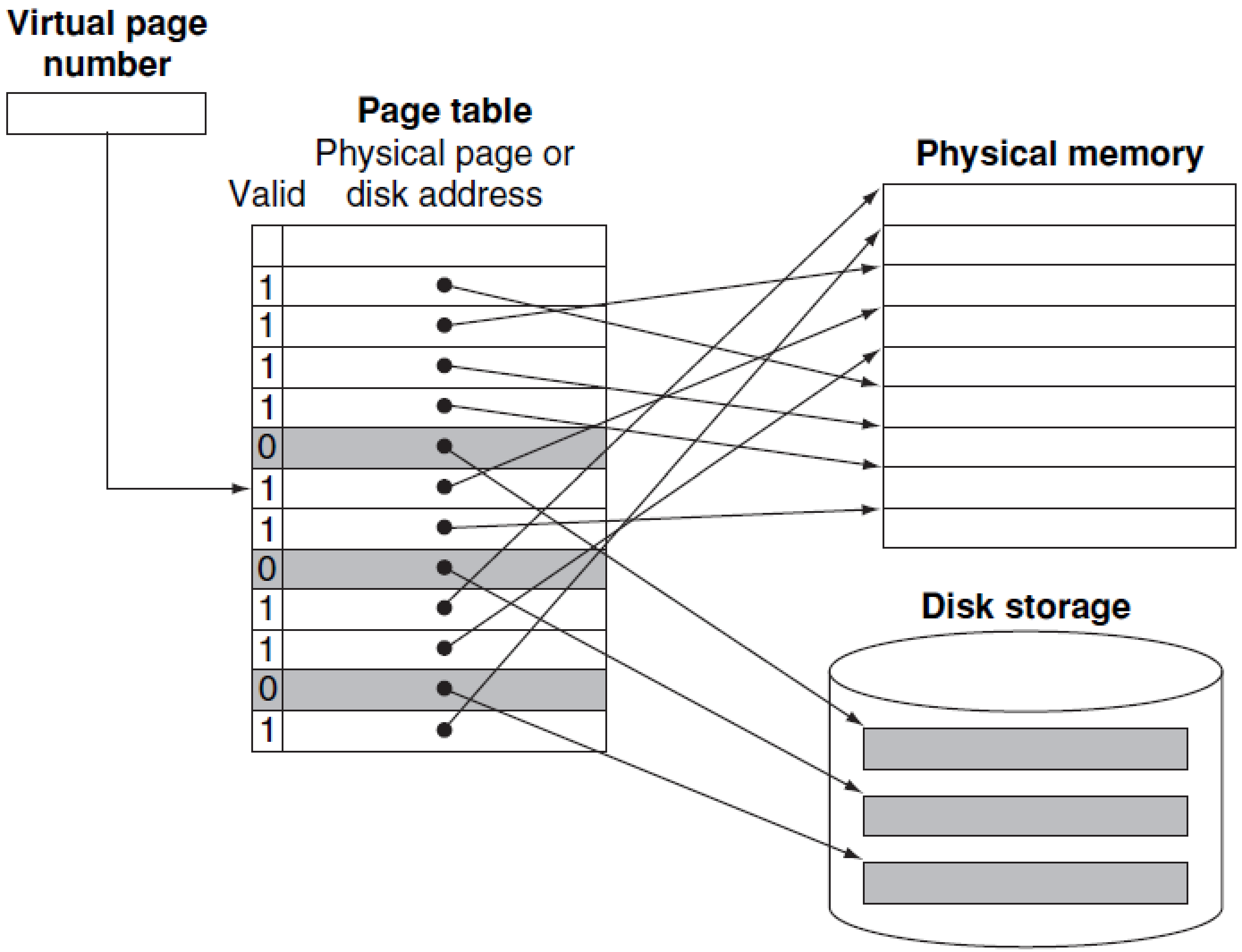
  = $2^{20} \times 4$ bytes = 4 MB (*huge*!)

- Note, to avoid large page table size:
  - each program has its own page table
    - *page table register* points to start of program's page table
  - to reduce storage required per program page table
    - page table for a program covers the span of virtual memory containing its own code and data
    - other techniques, e.g., multiple-level page tables, hashing virtual address, etc.

Page Faults

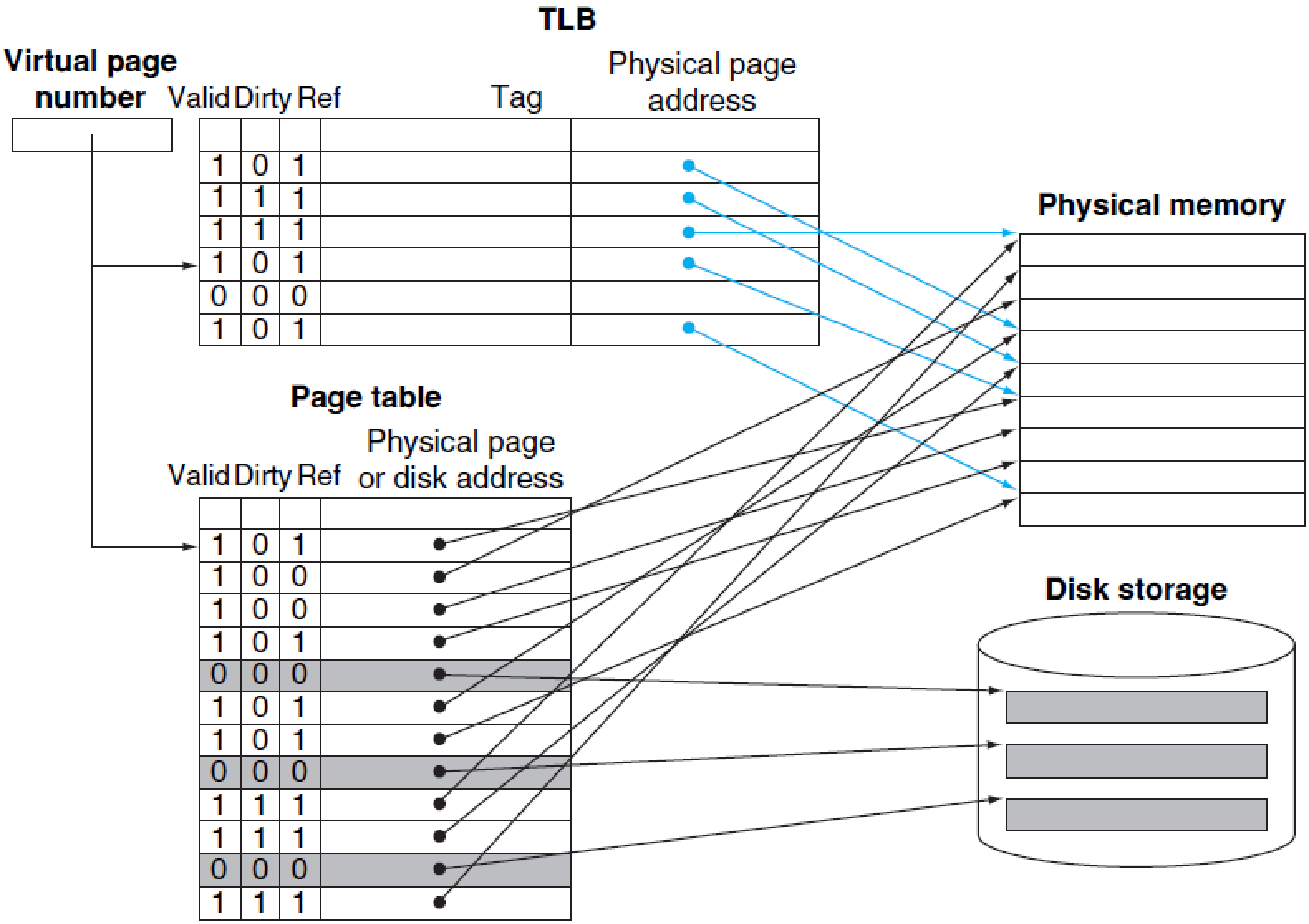
- *Page fault*: page is not in memory, must retrieve it from disk
  - *enormous miss penalty* = millions of cycles
  - therefore, page size should be *large* (e.g., 32 or 64 KB)
    - to make one trip to disk worth a lot
  - reducing page faults is *critical*
    - *LRU replacement* policy – implemented approximately by setting a *use bit* each time a page is accessed, and then periodically clearing all these bits so that pages accessed in a fixed time period are known
    - *fully associative* page placement – consequence of page table
  - handle faults in software instead of hardware
    - as software overhead is still small compared to disk access time
  - using write-through is too expensive, so always use write-back

Resolving Page Faults using the Page



Page table maps virtual page to either physical page or disk page

Making Address Translation Fast with the Translation-lookaside Buffer



Valid	Dirty	Ref	Tag	Physical page address
1	0	1		●
1	1	1		●
1	1	1		●
1	0	1		●
0	0	0		
1	0	1		●

Valid	Dirty	Ref	Physical page or disk address
1	0	1	●
1	0	0	●
1	0	0	●
1	0	1	●
0	0	0	●
1	0	1	●
1	0	1	●
0	0	0	●
1	1	1	●
1	1	1	●
0	0	0	●
1	1	1	●

On a page reference, first look up the virtual page number in the TLB; if there is a TLB miss look up the page table; if miss again then true page fault

Summary

- Memory hierarchy
- Cache basics
- Measuring cache perfromance
- Improving cache performance
- Virtual memory

UNIT IV MEMORY AND I/O SYSTEMS

Input/output system, programmed I/O, DMA and interrupts, I/O processors.

Reference:

- Chapter 7 – Input Output System
- Book - William Stallings , “Computer Organization and Architecture‟, 8th Edition.

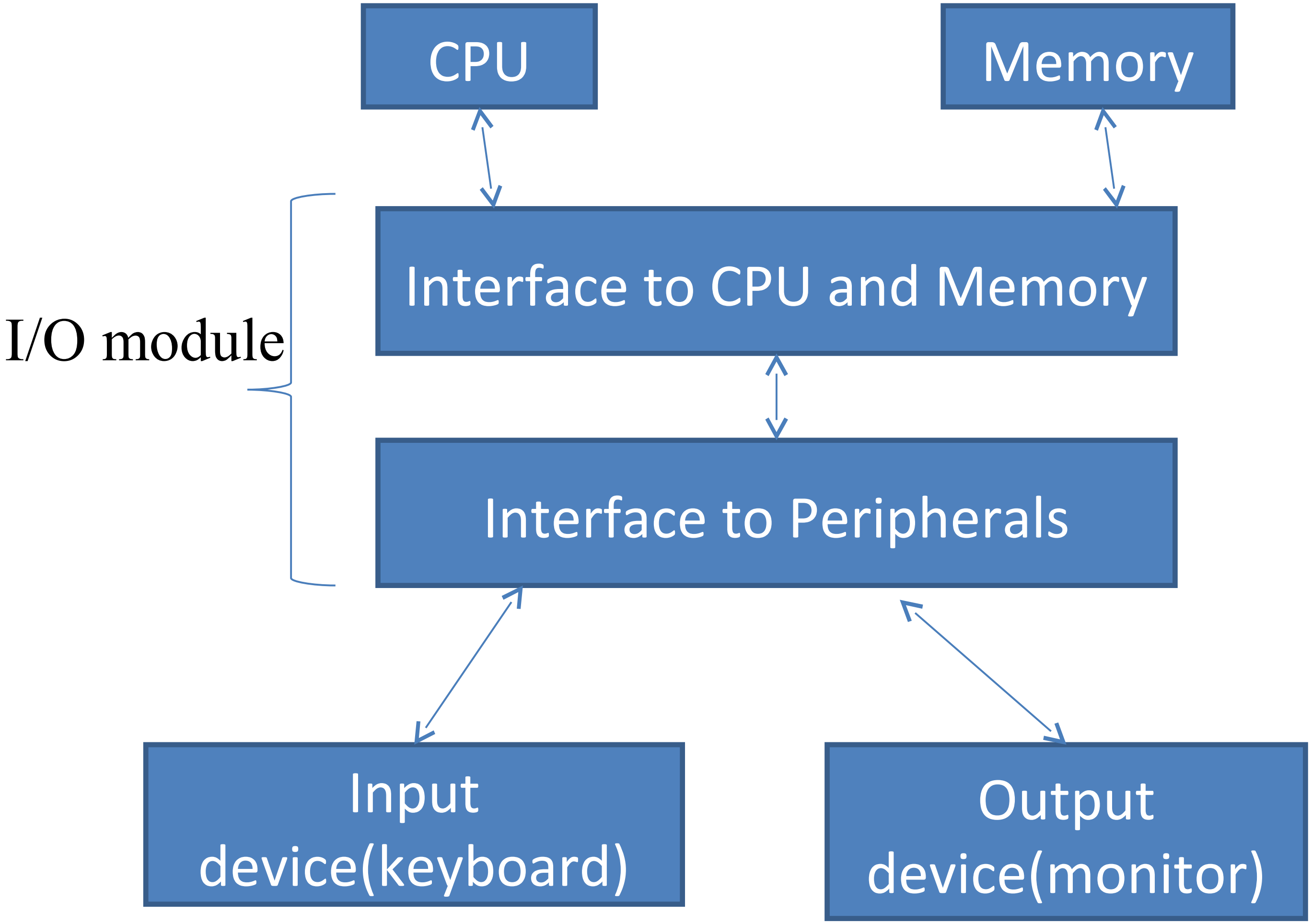
Input Output System

Need for Input/Output Module

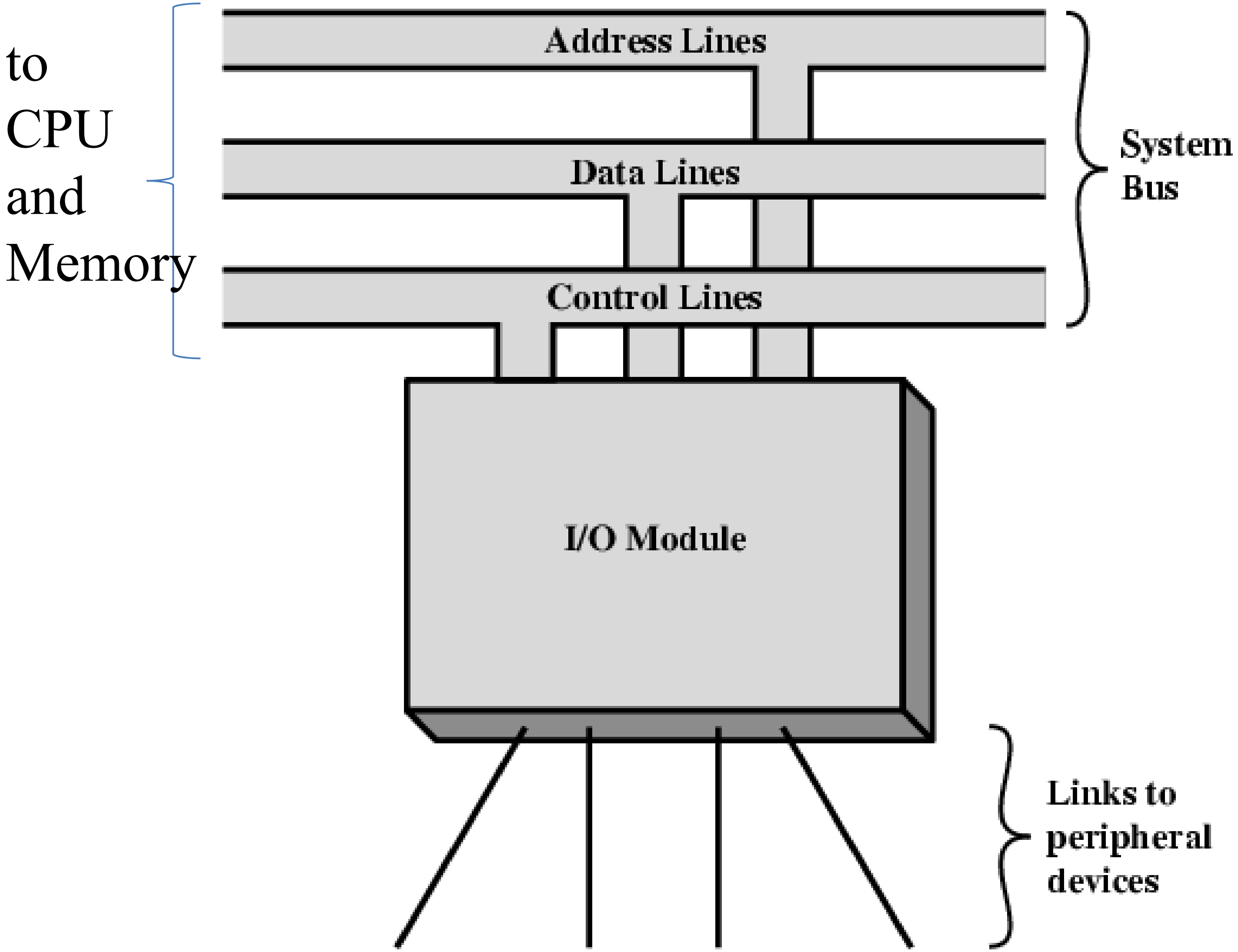
- Wide variety of peripherals
  - Delivering different amounts of data
  - At different speeds
  - In different formats
- All slower than CPU and RAM
- Need I/O modules

Input/Output Module

- Interface to CPU and Memory
- Interface to one or more peripherals



Generic Model of I/O Module

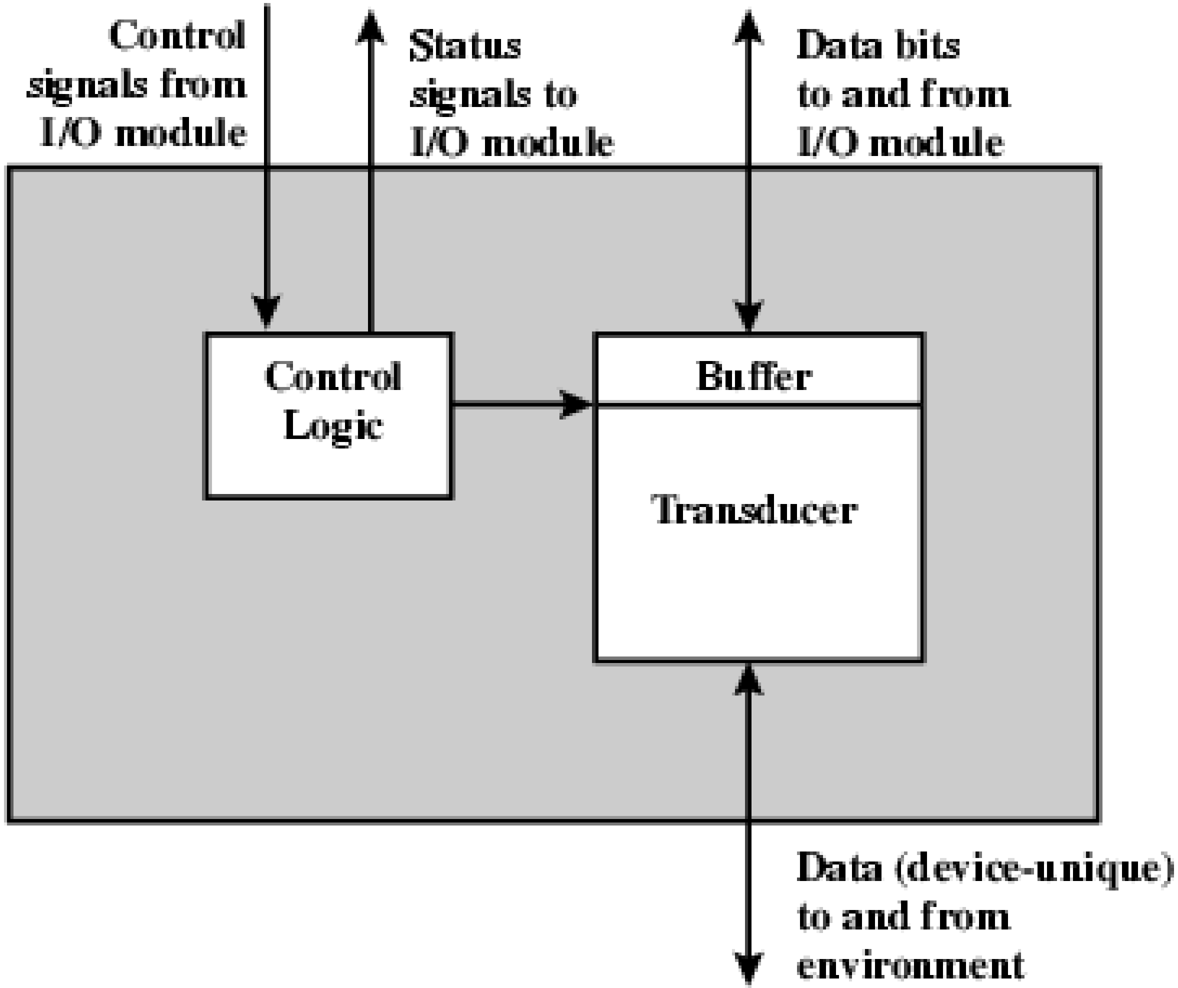


to
CPU
and
Memory

External Devices

- Monitor Screen
- Printer
- Keyboard
- Modem

External Device Block Diagram


Control signals from I/O module
Status signals to I/O module
Data bits to and from I/O module
Control Logic
Buffer
Transducer
Data (device-unique) to and from environment

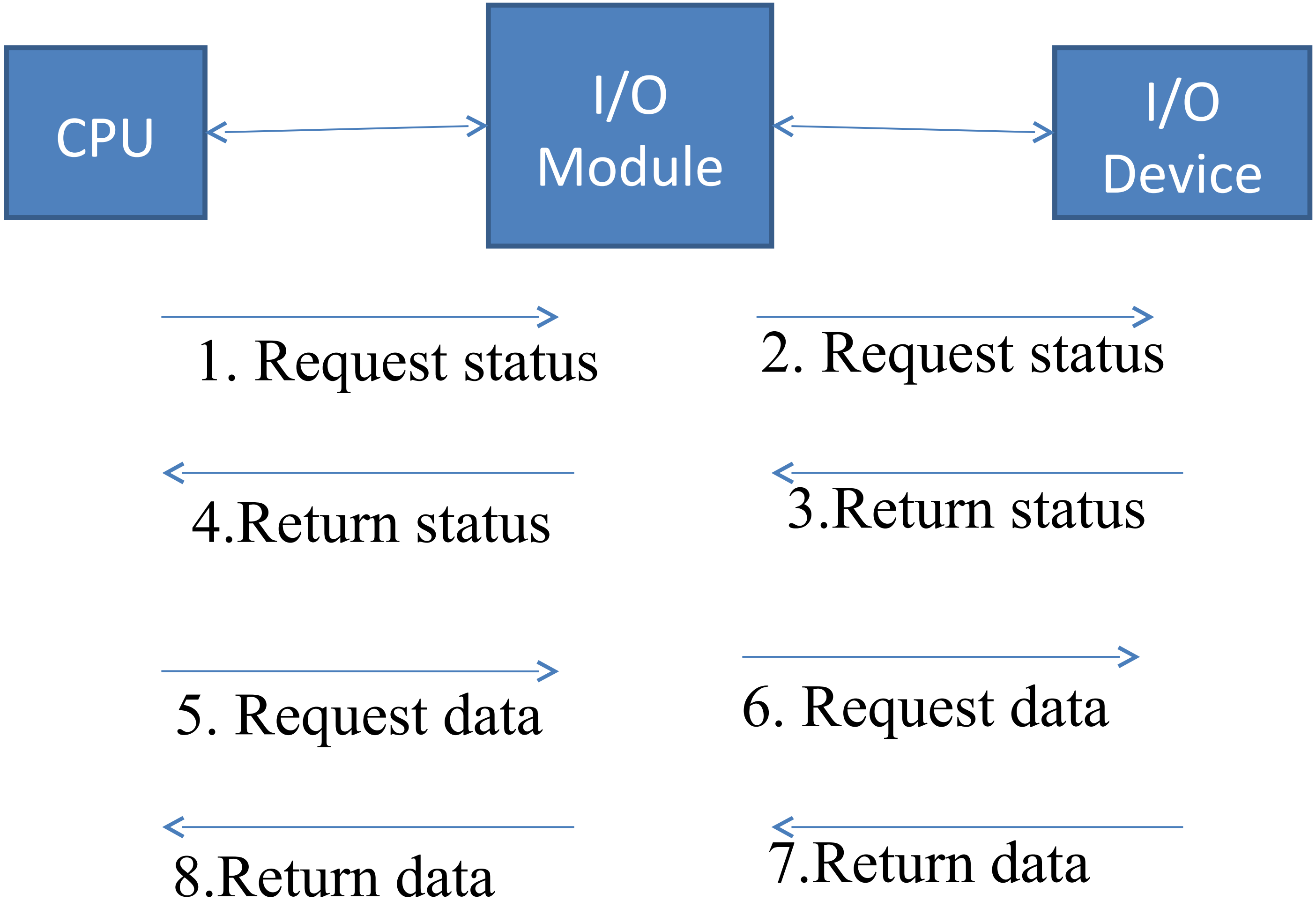
I/O Module Function

- Control & Timing
- CPU Communication
- Device Communication
- Data Buffering
- Error Detection

I/O Steps

- CPU checks I/O module device status
- I/O module returns status
- If ready, CPU requests data transfer
- I/O module gets data from device
- I/O module transfers data to CPU
- Variations for output, DMA, etc.

I/O Function and Steps



CPU

I/O Module

I/O Device

1. Request status

2. Request status

3.Return status

4.Return status

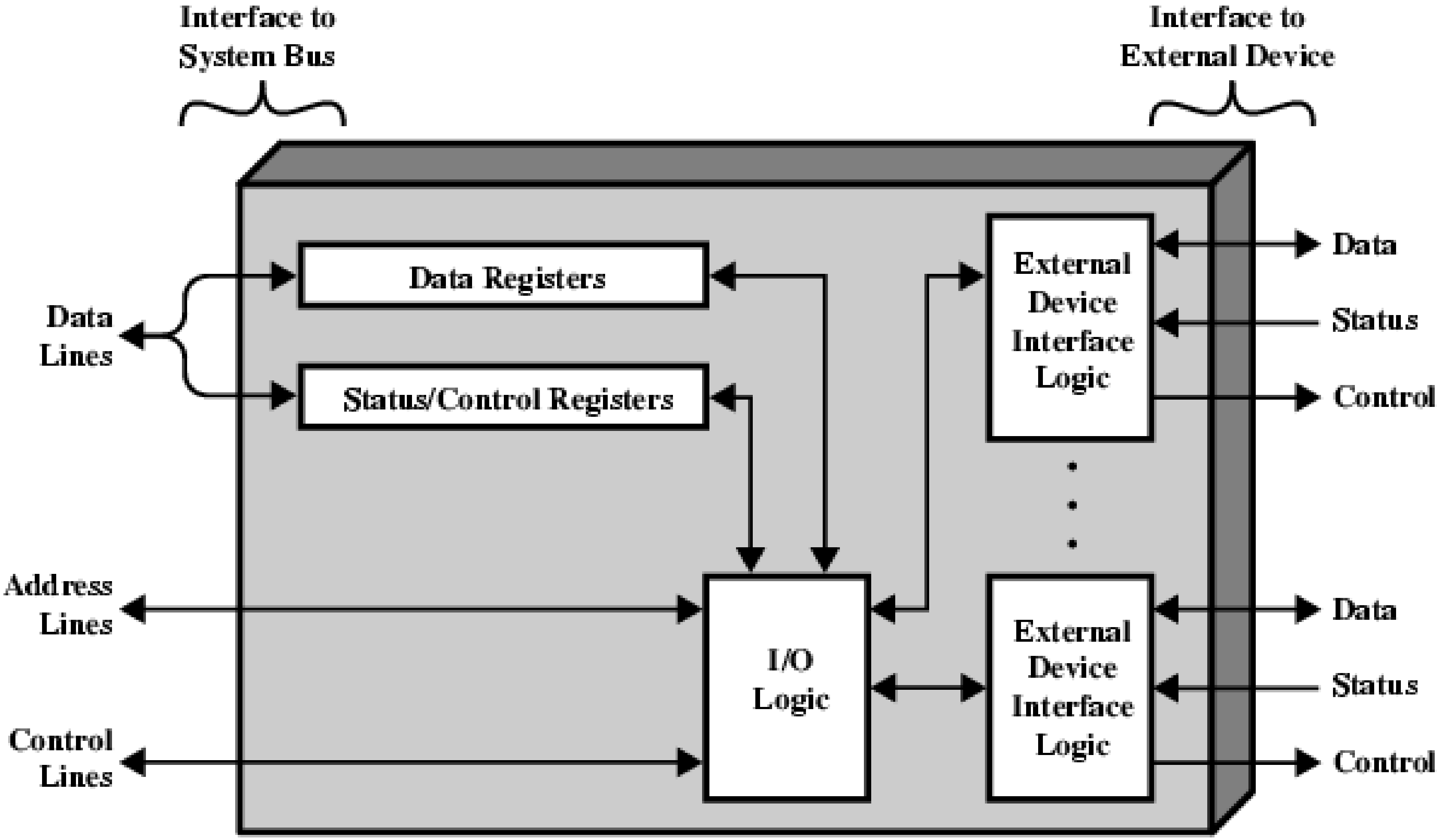
5. Request data

6. Request data

7.Return data

8.Return data

I/O Module Diagram



I/O Module Decisions

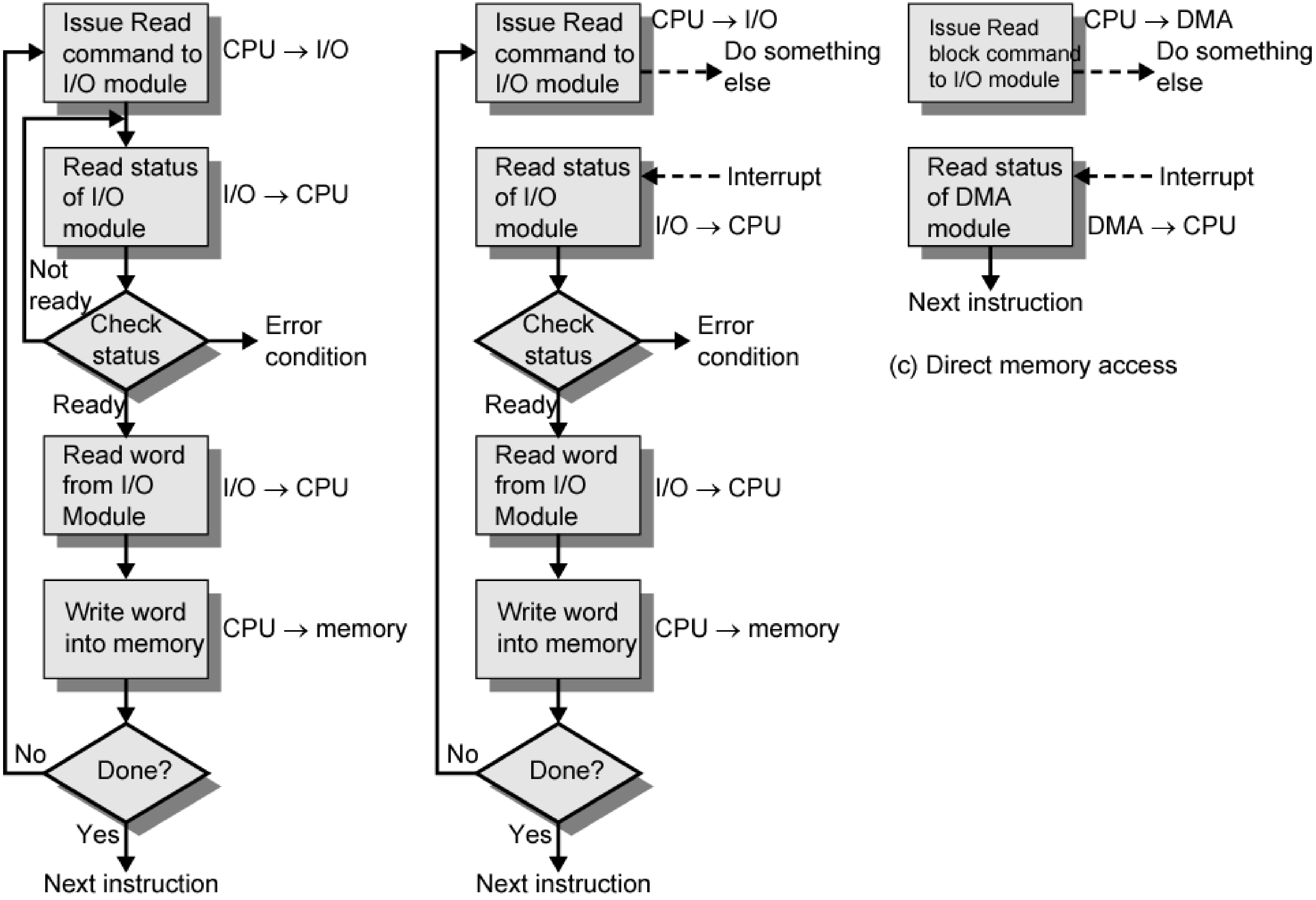
- Hide or reveal device properties to CPU
- Support multiple or single device
- Control device functions or leave for CPU
- Also O/S decisions
  - e.g. Unix treats everything it can as a file

Programmed I/O

Input Output Techniques

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access (DMA)

Three Techniques for Input of a Block of Data



Issue Read command to I/O module

CPU → I/O

Read status of I/O module

I/O → CPU

Check status

Not ready

Error condition

Ready

Read word from I/O Module

I/O → CPU

Write word into memory

CPU → memory

Done?

No

Yes

Next instruction

(a) Programmed I/O

Issue Read command to I/O module

CPU → I/O

Do something else

Read status of I/O module

Interrupt

I/O → CPU

Check status

Error condition

Ready

Read word from I/O Module

I/O → CPU

Write word into memory

CPU → memory

Done?

No

Yes

Next instruction

(b) Interrupt-driven I/O

Issue Read block command to I/O module

CPU → DMA

Do something else

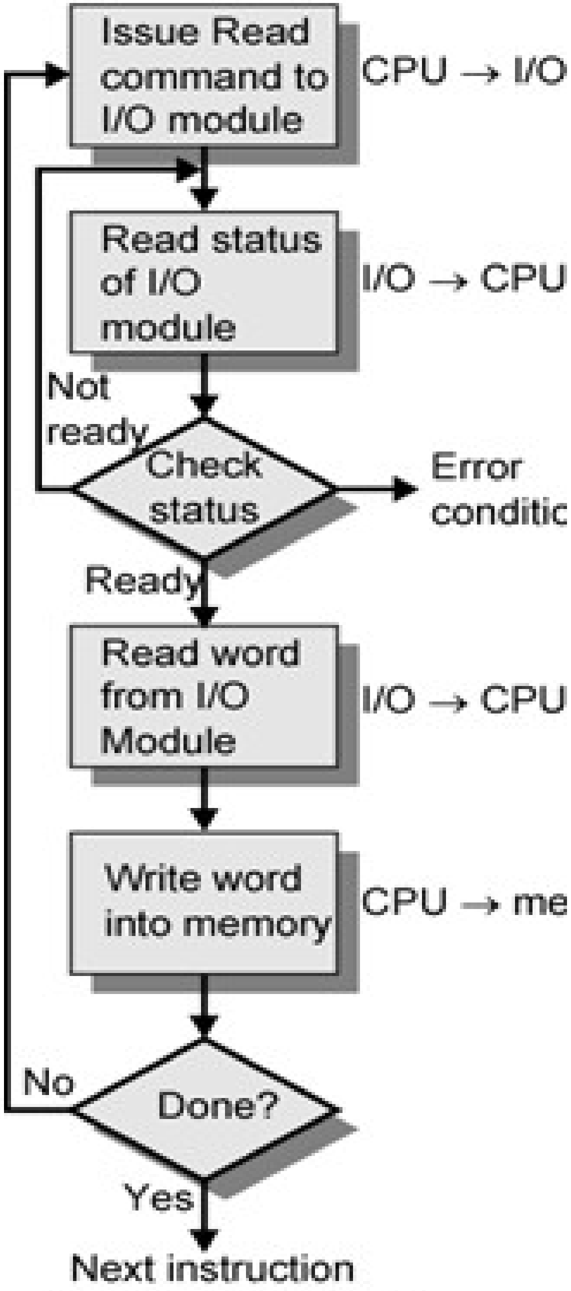
Read status of DMA module

Interrupt

DMA → CPU

Next instruction

(c) Direct memory access



(a) Programmed I/O

Programmed I/O

- CPU has direct control over I/O
  - Sensing status
  - Read/write commands
  - Transferring data
- CPU waits for I/O module to complete operation
- Wastes CPU time

Programmed I/O - detail

- CPU requests I/O operation
- I/O module performs operation
- I/O module sets status bits
- CPU checks status bits periodically
- I/O module does not inform CPU directly
- I/O module does not interrupt CPU
- CPU may wait or come back later

I/O Commands

- CPU issues address
  - Identifies module (& device if >1 per module)
- CPU issues command
  - Control - telling module what to do
    - e.g. spin up disk
  - Test - check status
    - e.g. power? Error?
  - Read/Write
    - Module transfers data via buffer from/to device

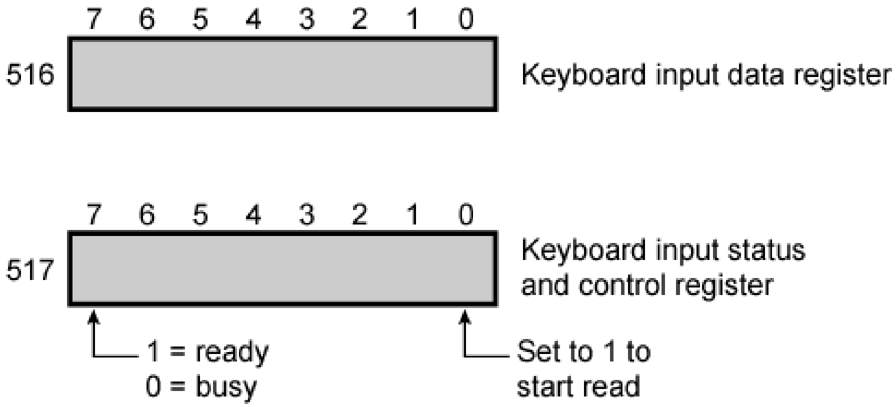
Addressing I/O Devices

- Under programmed I/O data transfer is very like memory access (CPU viewpoint)
- Each device given unique identifier
- CPU commands contain identifier (address)

I/O Mapping

- Memory mapped I/O
  - Devices and memory share an address space
  - I/O looks just like memory read/write
  - No special commands for I/O
    - Large selection of memory access commands available
- Isolated I/O
  - Separate address spaces
  - Need I/O or memory select lines
  - Special commands for I/O
    - Limited set

Memory Mapped and Isolated I/O



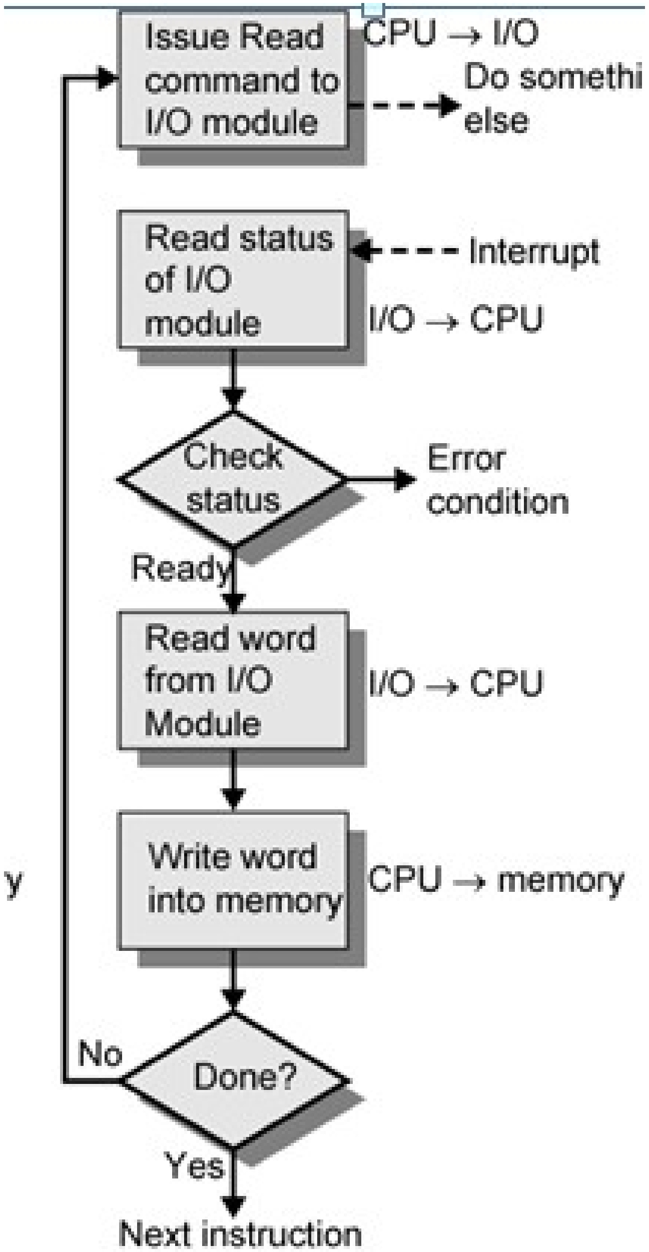
ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load AC	"1"	Load accumulator
	Store AC	517	Initiate keyboard read
202	Load AC	517	Get status byte
	Branch if Sign = 0	202	Loop until ready
	Load AC	516	Load data byte

(a) Memory-mapped I/O

ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load I/O	5	Initiate keyboard read
201	Test I/O	5	Check for completion
	Branch Not Ready	201	Loop until complete
	In	5	Load data byte

(b) Isolated I/O

Interrupt Driven I/O



(b) Interrupt-driven I/O

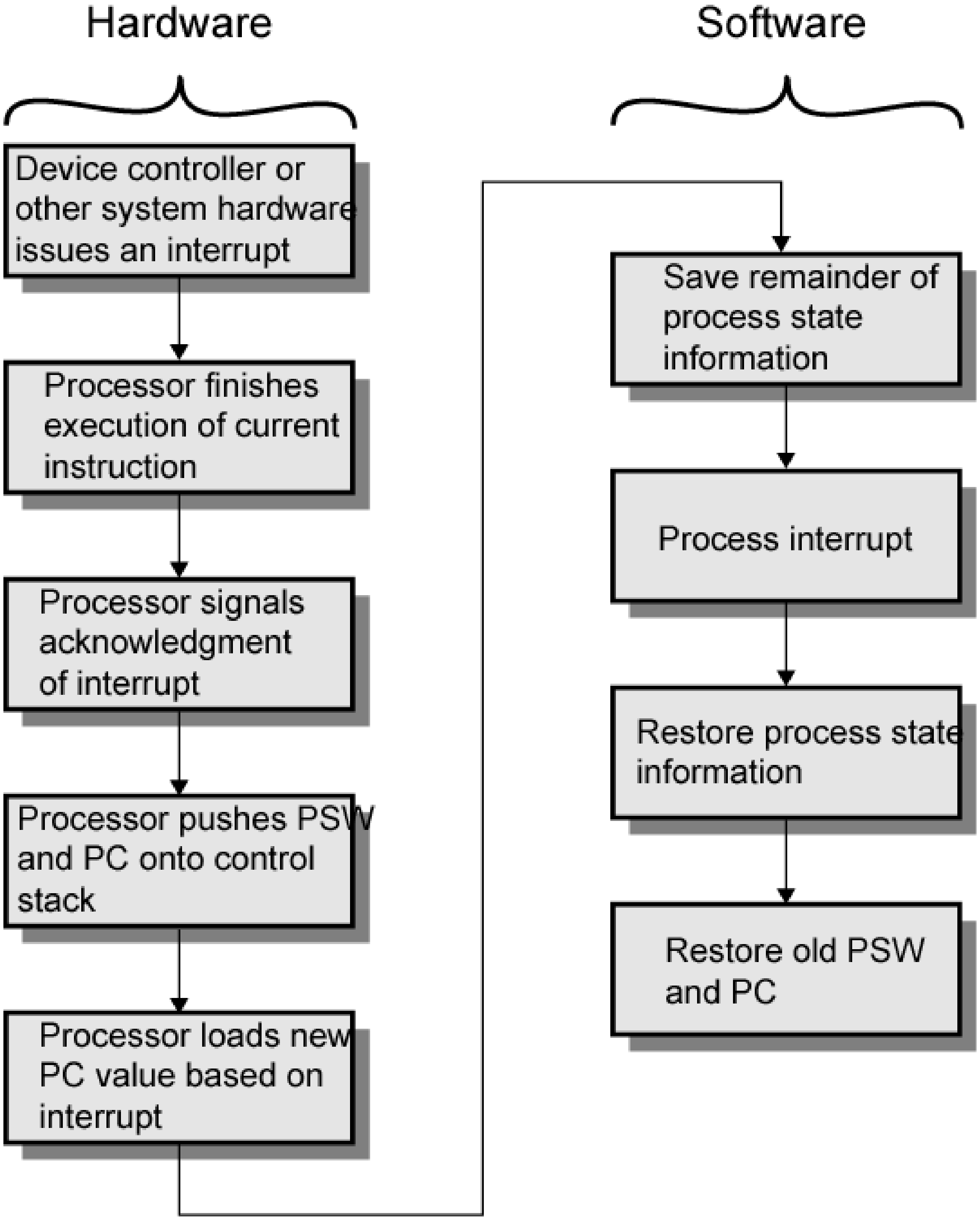
Interrupt Driven I/O

- Overcomes CPU waiting
- No repeated CPU checking of device
- I/O module interrupts when ready

Interrupt Driven I/O Basic Operation

- CPU issues read command
- I/O module gets data from peripheral whilst CPU does other work
- I/O module interrupts CPU
- CPU requests data
- I/O module transfers data

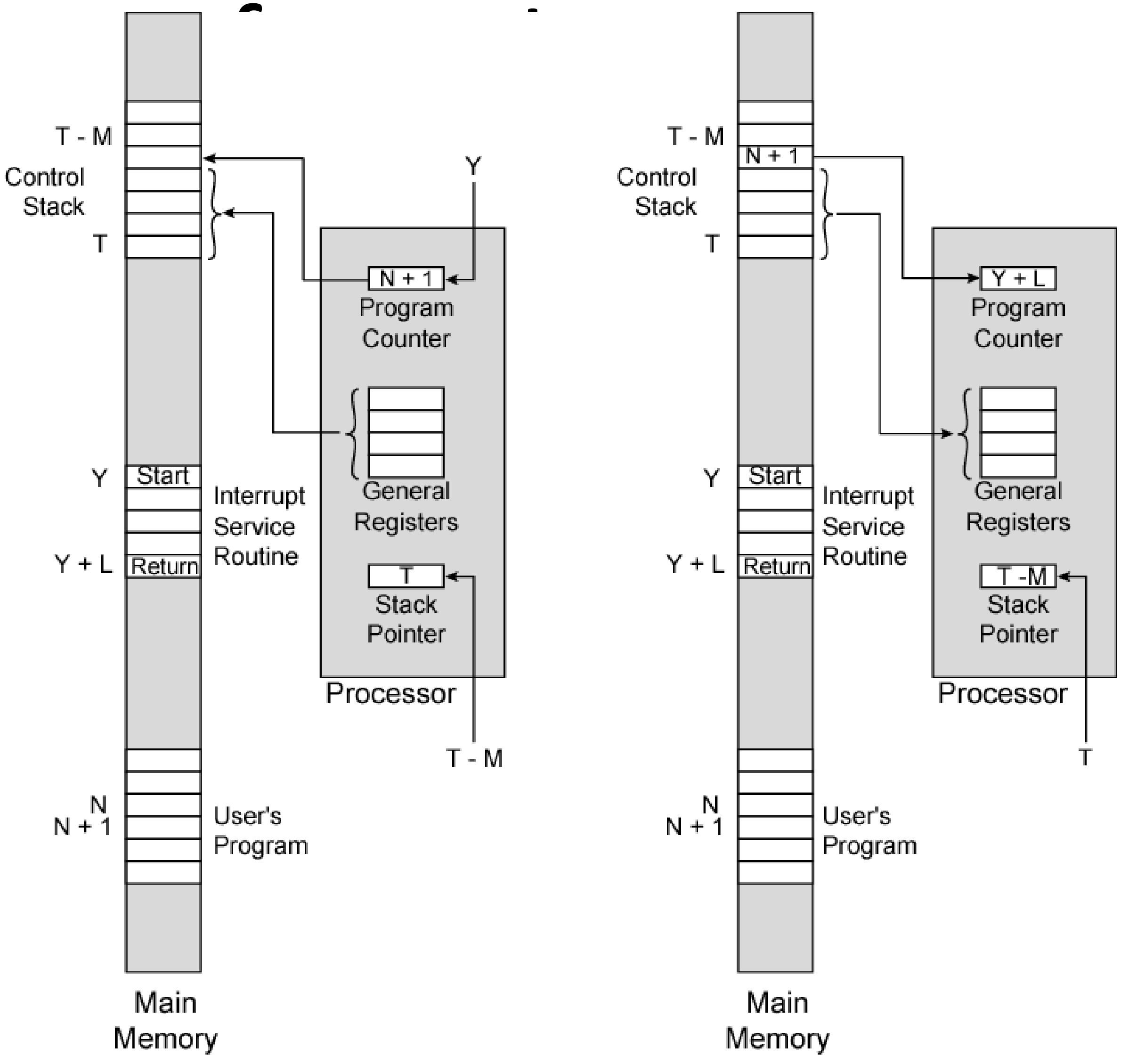
Simple Interrupt Processing


Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

CPU Viewpoint

- Issue read command
- Do other work
- Check for interrupt at end of each instruction cycle
- If interrupted:-
  - Save context (registers)
  - Process interrupt
    - Fetch data & store
- See Operating Systems notes

Changes in Memory and Registers



(a) Interrupt occurs after instruction at location N

(b) Return from interrupt

Design Issues

- How do you identify the module issuing the interrupt?
- How do you deal with multiple interrupts?
  - i.e. an interrupt handler being interrupted

Identifying Interrupting Module (1)

- Different line for each module
  - PC
  - Limits number of devices
- Software poll
  - CPU asks each module in turn
  - Slow

Identifying Interrupting Module (2)

- Daisy Chain or Hardware poll
  - Interrupt Acknowledge sent down a chain
  - Module responsible places vector on bus
  - CPU uses vector to identify handler routine
- Bus Master
  - Module must claim the bus before it can raise interrupt

Multiple Interrupts

1) Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

- Each line has different priority levels.
- Interrupt acknowledgement will be sent only if it has highest priority than the one assigned currently.
- **Adv** – accept interrupt request from **some devices based on priority.**

2) **Daisy chain**

- **Adv** - fewer lines

3) Arrangement of priority groups

- **Adv –** combine the advantage of both the above schemes

Example - PC Bus

- 80x86 has one interrupt line
- 8086 based systems use one 8259A interrupt controller
- 8259A has 8 interrupt lines

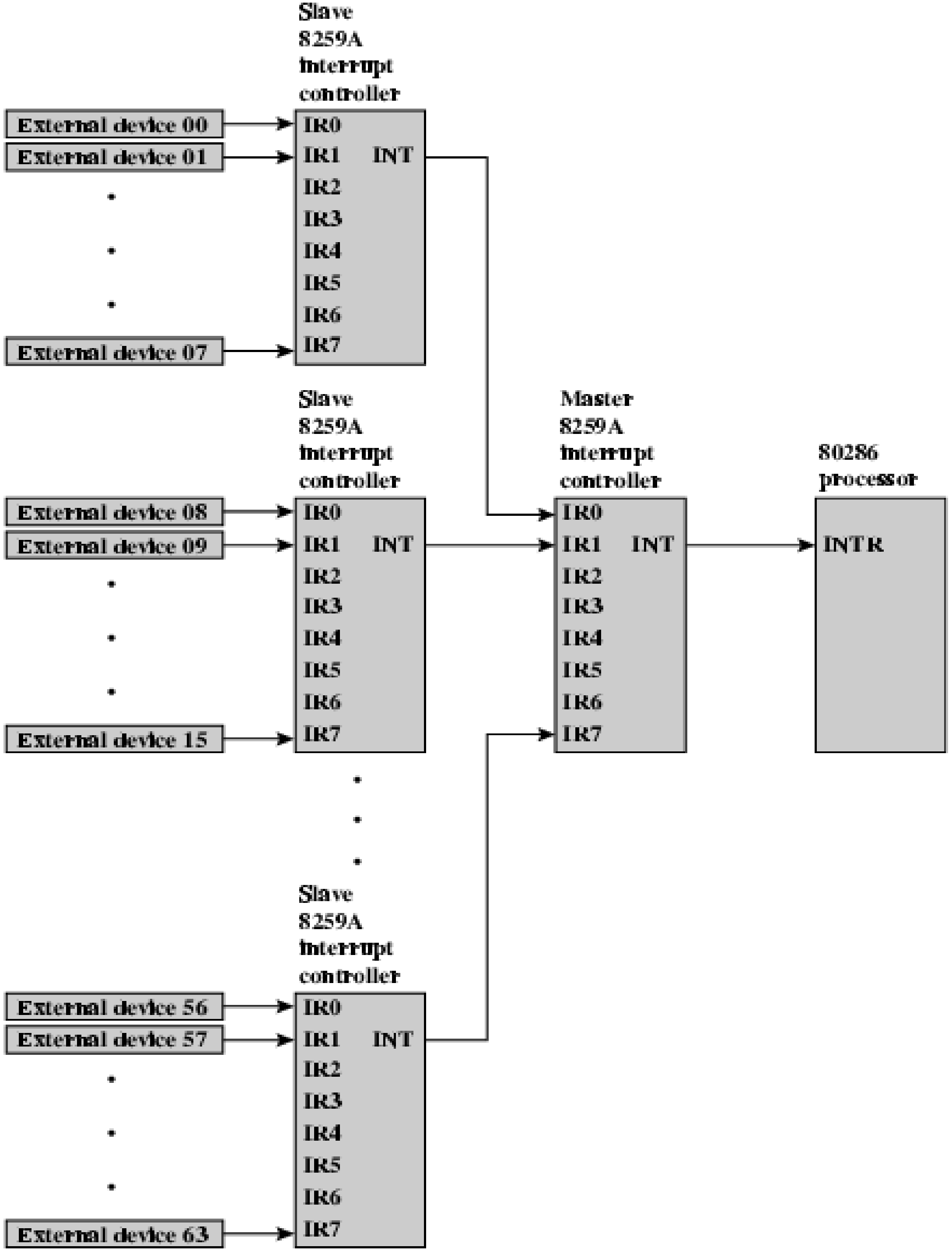
Sequence of Events

- 8259A accepts interrupts
- 8259A determines priority
- 8259A signals 8086 (raises INTR line)
- CPU Acknowledges
- 8259A puts correct vector on data bus
- CPU processes interrupt

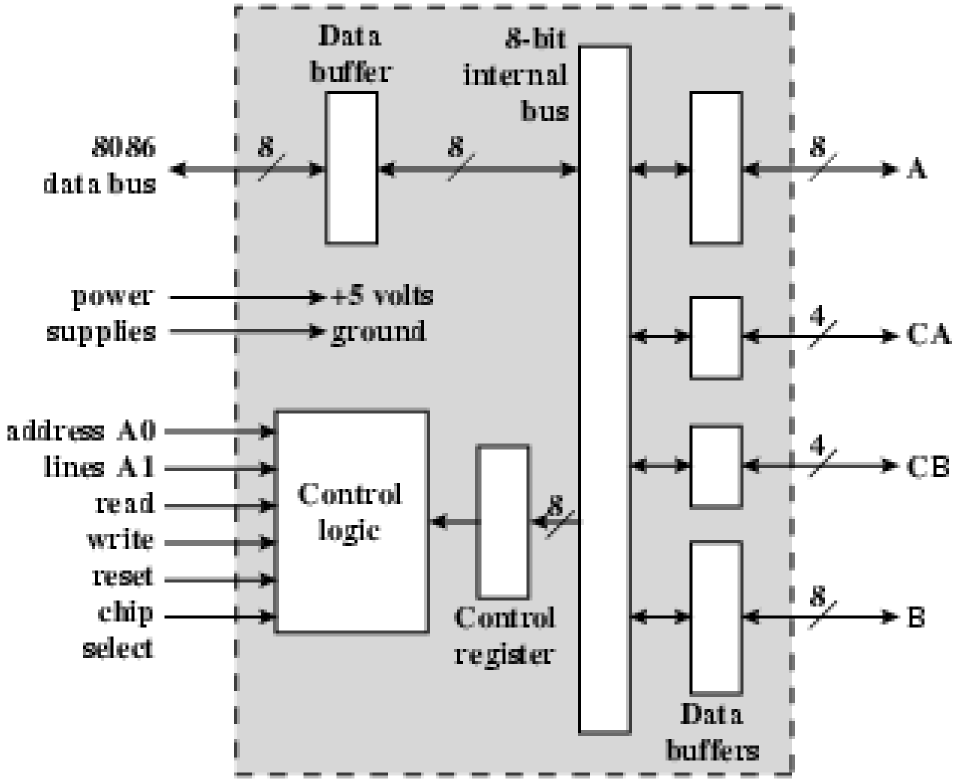
ISA Bus Interrupt System

- ISA bus chains two 8259As together
- Link is via interrupt 2
- Gives 15 lines
  - 16 lines less one for link
- IRQ 9 is used to re-route anything trying to use IRQ 2
  - Backwards compatibility
- Incorporated in chip set

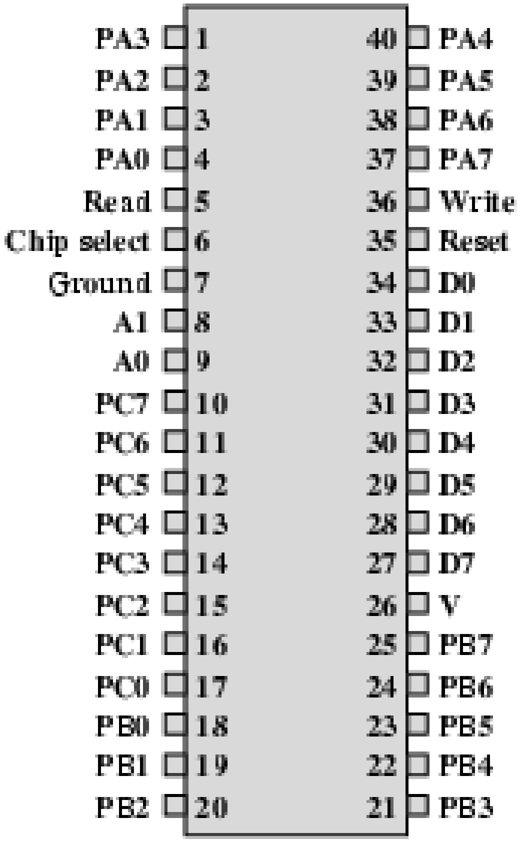
82C59A Interrupt Controller



Intel 82C55A Programmable Peripheral Interface

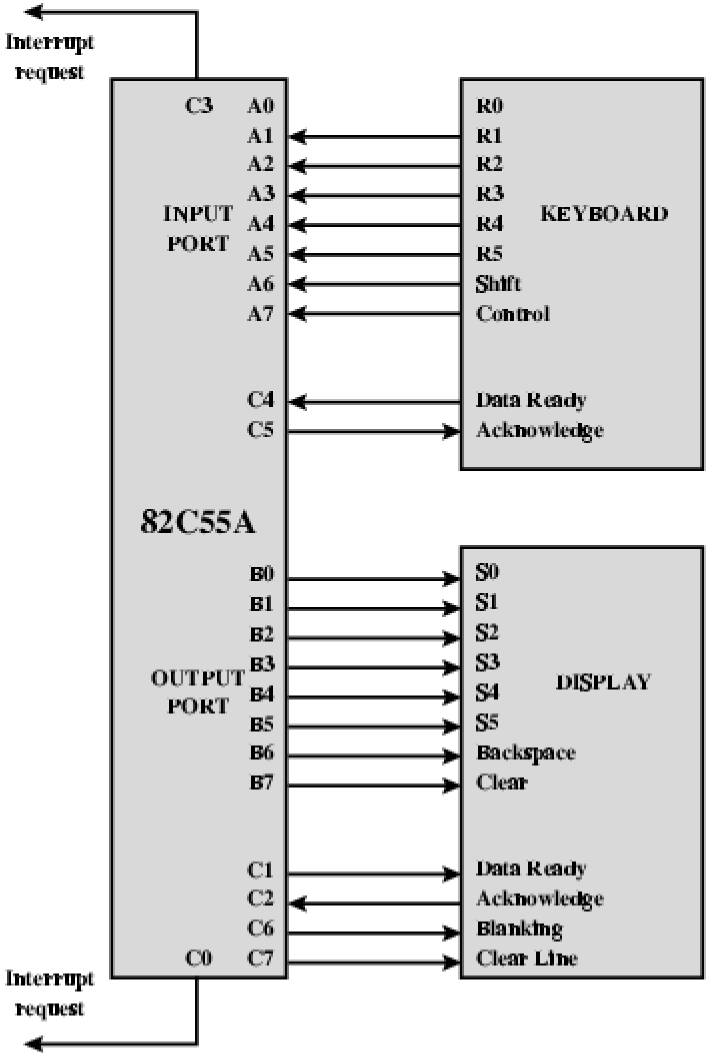


(a) Block diagram



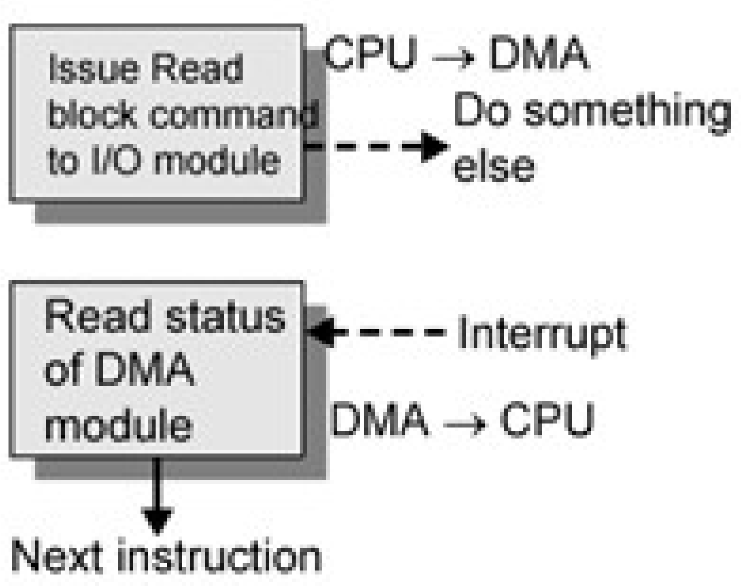
(b) Pin layout

Keyboard/Display Interfaces to 82C55A



DMA

DMA working


Issue Read block command to I/O module
CPU → DMA
Do something else
Read status of DMA module
Interrupt
DMA → CPU
Next instruction

Direct Memory Access

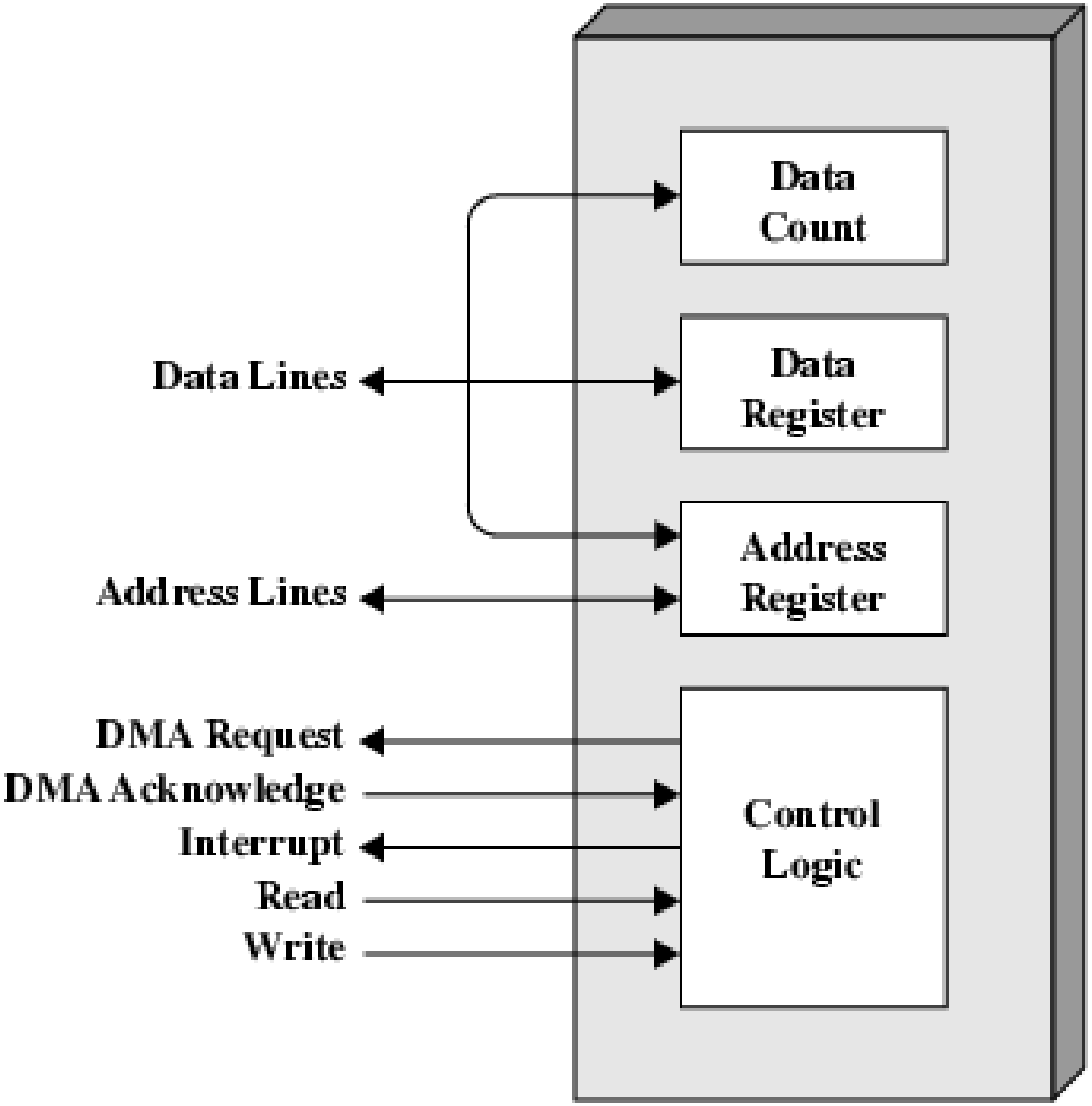
- DMA – used to transfer large block of data

- Interrupt driven and programmed I/O require active CPU intervention
  - Transfer rate is limited
  - CPU is tied up

DMA Function

- Additional Module (hardware) on bus
- DMA controller takes over from CPU for I/O

Typical DMA Module Diagram



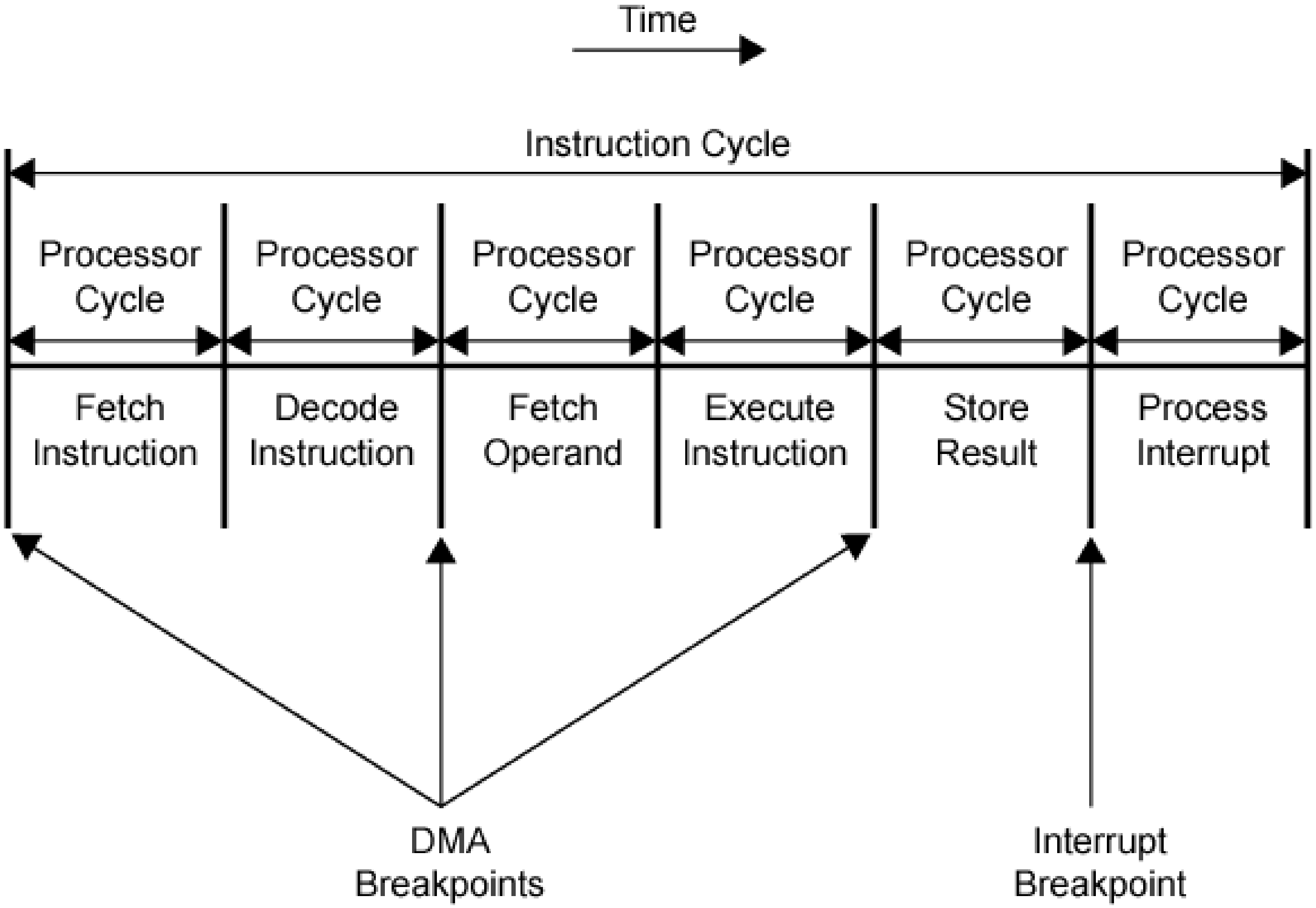
DMA Operation

- CPU tells DMA controller:-
  - Read/Write
  - Device address
  - Starting address of memory block for data
  - Amount of data to be transferred
- CPU carries on with other work
- DMA controller deals with transfer
- DMA controller sends interrupt when finished

Memory Cycle Stealing

- DMA controller has higher priority over the system bus than the processor.
  - This causes the DMA controller to steal the memory clock cycles of the processor.

DMA and Interrupt Breakpoints During an Instruction Cycle



Time

Instruction Cycle

Processor Cycle	Processor Cycle	Processor Cycle	Processor Cycle	Processor Cycle	Processor Cycle
Fetch Instruction	Decode Instruction	Fetch Operand	Execute Instruction	Store Result	Process Interrupt

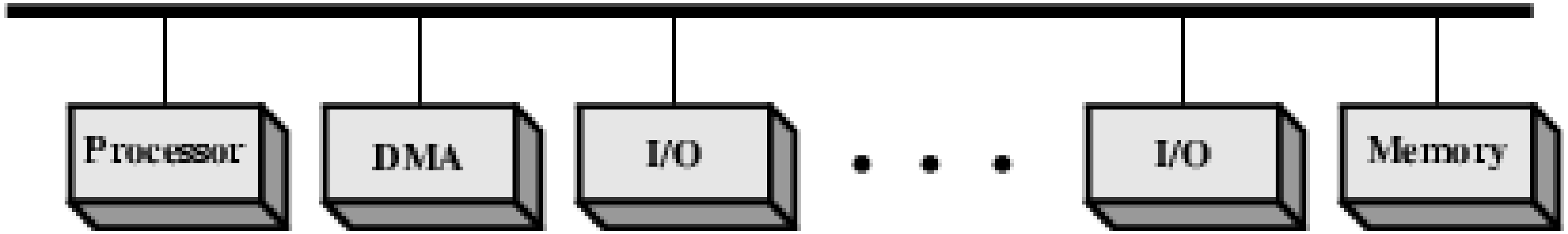
DMA Breakpoints

Interrupt Breakpoint

Aside

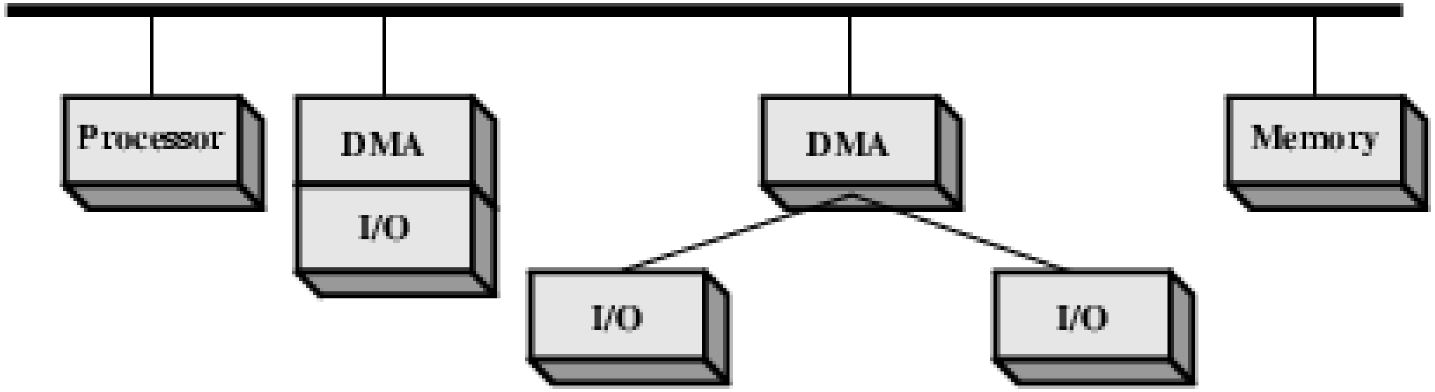
- What effect does caching memory have on DMA?
- What about on board cache?
- Hint: how much are the system buses available?

DMA Configurations (1)



- Single Bus, Detached DMA controller
- Each transfer uses bus twice
  - I/O to DMA then DMA to memory
- CPU is suspended twice

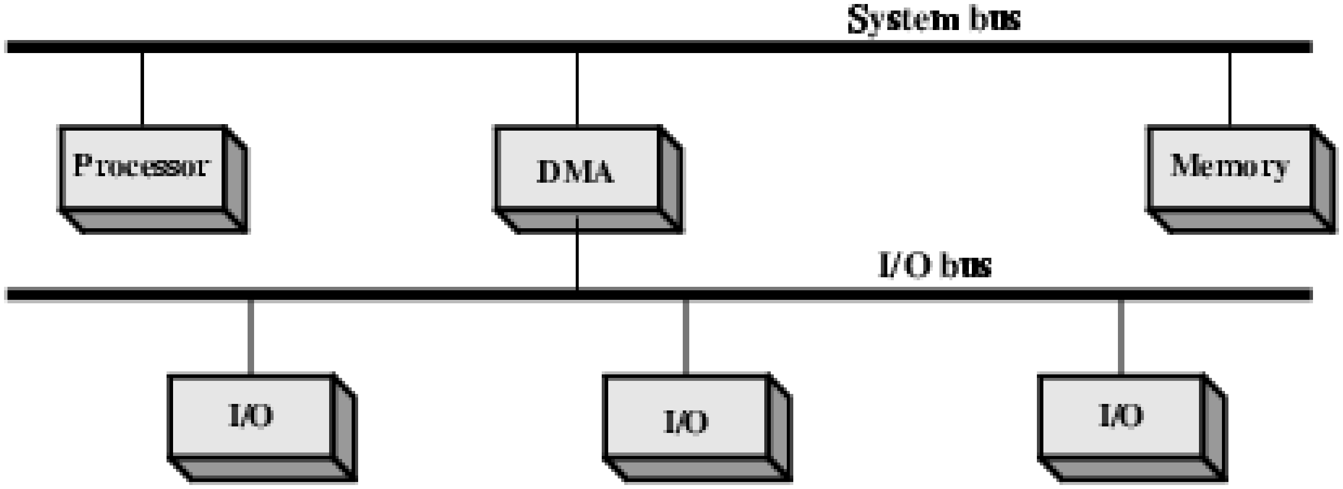
DMA Configurations (2)



(b) Single-bus, Integrated DMA-I/O

- Single Bus, Integrated DMA controller
- Controller may support >1 device
- Each transfer uses bus once
  - DMA to memory
- CPU is suspended once

DMA Configurations (3)



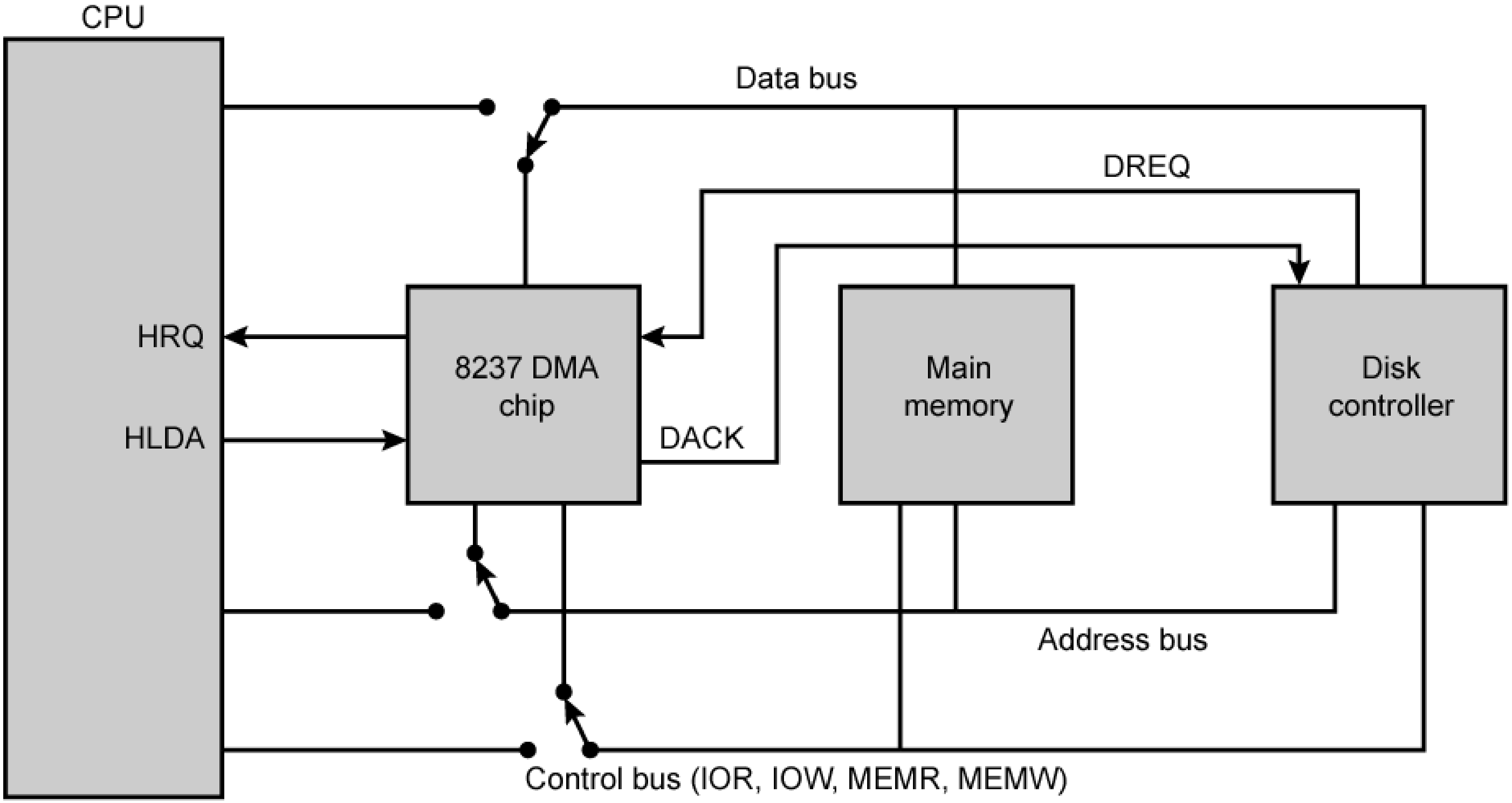
(c) I/O bus

- Separate I/O Bus
- Bus supports all DMA enabled devices
- Each transfer uses bus once
  – DMA to memory
- CPU is suspended once

Intel 8237A DMA Controller

- Interfaces to 80x86 family and DRAM
- When DMA module needs buses it sends HOLD signal to processor
- CPU responds HLDA (hold acknowledge)
  - DMA module can use buses
- E.g. transfer data from memory to disk
  1. Device requests service of DMA by pulling DREQ (DMA request) high
  2. DMA puts high on HRQ (hold request),
  3. CPU finishes present bus cycle (not necessarily present instruction) and puts high on HDLA (hold acknowledge). HOLD remains active for duration of DMA
  4. DMA activates DACK (DMA acknowledge), telling device to start transfer
  5. DMA starts transfer by putting address of first byte on address bus and activating MEMR; it then activates IOW to write to peripheral. DMA decrements counter and increments address pointer. Repeat until count reaches zero
  6. DMA deactivates HRQ, giving bus back to CPU

8237 DMA Usage of Systems Bus



DACK = DMA acknowledge
DREQ = DMA request
HLDA = HOLD acknowledge
HRQ = HOLD request

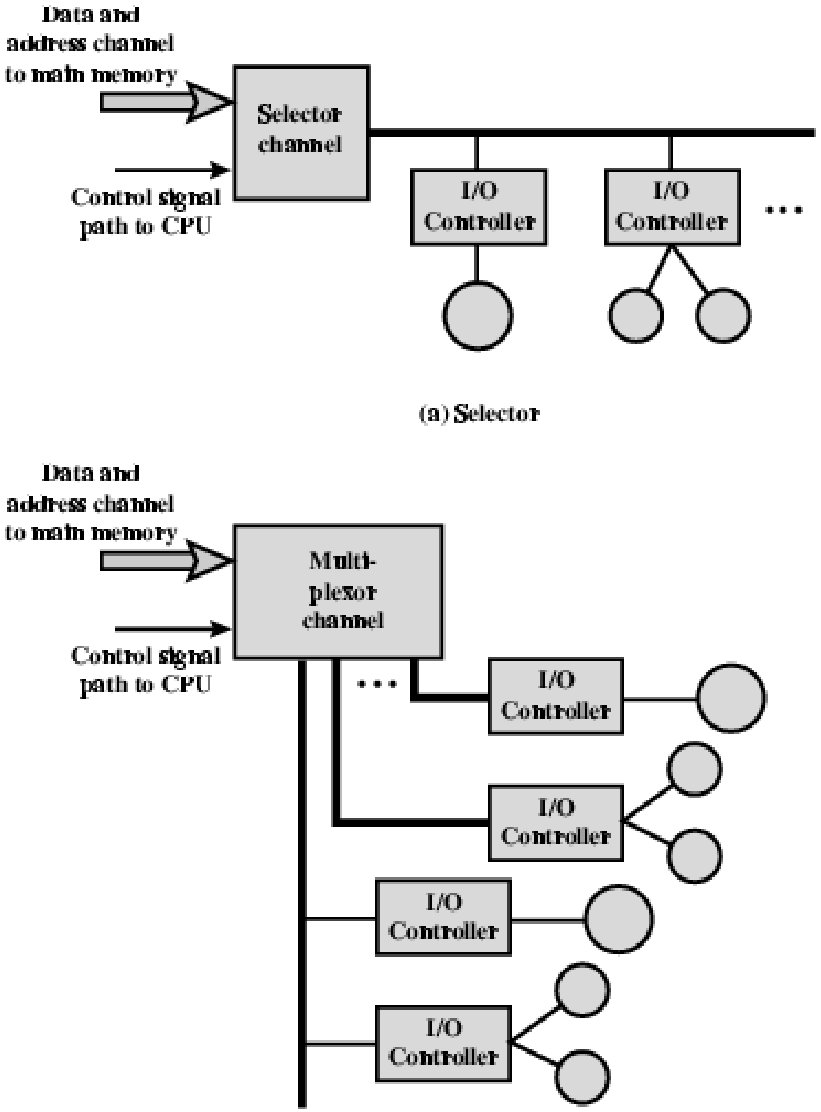
Fly-By

- While DMA using buses processor idle
- Processor using bus, DMA idle
  - Known as fly-by DMA controller
- Data does not pass through and is not stored in DMA chip
  - DMA only between I/O port and memory
  - Not between two I/O ports or two memory locations
- Can do memory to memory via register
- 8237 contains four DMA channels
  - Programmed independently
  - Any one active
  - Numbered 0, 1, 2, and 3

I/O Channels

- I/O devices getting more sophisticated
- e.g. 3D graphics cards
- CPU instructs I/O controller to do transfer
- I/O controller does entire transfer
- Improves speed
  - Takes load off CPU
  - Dedicated processor is faster

I/O Channel Architecture


Data and address channel to main memory
Selector channel
Control signal path to CPU
I/O Controller
I/O Controller
• • •
Data and address channel to main memory
Multi-plexor channel
Control signal path to CPU
• • •
I/O Controller
I/O Controller
I/O Controller
I/O Controller

(a) Selector

(b) Multiplexor

Interfacing

- Connecting devices together
- Bit of wire?
- Dedicated processor/memory/buses?
- E.g. FireWire, InfiniBand

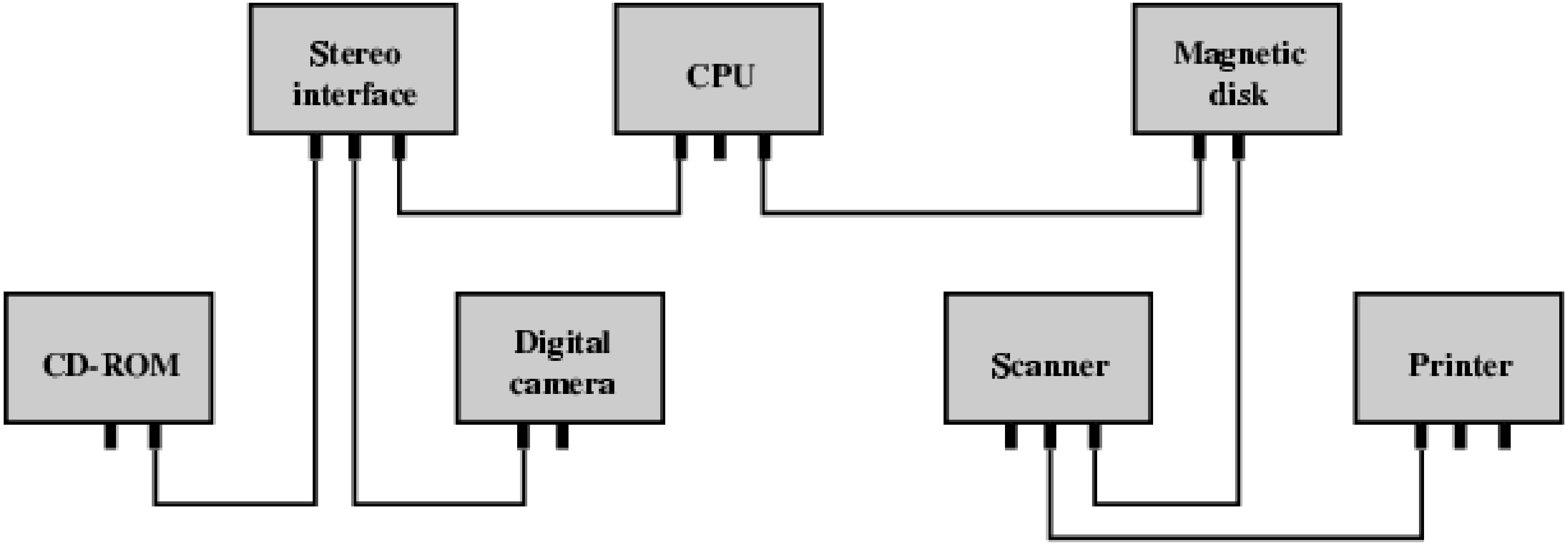
IEEE 1394 FireWire

- High performance serial bus
- Fast
- Low cost
- Easy to implement
- Also being used in digital cameras, VCRs and TV

FireWire Configuration

- Daisy chain
- Up to 63 devices on single port
  - Really 64 of which one is the interface itself
- Up to 1022 buses can be connected with bridges
- Automatic configuration
- No bus terminators
- May be tree structure

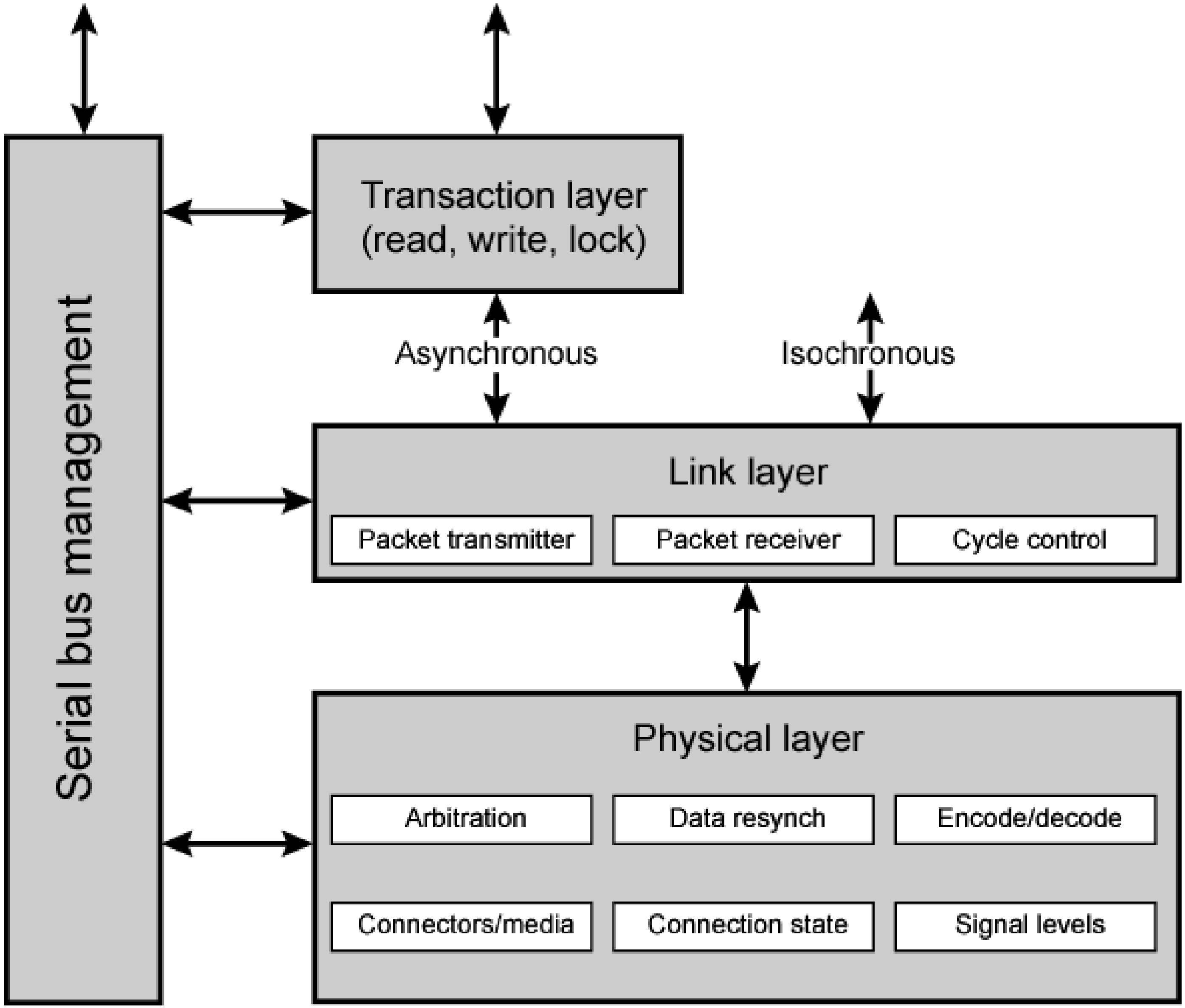
Simple FireWire Configuration



FireWire 3 Layer Stack

- Physical
  - Transmission medium, electrical and signaling characteristics
- Link
  - Transmission of data in packets
- Transaction
  - Request-response protocol

FireWire Protocol Stack



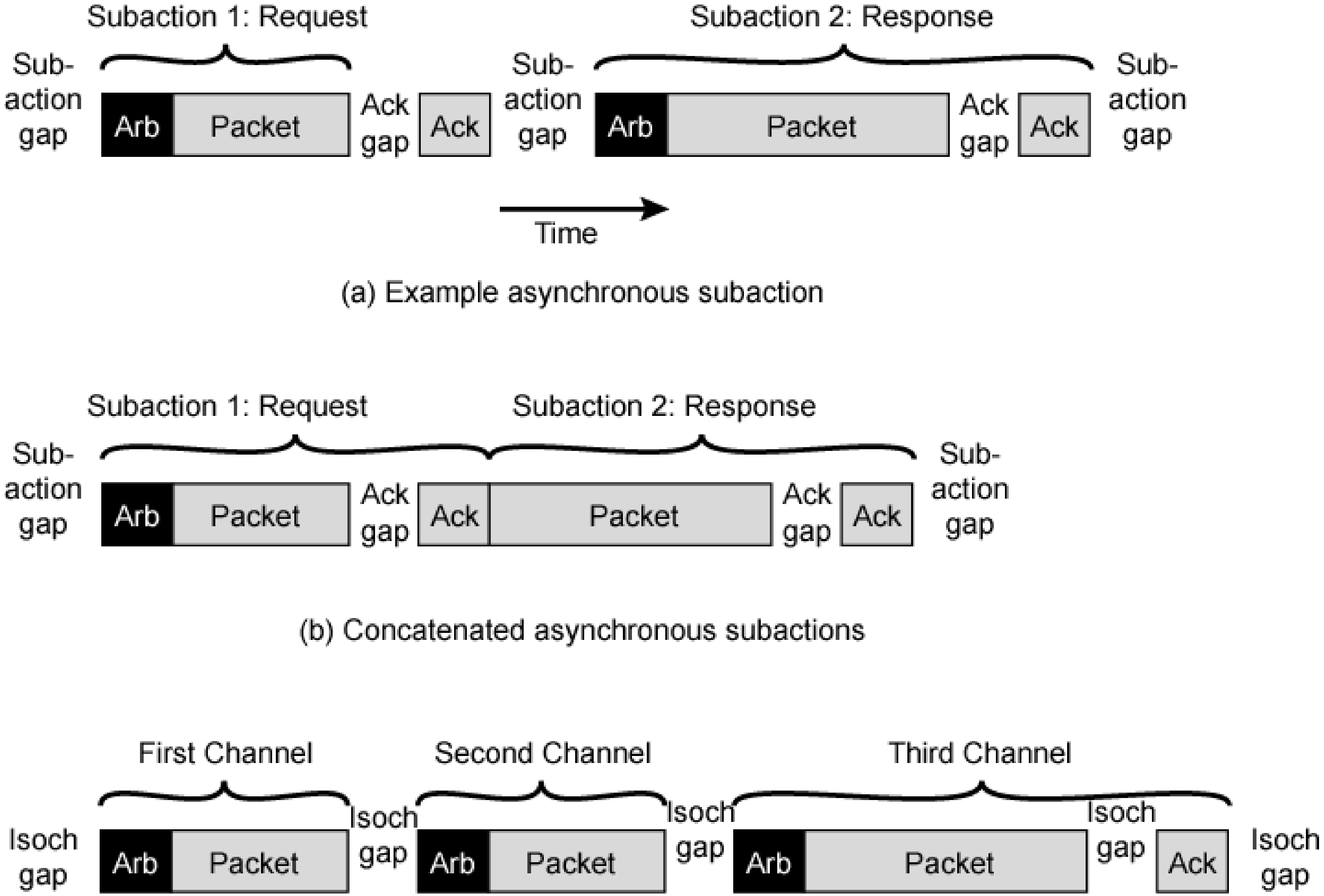
FireWire - Physical Layer

- Data rates from 25 to 400Mbps
- Two forms of arbitration
  - Based on tree structure
  - Root acts as arbiter
  - First come first served
  - Natural priority controls simultaneous requests
    - i.e. who is nearest to root
  - Fair arbitration
  - Urgent arbitration

FireWire - Link Layer

- Two transmission types
  - Asynchronous
    - Variable amount of data and several bytes of transaction data transferred as a packet
    - To explicit address
    - Acknowledgement returned
  - Isochronous
    - Variable amount of data in sequence of fixed size packets at regular intervals
    - Simplified addressing
    - No acknowledgement

FireWire Subactions



(a) Example asynchronous subaction

(b) Concatenated asynchronous subactions

(c) Example isochronous subactions

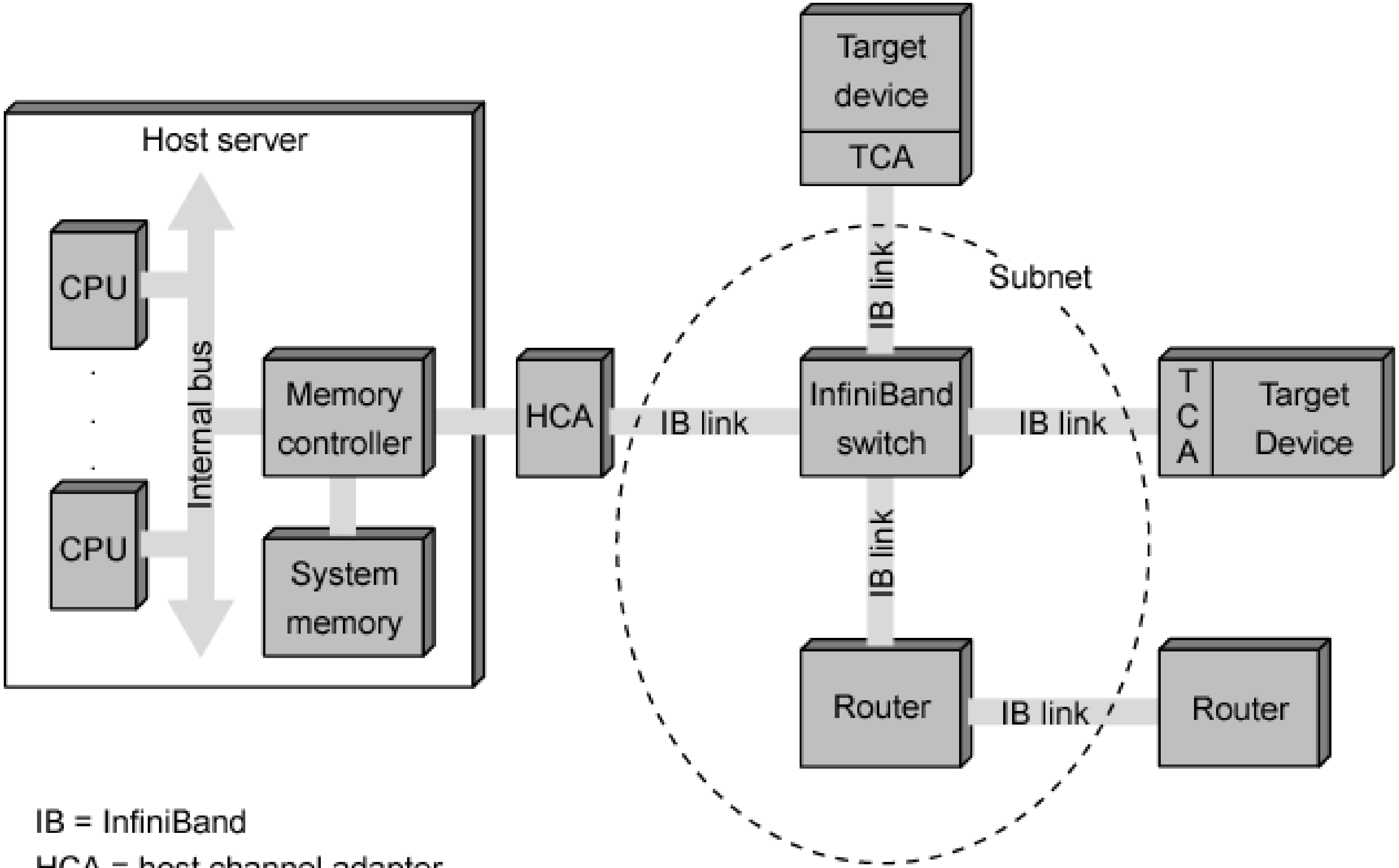
InfiniBand

- I/O specification aimed at high end servers
  - Merger of Future I/O (Cisco, HP, Compaq, IBM) and Next Generation I/O (Intel)
- Version 1 released early 2001
- Architecture and spec. for data flow between processor and intelligent I/O devices
- Intended to replace PCI in servers
- Increased capacity, expandability, flexibility

InfiniBand Architecture

- Remote storage, networking and connection between servers
- Attach servers, remote storage, network devices to central fabric of switches and links
- Greater server density
- Scalable data centre
- Independent nodes added as required
- I/O distance from server up to
  - 17m using copper
  - 300m multimode fibre optic
  - 10km single mode fibre
- Up to 30Gbps

InfiniBand Switch Fabric

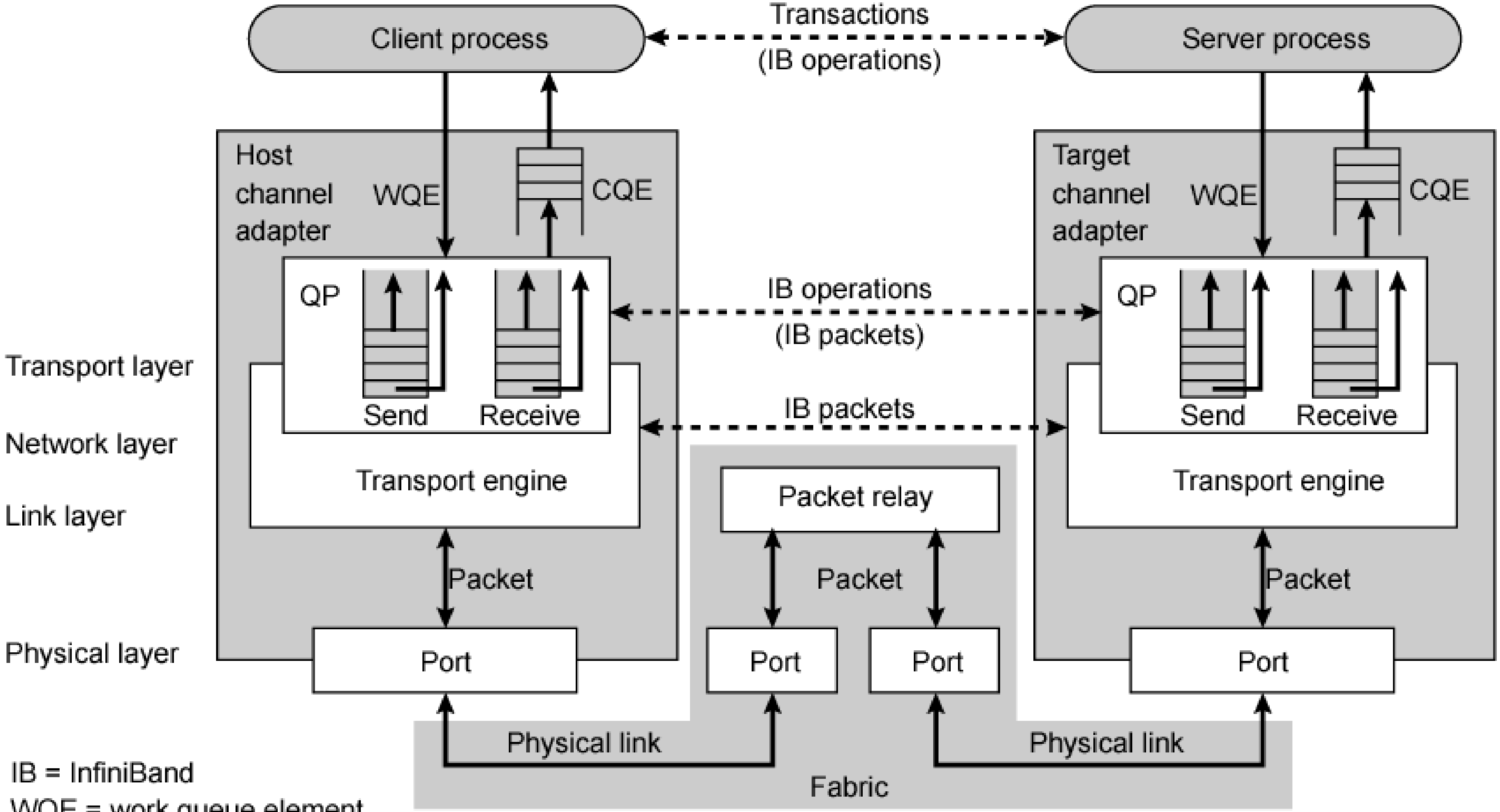


IB = InfiniBand
HCA = host channel adapter
TCA = target channel adapter

InfiniBand Operation

- 16 logical channels (virtual lanes) per physical link
- One lane for management, rest for data
- Data in stream of packets
- Virtual lane dedicated temporarily to end to end transfer
- Switch maps traffic from incoming to outgoing lane

InfiniBand Protocol Stack



IB = InfiniBand
WQE = work queue element
CQE = completion queue entry
QP = queue pair

Foreground Reading

- Check out Universal Serial Bus (USB)
- Compare with other communication standards e.g. Ethernet