COMPUTER SYSTEM ARCHITECTURE

THIRD EDITION

Dr.Chao Tan,
Carnegie Mellon University

M. Morris Mano

Chap. 1: Digital Logic Circuits

- Logic Gates, • Boolean Algebra
- Map Simplification, • Combinational Circuits
- Filp-Flops, • Sequential Circuits

Chap. 2: Digital Components

- Integrated Circuits, • Decoders, • Multiplexers
- Registers, • Shift Registers, • Binary Counters
- Memory Unit

Chap. 3: Data Representation

- Data Types, • Complements
- Fixed Point Representation
- Floating Point Representation
- Other Binary Codes, • Error Detection Codes

Chap. 4: Register Transfer and Microoperations

- Register Transfer Language, • Register Transfer
- Bus and Memory Transfers
- Arithmetic Microoperations
- Logic Microoperations, • Shift Microoperations
- Arithmetic Logic Shift Unit

Chap. 5: 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 Logic

Chap. 6: Programming the Basic Computer

- Machine Language, • Assembly Language
- Assembler, • Program Loops
- Programming Arithmetic and Logic Operations
- Subroutines, • Input-Output Programming

Chap. 7: Microprogrammed Control

- Control Memory, • Sequencing Microinstructions
- Microprogram Example, • Design of Control Unit
- Microinstruction Format

Chap. 8: Central Processing Unit

- General Register Organization
- Stack Organization, • Instruction Formats
- Addressing Modes
- Data Transfer and Manipulation
- Program Control
- Reduced Instruction Set Computer

Chap. 9: Pipeline and Vector Processing

- Parallel Processing, • Pipelining
- Arithmetic Pipeline, • Instruction Pipeline
- RISC Pipeline, • Vector Processing

Chap. 10: Computer Arithmetic

- Arithmetic with Signed-2's Complement Numbers
- Multiplication and Division Algorithms
- Floating-Point Arithmetic Operations
- Decimal Arithmetic Unit
- Decimal Arithmetic Operations

Chap. 11: Input-Output Organization

- Peripheral Devices, • Input-Output Interface
- Asynchronous Data Transfer, • Modes of Transfer
- Priority Interrupt, • Direct Memory Access

Chap. 12: Memory Organization

- Memory Hierarchy, • Main Memory
- Auxiliary Memory. • Associative Memory
- Cache Memory, • Virtual Memory

Chap. 13: Multiprocessors (Δ)

- Characteristics of Multiprocessors
- Interconnection Structures
- Interprocessor Arbitration
- Interprocessor Communication/Synchronization
- Cache Coherence

SIMPLE DIGITAL SYSTEMS

- Combinational and sequential circuits (learned in Chapters 1 and 2) 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

REGISTER TRANSFER AND MICROOPERATIONS

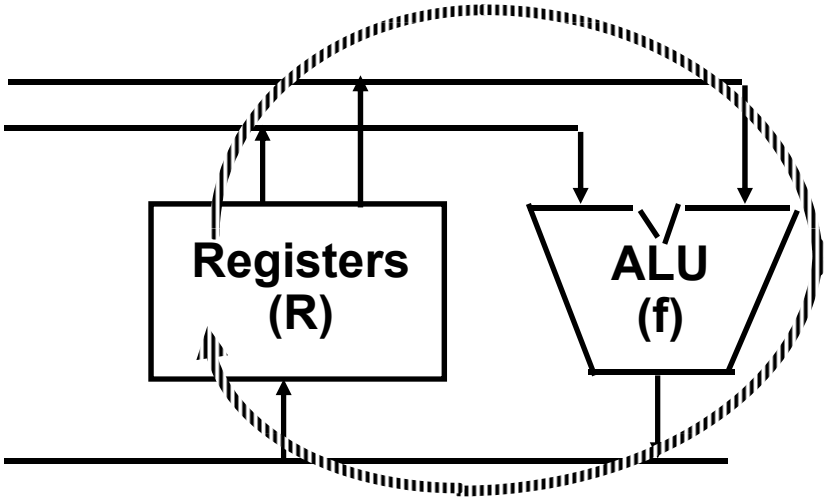
- Register Transfer Language
- Register Transfer
- Bus and Memory Transfers
- Arithmetic Microoperations
- Logic Microoperations
- Shift Microoperations
- Arithmetic Logic Shift Unit

MICROOPERATIONS (1)

- The operations on the data in registers are called microoperations.
- The functions built into registers are examples of microoperations
  - Shift
  - Load
  - Clear
  - Increment
  - …

MICROOPERATION (2)

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



1 clock cycle

$R \leftarrow f(R, R)$

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

  - Microoperations set

    Set of allowable microoperations provided by the organization of the computer

  - Control signals that initiate the sequence of microoperations (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 systems.

DESIGNATION OF REGISTERS

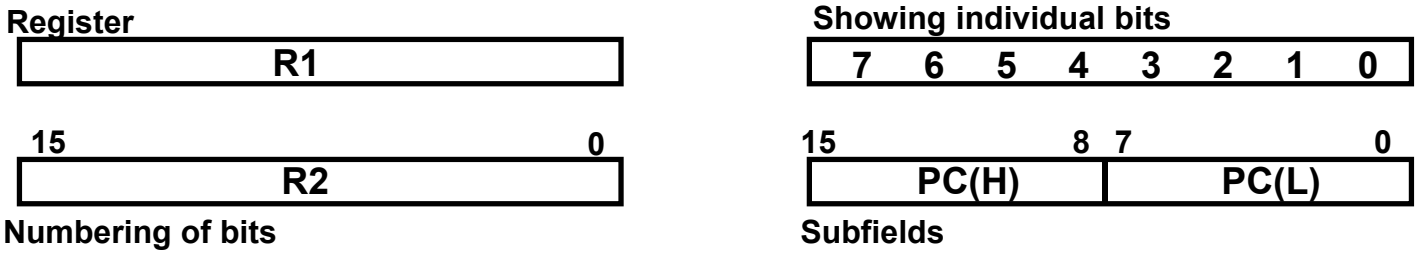
- Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)
- Often the names indicate function:
  - MAR - memory address register
  - PC - program counter
  - IR - instruction register
- Registers and their contents can be viewed and represented in *various ways*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

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

DESIGNATION OF REGISTERS

- 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

  $R2 \leftarrow R1$

  – In this case the contents of register R2 are copied (loaded) into register R1
  – 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

REGISTER TRANSFER

- **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

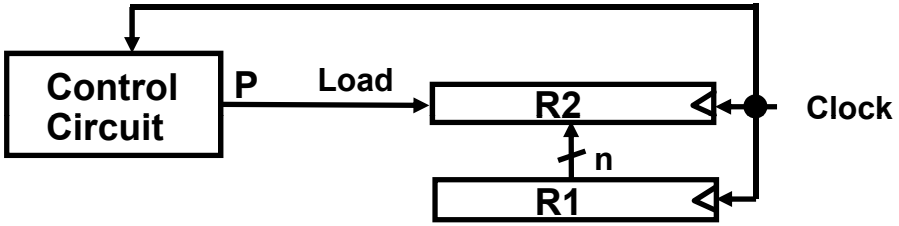
Which means "if P = 1, then load the contents of register R1 into register R2", i.e., if (P = 1) then (R2 ← R1)

HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

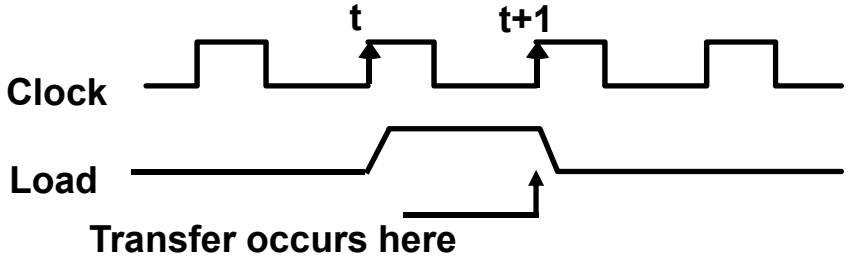
Implementation of controlled transfer

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

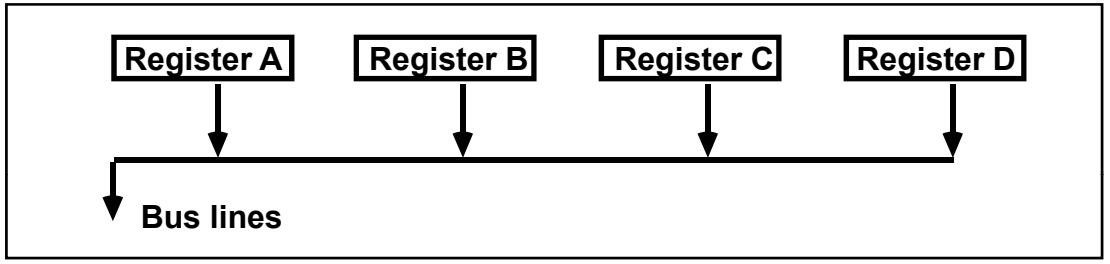
- To completely connect n registers → n(n-1) lines
- $O(n^2)$ cost
  - This is not a realistic approach to use in a large digital system

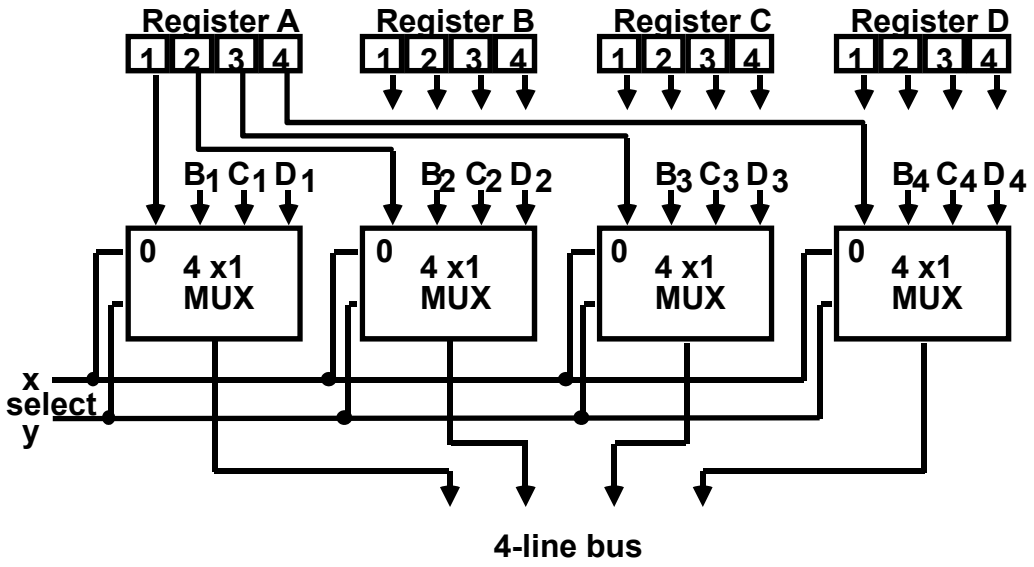
- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the bus
- Have control circuits to select which register is the source, and which is the destination

BUS AND BUS TRANSFER

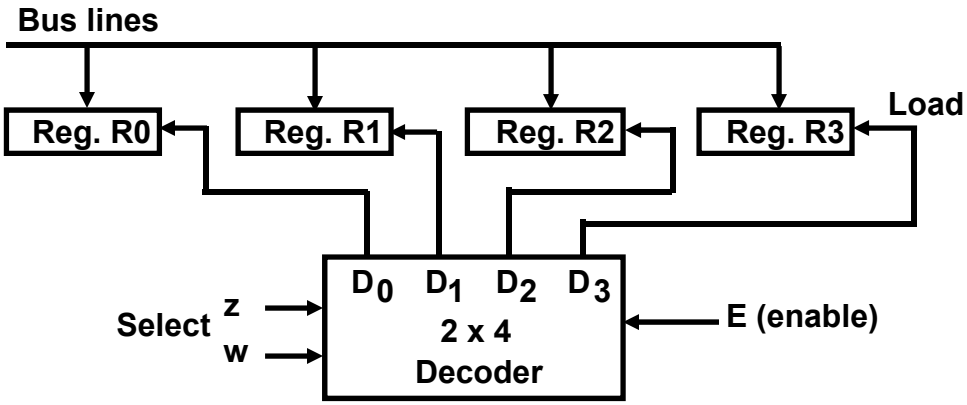
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

From a register to bus: BUS ← R





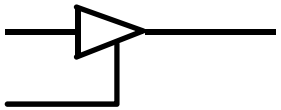
TRANSFER FROM BUS TO A DESTINATION REGISTER



Three-State Bus Buffers

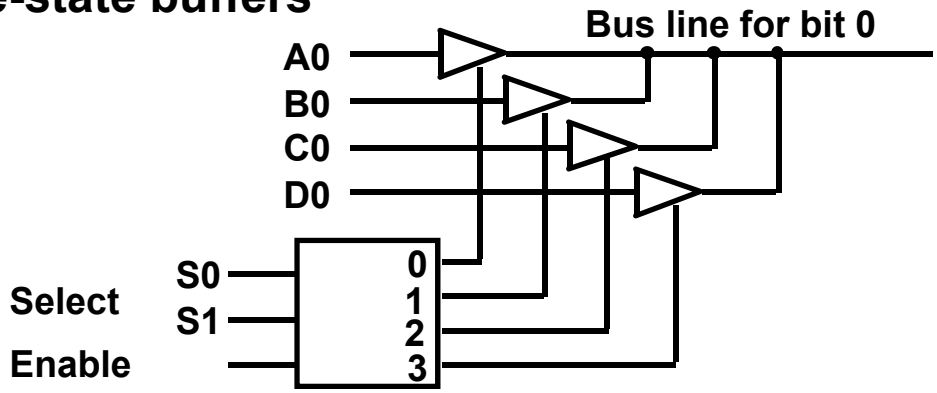
Normal input A

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

  $R2 \leftarrow R1$

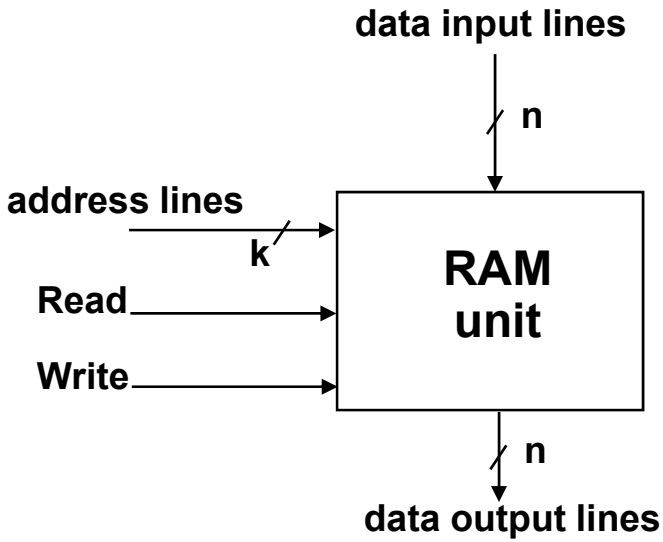
  or

  $BUS \leftarrow R1, R2 \leftarrow BUS$

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

MEMORY (RAM)

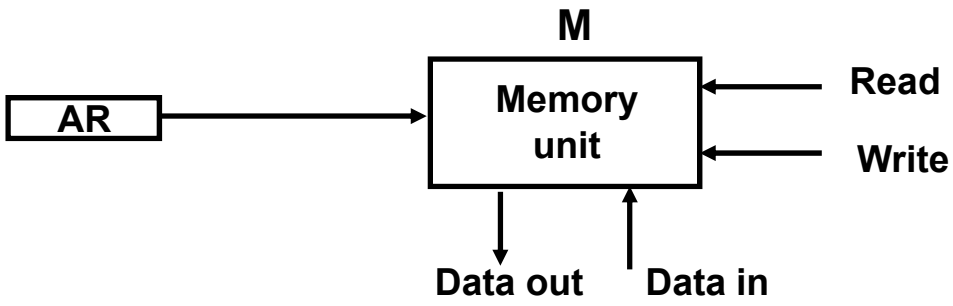
- Memory (RAM) can be thought as a sequential circuits containing some number of registers
- These registers hold the *words* of memory
- Each of the r registers is indicated by an *address*
- These addresses range from 0 to r-1
- Each register (word) can hold n bits of data
- Assume the RAM contains 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

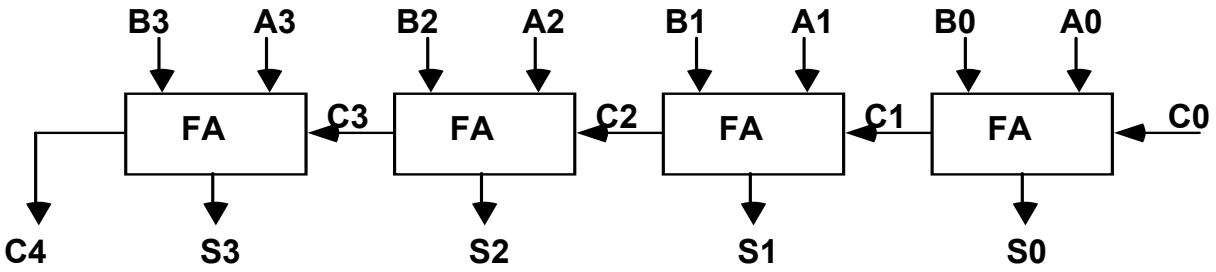
- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

Summary of Typical Arithmetic Micro-Operations

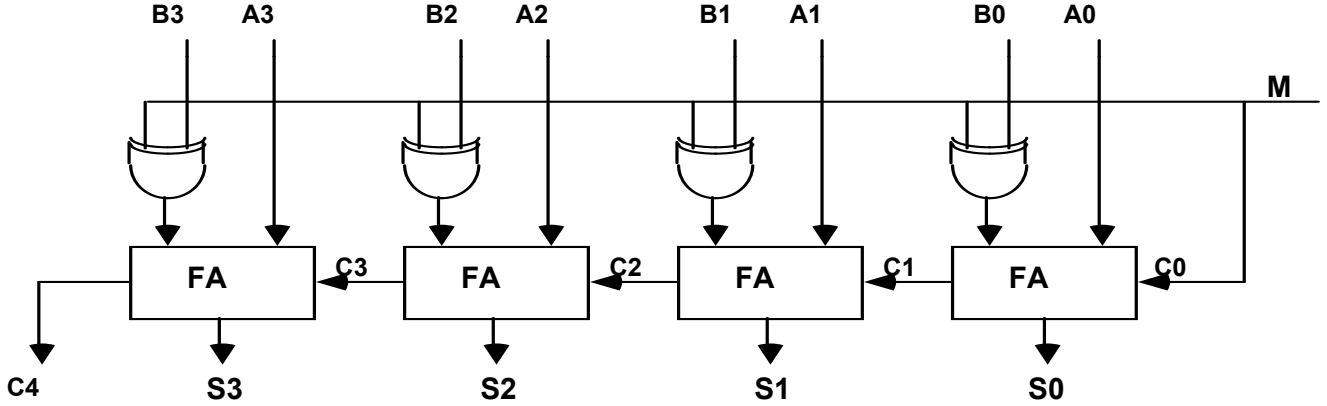
R3 ← R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ← R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ← R2’	Complement the contents of R2
R2 ← R2’+ 1	2's complement the contents of R2 (negate)
R3 ← R1 + R2’+ 1	subtraction
R1 ← R1 + 1	Increment
R1 ← R1 - 1	Decrement

BINARY ADDER / SUBTRACTOR / INCREMENTER

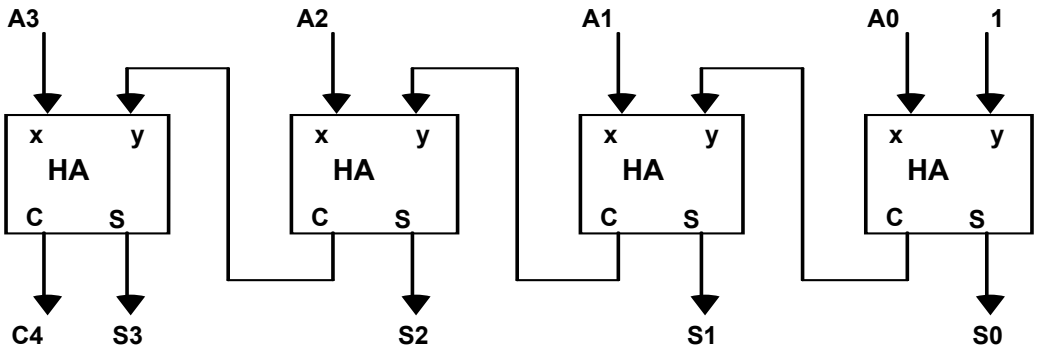
Binary Adder



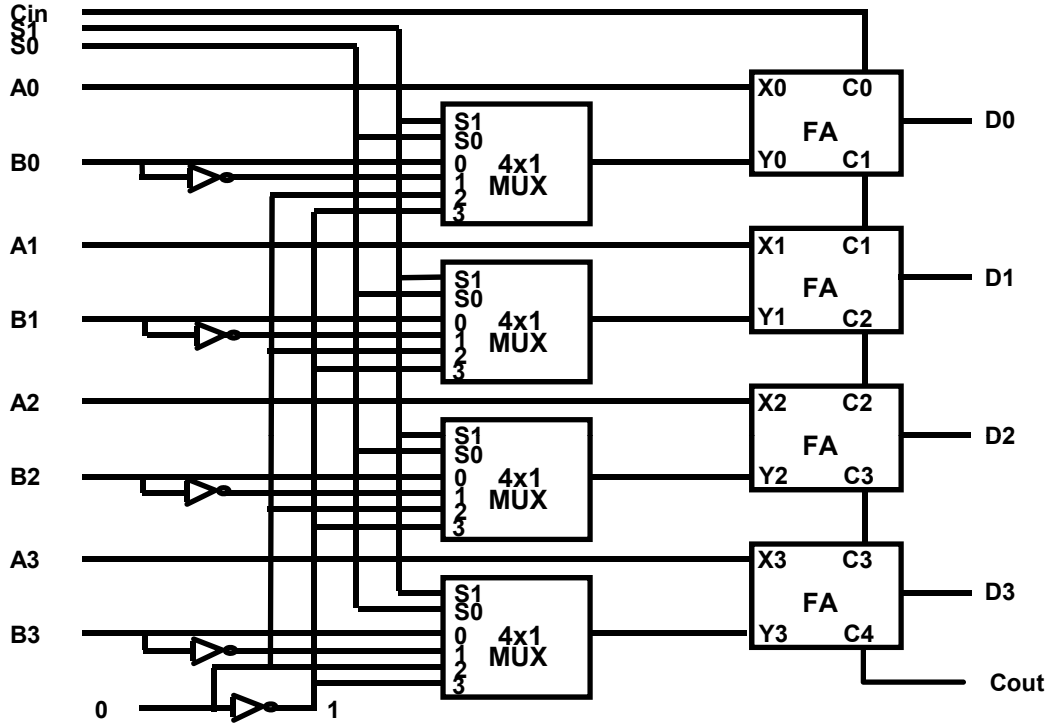
Binary Adder-Subtractor



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 microoperations 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	F_0	F_1	F_2	…	F_{13}	F_{14}	F_{15}
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
- The others can be created from combination of these

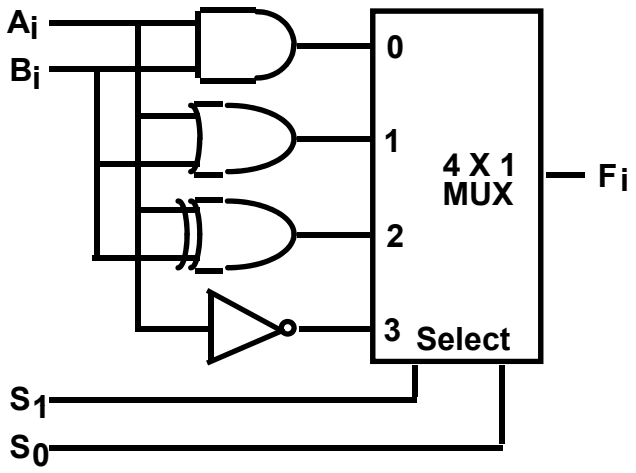
LIST OF LOGIC MICROOPERATIONS

- **List of Logic Microoperations**
  - 16 different logic operations with 2 binary vars.
  - n binary vars $\rightarrow 2^{2^n}$ functions

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

x	0 0 1 1	*Boolean Function*	*Micro-Operations*	*Name*
y	0 1 0 1			
	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

S_1	S_0	Output	μ-operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	F = A'	Complement

APPLICATIONS OF LOGIC MICROOPERATIONS

- **Logic microoperations 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	$A \leftarrow A \cdot B'$
– Mask (Delete)	$A \leftarrow A \cdot B$
– Clear	$A \leftarrow A \oplus B$
– Insert	$A \leftarrow (A \cdot B) + C$
– Compare	$A \leftarrow A \oplus B$
– . . .	

SELECTIVE SET

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

1 1 0 0	A_t	
1 0 1 0	B	
1 1 1 0	A_{t+1}	(A $\leftarrow$ 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	A_t	
1 0 1 0	B	
0 1 1 0	A_{t+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	A_t
1 0 1 0	B
0 1 0 0	A_{t+1}

$(A \leftarrow A \cdot B')$

- 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	A_t
1 0 1 0	B
1 0 0 0	A_{t+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	A_t	
1 0 1 0	B	
0 1 1 0	A_{t+1}	$(A \leftarrow A \oplus B)$

INSERT OPERATION

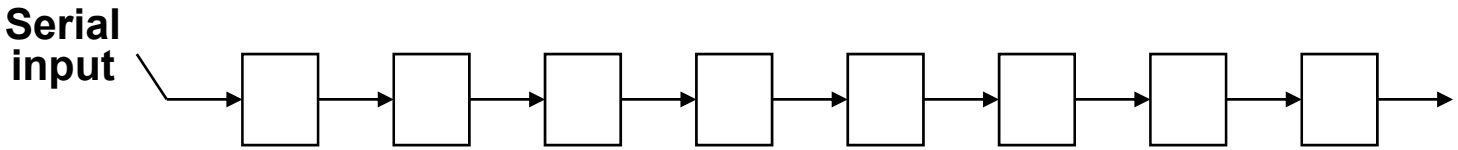
- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged
- This is done as
  - A mask operation to clear the desired bit positions, followed by
  - An OR operation to introduce the new bits into the desired positions
  - Example
    - » 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)
```

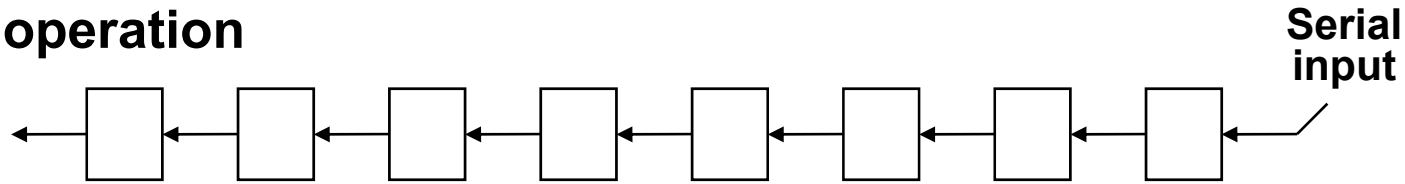
SHIFT MICROOPERATIONS

- 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

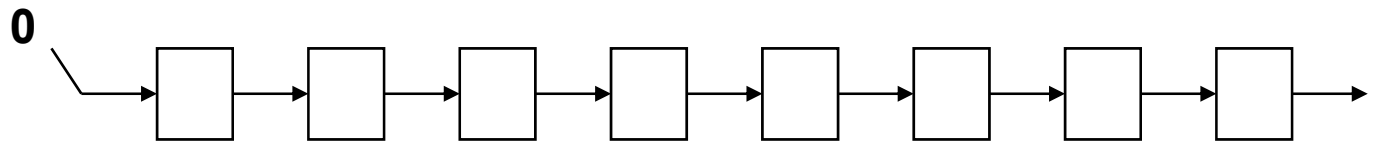


- A left shift operation

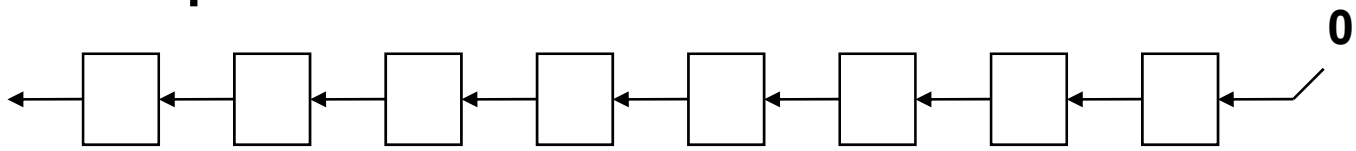


LOGICAL SHIFT

- In a logical shift the serial input to the shift is a 0.
- A right logical shift operation:



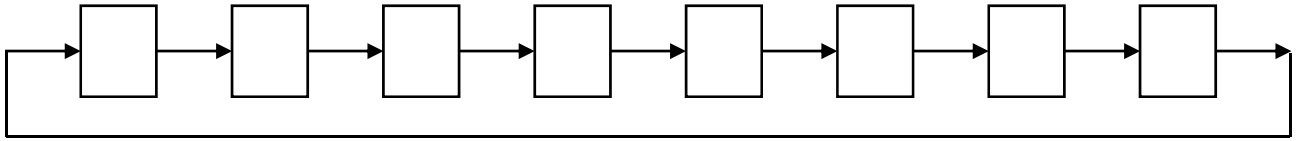
- A left logical shift operation:



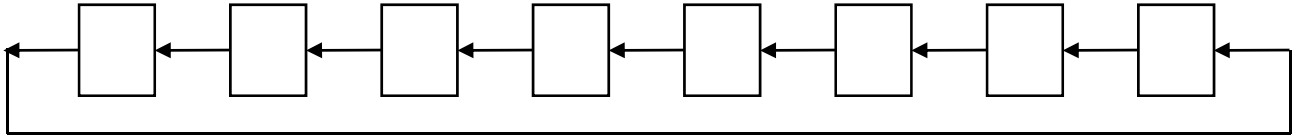
- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right
  - Examples:
    - » R2 ← *shr* R2
    - » R3 ← *shl* R3

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:

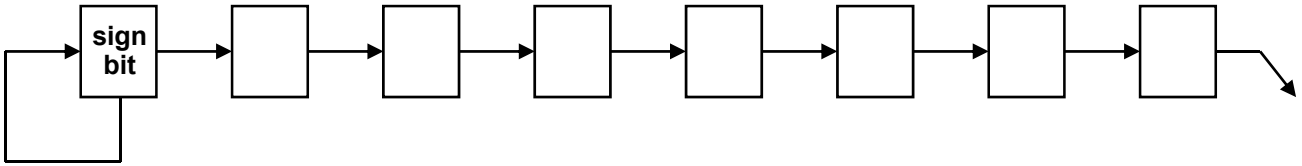


- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - » R2 ← *cir* R2
    - » R3 ← *cil* R3

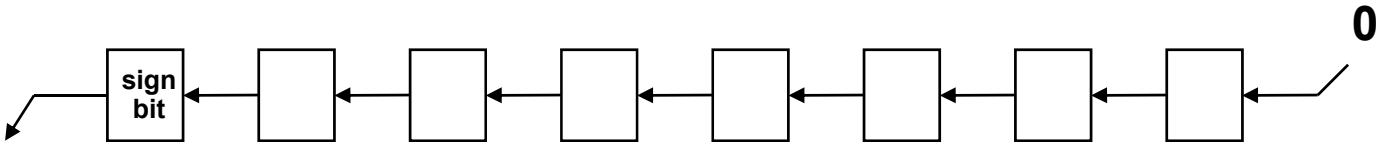
ARITHMETIC SHIFT

- 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:

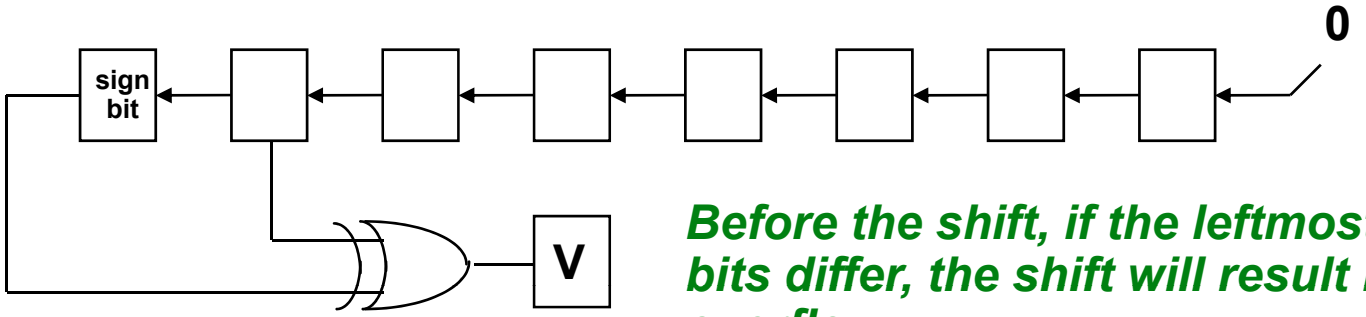


- A left arithmetic shift operation:



ARITHMETIC SHIFT

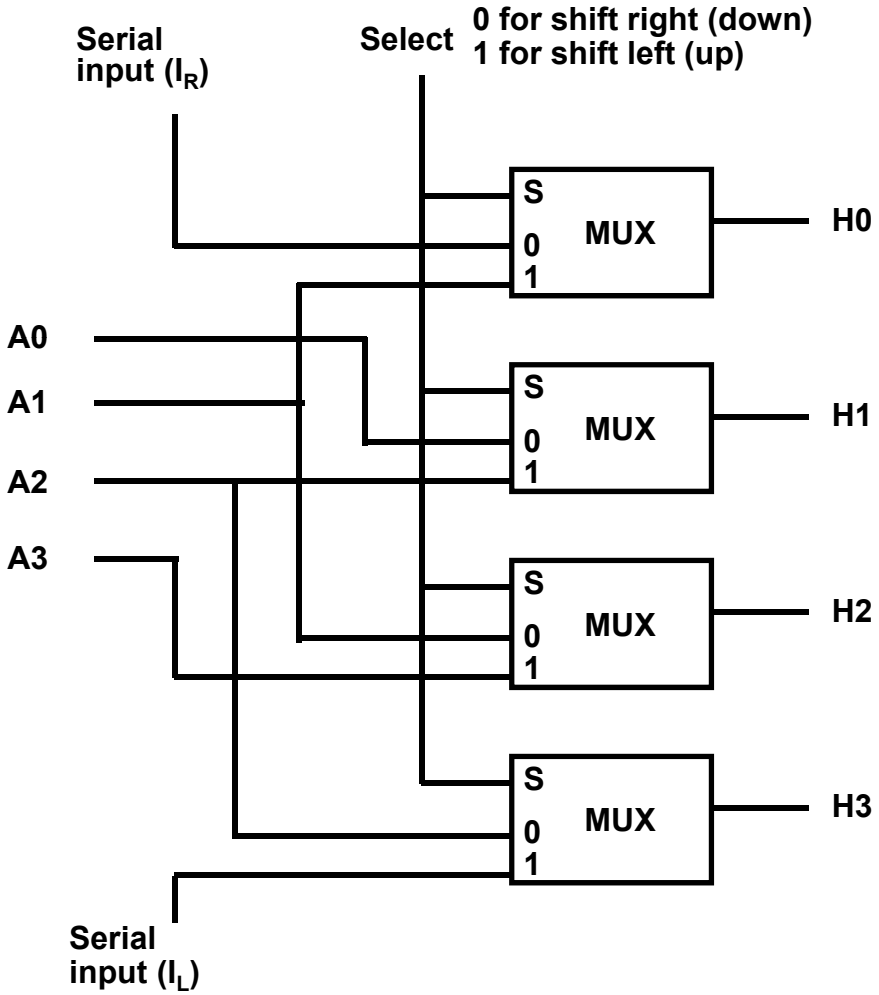
- An left arithmetic shift operation must be checked for the overflow



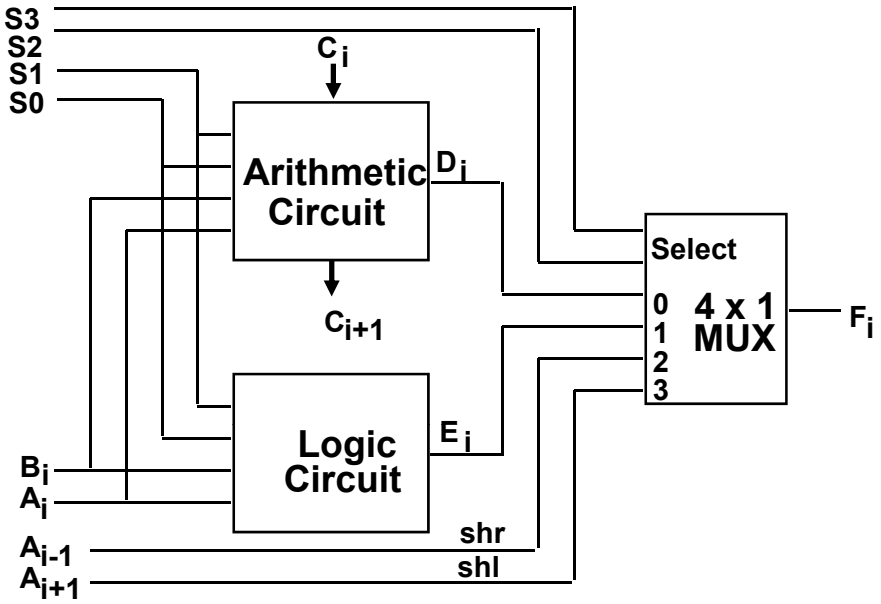
Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- 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

BASIC COMPUTER ORGANIZATION AND DESIGN

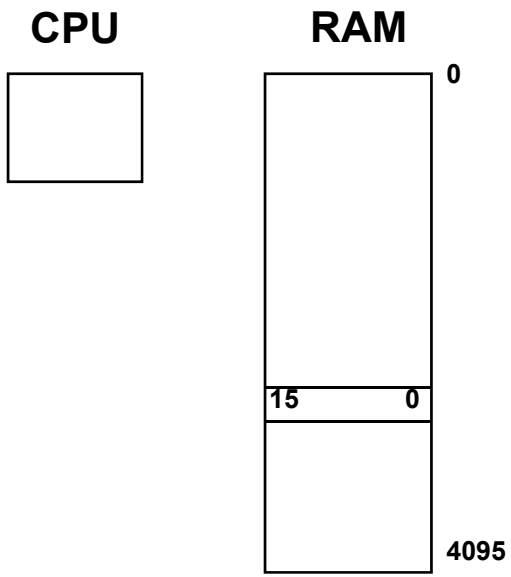
- Instruction Codes
- Computer Registers
- Computer Instructions
- Timing and Control
- Instruction Cycle
- Memory Reference Instructions
- Input-Output and Interrupt
- Complete Computer Description
- Design of Basic Computer
- Design of Accumulator Logic

INTRODUCTION

- Every different processor type has its own design (different registers, buses, microoperations, machine instructions, etc)
- Modern processor is a very complex device
- It contains
  - Many registers
  - Multiple arithmetic units, for both integer and floating point calculations
  - The ability to pipeline several consecutive instructions to speed execution
  - Etc.
- However, to understand how processors work, we will start with a simplified processor model
- This is similar to what real processors were like ~25 years ago
- M. Morris Mano introduces a simple processor model he calls the *Basic Computer*
- We will use this to introduce processor organization and the relationship of the RTL model to the higher level computer processor

THE BASIC COMPUTER

- The Basic Computer has two components, a processor and memory
- The memory has 4096 words in it
  - 4096 = 2^{12}, so it takes 12 bits to select a word in memory
- Each word is 16 bits long



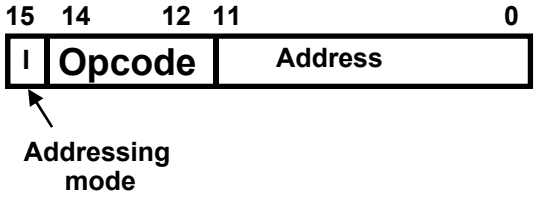
INSTRUCTIONS

- **Program**
  - **A sequence of (machine) instructions**
- **(Machine) Instruction**
  - **A group of bits that tell the computer to *perform a specific operation* (a sequence of micro-operation)**
- **The instructions of a program, along with any needed data are stored in memory**
- **The CPU reads the next instruction from memory**
- **It is placed in an *Instruction Register* (IR)**
- **Control circuitry in control unit then translates the instruction into the sequence of microoperations necessary to implement it**

INSTRUCTION FORMAT

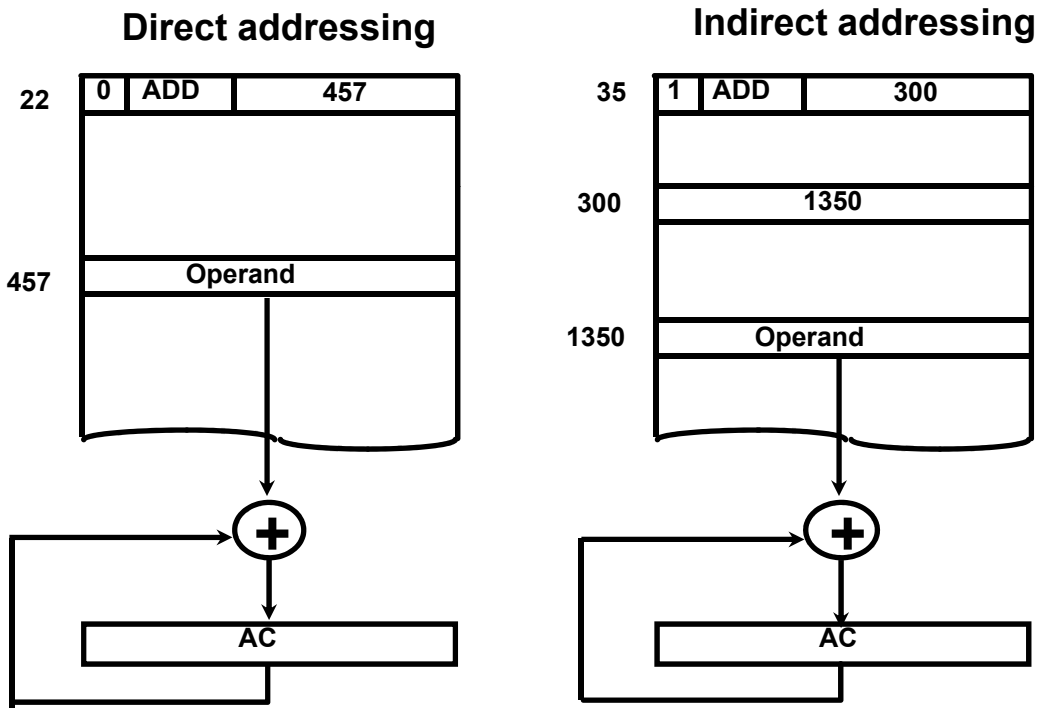
- A computer instruction is often divided into two parts
  - An *opcode* (Operation Code) that specifies the operation for that instruction
  - An *address* that specifies the registers and/or locations in memory to use for that operation
- In the Basic Computer, since the memory contains 4096 (= 2^{12}) words, we needs 12 bit to specify which memory address this instruction will use
- In the Basic Computer, bit 15 of the instruction specifies the *addressing mode* (0: direct addressing, 1: indirect addressing)
- Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's opcode

Instruction Format



ADDRESSING MODES

- The address field of an instruction can represent either
  - Direct address: the address in memory of the data to use (the address of the operand), or
  - Indirect address: the address in memory of the address in memory of the data to use



- Effective Address (EA)
  - The address, that can be directly used without modification to access an operand for a computation-type instruction, or as the target address for a branch-type instruction

PROCESSOR REGISTERS

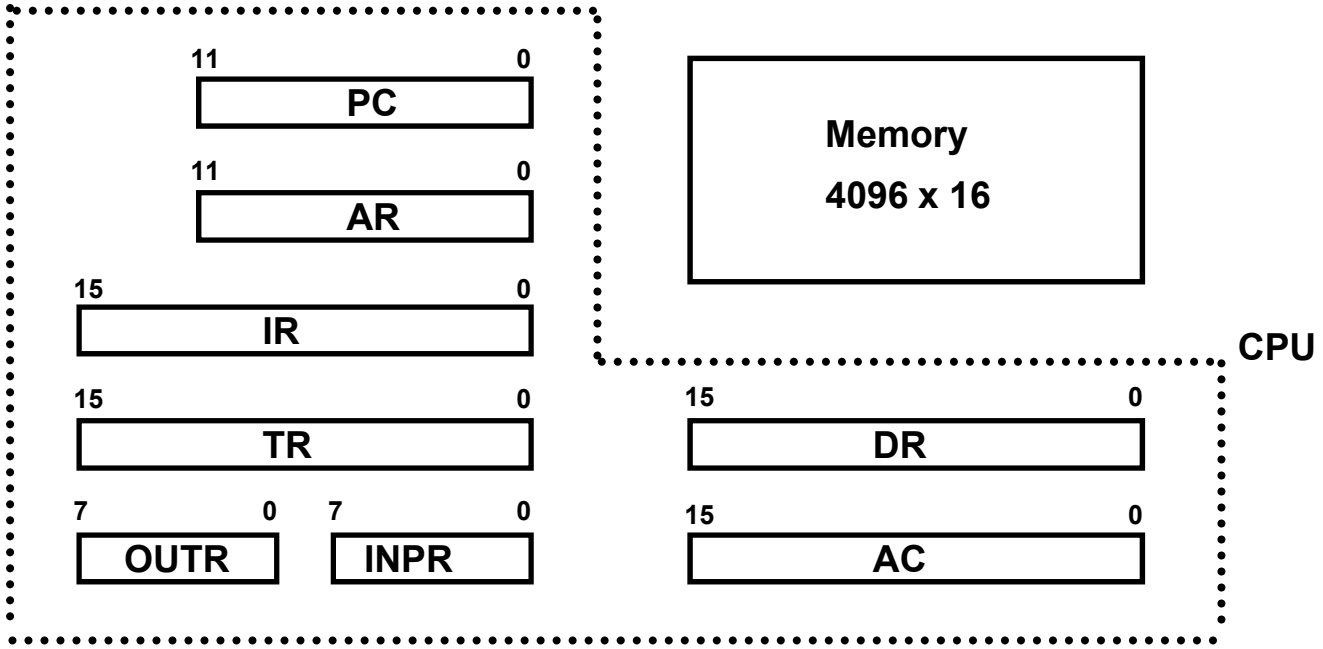
- A processor has many registers to hold instructions, addresses, data, etc
- The processor has a register, the *Program Counter* (PC) that holds the memory address of the next instruction to get
  - Since the memory in the Basic Computer only has 4096 locations, the PC only needs 12 bits
- In a direct or indirect addressing, the processor needs to keep track of what locations in memory it is addressing: The *Address Register* (AR) is used for this
  - The AR is a 12 bit register in the Basic Computer
- When an operand is found, using either direct or indirect addressing, it is placed in the *Data Register* (DR). The processor then uses this value as data for its operation
- The Basic Computer has a single *general purpose register* – the *Accumulator* (AC)

PROCESSOR REGISTERS

- The significance of a general purpose register is that it can be referred to in instructions
  - e.g. load AC with the contents of a specific memory location; store the contents of AC into a specified memory location
- Often a processor will need a scratch register to store intermediate results or other temporary data; in the Basic Computer this is the *Temporary Register* (TR)
- The Basic Computer uses a very simple model of input/output (I/O) operations
  - Input devices are considered to send 8 bits of character data to the processor
  - The processor can send 8 bits of character data to output devices
- The *Input Register* (INPR) holds an 8 bit character gotten from an input device
- The *Output Register* (OUTR) holds an 8 bit character to be send to an output device

BASIC COMPUTER REGISTERS

Registers in the Basic Computer



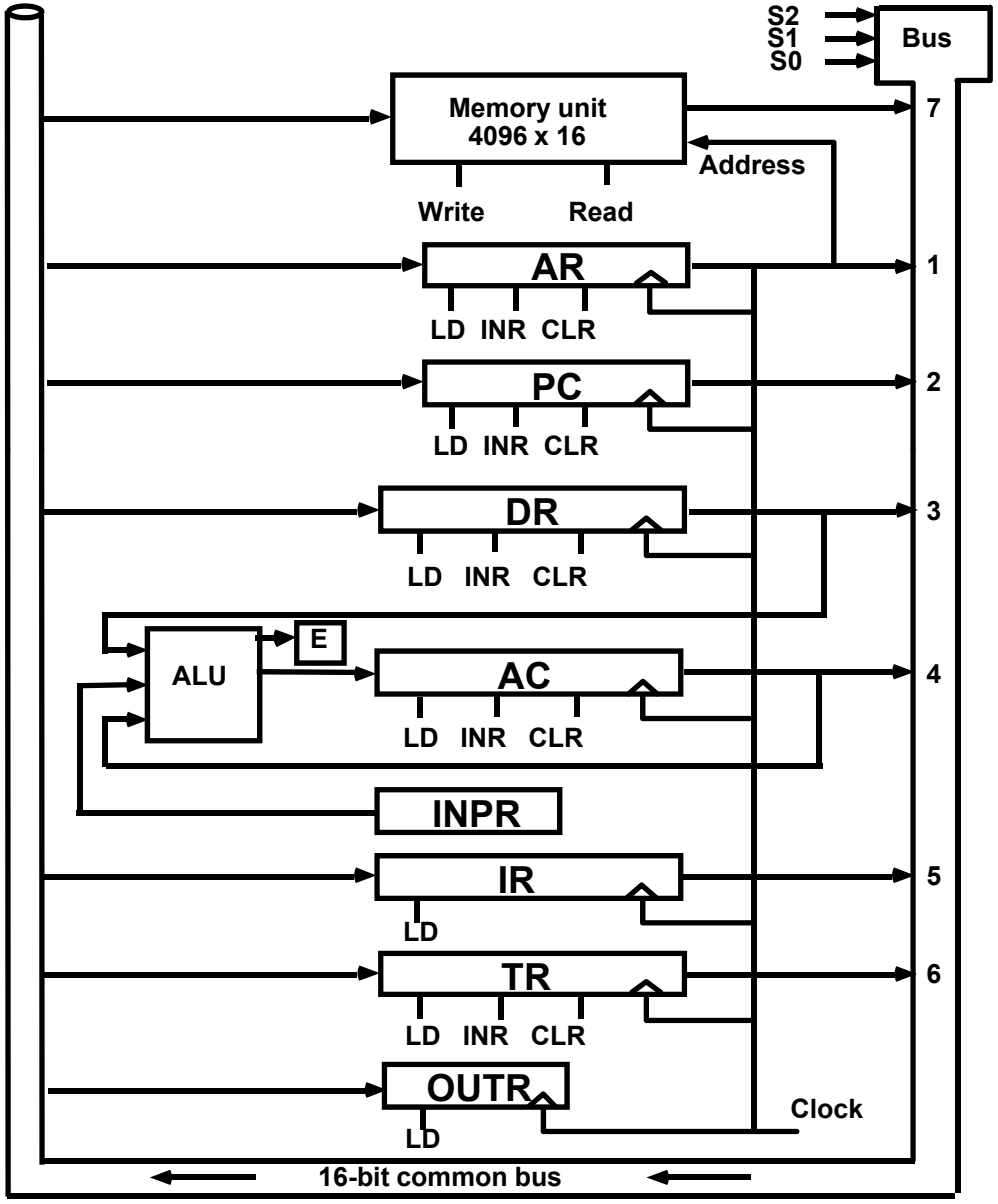
List of BC Registers

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

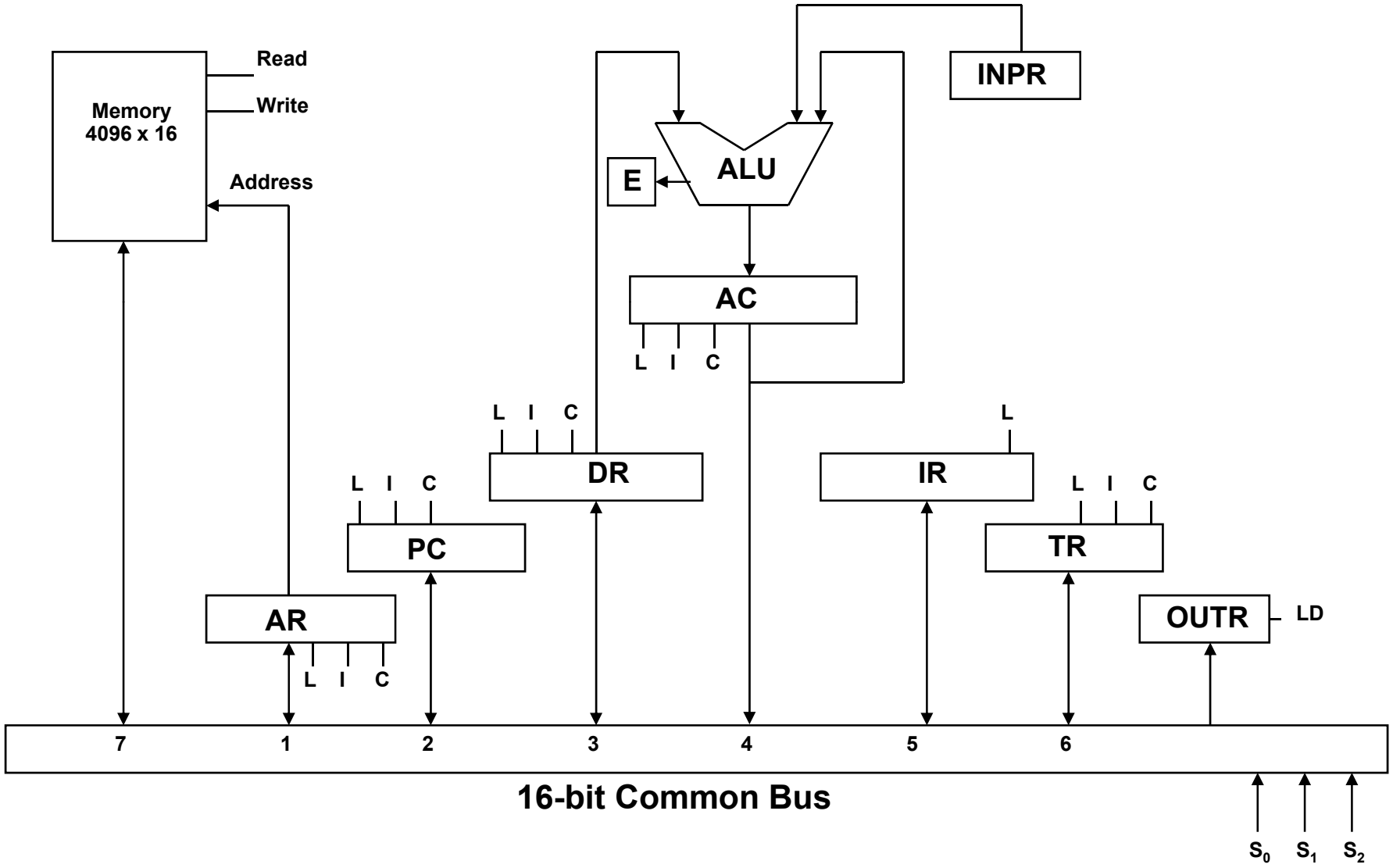
COMMON BUS SYSTEM

- The registers in the Basic Computer are connected using a bus
- This gives a savings in circuitry over complete connections between registers

COMMON BUS SYSTEM



COMMON BUS SYSTEM



COMMON BUS SYSTEM

- Three control lines, S_2, S_1, and S_0 control which register the bus selects as its input

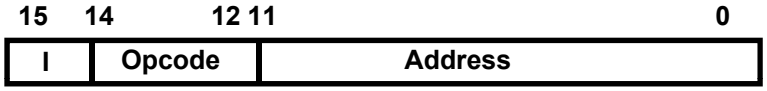
S_2	S_1	S_0	Register
0	0	0	x
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

- Either one of the registers will have its load signal activated, or the memory will have its read signal activated
  - Will determine where the data from the bus gets loaded
- The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions
- When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus

BASIC COMPUTER INSTRUCTIONS

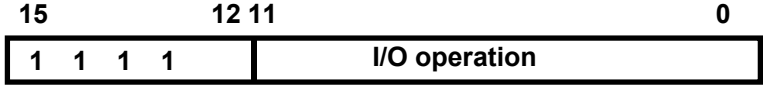
- **Basic Computer Instruction Format**

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

Symbol	Hex Code I = 0	Hex Code 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

INSTRUCTION SET COMPLETENESS

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

- **Instruction Types**

  Functional Instructions
  - **Arithmetic, logic, and shift instructions**
  - **ADD, CMA, INC, CIR, CIL, AND, CLA**

  Transfer Instructions
  - **Data transfers between the main memory and the processor registers**
  - **LDA, STA**

  Control Instructions
  - **Program sequencing and control**
  - **BUN, BSA, ISZ**

  Input/Output Instructions
  - **Input and output**
  - **INP, OUT**

CONTROL UNIT

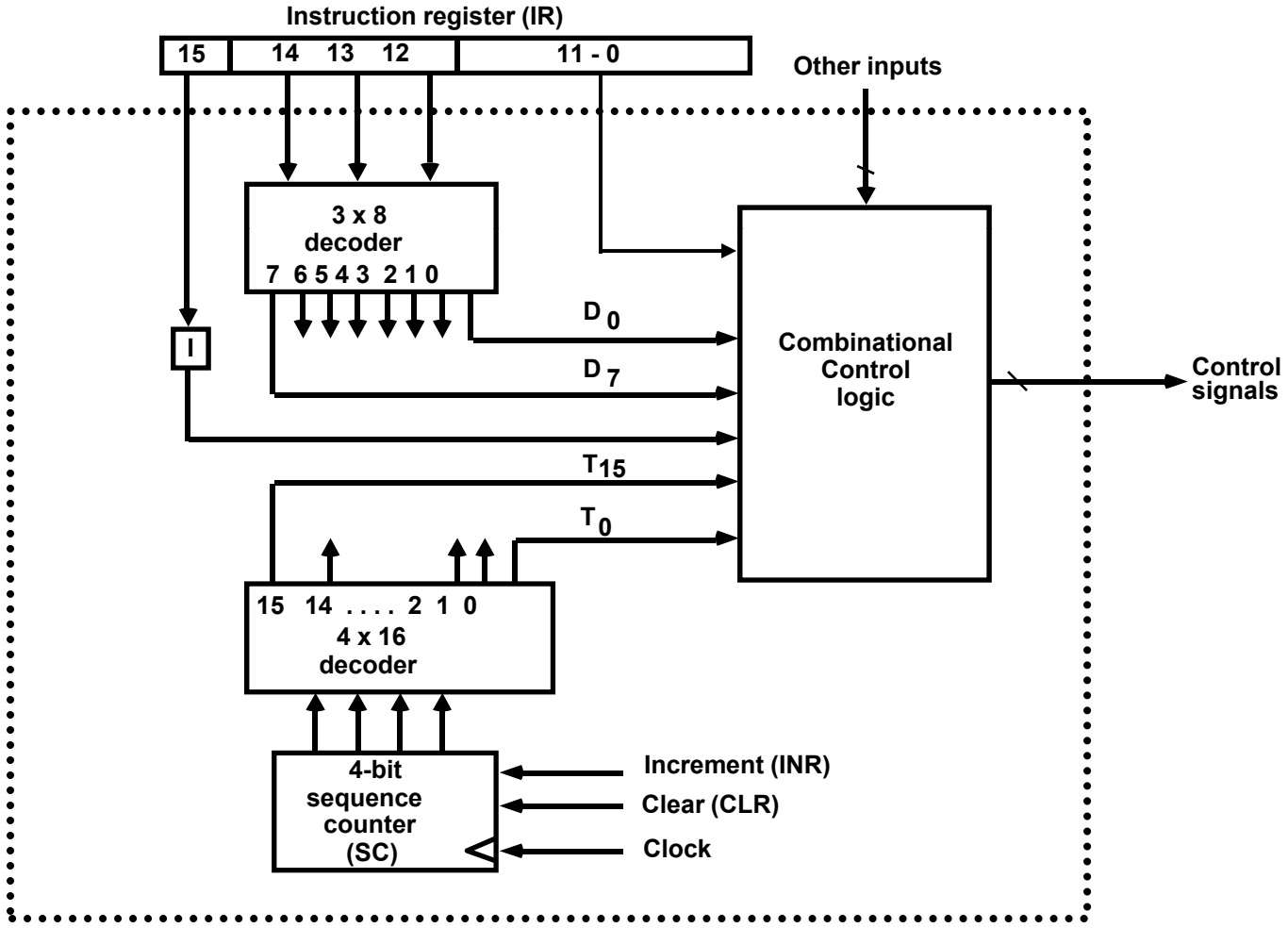
- Control unit (CU) of a processor translates from machine instructions to the control signals for the microoperations that implement them

- Control units are implemented in one of two ways
- *Hardwired* Control
  - CU is made up of sequential and combinational circuits to generate the control signals
- *Microprogrammed* Control
  - A control memory on the processor contains microprograms that activate the necessary control signals

- We will consider a hardwired implementation of the control unit for the Basic Computer

TIMING AND CONTROL

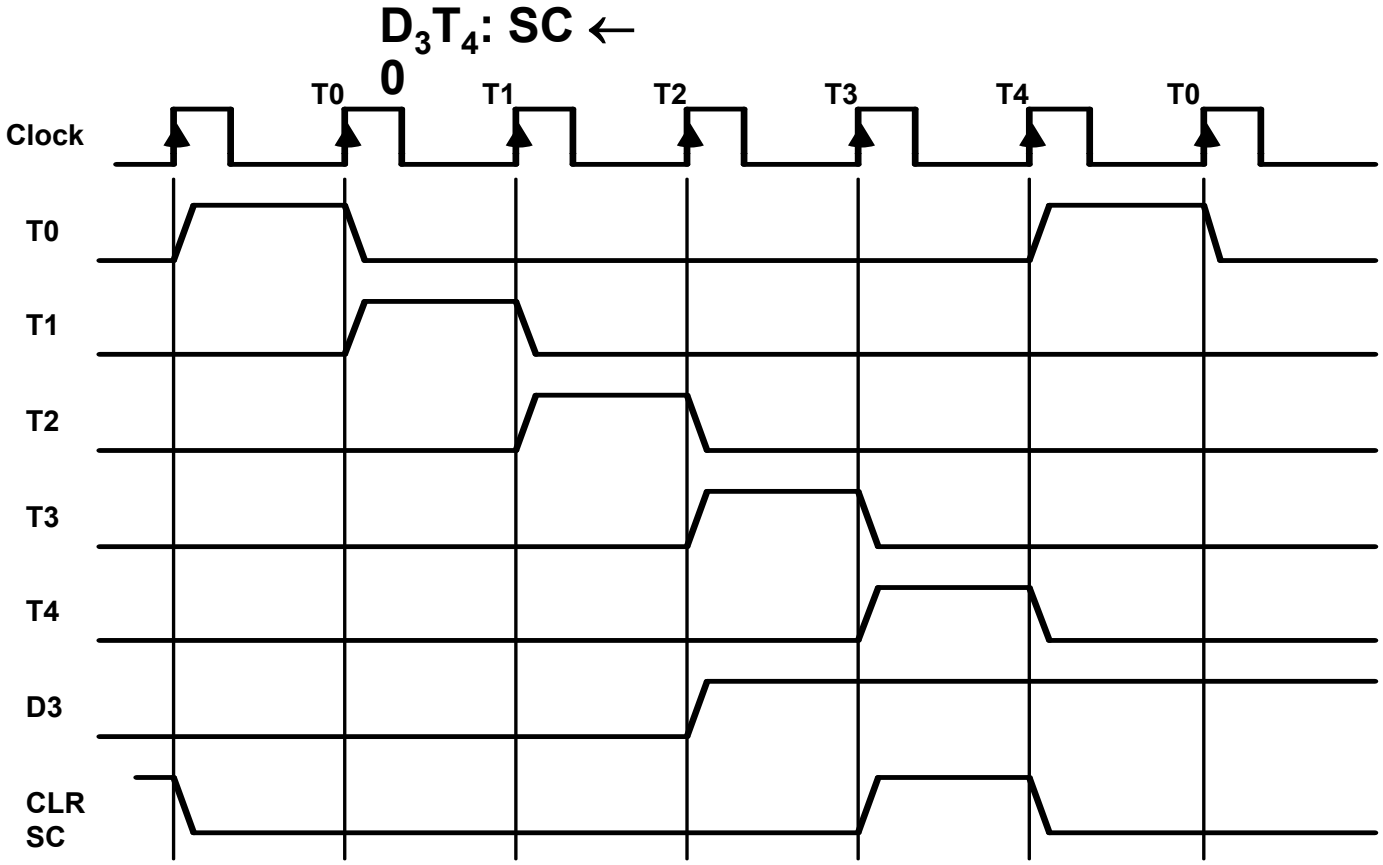
Control unit of Basic Computer



TIMING SIGNALS

- Generated by 4-bit sequence counter and 4×16 decoder
- The SC can be incremented or cleared.

- Example: $T_0, T_1, T_2, T_3, T_4, T_0, T_1, \ldots$
  Assume: At time T_4, SC is cleared to 0 if decoder output D3 is active.



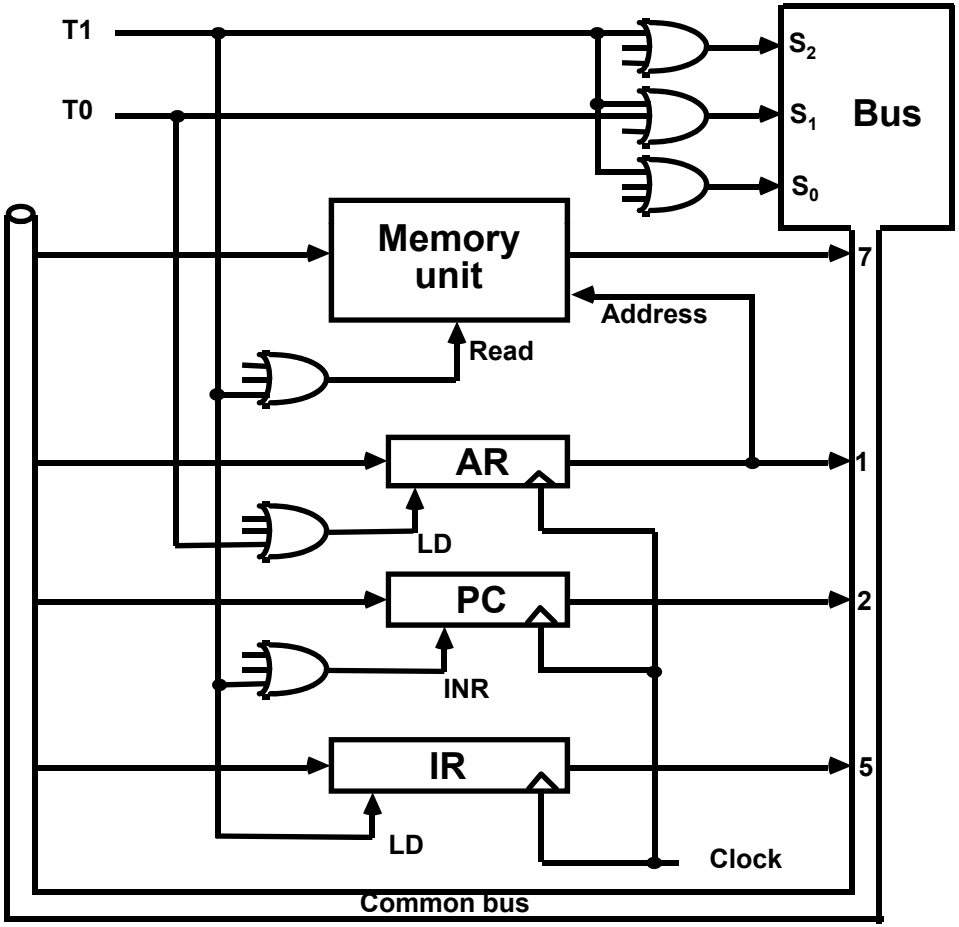
INSTRUCTION CYCLE

- In Basic Computer, a machine instruction is executed in the following cycle:
  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
- After an instruction is executed, the cycle starts again at step 1, for the next instruction
- *Note*: Every different processor has its own (different) instruction cycle

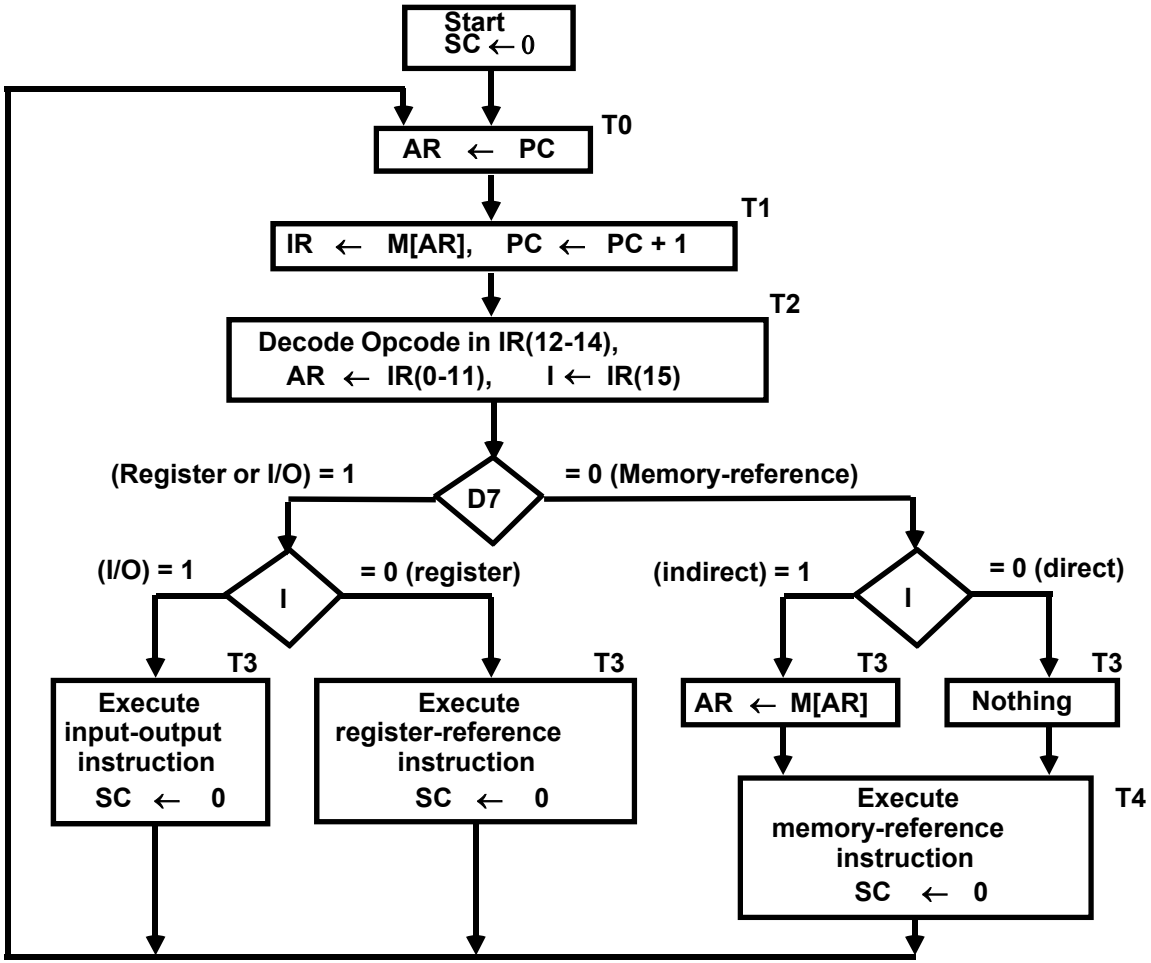
FETCH and DECODE

- **Fetch and Decode**

T0: AR ← PC ($S_0S_1S_2$=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



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

- $D_7 = 1$, $I = 0$
- Register Ref. Instr. is specified in $b_0 \sim b_{11}$ of IR
- Execution starts with timing signal T_3

$r = D_7 I'T_3$ => Register Reference Instruction
$B_i = IR(i)$, i=0,1,2,...,11

	r:	SC ← 0
CLA	rB_{11}:	AC ← 0
CLE	rB_{10}:	E ← 0
CMA	rB_9:	AC ← AC'
CME	rB_8:	E ← E'
CIR	rB_7:	AC ← shr AC, AC(15) ← E, E ← AC(0)
CIL	rB_6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB_5:	AC ← AC + 1
SPA	rB_4:	if (AC(15) = 0) then (PC ← PC+1)
SNA	rB_3:	if (AC(15) = 1) then (PC ← PC+1)
SZA	rB_2:	if (AC = 0) then (PC ← PC+1)
SZE	rB_1:	if (E = 0) then (PC ← PC+1)
HLT	rB_0:	S ← 0 (S is a start-stop flip-flop)

MEMORY REFERENCE INSTRUCTIONS

Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, 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 T_2 when I = 0, or during timing signal T_3 when I = 1
- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of MR instruction starts with T_4

AND to AC

D_0T_4: $DR \leftarrow M[AR]$ Read operand

D_0T_5: $AC \leftarrow AC \wedge DR, SC \leftarrow 0$ AND with AC

ADD to AC

D_1T_4: $DR \leftarrow M[AR]$ Read operand

D_1T_5: $AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$ Add to AC and store carry in E

MEMORY REFERENCE INSTRUCTIONS

LDA: Load to AC

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ← 0

STA: Store AC

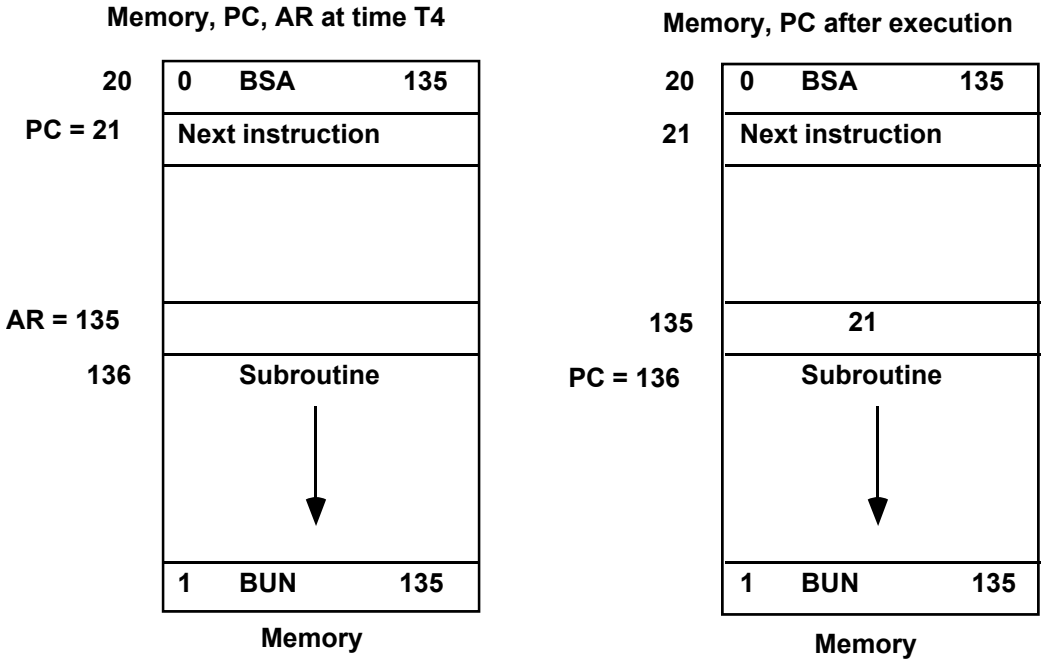
D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

M[AR] ← PC, PC ← AR + 1



MEMORY REFERENCE INSTRUCTIONS

BSA:

D_5T_4: $M[AR] \leftarrow PC,\ AR \leftarrow AR + 1$

D_5T_5: $PC \leftarrow AR,\ SC \leftarrow 0$

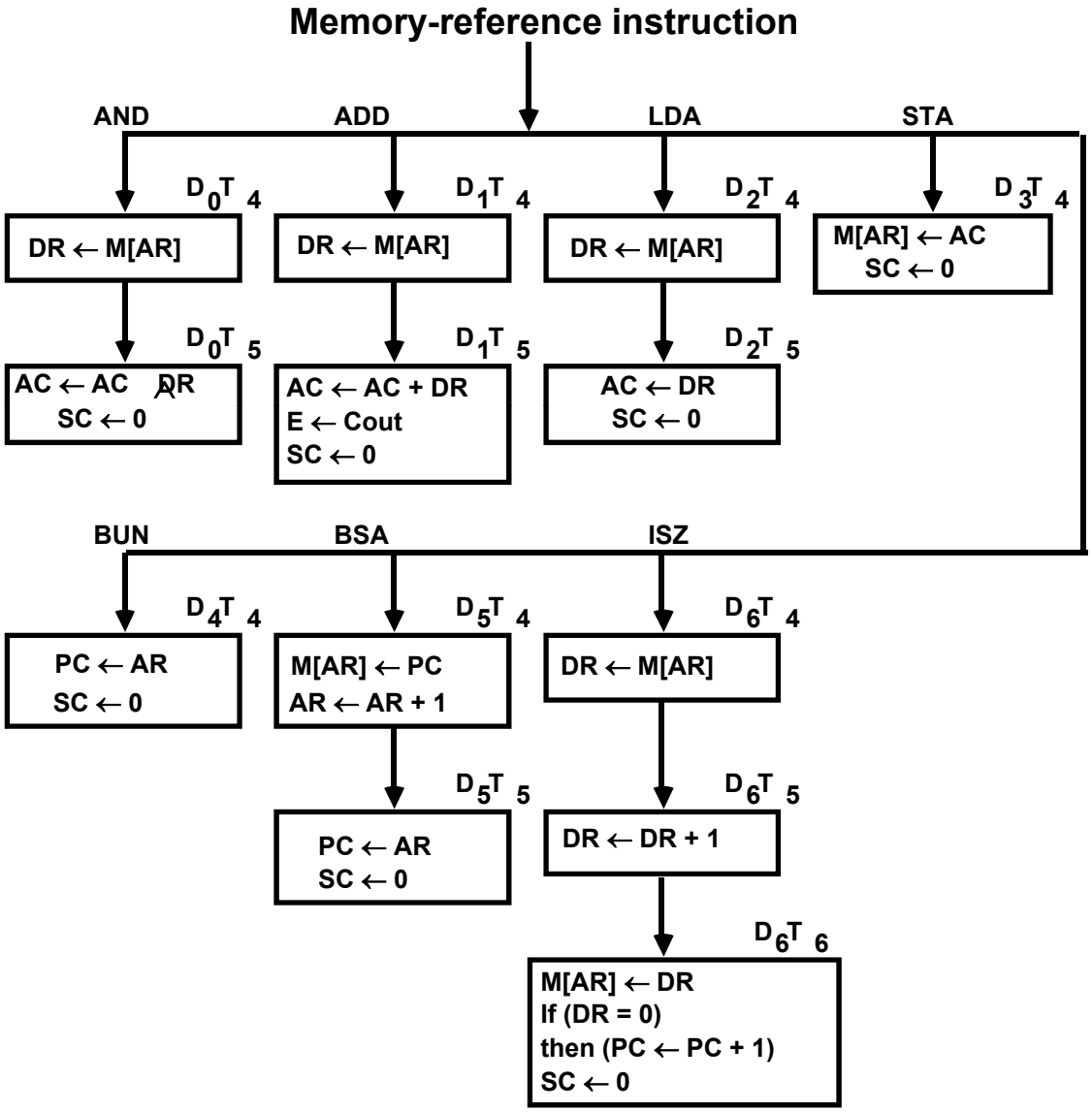
ISZ: Increment and Skip-if-Zero

D_6T_4: $DR \leftarrow M[AR]$

D_6T_5: $DR \leftarrow DR + 1$

D_6T_4: $M[AR] \leftarrow DR,\ \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1),\ SC \leftarrow 0$

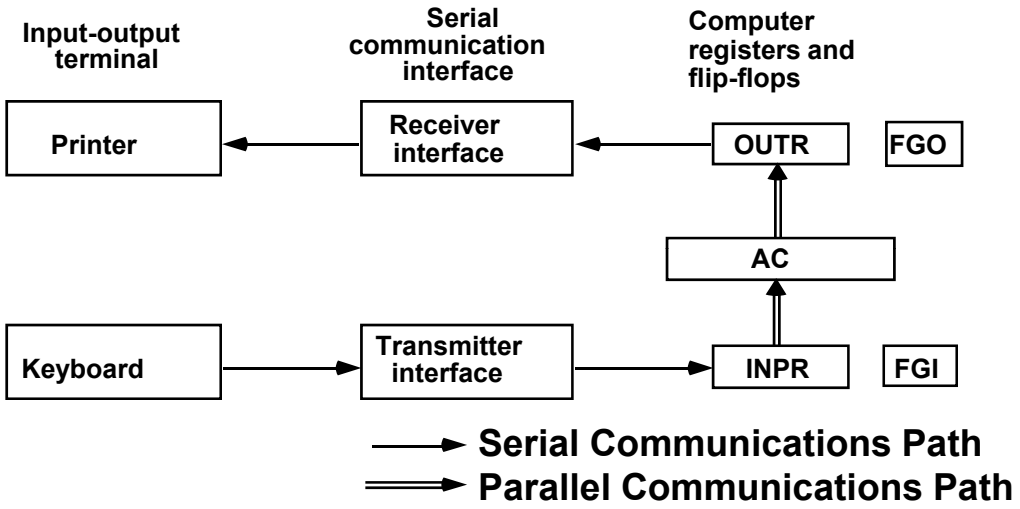
FLOWCHART FOR MEMORY REFERENCE INSTRUCTIONS



INPUT-OUTPUT AND INTERRUPT

A Terminal with a keyboard and a Printer

- **Input-Output Configuration**



INPR	Input register - 8 bits
OUTR	Output register - 8 bits
FGI	Input flag - 1 bit
FGO	Output flag - 1 bit
IEN	Interrupt enable - 1 bit

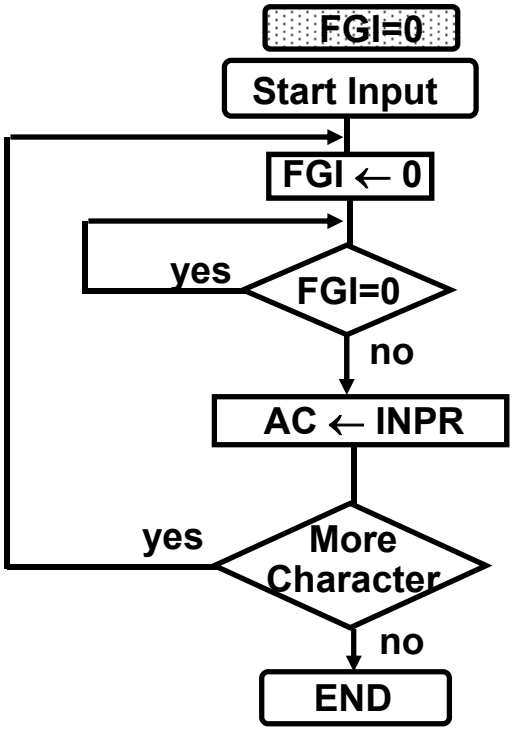
- The terminal sends and receives serial information
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to *synchronize* the timing difference between I/O device and the computer

PROGRAM CONTROLLED DATA TRANSFER

-- CPU --

/* Input */ /* Initially FGI = 0 */
loop: If FGI = 0 goto loop
AC ← INPR, FGI ← 0

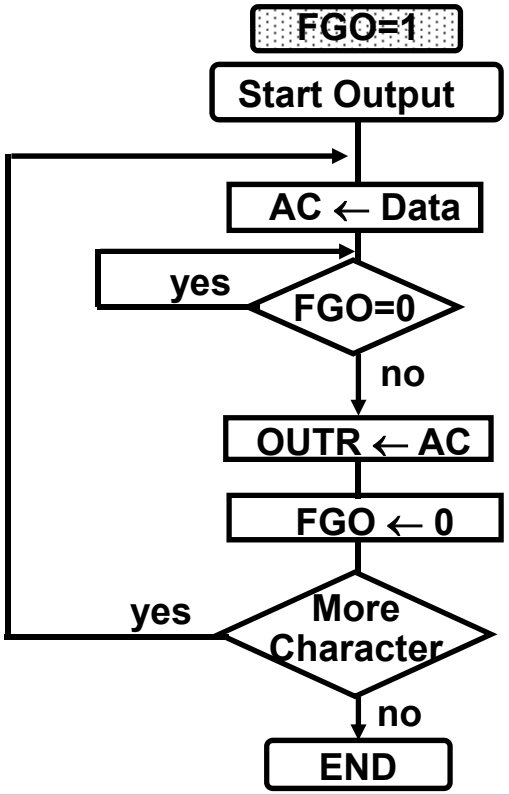
/* Output */ /* Initially FGO = 1 */
loop: If FGO = 0 goto loop
OUTR ← AC, FGO ← 0



-- I/O Device --

loop: If FGI = 1 goto loop
INPR ← new data, FGI ← 1

loop: If FGO = 1 goto loop
consume OUTR, FGO ← 1



INPUT-OUTPUT INSTRUCTIONS

$D_7IT_3 = p$
$IR(i) = B_i$, i = 6, …, 11

	p:	SC ← 0	Clear SC
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB_7:	IEN ← 1	Interrupt enable on
IOF	pB_6:	IEN ← 0	Interrupt enable off

PROGRAM-CONTROLLED INPUT/OUTPUT

- Program-controlled I/O
  - Continuous CPU involvement
    I/O takes valuable CPU time
  - CPU slowed down to I/O speed
  - Simple
  - Least hardware

Input

```
LOOP,   SKI   DEV
        BUN   LOOP
        INP   DEV
```

Output

```
LOOP,   LDA   DATA
LOP,    SKO   DEV
        BUN   LOP
        OUT   DEV
```

INTERRUPT INITIATED INPUT/OUTPUT

- Open communication only when some data has to be passed --> *interrupt*.

- The I/O interface, instead of the CPU, monitors the I/O device.

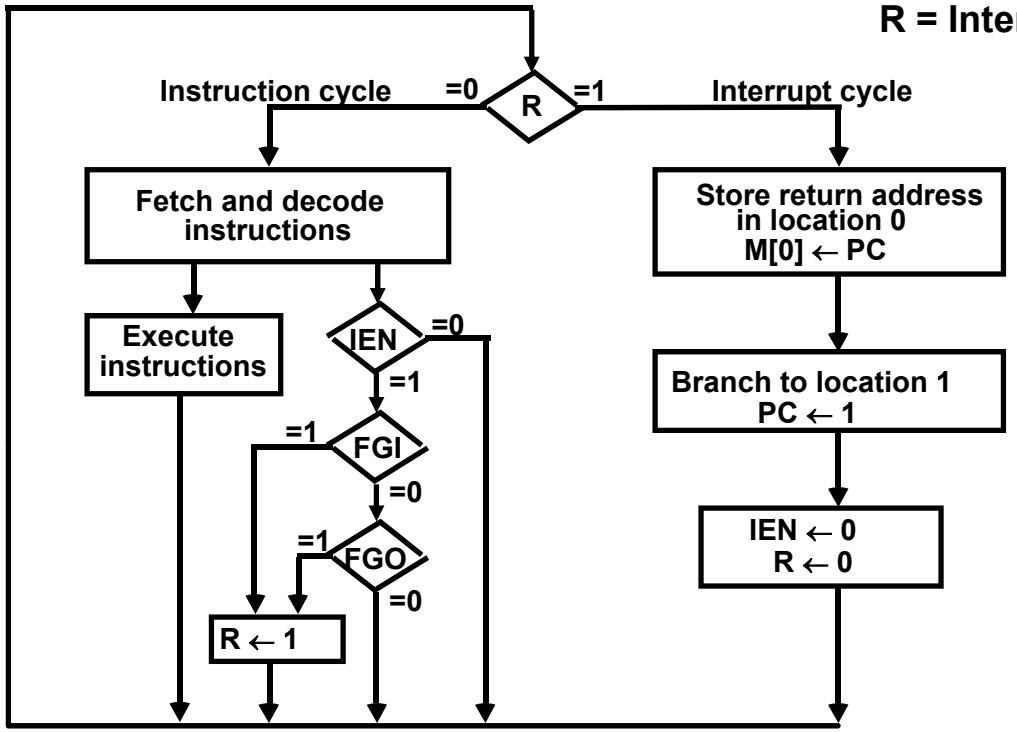
- When the interface founds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU

- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

* IEN (Interrupt-enable flip-flop)

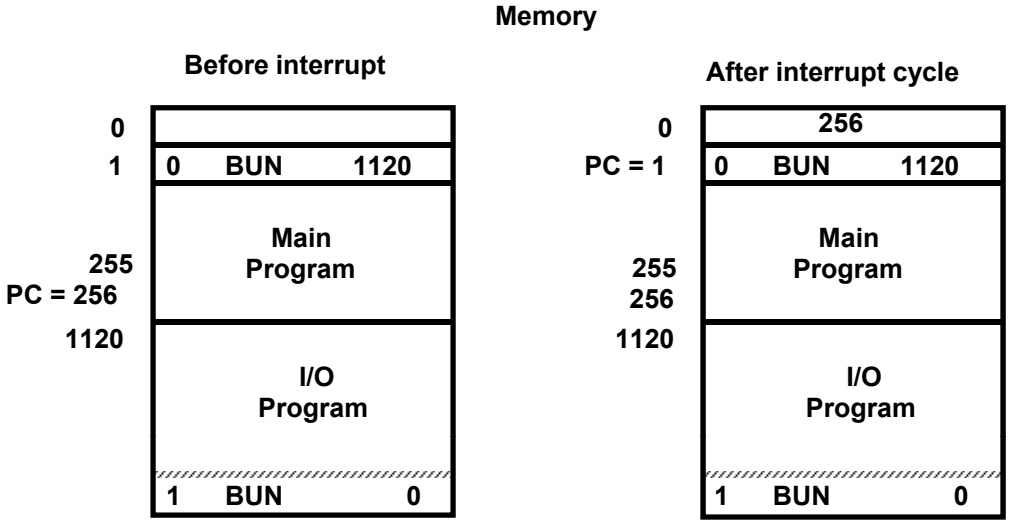
  - can be set and cleared by instructions
  - when cleared, the computer cannot be interrupted

FLOWCHART FOR INTERRUPT CYCLE



- The interrupt cycle is a HW implementation of a branch and save return address operation.
- At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1.
- At memory address 1, the programmer must store a branch instruction that sends the control to an interrupt service routine
- The instruction that returns the control to the original program is "indirect BUN 0"

REGISTER TRANSFER OPERATIONS IN INTERRUPT CYCLE



Register Transfer Statements for Interrupt Cycle

- R F/F ← 1 if IEN (FGI + FGO)$T_0'T_1'T_2'$

  ⇔ $T_0'T_1'T_2'$ (IEN)(FGI + FGO): R ← 1

- The fetch and decode phases of the instruction cycle must be modified ➔Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$
- The interrupt cycle :

  RT_0: AR ← 0, TR ← PC

  RT_1: M[AR] ← TR, PC ← 0

  RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

FURTHER QUESTIONS ON INTERRUPT

How can the CPU recognize the device requesting an interrupt ?

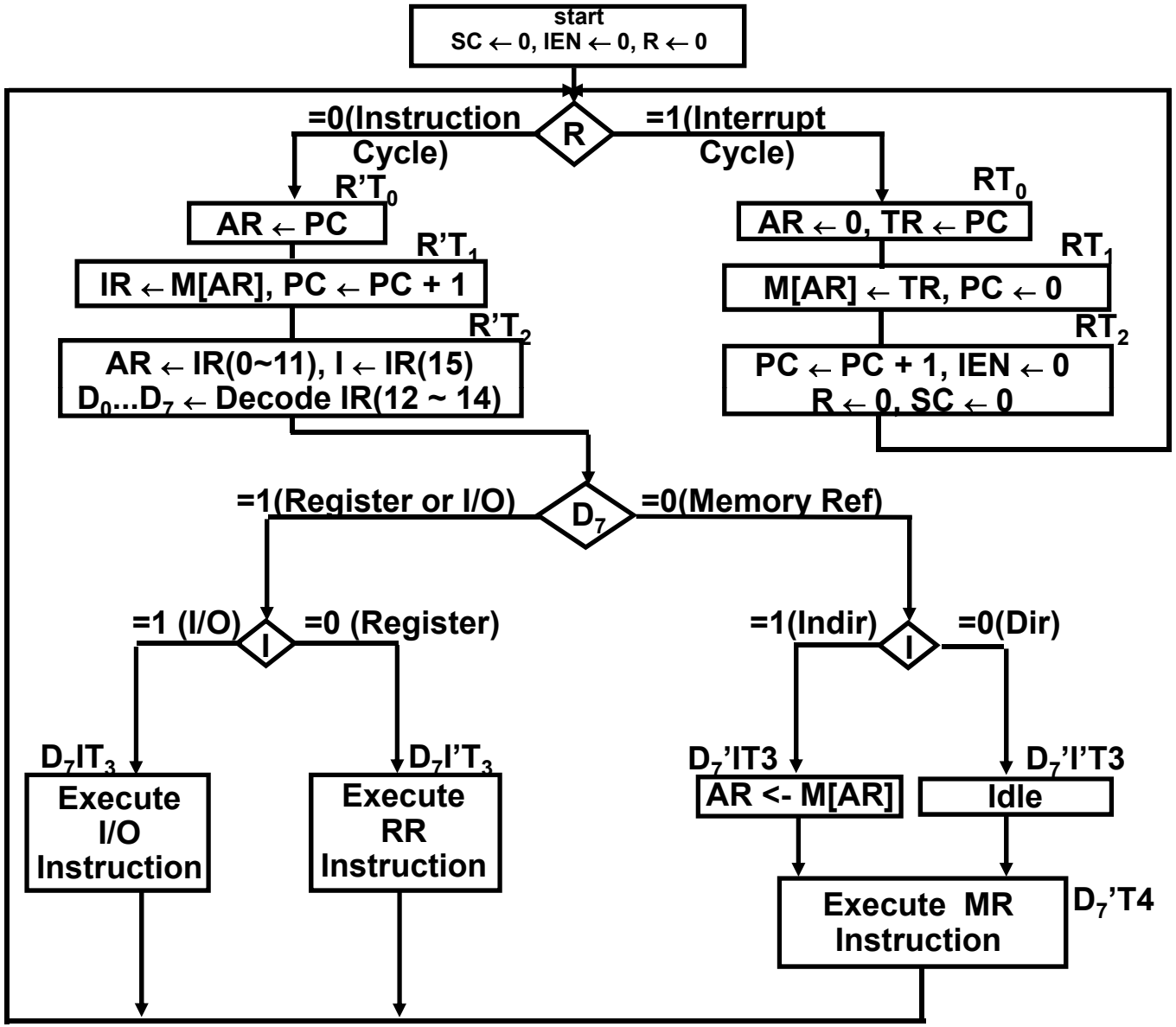
Since different devices are likely to require different interrupt service routines, how can the CPU obtain the starting address of the appropriate routine in each case ?

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

Flowchart of Operations



COMPLETE COMPUTER DESCRIPTION

Microoperations

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR], PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D0, ..., D7 \leftarrow$ Decode $IR(12 \sim 14)$, $AR \leftarrow IR(0 \sim 11), I \leftarrow IR(15)$
Indirect	$D_7'IT_3$:	$AR \leftarrow M[AR]$
Interrupt		
$T_0'T_1'T_2'(IEN)(FGI + FGO)$:		$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0, TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR, PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$
Memory-Reference		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR, SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR, SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC, SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR, SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC, AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR, SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if(DR=0) then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

COMPLETE COMPUTER DESCRIPTION

Microoperations

Register-Reference		
	$D_7I'T_3 = r$	(Common to all register-reference instr)
	$IR(i) = B_i$	(i = 0,1,2, ..., 11)
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow shr\ AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow shl\ AC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If(AC(15) =0) then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If(AC(15) =1) then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If(AC = 0) then $(PC \leftarrow PC + 1)$
SZE	rB_1:	If(E=0) then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-Output	$D_7IT_3 = p$	(Common to all input-output instructions)
	$IR(i) = B_i$	(i = 6,7,8,9,10,11)
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR,\ FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7),\ FGO \leftarrow 0$
SKI	pB_9:	If(FGI=1) then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If(FGO=1) then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

DESIGN OF BASIC COMPUTER(BC)

Hardware Components of BC

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

Control Logic Gates

- Input Controls of the nine registers
- Read and Write Controls of memory
- Set, Clear, or Complement Controls of the flip-flops
- S_2, S_1, S_0 Controls to select a register for the bus
- AC, and Adder and Logic circuit

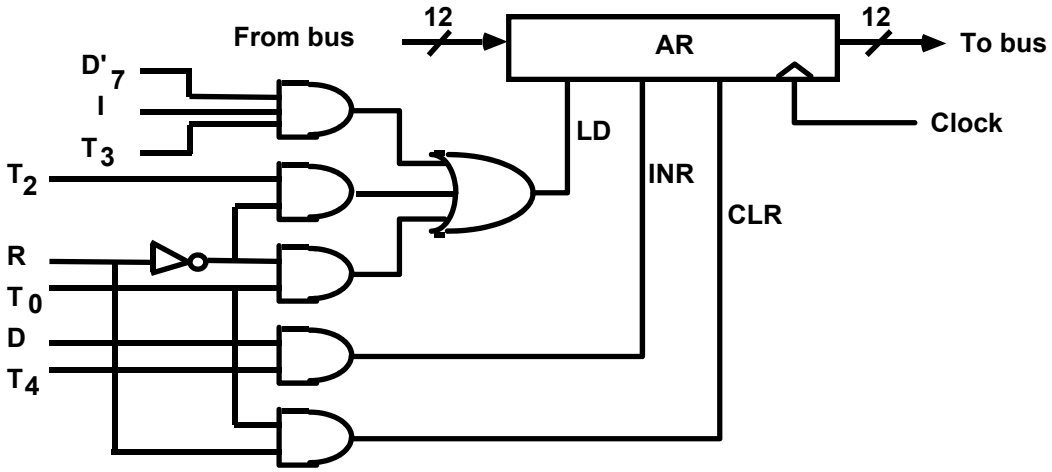
CONTROL OF REGISTERS AND MEMORY

Address Register; AR

Scan all of the register transfer statements that change the content of AR:

$R'T_0$:	$AR \leftarrow PC$	$LD(AR)$
$R'T_2$:	$AR \leftarrow IR(0\text{-}11)$	$LD(AR)$
D'_7IT_3:	$AR \leftarrow M[AR]$	$LD(AR)$
RT_0:	$AR \leftarrow 0$	$CLR(AR)$
D_5T_4:	$AR \leftarrow AR + 1$	$INR(AR)$

$$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$$
$$CLR(AR) = RT_0$$
$$INR(AR) = D_5T_4$$

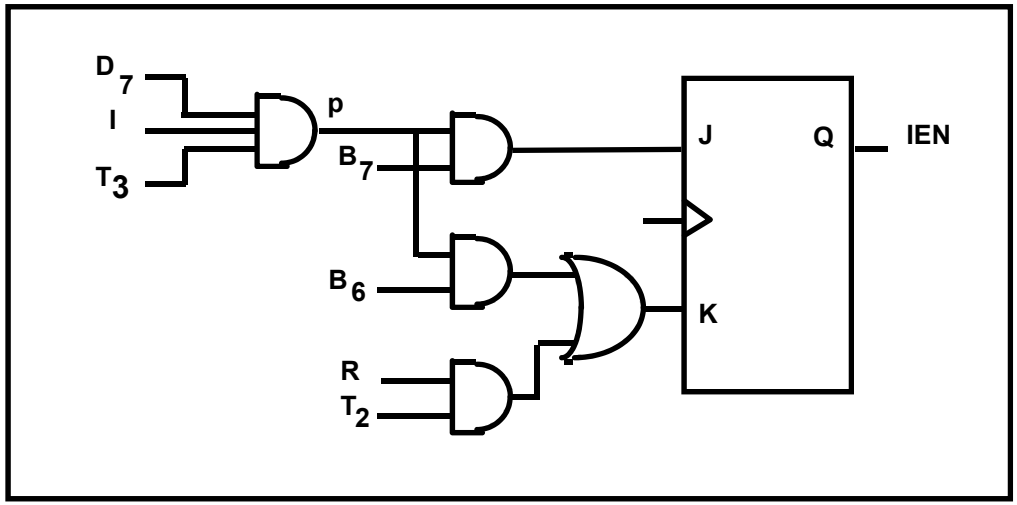


CONTROL OF FLAGS

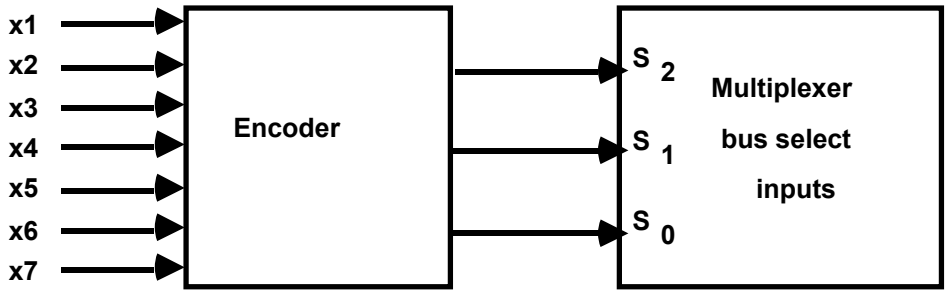
IEN: Interrupt Enable Flag

pB_7: IEN ← 1 (I/O Instruction)
pB_6: IEN ← 0 (I/O Instruction)
RT_2: IEN ← 0 (Interrupt)

$p = D_7IT_3$ (Input/Output Instruction)



CONTROL OF COMMON BUS



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	

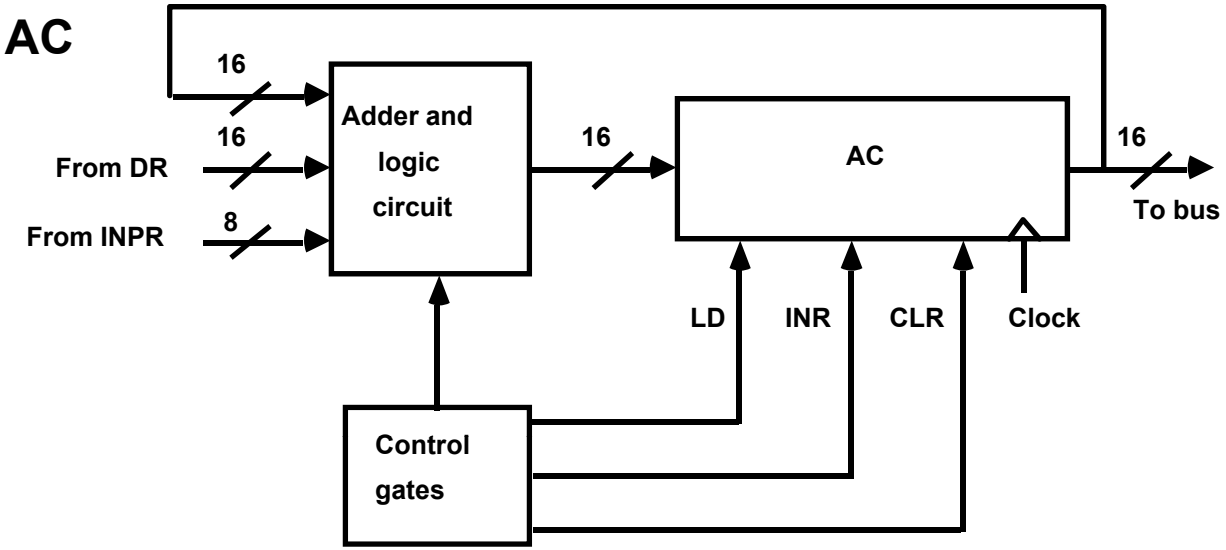
For AR

D_4T_4: PC ← AR
D_5T_5: PC ← AR

$x1 = D_4T_4 + D_5T_5$

DESIGN OF ACCUMULATOR LOGIC

Circuits associated with AC

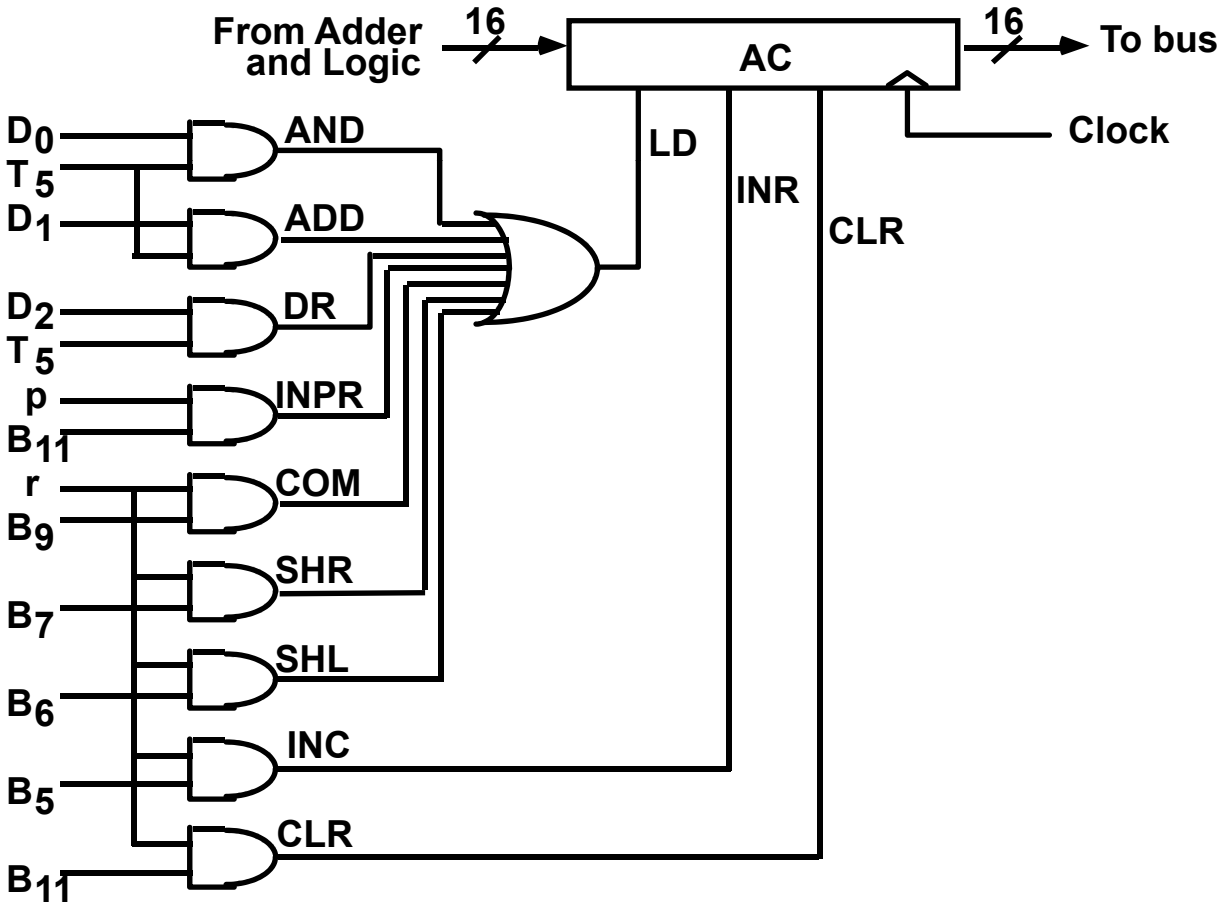


All the statements that change the content of AC

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$	Transfer from INPR
rB_9:	$AC \leftarrow AC'$	Complement
rB_7 :	$AC \leftarrow shr\ AC, AC(15) \leftarrow E$	Shift right
rB_6 :	$AC \leftarrow shl\ AC, AC(0) \leftarrow E$	Shift left
rB_{11} :	$AC \leftarrow 0$	Clear
rB_5 :	$AC \leftarrow AC + 1$	Increment

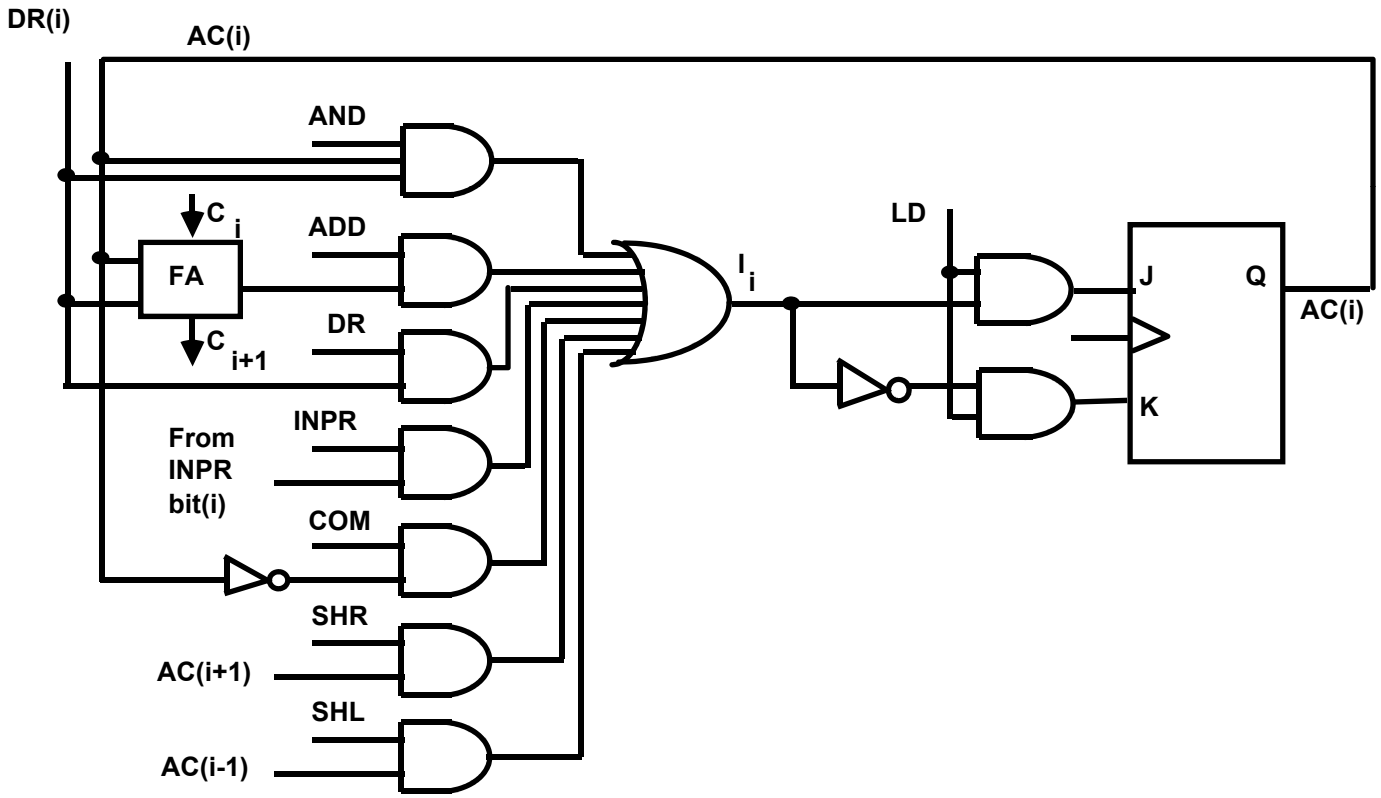
CONTROL OF AC REGISTER

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



ALU (ADDER AND LOGIC CIRCUIT)

One stage of Adder and Logic circuit



PROGRAMMING THE BASIC COMPUTER

Introduction

Machine Language

Assembly Language

Assembler

Program Loops

Programming Arithmetic and Logic Operations

Subroutines

Input-Output Programming

INTRODUCTION

Those concerned with computer architecture should have a knowledge of both hardware and software because the two branches influence each other.

Instruction Set of the *Basic Computer*

Symbol	*Hexa code*	*Description*
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to m+1
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC, carry to E
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

m: effective address
M: memory word (operand) found at m

MACHINE LANGUAGE

- Program

  A list of instructions or statements for directing the computer to perform a required data processing task

- Various types of programming languages
  - Hierarchy of programming languages

    - Machine-language
      - Binary code
      - Octal or hexadecimal code

    - Assembly-language (Assembler)
      - Symbolic code

    - High-level language (Compiler)

COMPARISON OF PROGRAMMING LANGUAGES

• Binary Program to Add Two Numbers

Location	Instruction Code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

• Hexa program

Location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0053
005	FFE9
006	0000

• Program with Symbolic OP-Code

Location	Instruction		Comments
000	LDA	004	Load 1st operand into AC
001	ADD	005	Add 2nd operand to AC
002	STA	006	Store sum in location 006
003	HLT		Halt computer
004	0053		1st operand
005	FFE9		2nd operand (negative)
006	0000		Store sum here

• Assembly-Language Program

```
     ORG   0     /Origin of program is location 0
     LDA   A     /Load  operand from location A
     ADD   B     /Add  operand from location B
     STA   C     /Store sum in location C
     HLT         /Halt computer
A,   DEC   83    /Decimal operand
B,   DEC   -23   /Decimal operand
C,   DEC   0     /Sum stored in location C
     END         /End of symbolic program
```

• Fortran Program

```
INTEGER  A, B, C
DATA  A,83 / B,-23
C = A + B
END
```

ASSEMBLY LANGUAGE

Syntax of the BC assembly language

Each line is arranged in three columns called fields

Label field

- May be empty or may specify a symbolic address consists of up to 3 characters
- Terminated by a comma

Instruction field

- Specifies a machine or a pseudo instruction
- May specify one of
  * Memory reference instr. (MRI)
    MRI consists of two or three symbols separated by spaces.
    ADD OPR (direct address MRI)
    ADD PTR I (indirect address MRI)
  * Register reference or input-output instr.
    Non-MRI does not have an address part
  * Pseudo instr. with or without an operand
    Symbolic address used in the instruction field must be defined somewhere as a label

Comment field

- May be empty or may include a comment

PSEUDO-INSTRUCTIONS

ORG N
Hexadecimal number N is the memory loc.
for the instruction or operand listed in the following line

END
Denotes the end of symbolic program

DEC N
Signed decimal number N to be converted to the binary

HEX N
Hexadecimal number N to be converted to the binary

Example: Assembly language program to subtract two numbers

```
        ORG  100      / Origin of program is location 100
        LDA  SUB      / Load subtrahend to AC
        CMA           / Complement AC
        INC           / Increment AC
        ADD  MIN      / Add minuend to AC
        STA  DIF      / Store difference
        HLT           / Halt computer
MIN,    DEC  83       / Minuend
SUB,    DEC  -23      / Subtrahend
DIF,    HEX  0        / Difference stored here
        END           / End of symbolic program
```

TRANSLATION TO BINARY

Hexadecimal Code		Symbolic Program	
Location	Content		
			ORG 100
100	2107		LDA SUB
101	7200		CMA
102	7020		INC
103	1106		ADD MIN
104	3108		STA DIF
105	7001		HLT
106	0053	MIN,	DEC 83
107	FFE9	SUB,	DEC -23
108	0000	DIF,	HEX 0
			END

ASSEMBLER - FIRST PASS -

Assembler

Source Program - Symbolic Assembly Language Program

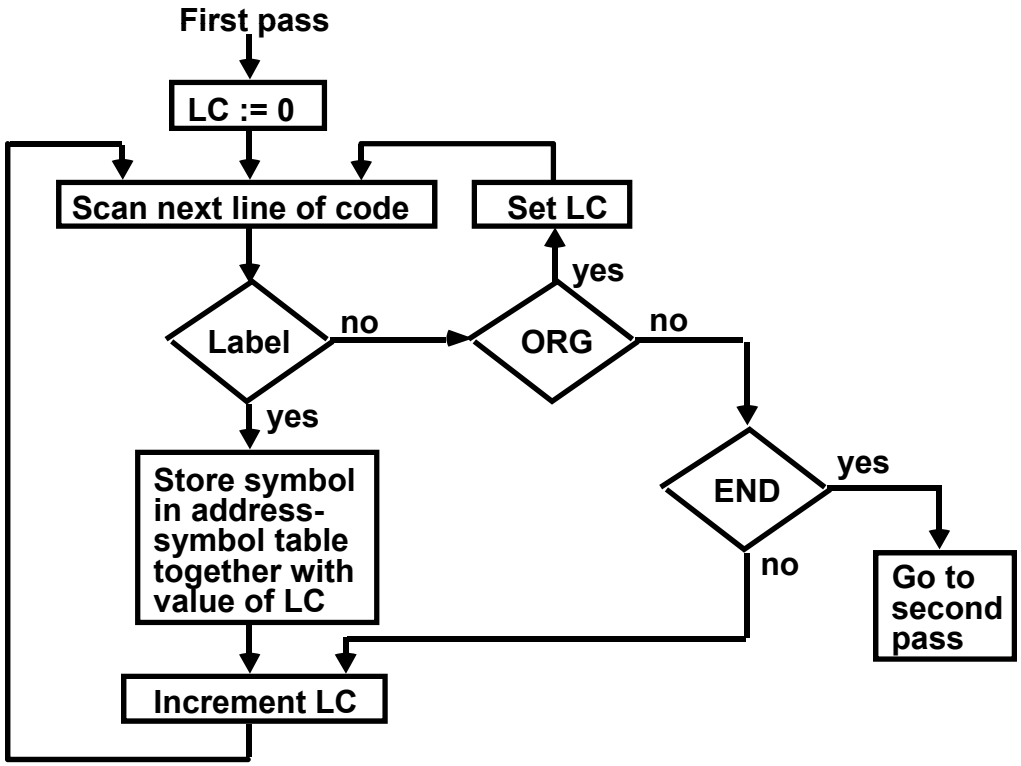
Object Program - Binary Machine Language Program

Two pass assembler

1st pass: generates a table that correlates all user defined (address) symbols with their binary equivalent value

2nd pass: binary translation

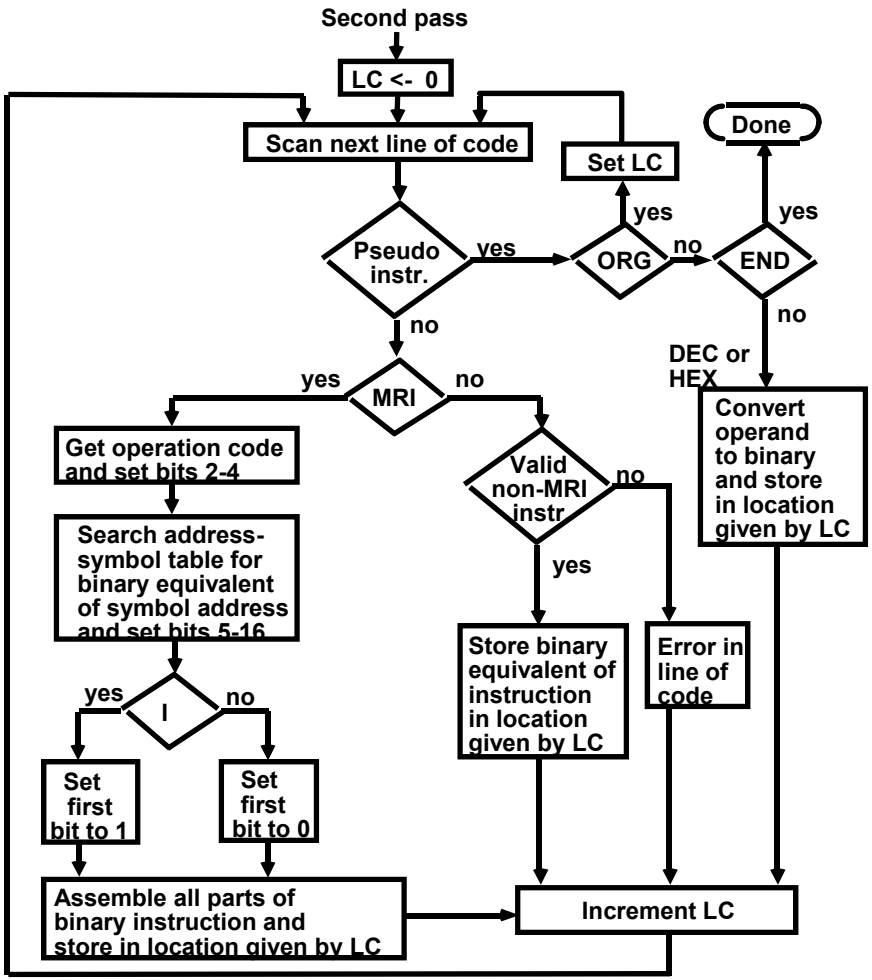
First pass



ASSEMBLER - SECOND PASS -

Second Pass

Machine instructions are translated by means of table-lookup procedures;
(1. Pseudo-Instruction Table, 2. MRI Table, 3. Non-MRI Table
4. Address Symbol Table)



PROGRAM LOOPS

Loop: A sequence of instructions that are executed many times, each with a different set of data

Fortran program to add 100 numbers:

```
      DIMENSION  A(100)
      INTEGER  SUM,  A
      SUM = 0
      DO  3  J = 1,  100
3     SUM = SUM + A(J)
```

Assembly-language program to add 100 numbers:

```
          ORG  100          / Origin of program is HEX 100
          LDA  ADS          / Load first address of operand
          STA  PTR          / Store in pointer
          LDA  NBR          / Load -100
          STA  CTR          / Store in counter
          CLA               / Clear AC
LOP,      ADD  PTR  I       / Add an operand to AC
          ISZ  PTR          / Increment pointer
          ISZ  CTR          / Increment counter
          BUN  LOP          / Repeat loop again
          STA  SUM          / Store sum
          HLT               / Halt
ADS,      HEX  150          / First address of operands
PTR,      HEX  0            / Reserved for a pointer
NBR,      DEC  -100         / Initial value for a counter
CTR,      HEX  0            / Reserved for a counter
SUM,      HEX  0            / Sum is stored here
          ORG  150          / Origin of operands is HEX 150
          DEC  75           / First operand
            .
            .
            .
          DEC  23           / Last operand
          END               / End of symbolic program
```

PROGRAMMING ARITHMETIC AND LOGIC OPERATIONS

Implementation of Arithmetic and Logic Operations

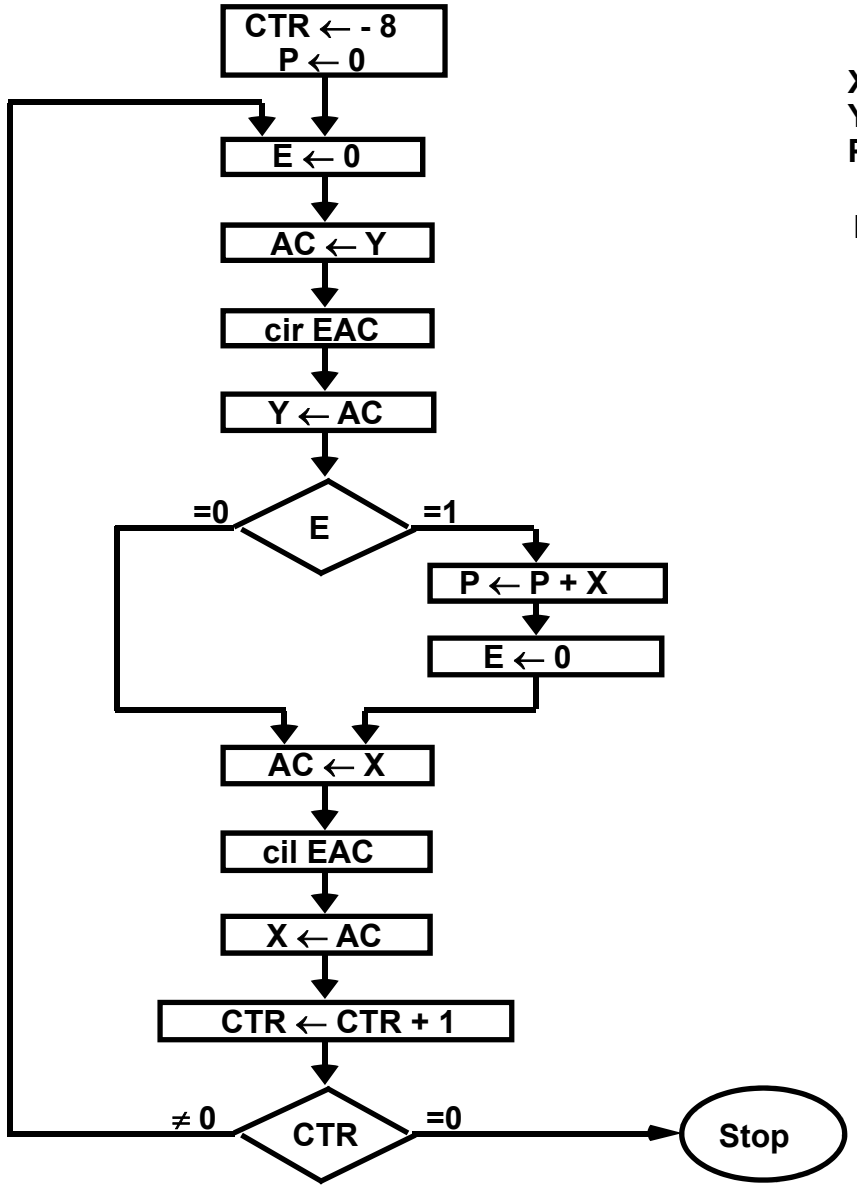
- Software Implementation
  - Implementation of an operation with a program using machine instruction set
  - Usually when the operation is not included in the instruction set

- Hardware Implementation
  - Implementation of an operation in a computer with one machine instruction

Software Implementation example:

* Multiplication
  - For simplicity, unsigned positive numbers
  - 8-bit numbers -> 16-bit product

FLOWCHART OF A PROGRAM - Multiplication -



X holds the multiplicand
Y holds the multiplier
P holds the product

Example with four significant digits

		P
X =	0000 1111	
Y =	0000 1011	0000 0000
	0000 1111	0000 1111
	0001 1110	0010 1101
	0000 0000	0010 1101
	0111 1000	1010 0101
	1010 0101	

ASSEMBLY LANGUAGE PROGRAM - Multiplication -

```
        ORG  100
LOP,    CLE          / Clear E
        LDA  Y       / Load multiplier
        CIR          / Transfer multiplier bit to E
        STA  Y       / Store shifted multiplier
        SZE          / Check if bit is zero
        BUN  ONE     / Bit is one; goto ONE
        BUN  ZRO     / Bit is zero; goto ZRO
ONE,    LDA  X       / Load multiplicand
        ADD  P       / Add to partial product
        STA  P       / Store partial product
        CLE          / Clear E
ZRO,    LDA  X       / Load multiplicand
        CIL          / Shift left
        STA  X       / Store shifted multiplicand
        ISZ  CTR     / Increment counter
        BUN  LOP     / Counter not zero; repeat loop
        HLT          / Counter is zero; halt
CTR,    DEC  -8      / This location serves as a counter
X,      HEX  000F    / Multiplicand stored here
Y,      HEX  000B    / Multiplier stored here
P,      HEX  0       / Product formed here
        END
```

ASSEMBLY LANGUAGE PROGRAM
- Double Precision Addition -

```
LDA   AL        / Load A low
ADD   BL        / Add B low, carry in E
STA   CL        / Store in C low
CLA             / Clear AC
CIL             / Circulate to bring carry into AC(16)
ADD   AH        / Add A high and carry
ADD   BH        / Add B high
STA   CH        / Store in C high
HLT
```

ASSEMBLY LANGUAGE PROGRAM
- Logic and Shift Operations -

• Logic operations

- BC instructions : AND, CMA, CLA
- Program for OR operation

```
LDA   A      / Load 1st operand
CMA          / Complement to get A'
STA   TMP    / Store in a temporary location
LDA   B      / Load 2nd operand B
CMA          / Complement to get B'
AND   TMP    / AND with A' to get A' AND B'
CMA          / Complement again to get A OR B
```

• Shift operations - BC has *Circular Shift* only

- Logical shift-right operation

```
CLE
CIR
```

- Logical shift-left operation

```
CLE
CIL
```

- Arithmetic right-shift operation

```
CLE          / Clear E to 0
SPA          / Skip if AC is positive
CME          / AC is negative
CIR          / Circulate E and AC
```

SUBROUTINES

Subroutine

- A set of common instructions that can be used in a program many times.
- Subroutine *linkage* : a procedure for branching to a subroutine and returning to the main program

Example

Loc.			
100		ORG 100	/ Main program
101		LDA X	/ Load X
102		BSA SH4	/ Branch to subroutine
103		STA X	/ Store shifted number
104		LDA Y	/ Load Y
105		BSA SH4	/ Branch to subroutine again
106		STA Y	/ Store shifted number
107		HLT	
108	X,	HEX 1234	
	Y,	HEX 4321	
			/ Subroutine to shift left 4 times
109	SH4,	HEX 0	/ Store return address here
10A		CIL	/ Circulate left once
10B		CIL	
10C		CIL	
10D		CIL	/ Circulate left fourth time
10E		AND MSK	/ Set AC(13-16) to zero
10F		BUN SH4 I	/ Return to main program
110	MSK,	HEX FFF0	/ Mask operand
		END	

SUBROUTINE PARAMETERS AND DATA LINKAGE

Linkage of Parameters and Data between the Main Program and a Subroutine
- via Registers
- via Memory locations
- ....

Example: **Subroutine performing *LOGICAL OR operation*; Need two parameters**

Loc.			
200		ORG 200	
200		LDA X	/ Load 1st operand into AC
201		BSA OR	/ Branch to subroutine OR
202		HEX 3AF6	/ 2nd operand stored here
203		STA Y	/ Subroutine returns here
204		HLT	
205	X,	HEX 7B95	/ 1st operand stored here
206	Y,	HEX 0	/ Result stored here
207	OR,	HEX 0	/ Subroutine OR
208		CMA	/ Complement 1st operand
209		STA TMP	/ Store in temporary location
20A		LDA OR I	/ Load 2nd operand
20B		CMA	/ Complement 2nd operand
20C		AND TMP	/ AND complemented 1st operand
20D		CMA	/ Complement again to get OR
20E		ISZ OR	/ Increment return address
20F		BUN OR I	/ Return to main program
210	TMP,	HEX 0	/ Temporary storage
		END	

SUBROUTINE - Moving a Block of Data -

```
                        / Main program
        BSA  MVE        / Branch to subroutine
        HEX  100        / 1st address of source data
        HEX  200        / 1st address of destination data
        DEC  -16        / Number of items to move
        HLT
MVE,    HEX  0          / Subroutine MVE
        LDA  MVE  I     / Bring address of source
        STA  PT1        / Store in 1st pointer
        ISZ  MVE        / Increment return address
        LDA  MVE  I     / Bring address of destination
        STA  PT2        / Store in 2nd pointer
        ISZ  MVE        / Increment return address
        LDA  MVE  I     / Bring number of items
        STA  CTR        / Store in counter
        ISZ  MVE        / Increment return address
LOP,    LDA  PT1  I     / Load source item
        STA  PT2  I     / Store in destination
        ISZ  PT1        / Increment source pointer
        ISZ  PT2        / Increment destination pointer
        ISZ  CTR        / Increment counter
        BUN  LOP        / Repeat 16 times
        BUN  MVE  I     / Return to main program
PT1,    --
PT2,    --
CTR,    --
```

• Fortran subroutine

```
      SUBROUTINE  MVE (SOURCE, DEST, N)
      DIMENSION  SOURCE(N), DEST(N)
      DO  20  I = 1, N
20    DEST(I) = SOURCE(I)
      RETURN
      END
```

INPUT OUTPUT PROGRAM

Program to Input one Character(Byte)

```
CIF,    SKI         / Check input flag
        BUN  CIF    / Flag=0, branch to check again
        INP         / Flag=1, input character
        OUT         / Display to ensure correctness
        STA  CHR    / Store character
        HLT
CHR,    --          / Store character here
```

Program to Output a Character

```
        LDA  CHR    / Load character into AC
COF,    SKO         / Check output flag
        BUN  COF    / Flag=0, branch to check again
        OUT         / Flag=1, output character
        HLT
CHR,    HEX  0057   / Character is "W"
```

CHARACTER MANIPULATION

Subroutine to Input 2 Characters and pack into a word

```
IN2,    --          / Subroutine entry
FST,    SKI
        BUN  FST
        INP         / Input 1st character
        OUT
        BSA  SH4    / Logical Shift left 4 bits
        BSA  SH4    / 4 more bits
SCD,    SKI
        BUN  SCD
        INP         / Input 2nd character
        OUT
        BUN  IN2  I / Return
```

PROGRAM INTERRUPT

Tasks of Interrupt Service Routine

- Save the Status of CPU
  Contents of processor registers and Flags

- Identify the source of Interrupt
  Check which flag is set

- Service the device whose flag is set
  (Input Output Subroutine)

- Restore contents of processor registers and flags

- Turn the interrupt facility on

- Return to the running program
  Load PC of the interrupted program

INTERRUPT SERVICE ROUTINE

```
Loc.
  0     ZRO,    -           / Return address stored here
  1             BUN  SRV    / Branch to service routine
100             CLA         / Portion of running program
101             ION         / Turn on interrupt facility
102             LDA  X
103             ADD  Y      / Interrupt occurs here
104             STA  Z      / Program returns here after interrupt

                            / Interrupt service routine
200     SRV,    STA  SAC    / Store content of AC
                CIR         / Move E into AC(1)
                STA  SE     / Store content of E
                SKI         / Check input flag
                BUN  NXT    / Flag is off, check next flag
                INP         / Flag is on, input character
                OUT         / Print character
                STA  PT1  I / Store it in input buffer
                ISZ  PT1    / Increment input pointer
        NXT,    SKO         / Check output flag
                BUN  EXT    / Flag is off, exit
                LDA  PT2  I / Load character from output buffer
                OUT         / Output character
                ISZ  PT2    / Increment output pointer
        EXT,    LDA  SE     / Restore value of AC(1)
                CIL         / Shift it to E
                LDA  SAC    / Restore content of AC
                ION         / Turn interrupt on
                BUN  ZRO  I / Return to running program
        SAC,    -           / AC is stored here
        SE,     -           / E is stored here
        PT1,    -           / Pointer of input buffer
        PT2,    -           / Pointer of output buffer
```

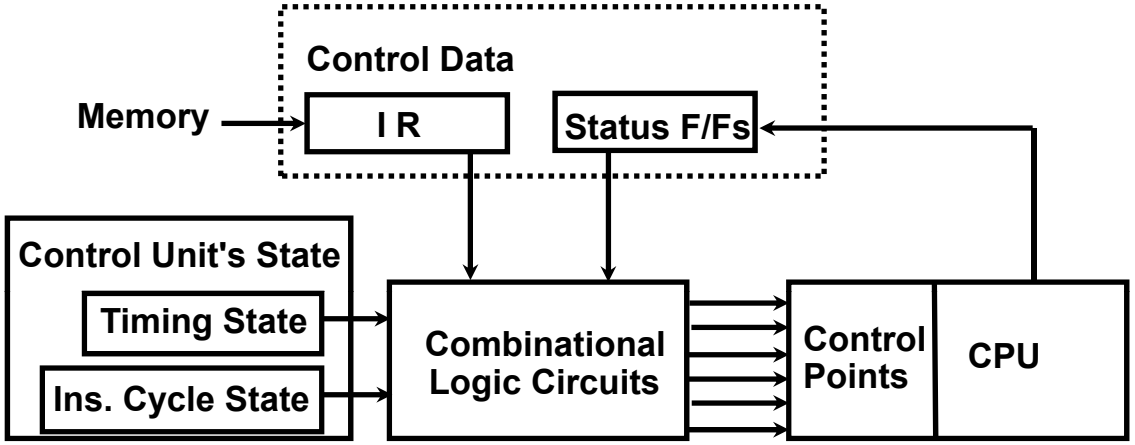
MICROPROGRAMMED CONTROL

- Control Memory
- Sequencing Microinstructions
- Microprogram Example
- Design of Control Unit
- Microinstruction Format
- Nanostorage and Nanoprogram

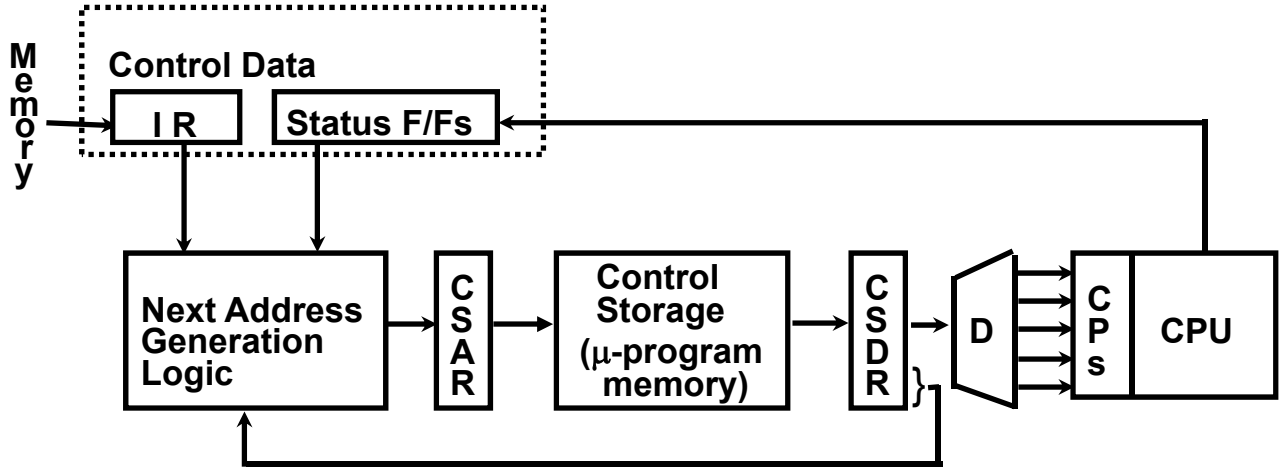
COMPARISON OF CONTROL UNIT IMPLEMENTATIONS

Control Unit Implementation

Combinational Logic Circuits (Hard-wired)



Microprogram



TERMINOLOGY

Microprogram
- Program stored in memory that generates all the control signals required to execute the instruction set correctly
- Consists of microinstructions

Microinstruction
- Contains a control word and a sequencing word
  - Control Word - All the control information required for one clock cycle
  - Sequencing Word - Information needed to decide the next microinstruction address
- Vocabulary to write a microprogram

Control Memory(Control Storage: CS)
- Storage in the microprogrammed control unit to store the microprogram

Writeable Control Memory(Writeable Control Storage:WCS)
- CS whose contents can be modified
  - -> Allows the microprogram can be changed
  - -> Instruction set can be changed or modified

Dynamic Microprogramming
- Computer system whose control unit is implemented with a microprogram in WCS
- Microprogram can be changed by a systems programmer or a user

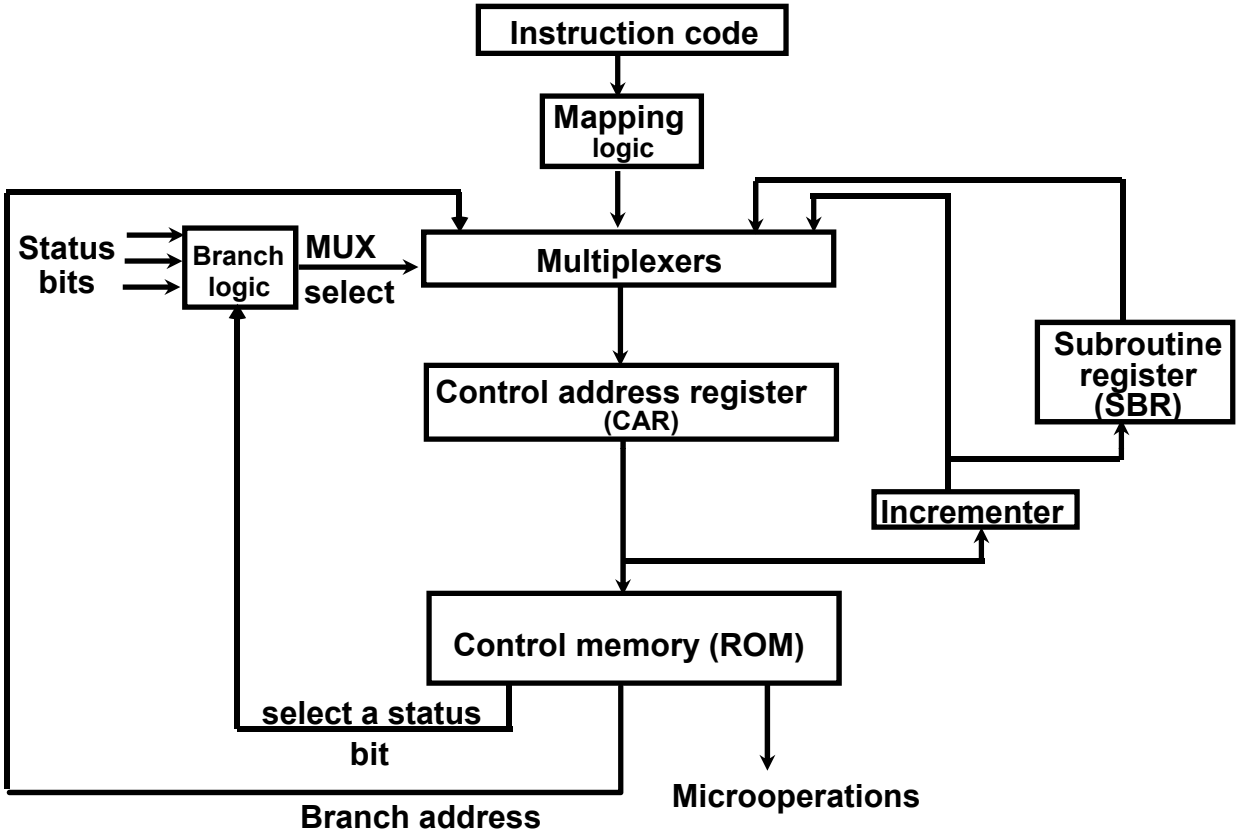
TERMINOLOGY

Sequencer (Microprogram Sequencer)

A Microprogram Control Unit that determines the Microinstruction Address to be executed in the next clock cycle

- **In-line Sequencing**
- **Branch**
- **Conditional Branch**
- **Subroutine**
- **Loop**
- **Instruction OP-code mapping**

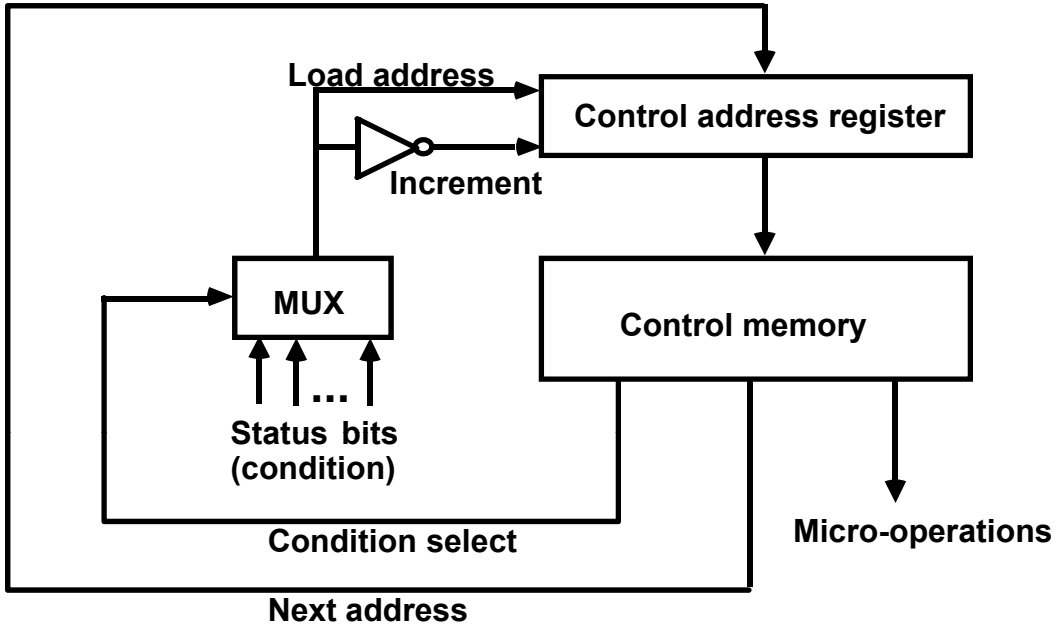
MICROINSTRUCTION SEQUENCING



Sequencing Capabilities Required in a Control Storage

- Incrementing of the control address register
- Unconditional and conditional branches
- A mapping process from the bits of the machine instruction to an address for control memory
- A facility for subroutine call and return

CONDITIONAL BRANCH



Conditional Branch

If *Condition* is true, then *Branch* (address from the next address field of the current microinstruction) else *Fall Through*

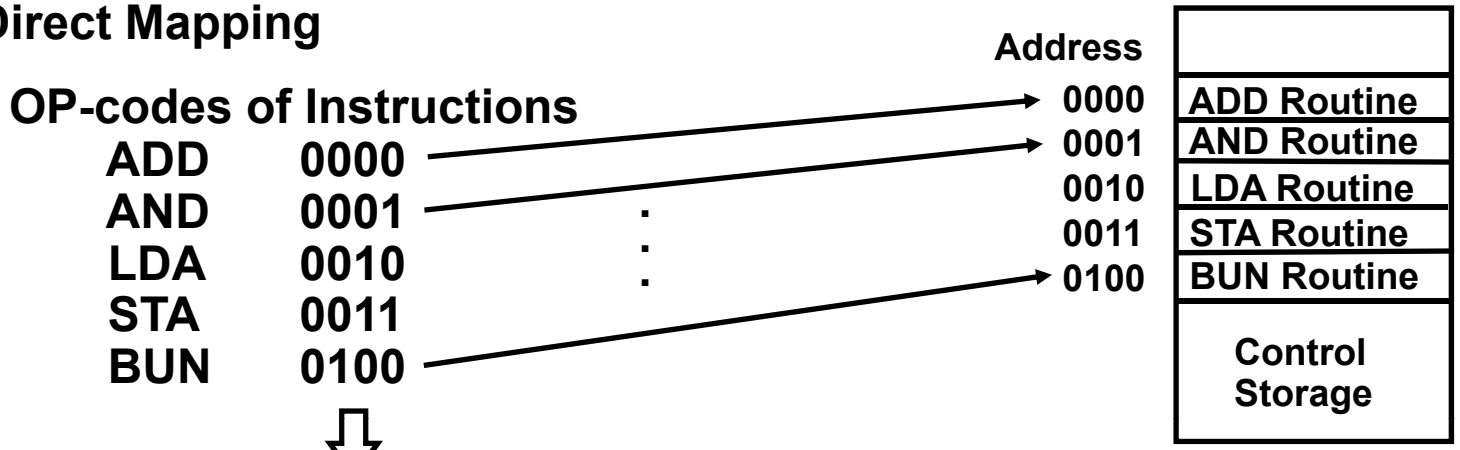
Conditions to Test: O(overflow), N(negative), Z(zero), C(carry), etc.

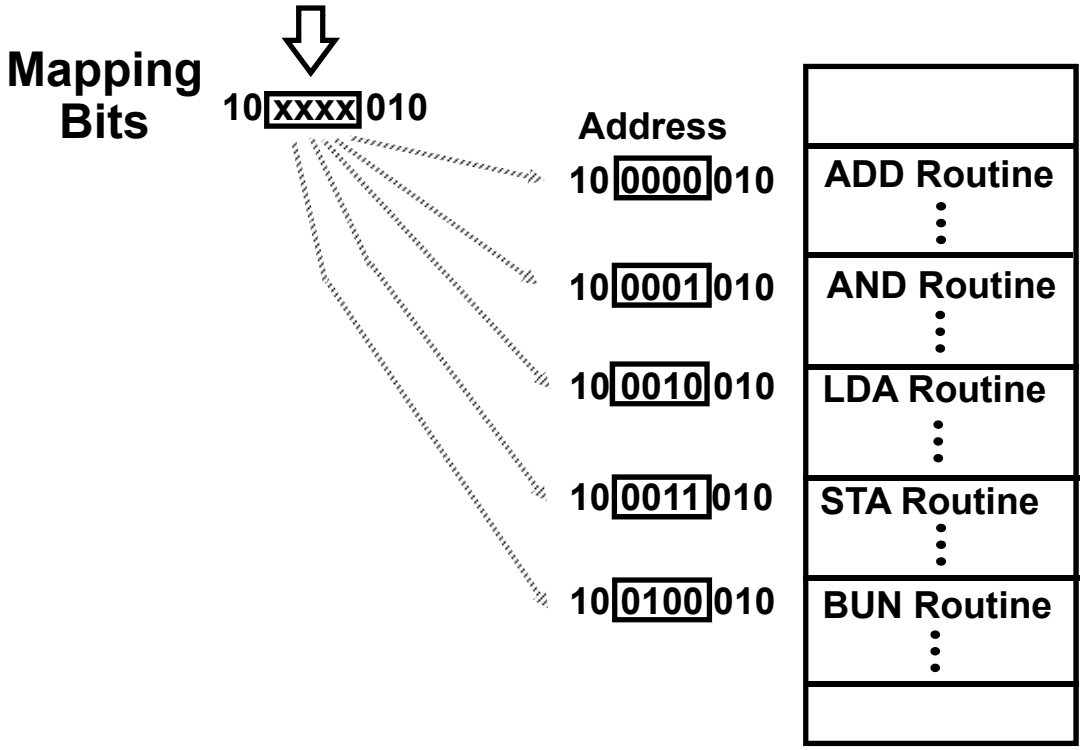
Unconditional Branch

Fixing the value of one status bit at the input of the multiplexer to 1

MAPPING OF INSTRUCTIONS

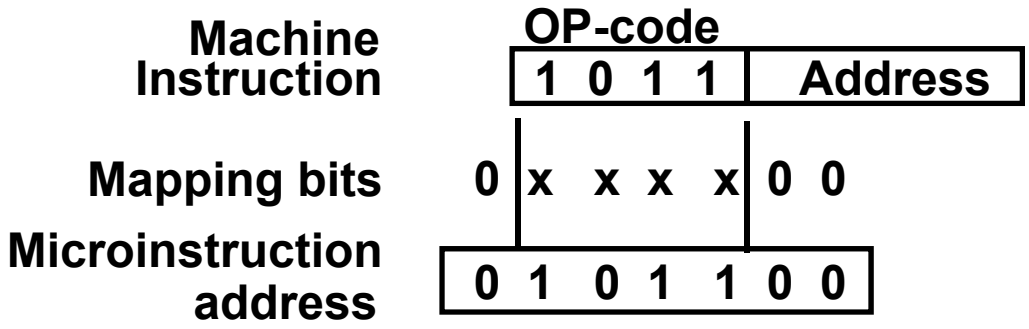
Direct Mapping



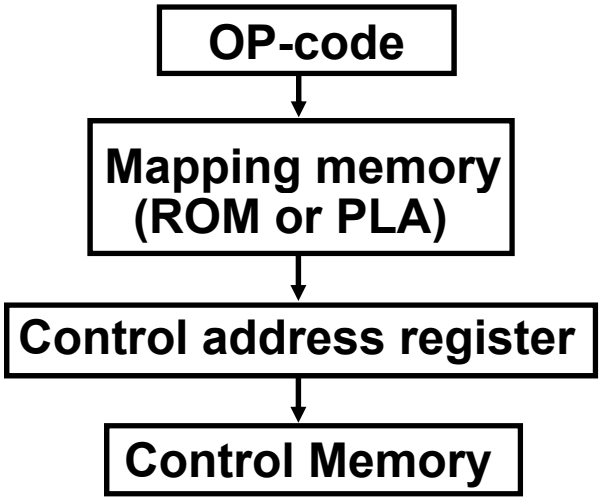


MAPPING OF INSTRUCTIONS TO MICROROUTINES

Mapping from the OP-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its execution microprogram

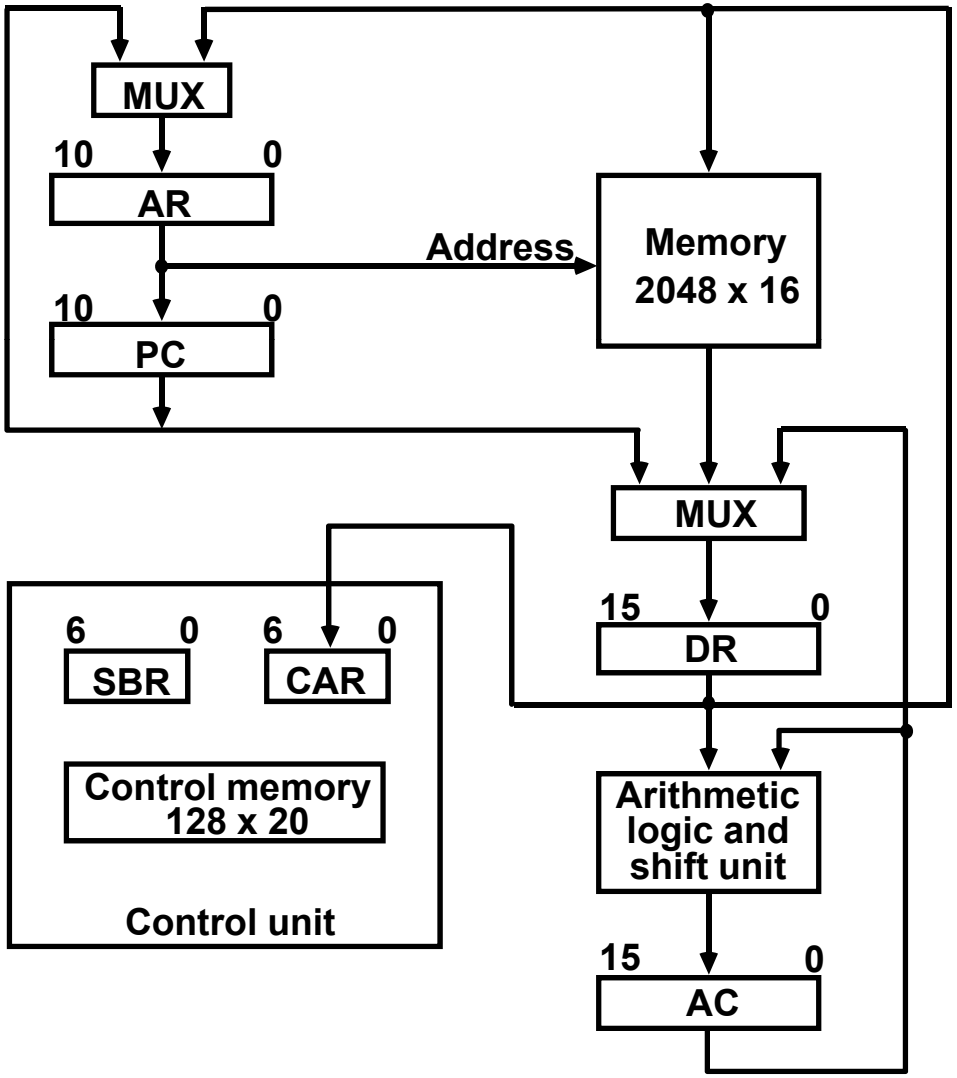


Mapping function implemented by ROM or PLA



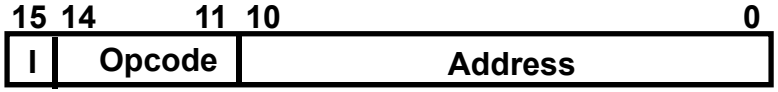
MICROPROGRAM EXAMPLE

Computer Configuration



MACHINE INSTRUCTION FORMAT

Machine instruction format

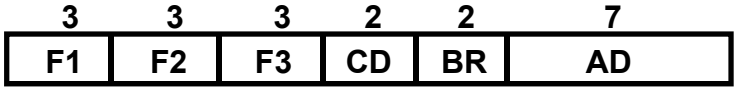


Sample machine instructions

Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

EA is the effective address

Microinstruction Format



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

MICROINSTRUCTION FIELD DESCRIPTIONS - F1,F2,F3

F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC - DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC'	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

MICROINSTRUCTION FIELD DESCRIPTIONS - CD, BR

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

SYMBOLIC MICROINSTRUCTIONS

- Symbols are used in microinstructions as in assembly language
- A symbolic microprogram can be translated into its binary equivalent by a microprogram assembler.

Sample Format

five fields: label; micro-ops; CD; BR; AD

Label: may be empty or may specify a symbolic address terminated with a colon

Micro-ops: consists of one, two, or three symbols separated by commas

CD: one of {U, I, S, Z}, where
- U: Unconditional Branch
- I: Indirect address bit
- S: Sign of AC
- Z: Zero value in AC

BR: one of {JMP, CALL, RET, MAP}

AD: one of {Symbolic address, NEXT, empty}

SYMBOLIC MICROPROGRAM - FETCH ROUTINE

During FETCH, Read an instruction from memory and decode the instruction and update PC

Sequence of microoperations in the fetch cycle:

```
AR ← PC
DR ← M[AR], PC ← PC + 1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0
```

Symbolic microprogram for the fetch cycle:

```
        ORG 64
FETCH:  PCTAR          U  JMP  NEXT
        READ, INCPC    U  JMP  NEXT
        DRTAR          U  MAP
```

Binary equivalents translated by an assembler

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

SYMBOLIC MICROPROGRAM

- Control Storage: 128 20-bit words
- The first 64 words: Routines for the 16 machine instructions
- The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: OP-code XXXX into 0XXXX00, the first address for the 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60

Partial Symbolic Microprogram

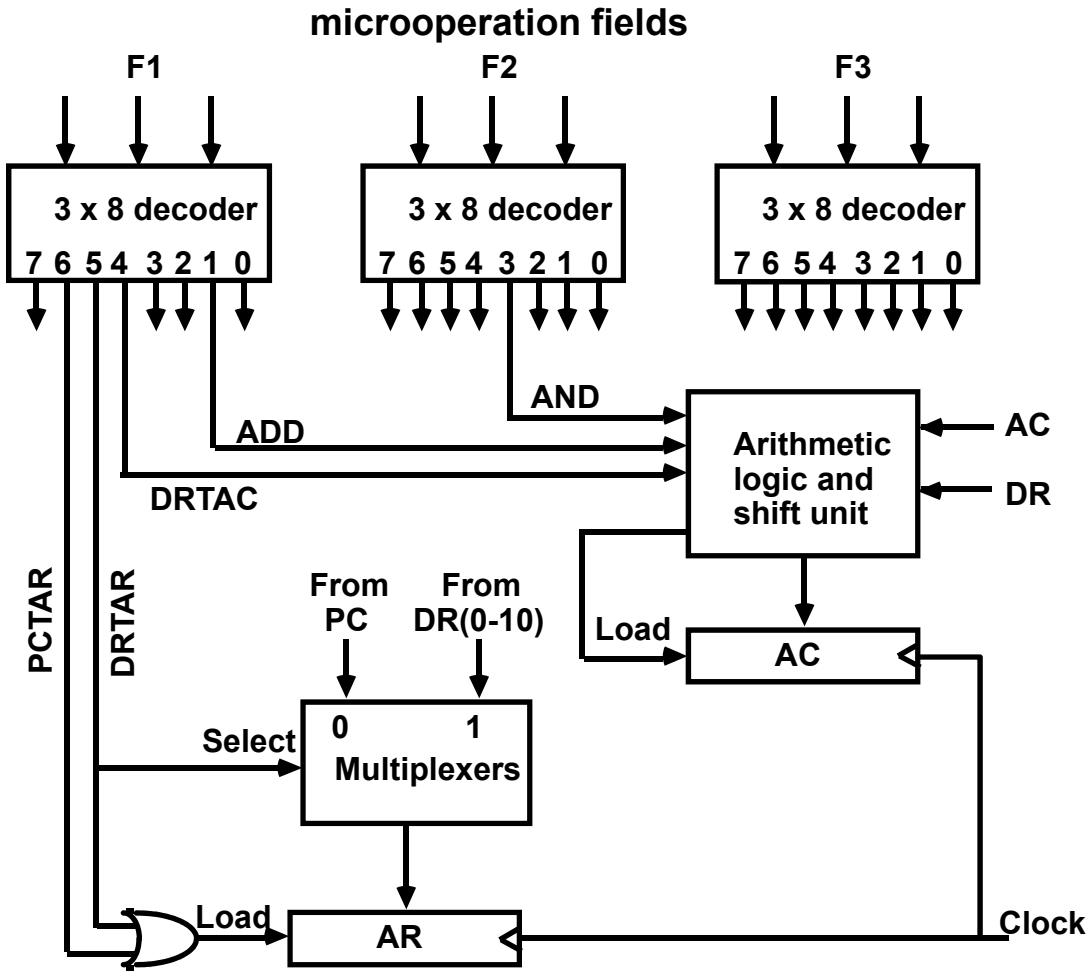
Label	Microops	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

BINARY MICROPROGRAM

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

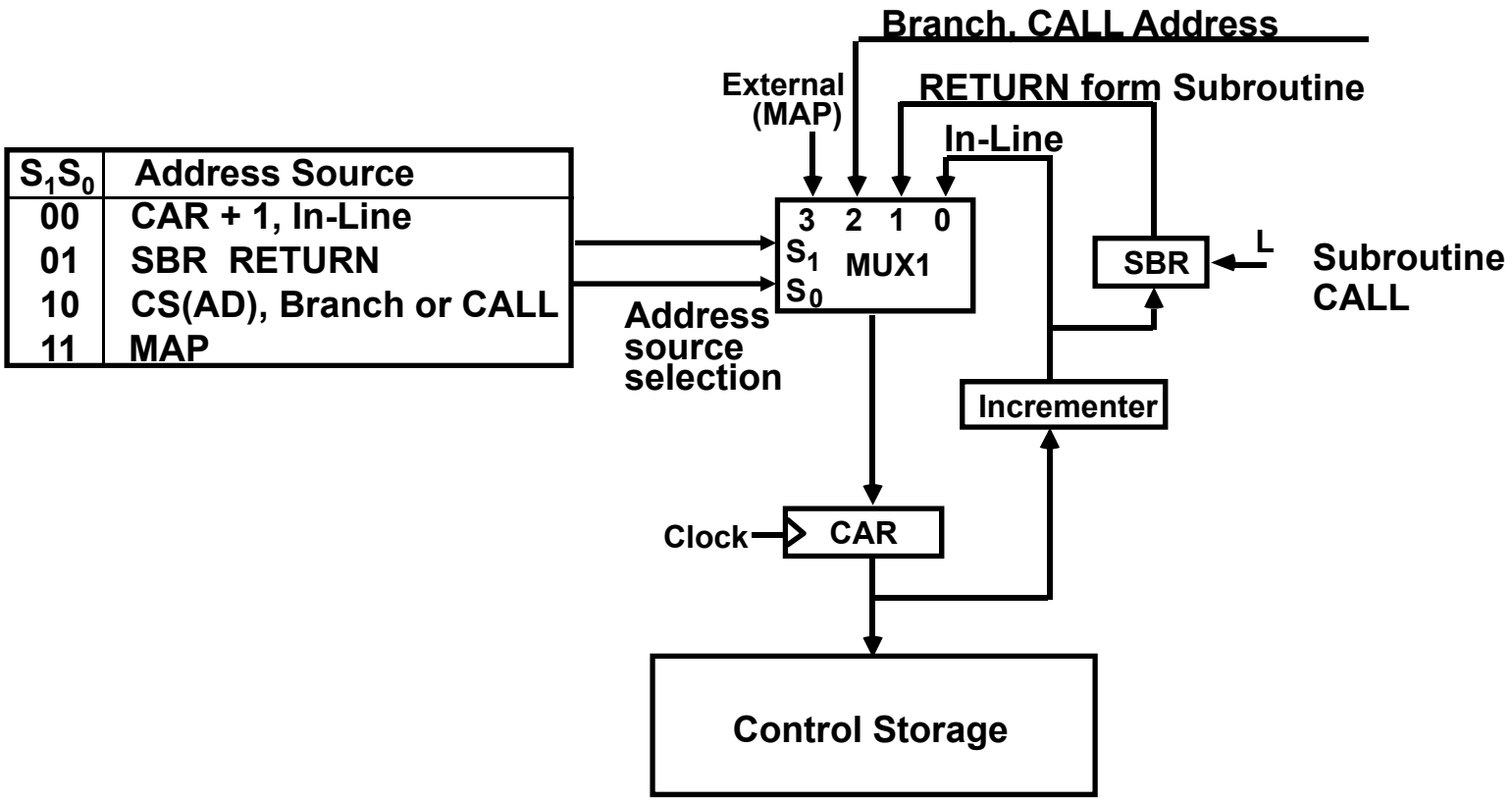
This microprogram can be implemented using ROM

DESIGN OF CONTROL UNIT
- DECODING ALU CONTROL INFORMATION -



Decoding of Microoperation Fields

MICROPROGRAM SEQUENCER
- NEXT MICROINSTRUCTION ADDRESS LOGIC -

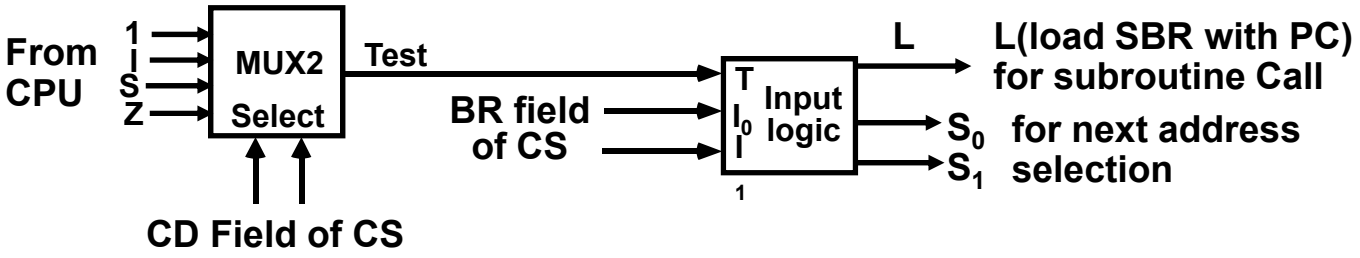


S_1S_0	Address Source
00	CAR + 1, In-Line
01	SBR RETURN
10	CS(AD), Branch or CALL
11	MAP

MUX-1 selects an address from one of four sources and routes it into a CAR

- In-Line Sequencing → CAR + 1
- Branch, Subroutine Call → CS(AD)
- Return from Subroutine → Output of SBR
- New Machine instruction → MAP

MICROPROGRAM SEQUENCER
- CONDITION AND BRANCH CONTROL -

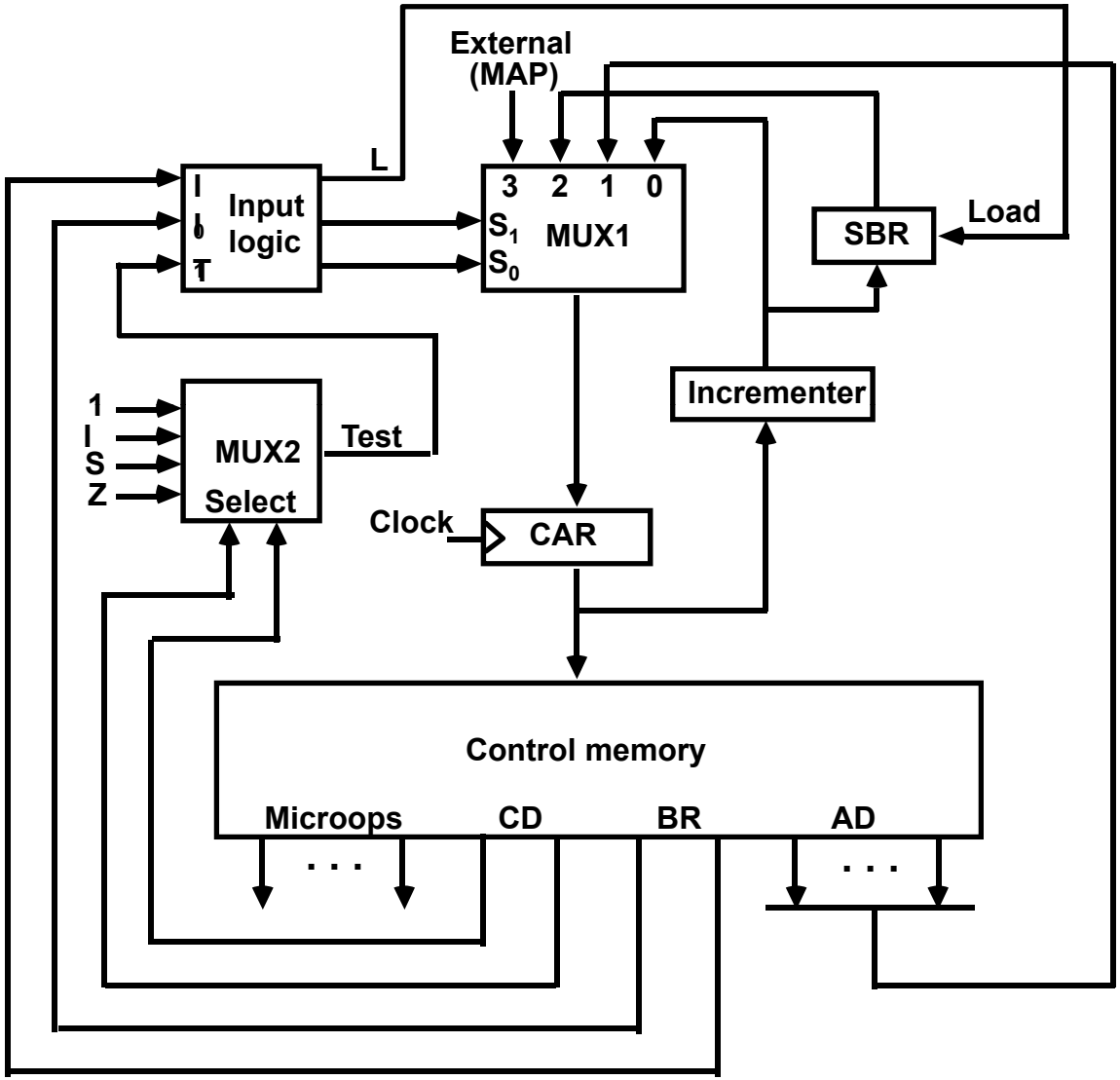


Input Logic

I_1I_0T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	01	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	01	1
10x	RET	SBR	10	0
11x	MAP	DR(11-14)	11	0

$$S_1 = I_1$$
$$S_0 = I_1I_0 + I_1'T$$
$$L = I_1'I_0T$$

MICROPROGRAM SEQUENCER



MICROINSTRUCTION FORMAT

Information in a Microinstruction
- Control Information
- Sequencing Information
- Constant
  Information which is useful when feeding into the system

These information needs to be organized in some way for
- Efficient use of the microinstruction bits
- Fast decoding

Field Encoding

- Encoding the microinstruction bits
- Encoding slows down the execution speed due to the decoding delay
- Encoding also reduces the flexibility due to the decoding hardware

HORIZONTAL AND VERTICAL MICROINSTRUCTION FORMAT

Horizontal Microinstructions

Each bit directly controls each micro-operation or each control point

Horizontal implies a long microinstruction word

Advantages: Can control a variety of components operating in parallel.
--> Advantage of efficient hardware utilization

Disadvantages: Control word bits are not fully utilized
--> CS becomes large --> Costly

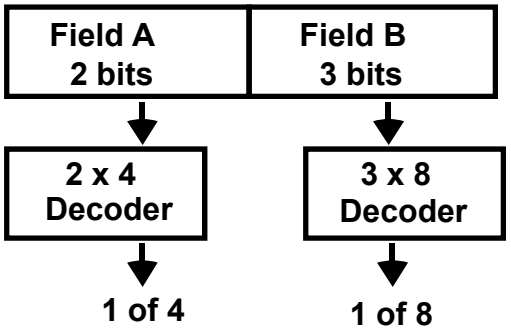
Vertical Microinstructions

A microinstruction format that is not horizontal

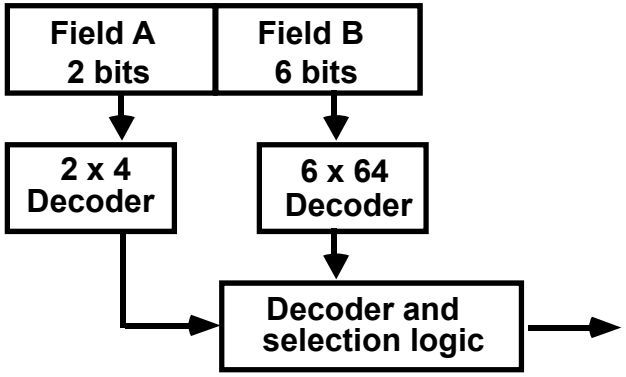
Vertical implies a short microinstruction word

Encoded Microinstruction fields
--> Needs decoding circuits for one or two levels of decoding

One-level decoding



Two-level decoding



NANOSTORAGE AND NANOINSTRUCTION

The decoder circuits in a vertical microprogram storage organization can be replaced by a ROM

=> Two levels of control storage

First level - *Control Storage*

Second level - *Nano Storage*

Two-level microprogram

First level

-*Vertical* format Microprogram

Second level

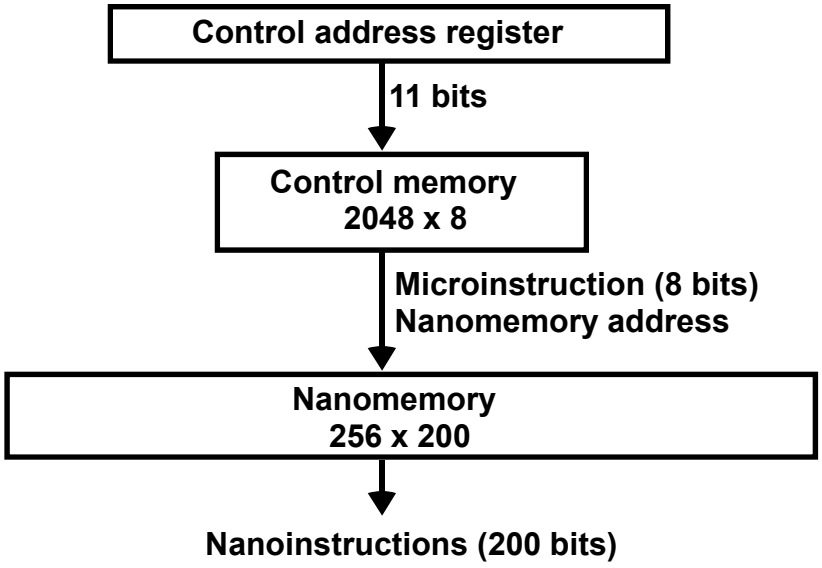
-*Horizontal* format Nanoprogram

- Interprets the microinstruction fields, thus converts a vertical microinstruction format into a horizontal nanoinstruction format.

Usually, the microprogram consists of a large number of short microinstructions, while the nanoprogram contains fewer words with longer nanoinstructions.

TWO-LEVEL MICROPROGRAMMING - EXAMPLE

* Microprogram: 2048 microinstructions of 200 bits each
* With 1-Level Control Storage: 2048 x 200 = 409,600 bits
* Assumption:
  256 distinct microinstructions among 2048
* With 2-Level Control Storage:
  Nano Storage: 256 x 200 bits to store 256 distinct nanoinstructions
  Control storage: 2048 x 8 bits
  To address 256 nano storage locations 8 bits are needed
* Total 1-Level control storage: 409,600 bits
  Total 2-Level control storage: 67,584 bits (256 x 200 + 2048 x 8)



Overview

- Instruction Set Processor (ISP)
- Central Processing Unit (CPU)
- A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.
- An instruction is executed by carrying out a sequence of more rudimentary operations.

Fundamental Concepts

- Processor fetches one instruction at a time and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

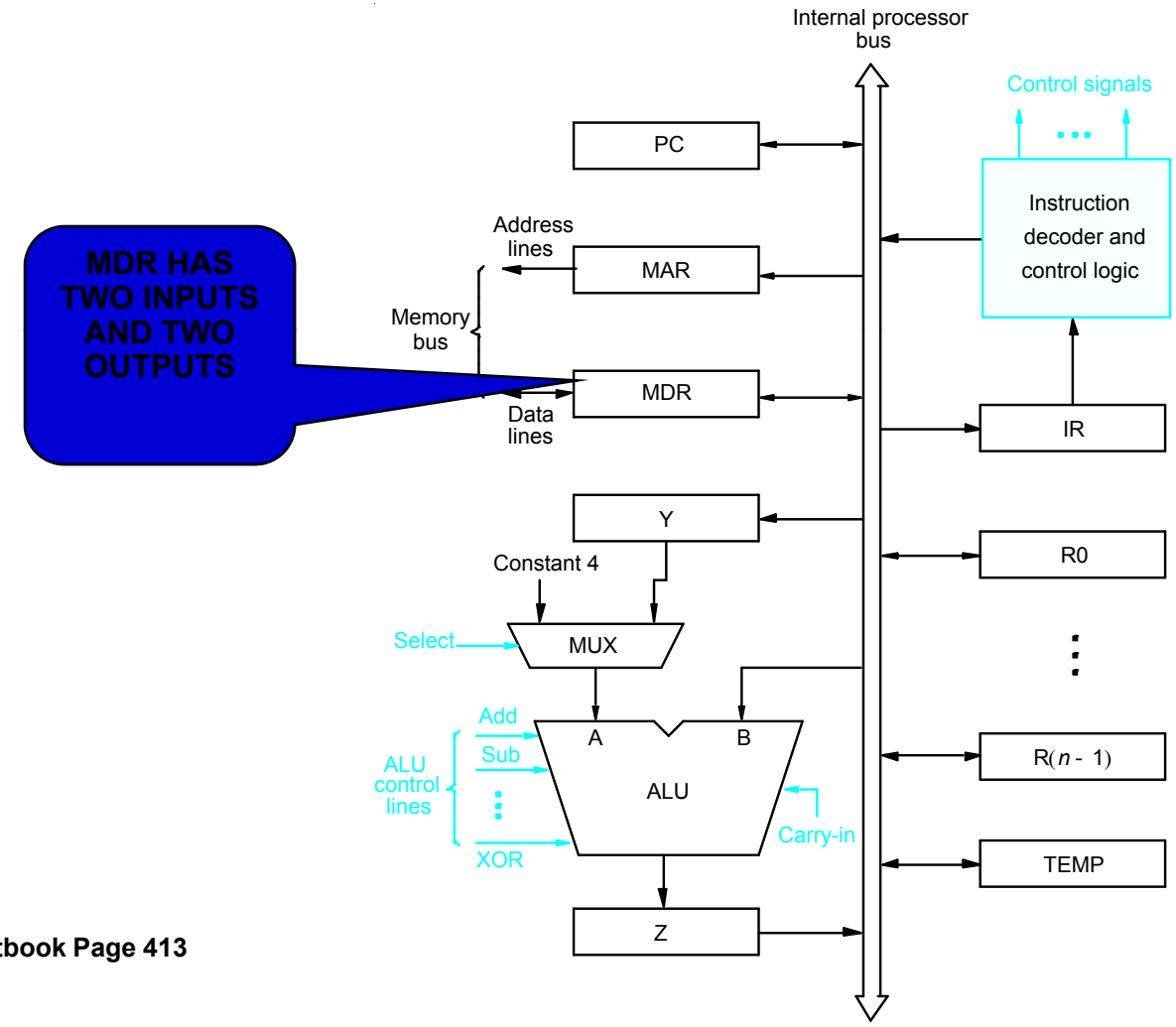
$$IR \leftarrow [[PC]]$$

- Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).

$$PC \leftarrow [PC] + 4$$

- Carry out the actions specified by the instruction in the IR (execution phase).

Processor Organization



Datapath

Textbook Page 413

Figure 7.1. Single-bus organization of the datapath inside a proc

Executing an Instruction

- Transfer a word of data from one processor register to another or to the ALU.
- Perform an arithmetic or a logic operation and store the result in a processor register.
- Fetch the contents of a given memory location and load them into a processor register.
- Store a word of data from a processor register into a given memory location.

Register Transfers

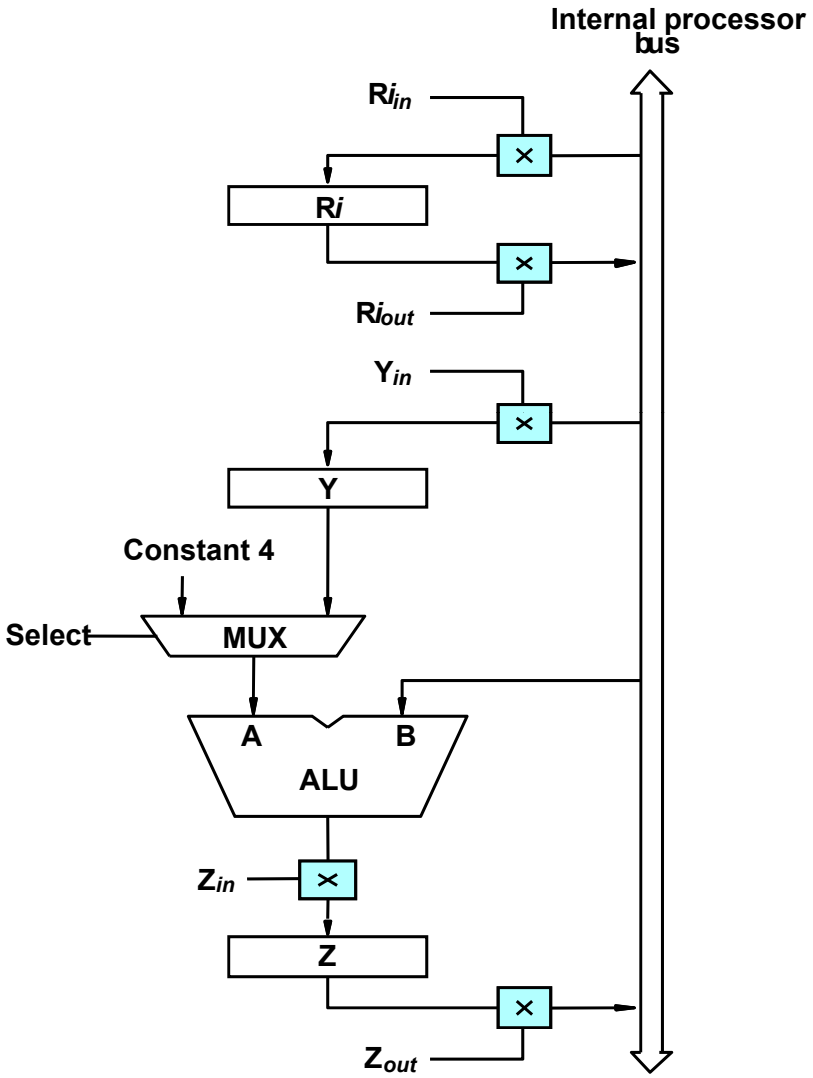


Figure 7.2. Input and output gating for the registers in Figure 7.1.

Register Transfers

- **All operations and data transfers are controlled by the processor clock.**

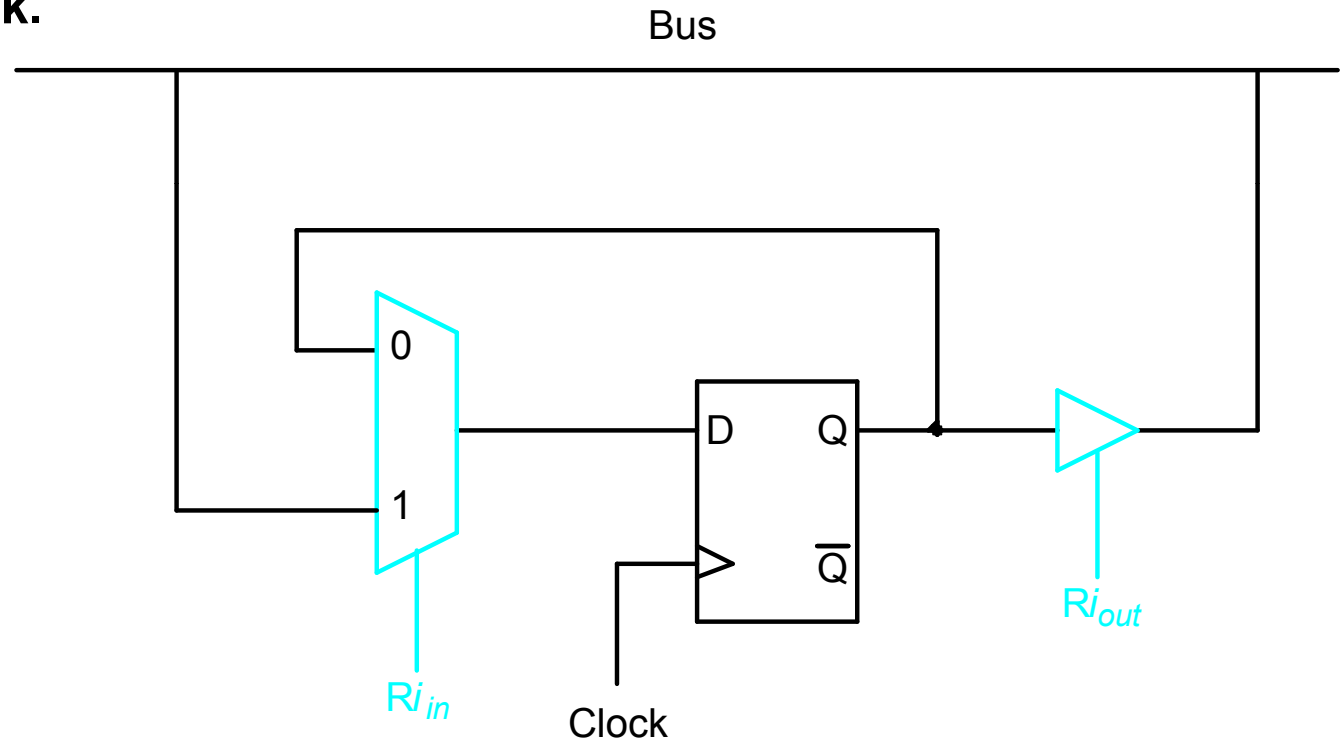


Figure 7.3. Input and output gating for one register bit.

Performing an Arithmetic or Logic Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?

1. R1out, Yin
2. R2out, SelectY, Add, Zin
3. Zout, R3in

Fetching a Word from Memory

- **Address into MAR; issue Read operation; data into MDR.**

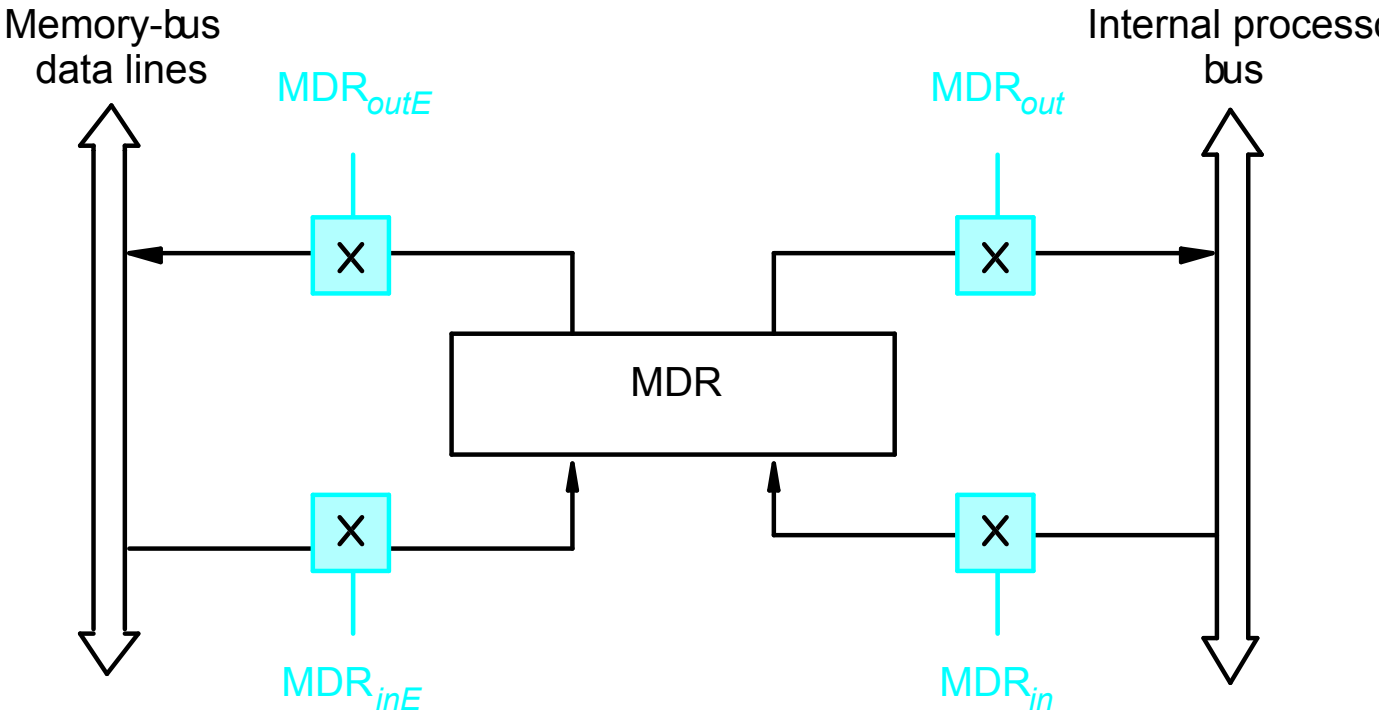


Figure 7.4. Connection and control signals for register MDR.

Fetching a Word from Memory

- The response time of each memory access varies (cache miss, memory-mapped I/O,…).
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed (Memory-Function-Completed, MFC).
- Move (R1), R2
  - MAR ← [R1]
  - Start a Read operation on the memory bus
  - Wait for the MFC response from the memory
  - Load MDR from the memory bus
  - R2 ← [MDR]

Timing

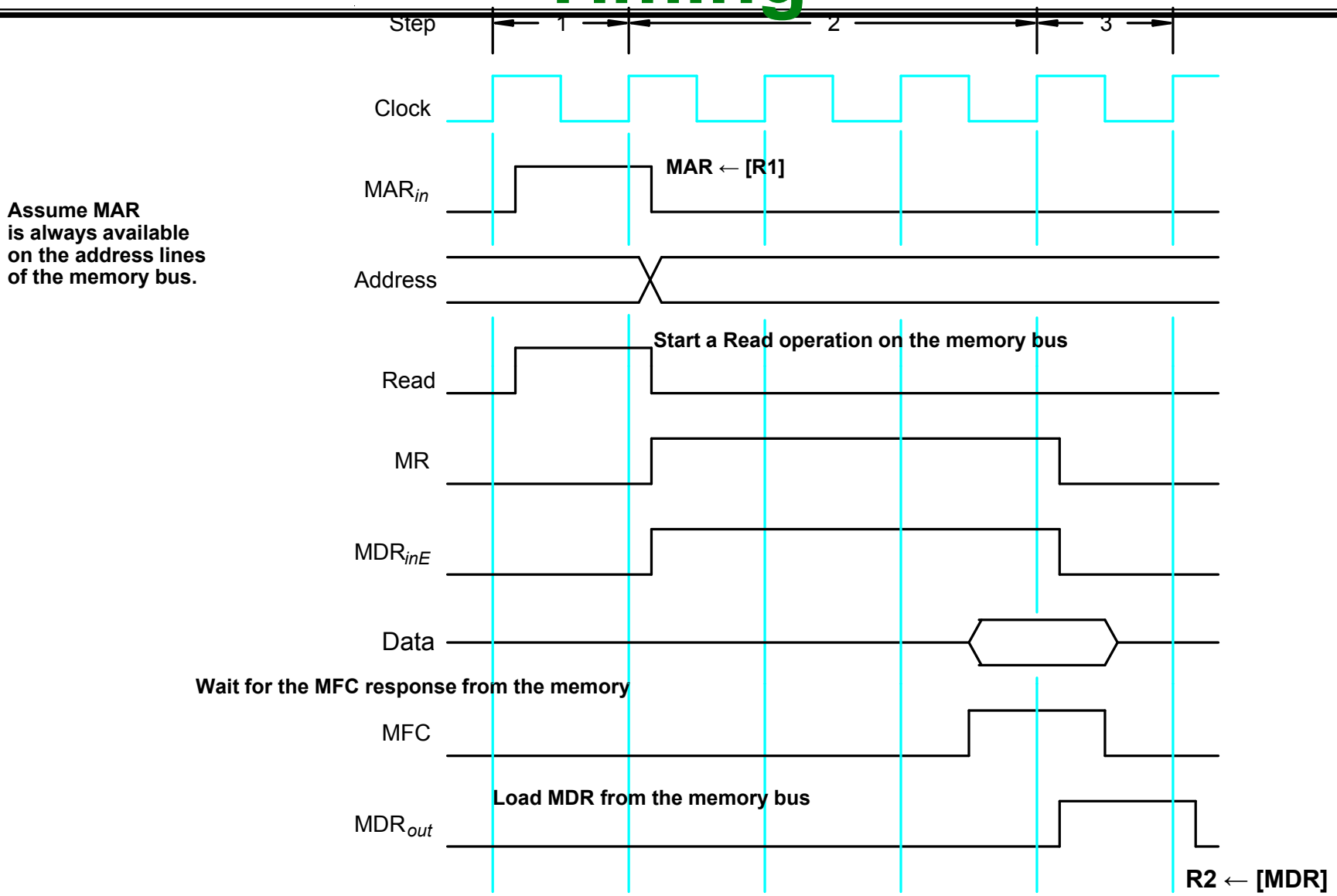


Figure 7.5. Timing of a memory Read operation

Execution of a Complete Instruction

- Add (R3), R1
- Fetch the instruction
- Fetch the first operand (the contents of the memory location pointed to by R3)
- Perform the addition
- Load the result into R1

Architecture

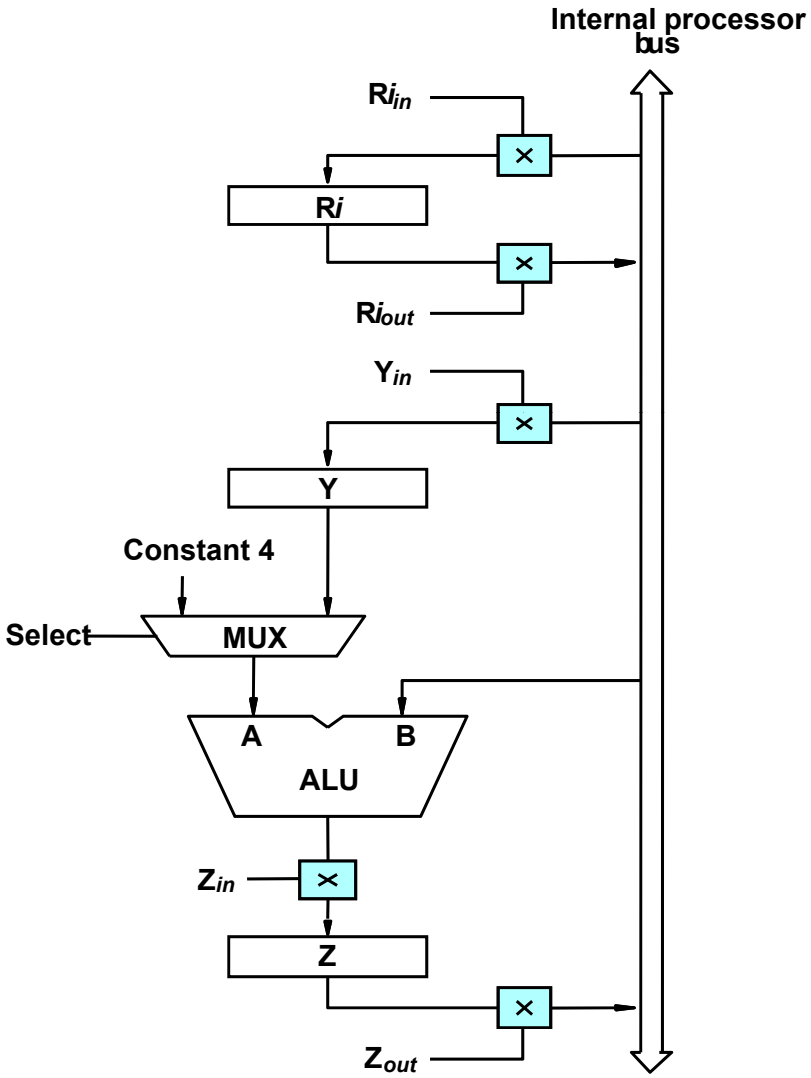


Figure 7.2. Input and output gating for the registers in Figure 7.1.

Execution of a Complete Instruction

Add (R3), R1

Step	Action
1	PC_{out} , MAR_{in} , Read, Select4,Add, Z_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMF C
3	MDR_{out} , IR_{in}
4	$R3_{out}$, MAR_{in} , Read
5	$R1_{out}$, Y_{in} , WMF C
6	MDR_{out} , SelectY, Add, Z_{in}
7	Z_{out} , $R1_{in}$, End

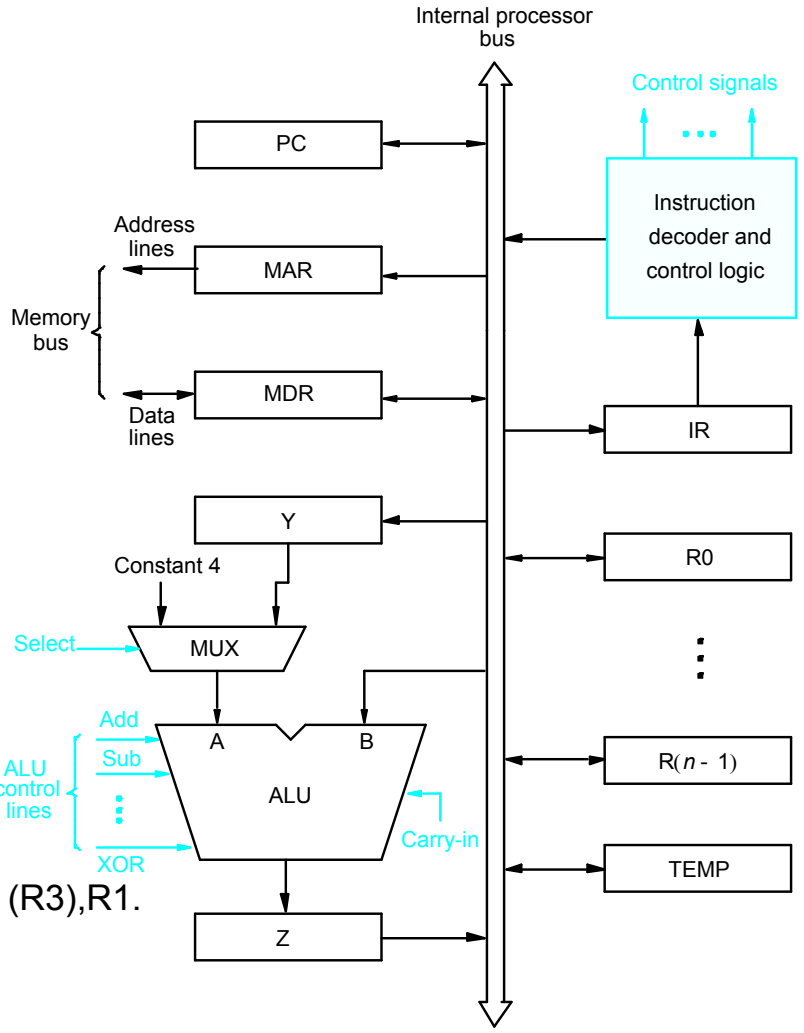


Figure 7.6. Control sequence for execution of the instruction Add (R3),R1.

Figure 7.1. Single-bus organization of the datapath inside a processor.

Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Conditional branch

Execution of Branch Instructions

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7. Control sequence for an unconditional branch instruction.

Multiple-Bus Organization

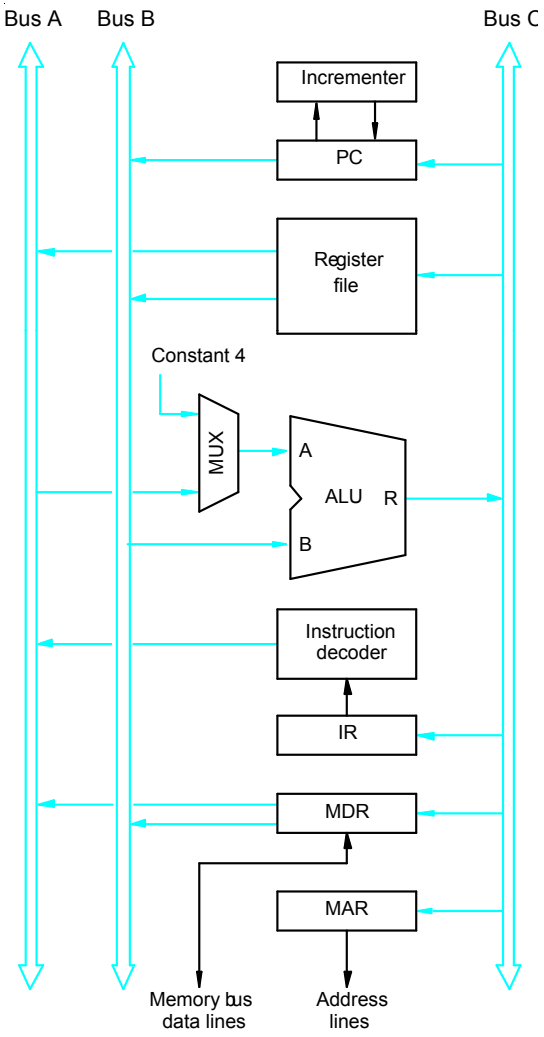


Figure 7.8. Three-bus organization of the datapath.

Multiple-Bus Organization

- **Add R4, R5, R6**

Step	Action
1	PC$_{out}$, R=B, MAR$_{in}$, Read, IncPC
2	WMF C
3	MDR$_{outB}$, R=B, IR$_{in}$
4	R4$_{outA}$, R5$_{outB}$, SelectA, Add, R6$_{in}$, End

Figure 7.9. Control sequence for the instruction. Add R4,R5,R6, for the three-bus organization in Figure 7.8.

Quiz

- **What is the control sequence for execution of the instruction**

  Add R1, R2

  including the instruction fetch phase? (Assume single bus architecture)

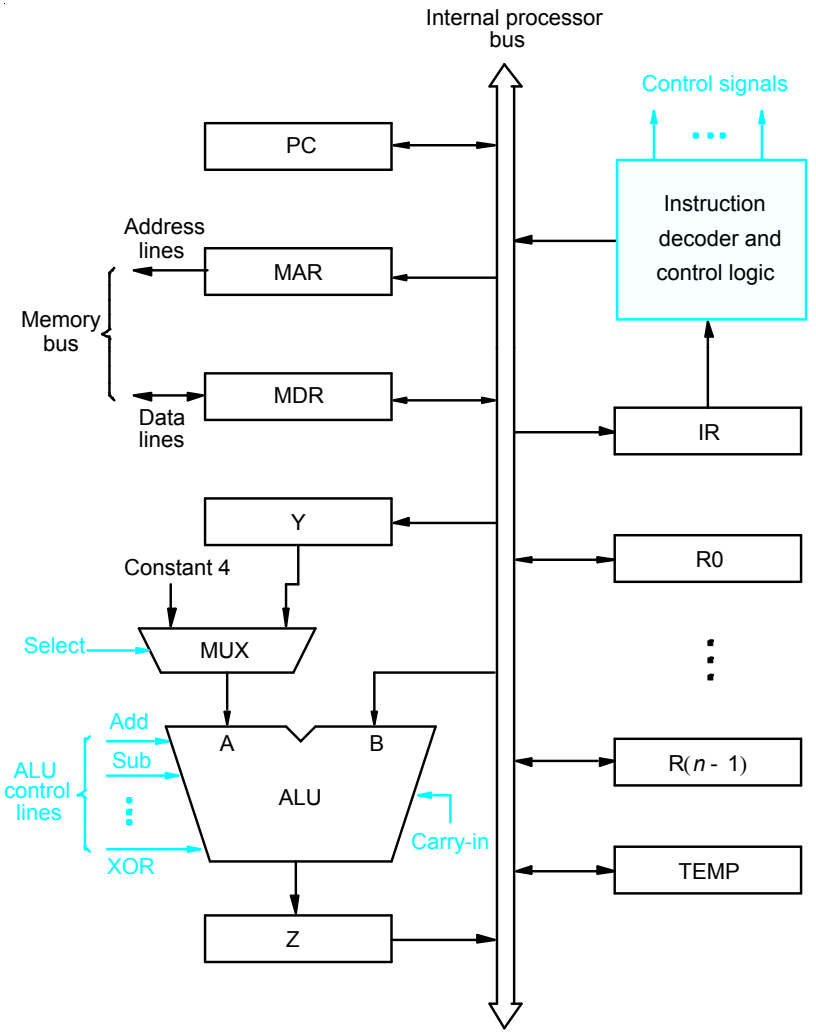


Figure 7.1. Single-bus organization of the datapath inside a proc

Control Unit Organization

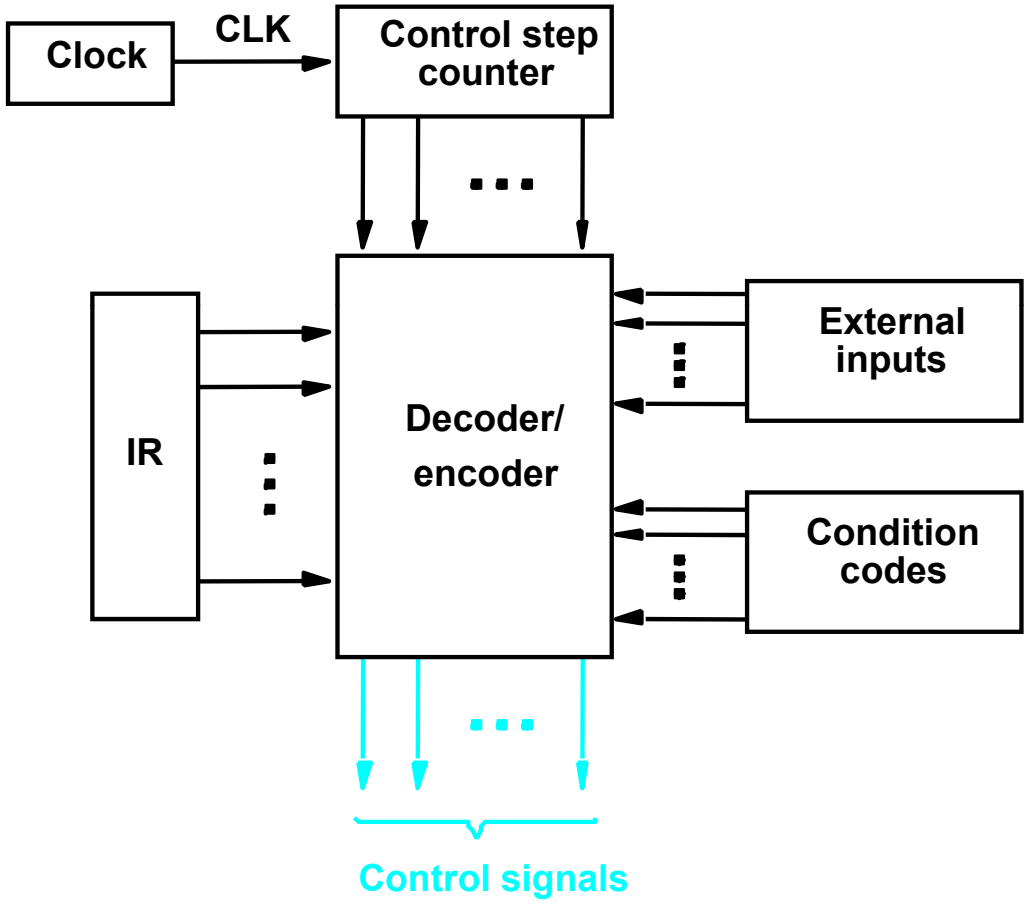


Figure 7.10. Control unit organization.

Detailed Block Description

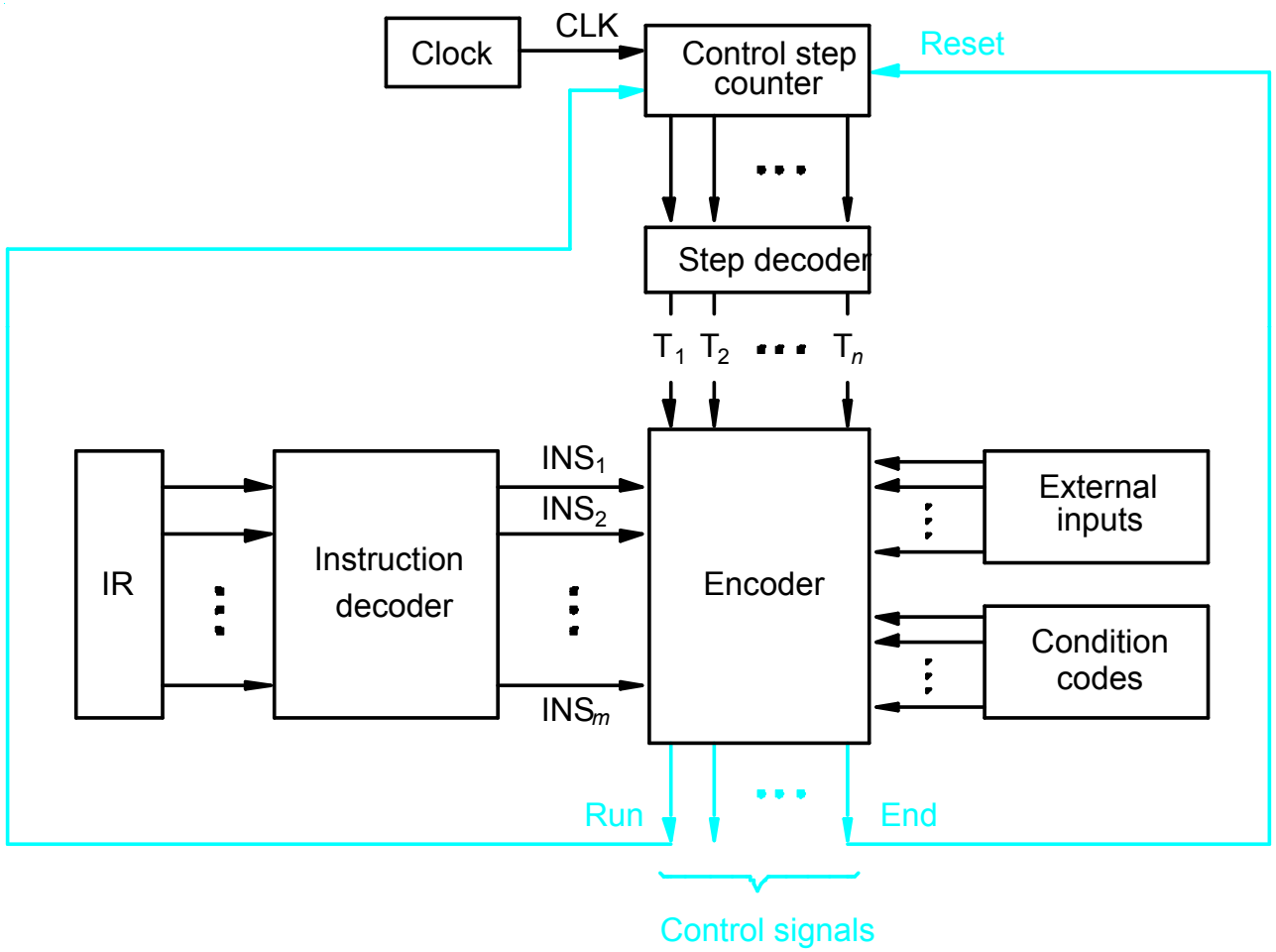


Figure 7.11. Separation of the decoding and encoding functio

Generating Z_{in}

- $Z_{in} = T_1 + T_6 \bullet ADD + T_4 \bullet BR + \ldots$

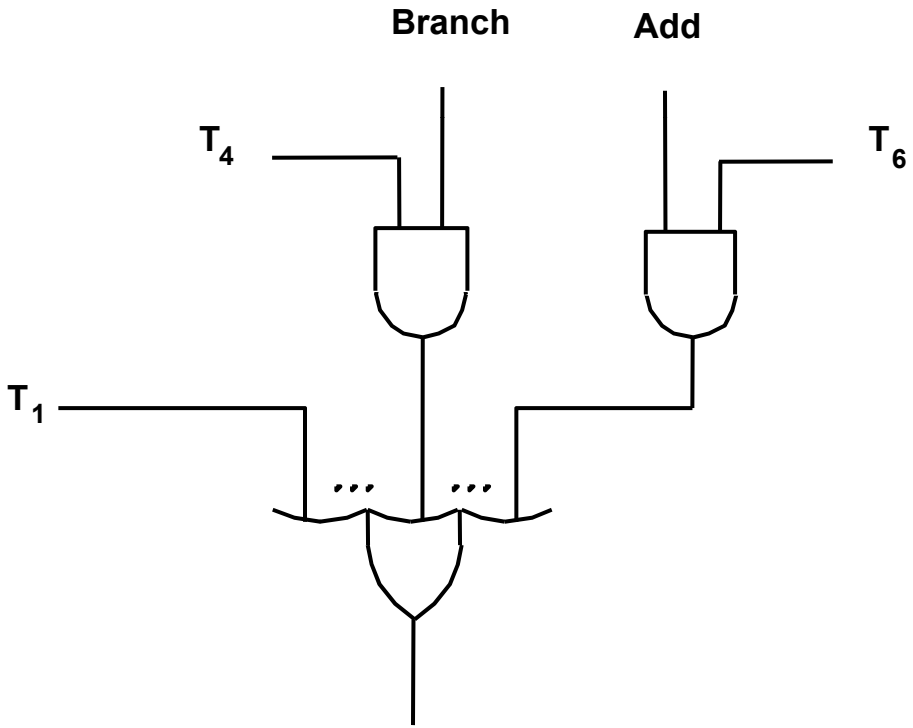


Figure 7.12. Generation of the Z_{in} control signal for the processor in Figure 7.1.

Generating End

- $End = T_7 \bullet ADD + T_5 \bullet BR + (T_5 \bullet N + T_4 \bullet \overline{N}) \bullet BRN + \ldots$

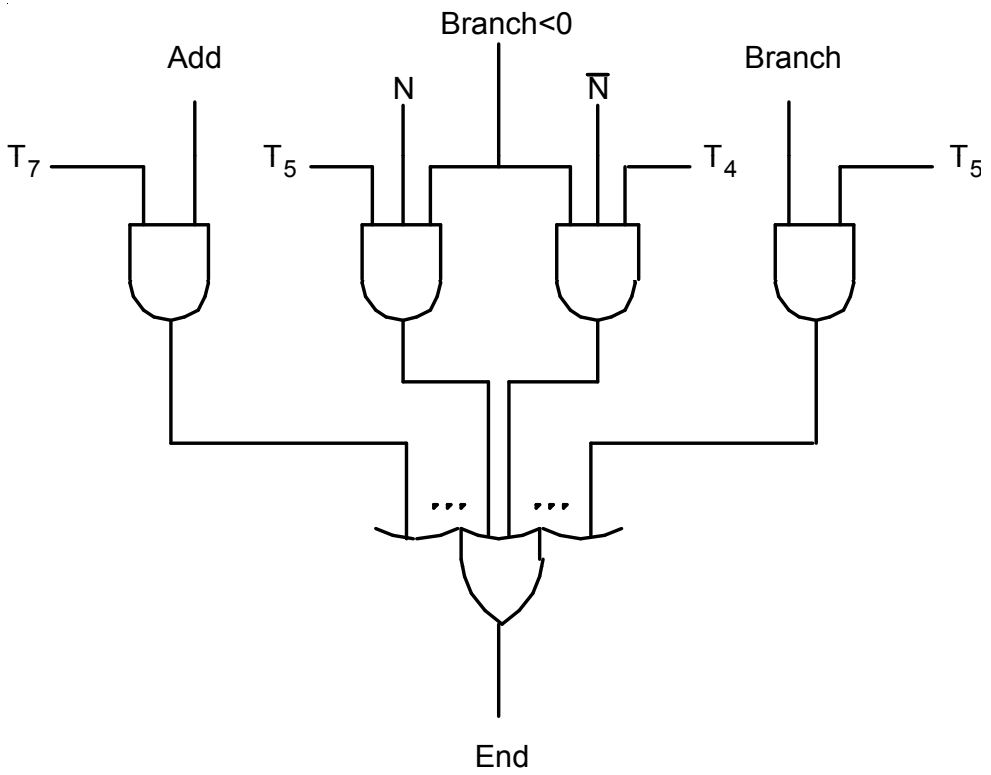


Figure 7.13. Generation of the End control signal.

A Complete Processor

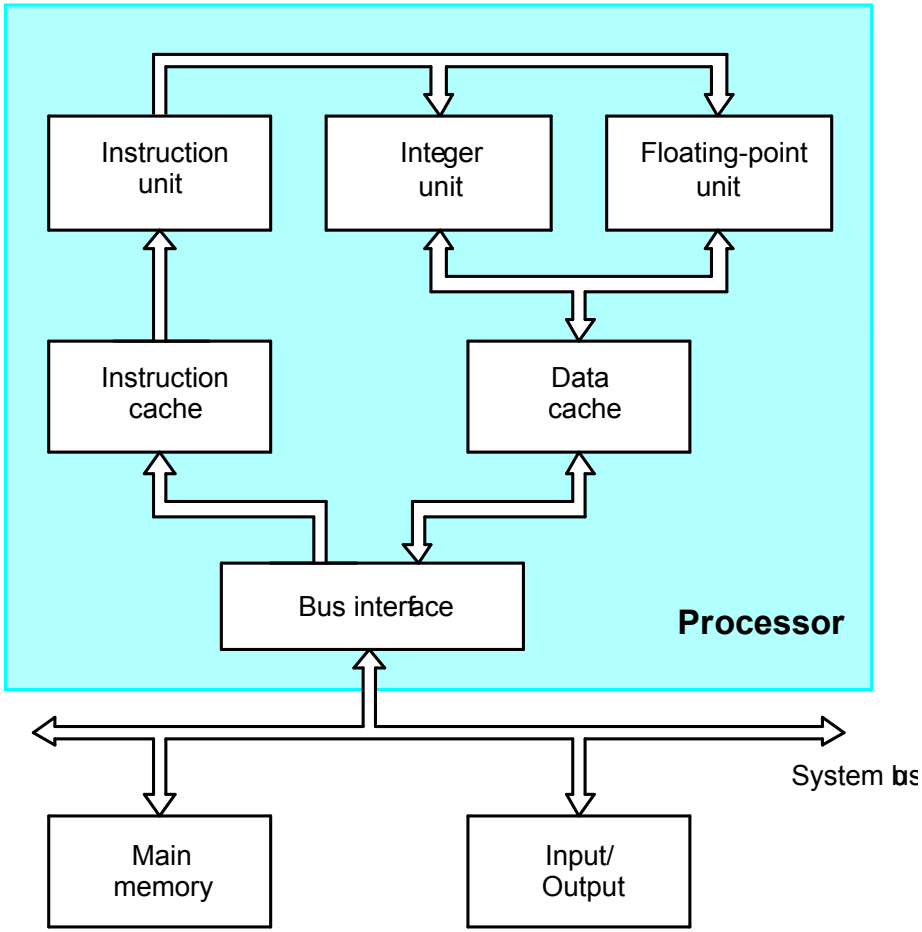


Figure 7.14. Block diagram of a complete processor

Overview

- **Control signals are generated by a program similar to machine language programs.**
- **Control Word (CW); microroutine; microinstruction**

Micro - instruction	...	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	...
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6.

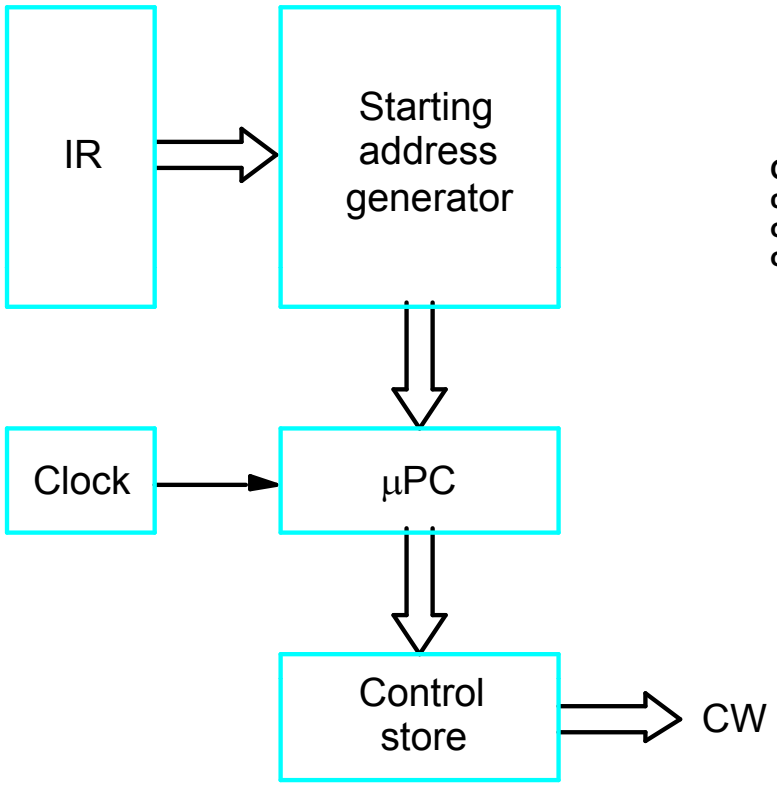
Overview

Step	Action
1	PC_{out} , MAR_{in} , Read, Select4,Add, Z_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMF C
3	MDR_{out} , IR_{in}
4	$R3_{out}$, MAR_{in} , Read
5	$R1_{out}$, Y_{in} , WMF C
6	MDR_{out} , SelectY, Add, Z_{in}
7	Z_{out} , $R1_{in}$, End

Figure 7.6. Control sequence for execution of the instruction Add (R3),R1.

Overview

- **Control store**



One function cannot be carried out by this simple organization.

Figure 7.16. Basic organization of a microprogrammed control

Overview

- The previous organization cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.
- Use conditional branch microinstruction.

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
...	...
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17. Microroutine for the instruction Branch<0.

Overview

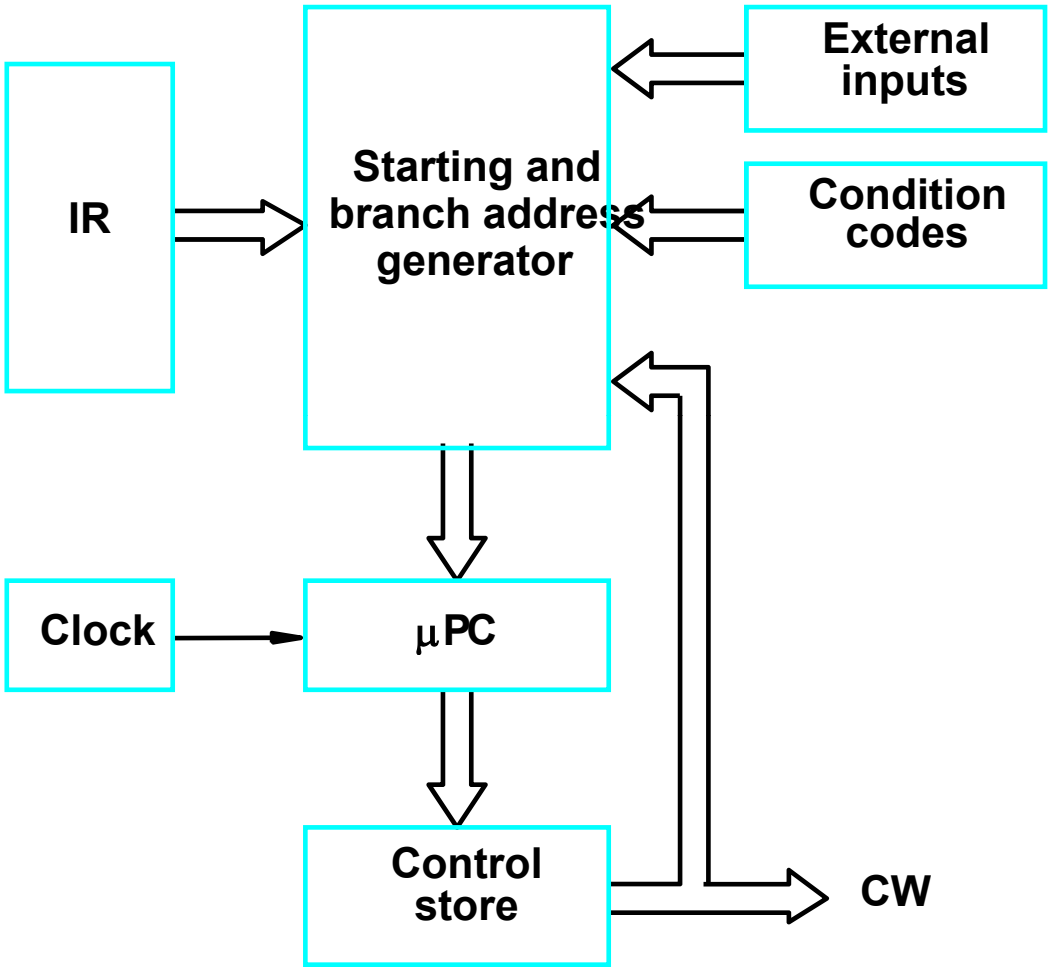


Figure 7.18. Organization of the control unit to allow conditional branching in the microprogram.

Microinstructions

- A straightforward way to structure microinstructions is to assign one bit position to each control signal.
- However, this is very inefficient.
- The length can be reduced: most signals are not needed simultaneously, and many signals are mutually exclusive.
- All mutually exclusive signals are placed in the same group in binary coding.

Partial Format for the Microinstructions

Microinstruction

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$	1111: XOR	
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	(16 ALU functions)	
0101: $R1_{out}$	101: $R1_{in}$			
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY 1: Select4	0: No action 1: WMFC	0: Continue 1: End

What is the price paid for this scheme?

Figure 7.10. An example of a partial format for field-encoded microinstructions

Further Improvement

- Enumerate the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can then be assigned a distinct code.
- Vertical organization
- Horizontal organization

Microprogram Sequencing

- If all microprograms require only straightforward sequential execution of microinstructions except for branches, letting a μPC governs the sequencing would be efficient.
- However, two disadvantages:
  - Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store.
  - Longer execution time because it takes more time to carry out the required branches.
- Example: Add src, Rdst
- Four addressing modes: register, autoincrement, autodecrement, and indexed (with indirect forms).

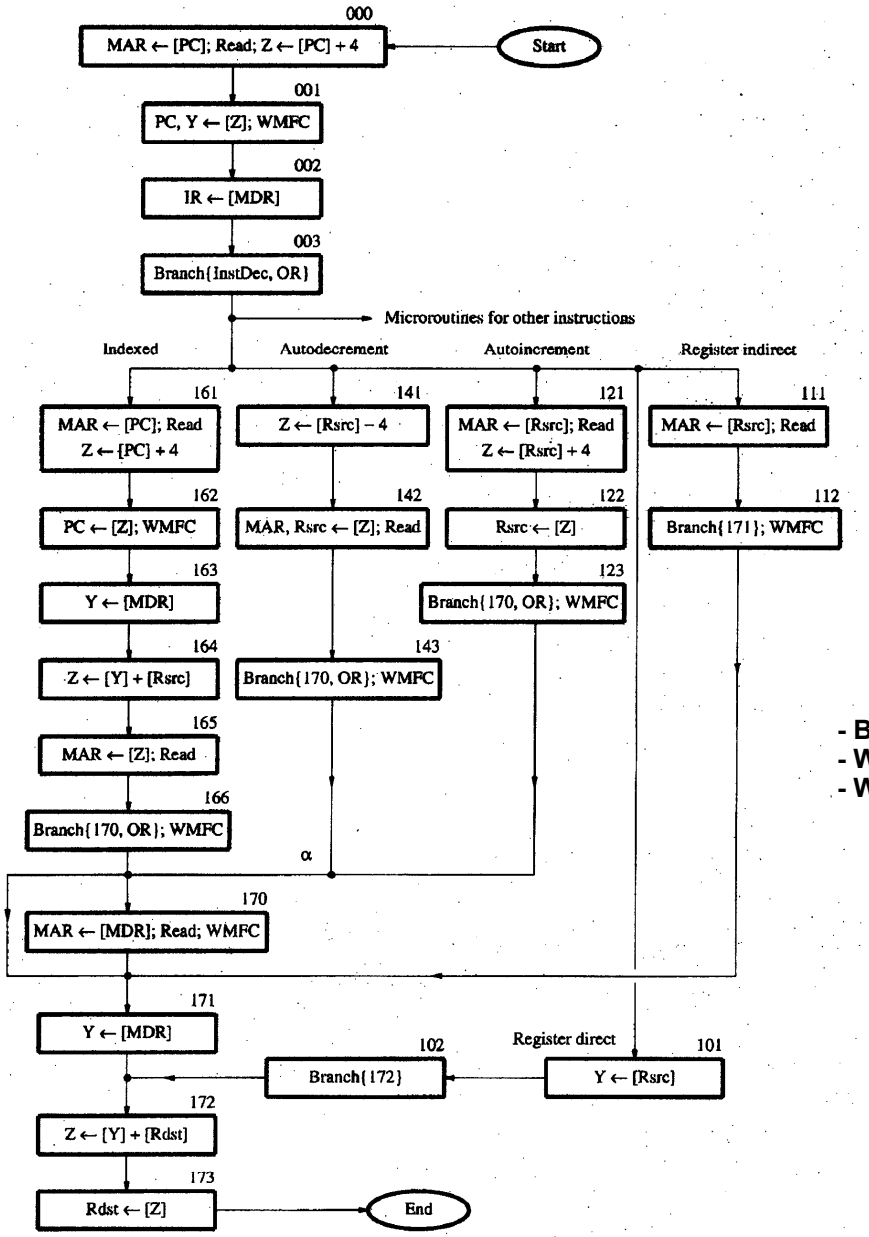
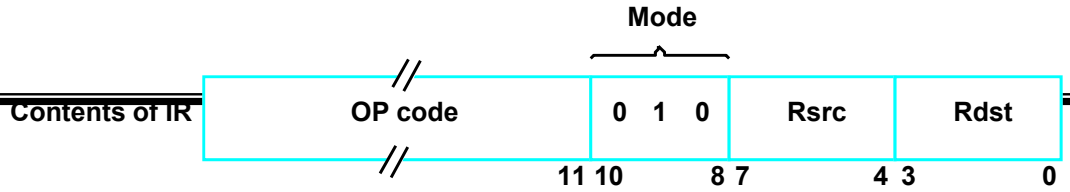


Figure 7.20. Flowchart of a microprogram for the Add src,Rdst instruction.

- Bit-ORing
- Wide-Branch Addressing
- WMFC



Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch μ{PC $\leftarrow$ 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \times [\overline{IR_9}] \times [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch μ{PC $\leftarrow$ 170; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, Select, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21. Microinstruction for Add (Rsrc)+,Rdst.

Note: Microinstruction at location 170 is not executed for this addressing mode.

Microinstructions with Next-Address Field

- The microprogram we discussed requires several branch microinstructions, which perform no useful operation in the datapath.
- A powerful alternative approach is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.
- Pros: separate branch microinstructions are virtually eliminated; few limitations in assigning addresses to microinstructions.
- Cons: additional bits for the address field (around 1/6)

Microinstructions with Next-Address Field

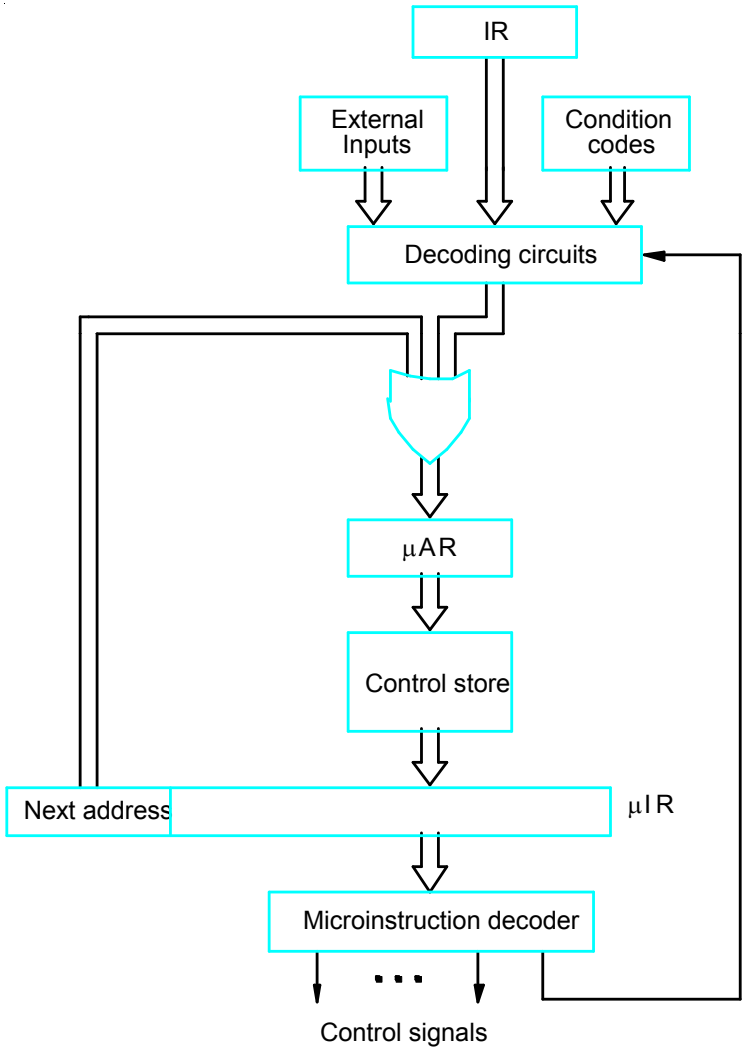


Figure 7.22. Microinstruction-sequencing organization.

Microinstruction

F0	F1	F2	F3

F0 (8 bits)	F1 (3 bits)	F2 (3 bits)	F3 (3 bits)
Address of next microinstruction	000: No transfer 001: PC_{out} 010: MDR_{out} 011: Z_{out} 100: $Rsrc_{out}$ 101: $Rdst_{out}$ 110: $TEMP_{out}$	000: No transfer 001: PC_{in} 010: IR_{in} 011: Z_{in} 100: $Rsrc_{in}$ 101: $Rdst_{in}$	000: No transfer 001: MAR_{in} 010: MDR_{in} 011: $TEMP_{in}$ 100: Y_{in}

F4	F5	F6	F7

F4 (4 bits)	F5 (2 bits)	F6 (1 bit)	F7 (1 bit)
0000: Add 0001: Sub ⋮ 1111: XOR	00: No action 01: Read 10: Write	0: SelectY 1: Select4	0: No action 1: WMFC

F8	F9	F10

F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
0: NextAdrs 1: InstDec	0: No action 1: OR_{mode}	0: No action 1: OR_{indsrc}

Figure 7.23. Format for microinstructions in the example of Section 7

Implementation of the Microroutine

Octal address	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
0 0 0	0 0 0 0 0 0 0 1	0 0 1	0 1 1	0 0 1	0 0 0 0	0 1	1	0	0	0	0
0 0 1	0 0 0 0 0 0 1 0	0 1 1	0 0 1	1 0 0	0 0 0 0	0 0	0	1	0	0	0
0 0 2	0 0 0 0 0 0 1 1	0 1 0	0 1 0	0 0 0	0 0 0 0	0 0	0	0	0	0	0
0 0 3	0 0 0 0 0 0 0 0	0 0 0	0 0 0	0 0 0	0 0 0 0	0 0	0	0	1	1	0
1 2 1	0 1 0 1 0 0 1 0	1 0 0	0 1 1	0 0 1	0 0 0 0	0 1	1	0	0	0	0
1 2 2	0 1 1 1 1 0 0 0	0 1 1	1 0 0	0 0 0	0 0 0 0	0 0	0	1	0	0	1
1 7 0	0 1 1 1 1 0 0 1	0 1 0	0 0 0	0 0 1	0 0 0 0	0 1	0	1	0	0	0
1 7 1	0 1 1 1 1 0 1 0	0 1 0	0 0 0	1 0 0	0 0 0 0	0 0	0	0	0	0	0
1 7 2	0 1 1 1 1 0 1 1	1 0 1	0 1 1	0 0 0	0 0 0 0	0 0	0	0	0	0	0
1 7 3	0 0 0 0 0 0 0 0	0 1 1	1 0 1	0 0 0	0 0 0 0	0 0	0	0	0	0	0

Figure 7.24. Implementation of the microroutine of Figure 7.21 usi
next-microinstruction address fie(See Figure 7.23 for encoded sign

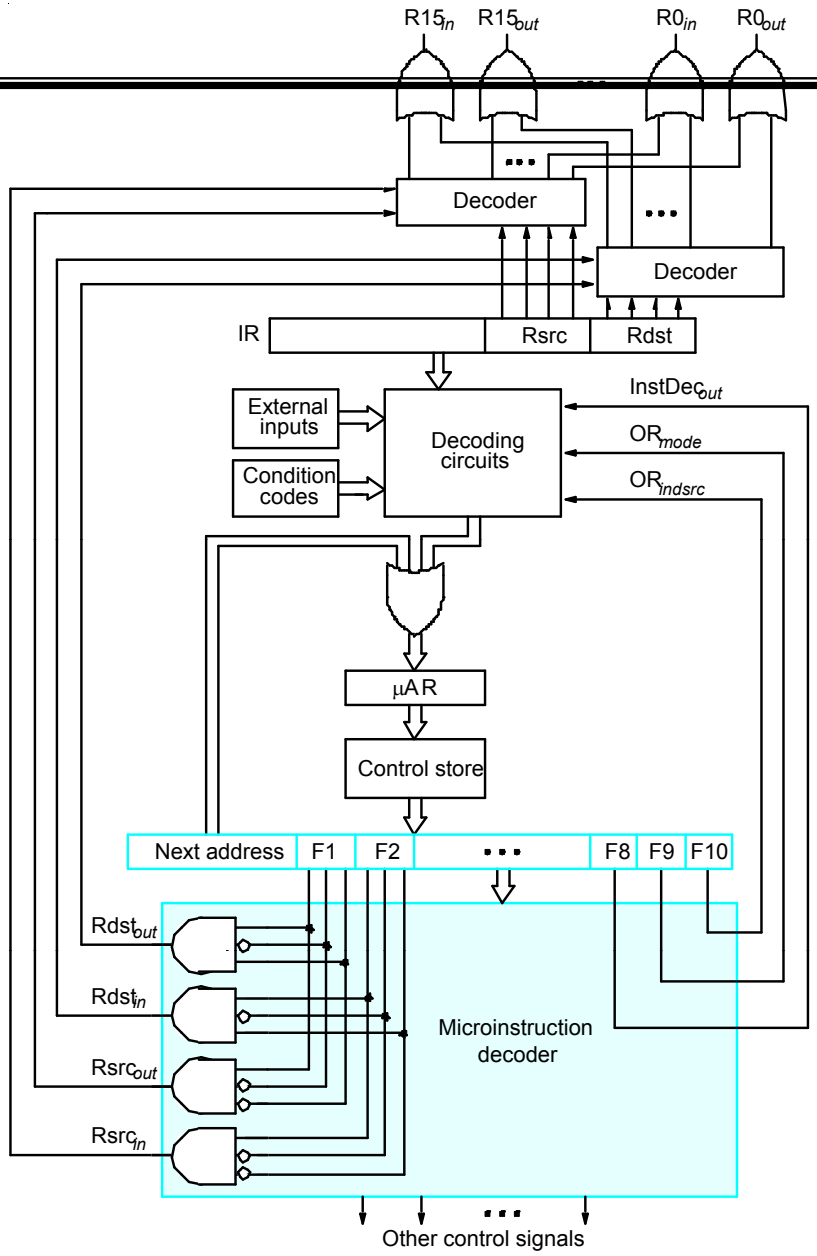


Figure 7.25. Some details of the control-signal-generating circuitry.

bit-ORing

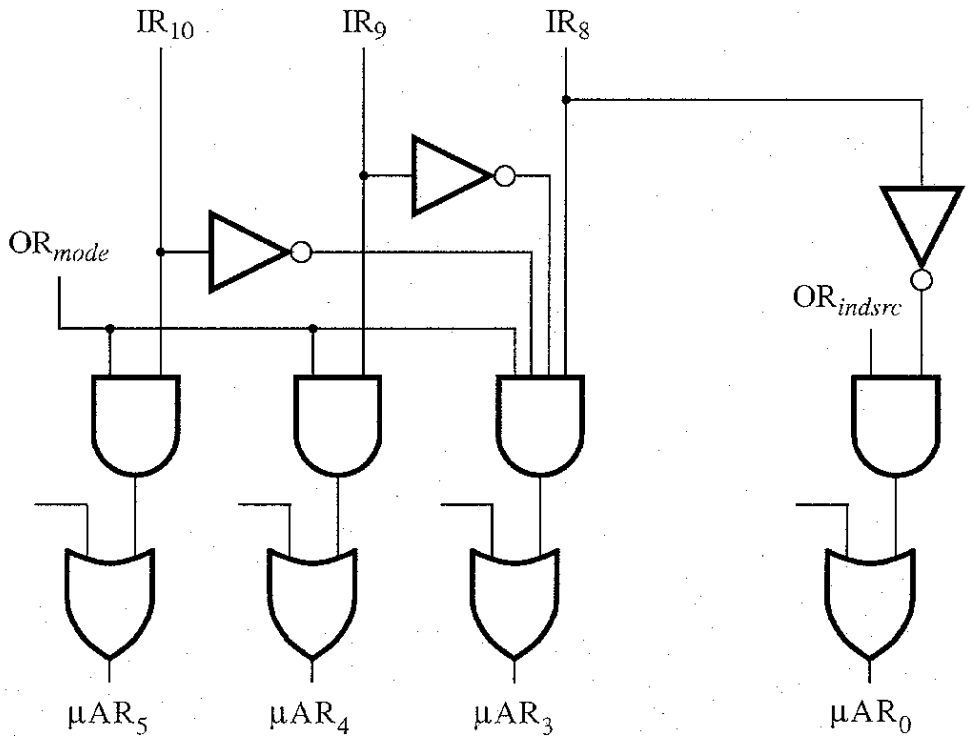


Figure 7.26. Control circuitry for bit-ORing

(part of the decoding circuits in Figure 7.25).

PIPELINING AND VECTOR PROCESSING

- Parallel Processing
- Pipelining
- Arithmetic Pipeline
- Instruction Pipeline
- RISC Pipeline
- Vector Processing
- Array Processors

PARALLEL PROCESSING

Execution of *Concurrent Events* in the computing process to achieve faster *Computational Speed*

Levels of Parallel Processing

- Job or Program level
- Task or Procedure level
- Inter-Instruction level
- Intra-Instruction level

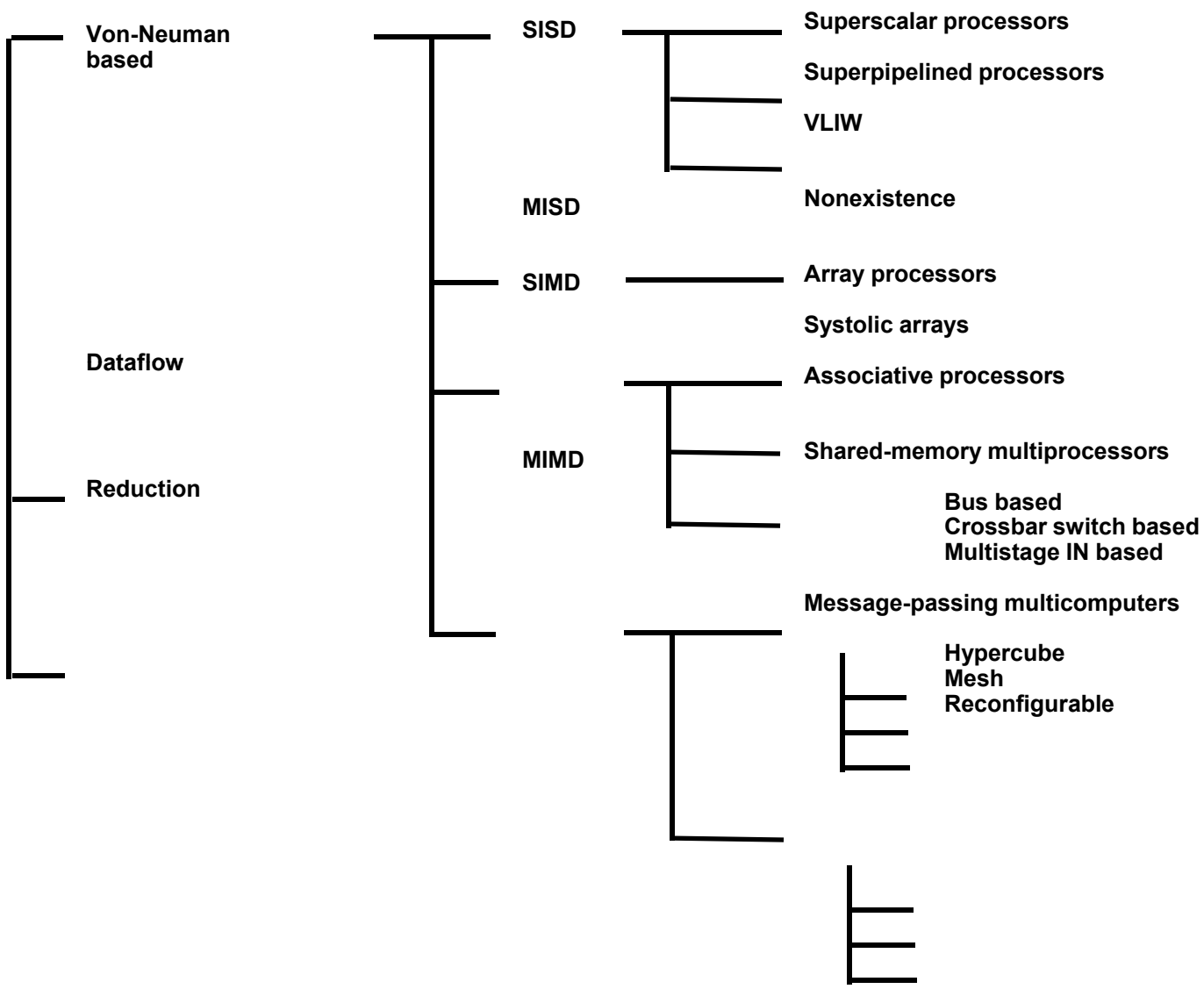
PARALLEL COMPUTERS

Architectural Classification

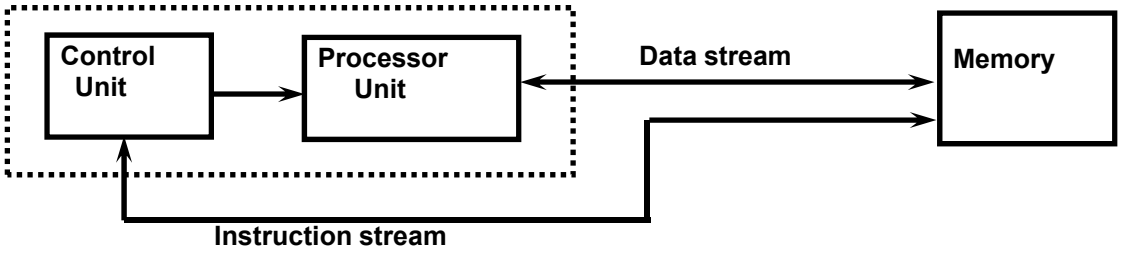
- Flynn's classification
  - » Based on the multiplicity of *Instruction Streams* and *Data Streams*
  - » Instruction Stream
    - Sequence of Instructions read from memory
  - » Data Stream
    - Operations performed on the data in the processor

		Number of *Data Streams*	
		Single	Multiple
Number of *Instruction Streams*	Single	SISD	SIMD
	Multiple	MISD	MIMD

COMPUTER ARCHITECTURES FOR PARALLEL PROCESSING



SISD COMPUTER SYSTEMS



Characteristics

- Standard von Neumann machine
- Instructions and data are stored in memory
- One operation at a time

Limitations

Von Neumann bottleneck

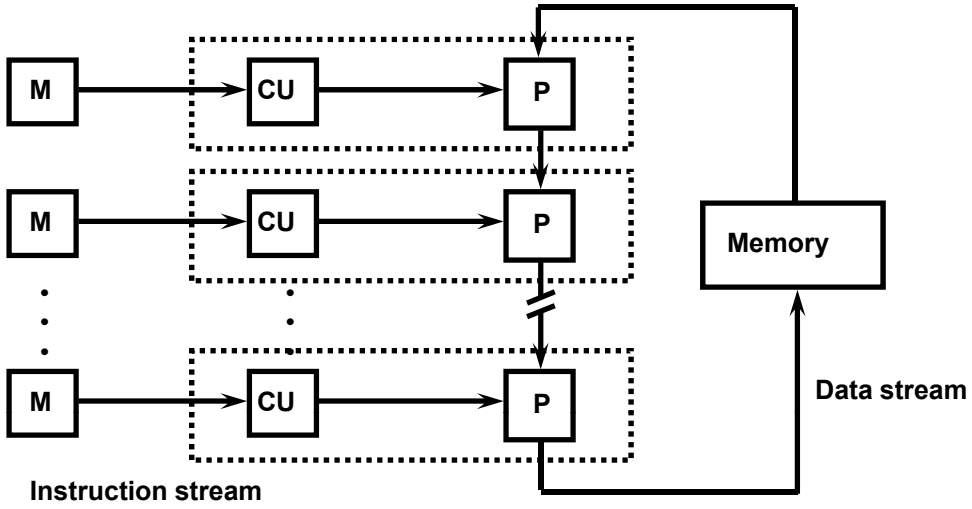
Maximum speed of the system is limited by the *Memory Bandwidth* (bits/sec or bytes/sec)

- Limitation on *Memory Bandwidth*
- Memory is shared by CPU and I/O

SISD PERFORMANCE IMPROVEMENTS

- Multiprogramming
- Spooling
- Multifunction processor
- Pipelining
- Exploiting instruction-level parallelism
  - Superscalar
  - Superpipelining
  - VLIW (Very Long Instruction Word)

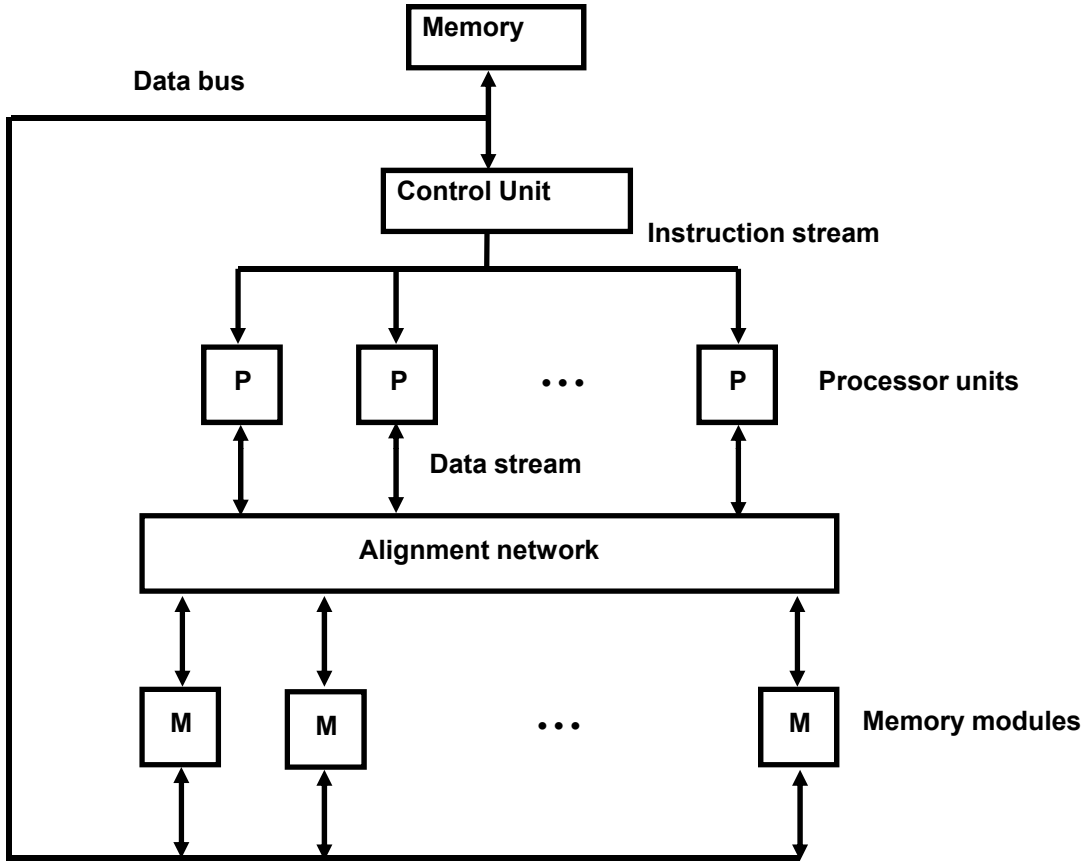
MISD COMPUTER SYSTEMS



Characteristics

- There is no computer at present that can be classified as MISD

SIMD COMPUTER SYSTEMS



Characteristics

- Only one copy of the program exists
- A single controller executes one instruction at a time

TYPES OF SIMD COMPUTERS

Array Processors

- The control unit broadcasts instructions to all PEs, and all active PEs execute the same instructions
- ILLIAC IV, GF-11, Connection Machine, DAP, MPP

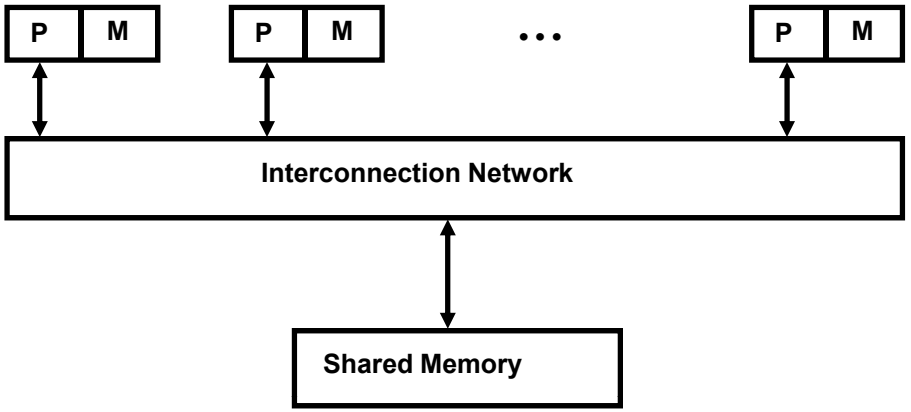
Systolic Arrays

- Regular arrangement of a large number of very simple processors constructed on VLSI circuits
- CMU Warp, Purdue CHiP

Associative Processors

- Content addressing
- Data transformation operations over many sets of arguments with a single instruction
- STARAN, PEPE

MIMD COMPUTER SYSTEMS



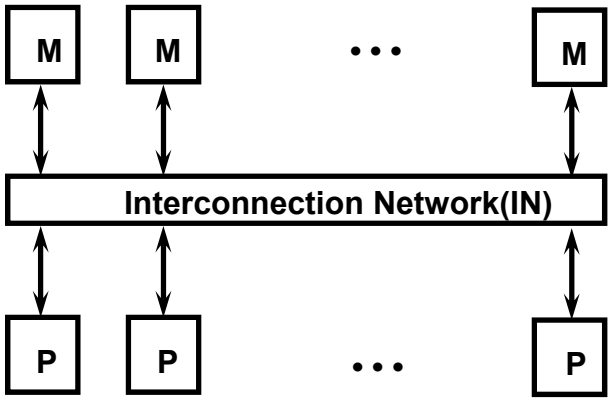
Characteristics

- Multiple processing units
- Execution of multiple instructions on multiple data

Types of MIMD computer systems

- Shared memory multiprocessors
- Message-passing multicomputers

SHARED MEMORY MULTIPROCESSORS



Buses,
Multistage IN,
Crossbar Switch

Characteristics

All processors have equally direct access to one large memory address space

Example systems

Bus and cache-based systems
- Sequent Balance, Encore Multimax

Multistage IN-based systems
- Ultracomputer, Butterfly, RP3, HEP

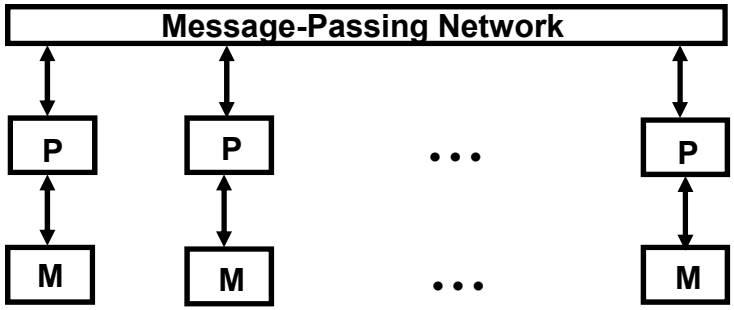
Crossbar switch-based systems
- C.mmp, Alliant FX/8

Limitations

Memory access latency

Hot spot problem

MESSAGE-PASSING MULTICOMPUTER



Point-to-point connections

Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

Example systems

- Tree structure: Teradata, DADO
- Mesh-connected: Rediflow, Series 2010, J-Machine
- Hypercube: Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III

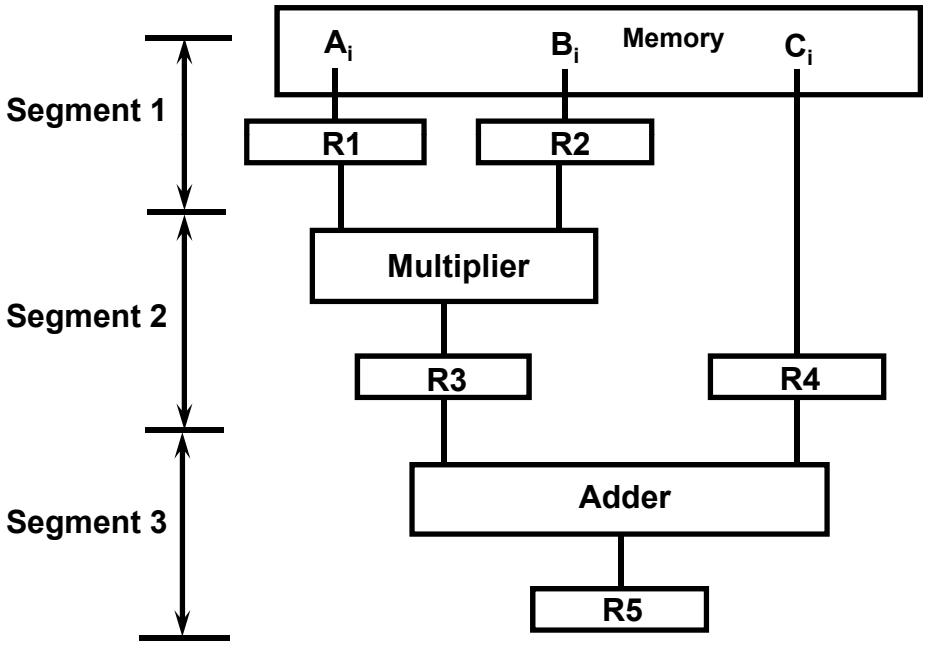
Limitations

- Communication overhead
- Hard to programming

PIPELINING

A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

$A_i * B_i + C_i$ for i = 1, 2, 3, ... , 7



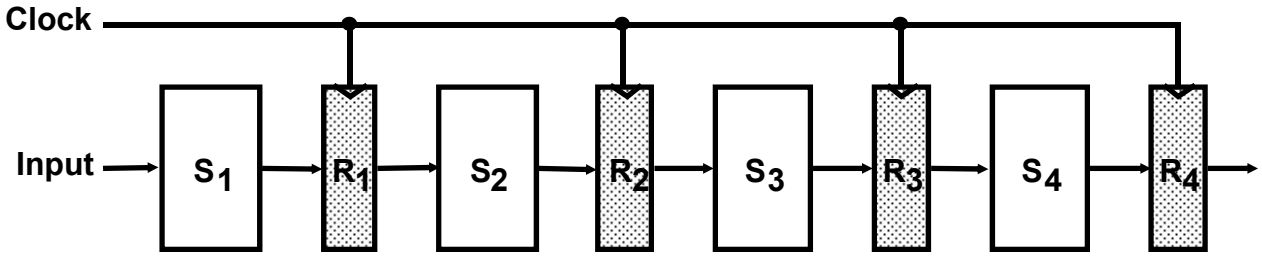
$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Load A_i and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply and load C_i
$R5 \leftarrow R3 + R4$	Add

OPERATIONS IN EACH PIPELINE STAGE

Clock Pulse	Segment 1			Segment 2		Segment 3
Number	R1	R2	R3	R4	R5	
1	A1	B1				
2	A2	B2	A1 * B1	C1		
3	A3	B3	A2 * B2	C2	A1 * B1 + C1	
4	A4	B4	A3 * B3	C3	A2 * B2 + C2	
5	A5	B5	A4 * B4	C4	A3 * B3 + C3	
6	A6	B6	A5 * B5	C5	A4 * B4 + C4	
7	A7	B7	A6 * B6	C6	A5 * B5 + C5	
8				A7 * B7	C7	A6 * B6 + C6
9						A7 * B7 + C7

GENERAL PIPELINE

General Structure of a 4-Segment Pipeline



Space-Time Diagram

Segment	1	2	3	4	5	6	7	8	9
1	T1	T2	T3	T4	T5	T6			
2		T1	T2	T3	T4	T5	T6		
3			T1	T2	T3	T4	T5	T6	
4				T1	T2	T3	T4	T5	T6

Clock cycles

PIPELINE SPEEDUP

n: Number of tasks to be performed

Conventional Machine (Non-Pipelined)

t_n: Clock cycle

τ_1: Time required to complete the n tasks

$\tau_1 = n * t_n$

Pipelined Machine (k stages)

t_p: Clock cycle (time to complete each suboperation)

τ_κ: Time required to complete the n tasks

$\tau_\kappa = (k + n - 1) * t_p$

Speedup

S_k: Speedup

$S_k = n*t_n / (k + n - 1)*t_p$

$$\lim_{n \to \infty} S_k = \frac{t_n}{t_p} \quad (= k, \text{ if } t_n = k * t_p)$$

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

- 4-stage pipeline
- subopertion in each stage; t_p = 20nS
- 100 tasks to be executed
- 1 task in non-pipelined system; 20*4 = 80nS

Pipelined System

$(k + n - 1)*t_p = (4 + 99) * 20 = 2060nS$

Non-Pipelined System

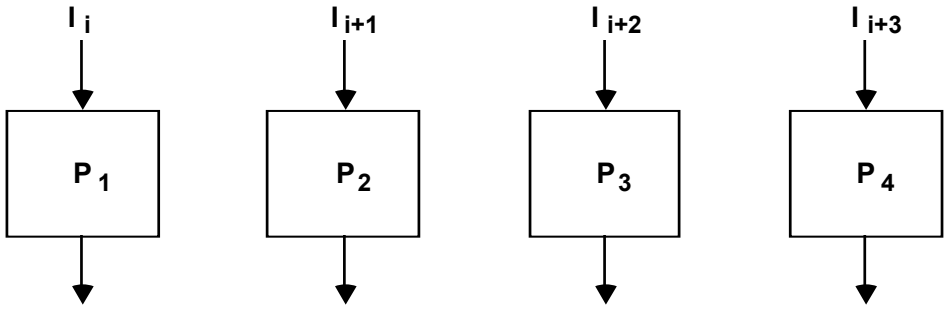
$n*k*t_p = 100 * 80 = 8000nS$

Speedup

$S_k = 8000 / 2060 = 3.88$

4-Stage Pipeline is basically identical to the system with 4 identical function units

Multiple Functional Units

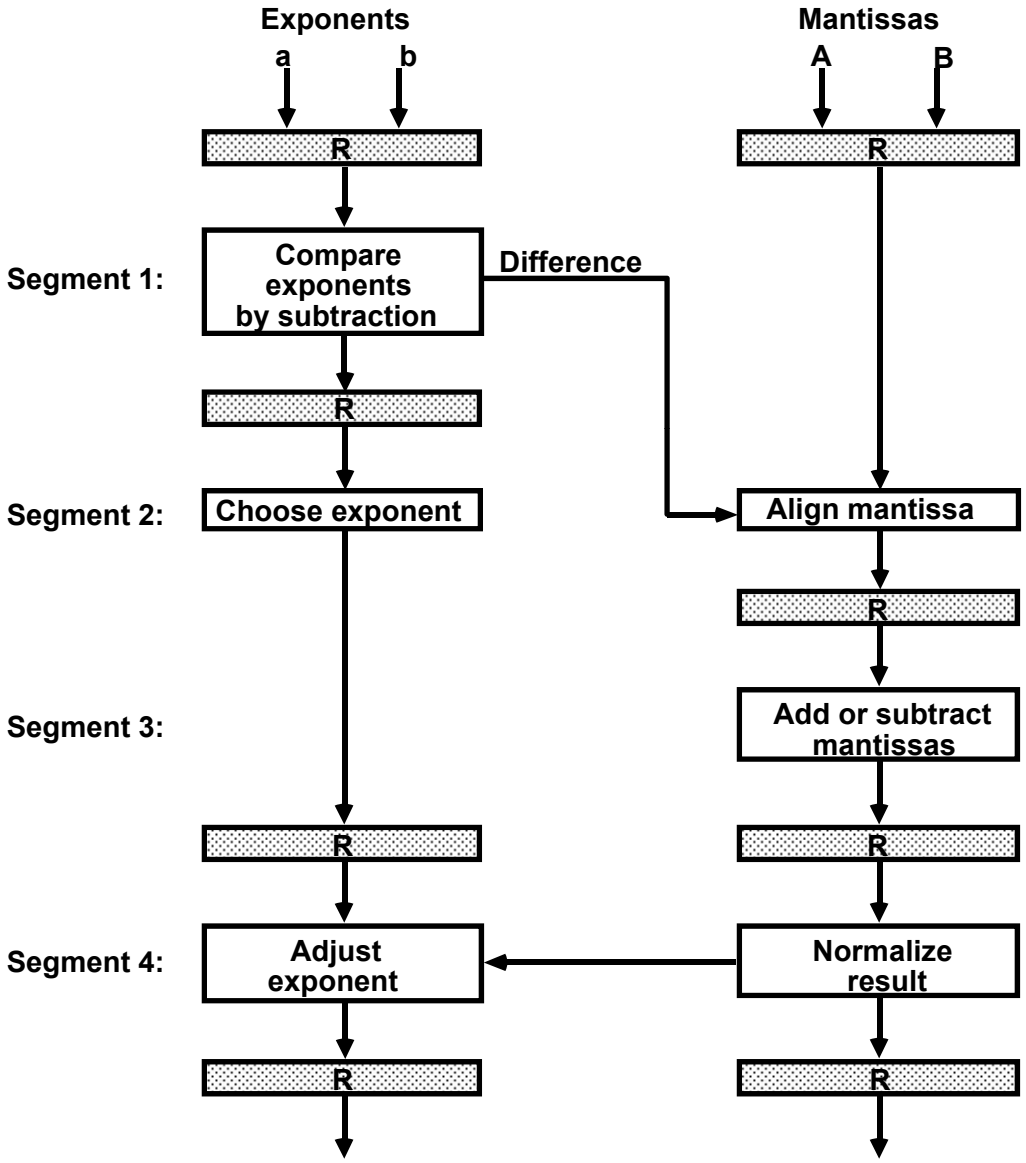


ARITHMETIC PIPELINE

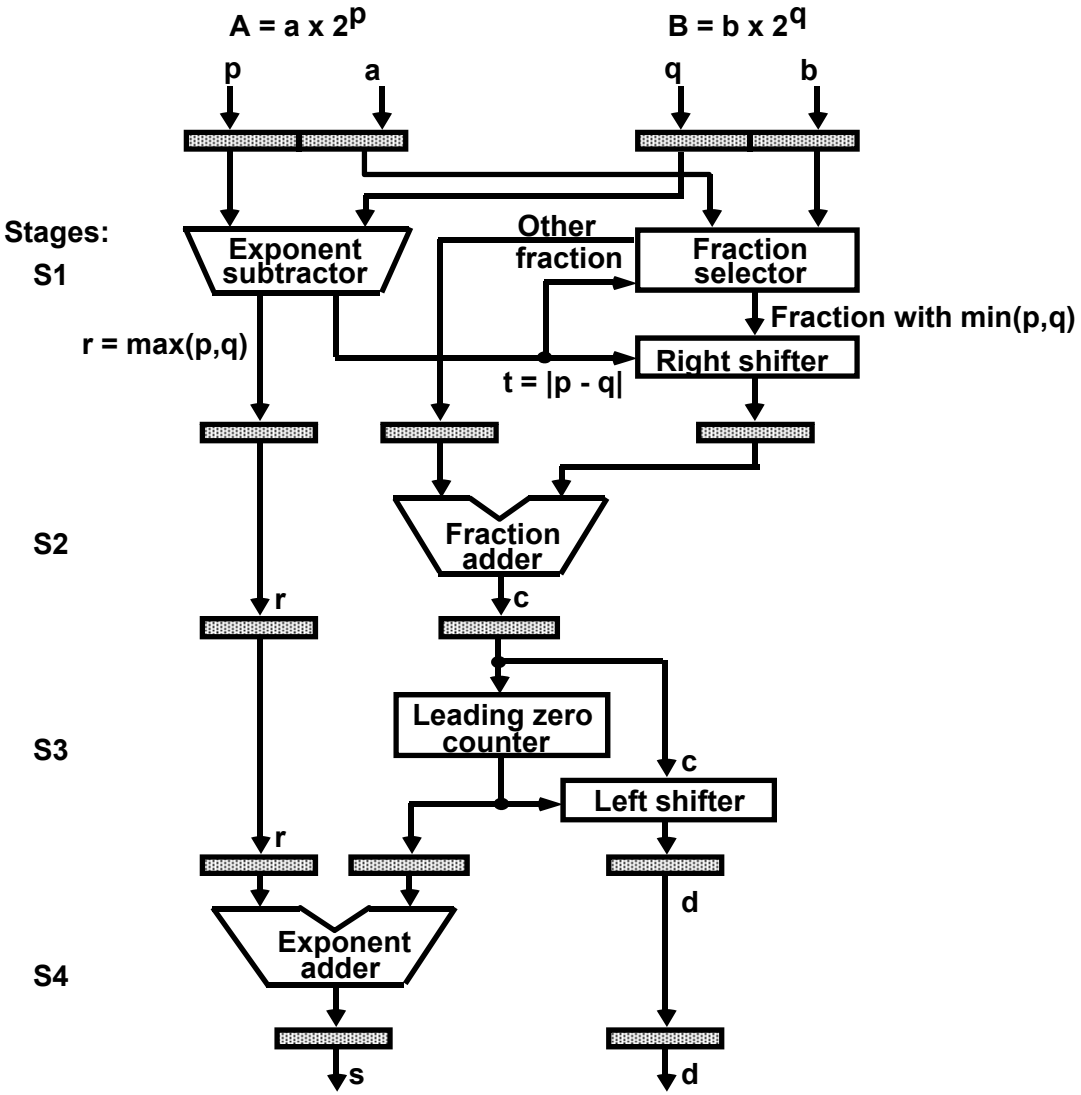
Floating-point adder

$X = A \times 2^a$
$Y = B \times 2^b$

[1] Compare the exponents
[2] Align the mantissa
[3] Add/sub the mantissa
[4] Normalize the result



4-STAGE FLOATING POINT ADDER



INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

[1] Fetch an instruction from memory
[2] Decode the instruction
[3] Calculate the effective address of the operand
[4] Fetch the operands from memory
[5] Execute the operation
[6] Store the result in the proper place

* Some instructions skip some phases
* Effective address calculation can be done in the part of the decoding phase
* Storage of the operation result into a register is done automatically in the execution phase

==> 4-Stage Pipeline

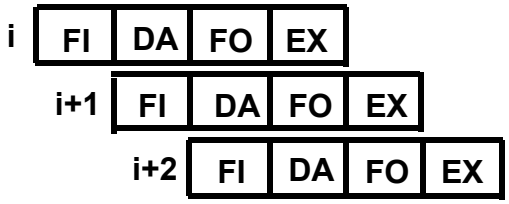
[1] FI: Fetch an instruction from memory
[2] DA: Decode the instruction and calculate the effective address of the operand
[3] FO: Fetch the operand
[4] EX: Execute the operation

INSTRUCTION PIPELINE

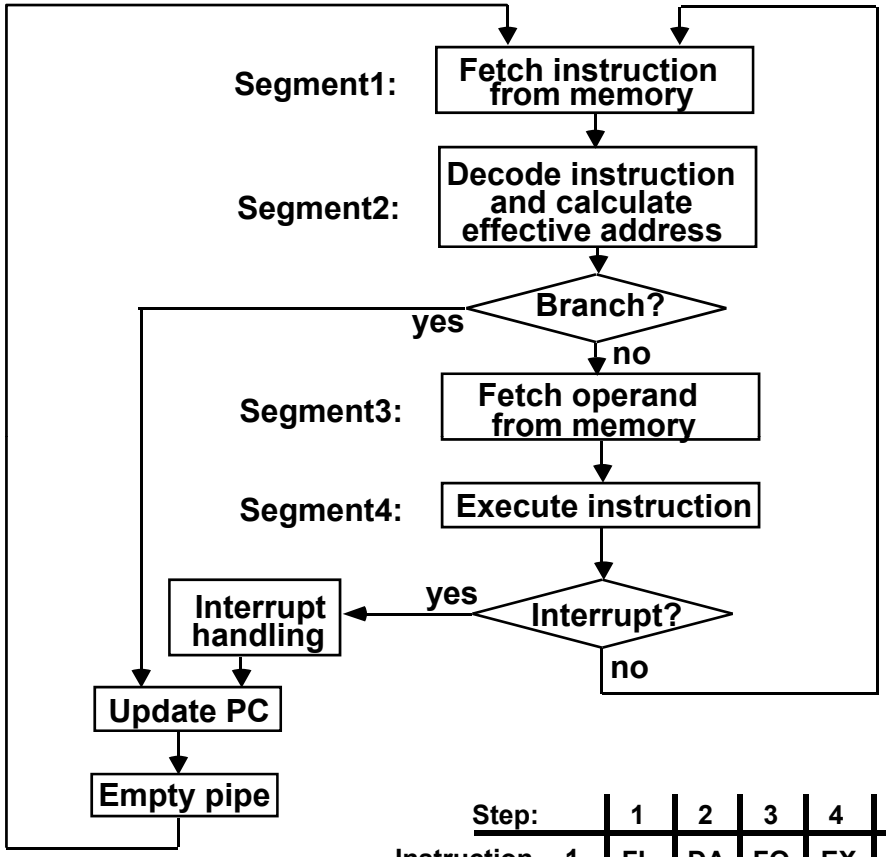
Execution of Three Instructions in a 4-Stage Pipeline

Conventional

Pipelined



INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

MAJOR HAZARDS IN PIPELINED EXECUTION

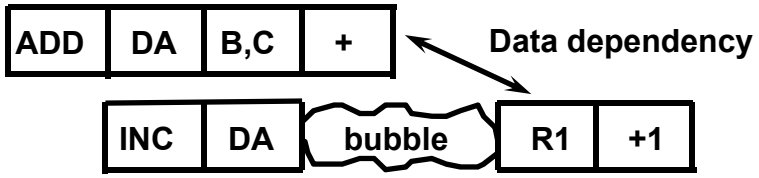
Structural hazards(Resource Conflicts)

Hardware Resources required by the instructions in simultaneous overlapped execution cannot be met

Data hazards (Data Dependency Conflicts)

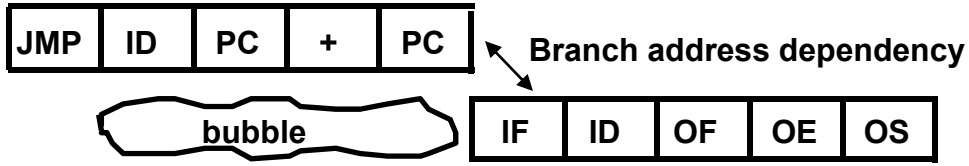
An instruction scheduled to be executed in the pipeline requires the result of a previous instruction, which is not yet available

R1 <- B + C
R1 <- R1 + 1



Control hazards

Branches and other instructions that change the PC make the fetch of the next instruction to be delayed



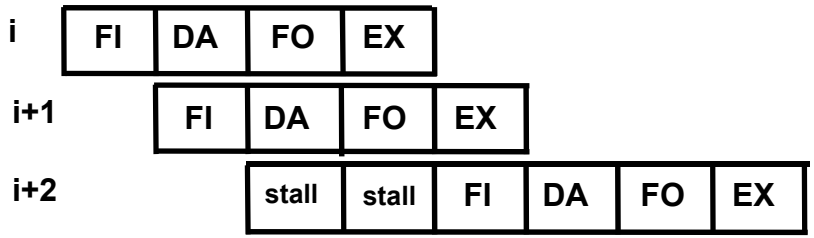
Hazards in pipelines may make it necessary to ***stall*** the pipeline ⇨ Pipeline Interlock: Detect Hazards Stall until it is cleared

STRUCTURAL HAZARDS

Structural Hazards

Occur when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute

Example: With one memory-port, a data and an instruction fetch cannot be initiated in the same clock



The Pipeline is stalled for a structural hazard
<- Two Loads with one port memory
-> Two-port memory will serve without stall

DATA HAZARDS

Data Hazards

Occurs when the execution of an instruction depends on the results of a previous instruction

```
ADD   R1, R2, R3
SUB   R4, R1, R5
```

Data hazard can be dealt with either hardware techniques or software technique

Hardware Technique

Interlock

- hardware detects the data dependencies and delays the scheduling of the dependent instruction by stalling enough clock cycles

Forwarding (bypassing, short-circuiting)

- Accomplished by a data path that routes a value from a source (usually an ALU) to a user, bypassing a designated register. This allows the value to be produced to be used at an earlier stage in the pipeline than would otherwise be possible

Software Technique

Instruction Scheduling(compiler) for *delayed load*

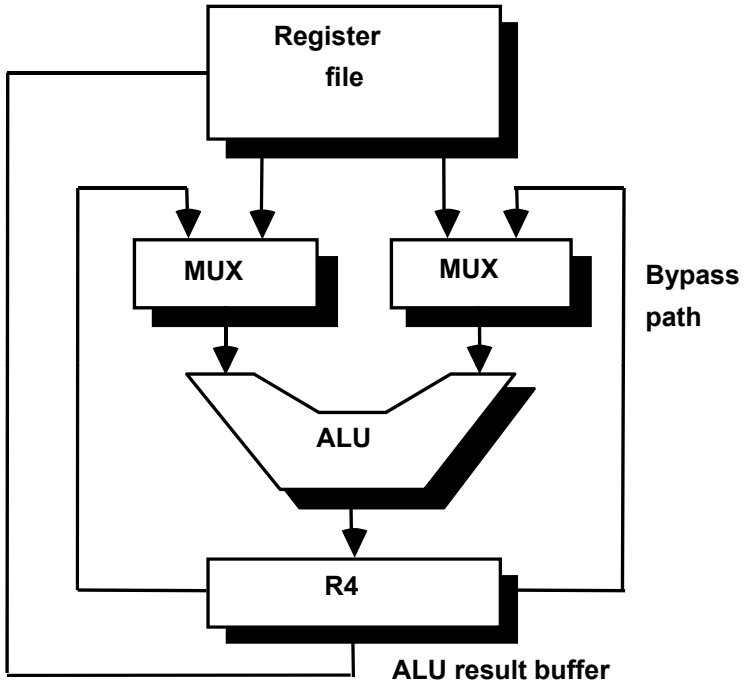
FORWARDING HARDWARE

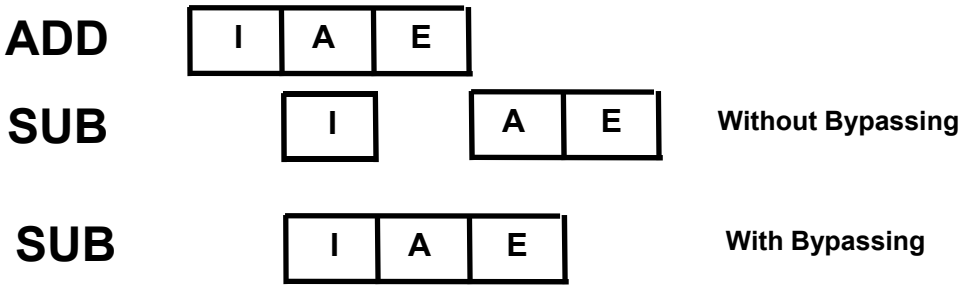
Example:

```
ADD     R1, R2, R3
SUB     R4, R1, R5
```

3-stage Pipeline

I: Instruction Fetch
A: Decode, Read Registers, ALU Operations
E: Write the result to the destination register





INSTRUCTION SCHEDULING

```
a = b + c;
d = e - f;
```

Unscheduled code:

```
      LW    Rb, b
      LW    Rc, c
→     ADD   Ra, Rb, Rc
→     SW    a, Ra
      LW    Re, e
      LW    Rf, f
→     SUB   Rd, Re, Rf
→     SW    d, Rd
```

Scheduled Code:

```
      LW    Rb, b
      LW    Rc, c
      LW    Re, e
      ADD   Ra, Rb, Rc
      LW    Rf, f
      SW    a, Ra
      SUB   Rd, Re, Rf
→     SW    d, Rd
```

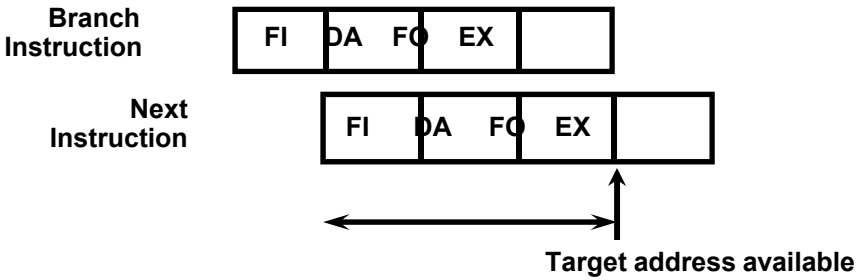
Delayed Load

A load requiring that the following instruction not use its result

CONTROL HAZARDS

Branch Instructions

- Branch target address is not known until the branch instruction is completed



- Stall -> waste of cycle times

Dealing with Control Hazards

* Prefetch Target Instruction
* Branch Target Buffer
* Loop Buffer
* Branch Prediction
* Delayed Branch

CONTROL HAZARDS

Prefetch Target Instruction

- Fetch instructions in both streams, branch not taken and branch taken
- Both are saved until branch branch is executed. Then, select the right instruction stream and discard the wrong stream

Branch Target Buffer(BTB; Associative Memory)

- Entry: Addr of previously executed branches; Target instruction and the next few instructions
- When fetching an instruction, search BTB.
- If found, fetch the instruction stream in BTB;
- If not, new stream is fetched and update BTB

Loop Buffer(High Speed Register file)

- Storage of entire loop that allows to execute a loop without accessing memory

Branch Prediction

- Guessing the branch condition, and fetch an instruction stream based on the guess. Correct guess eliminates the branch penalty

Delayed Branch

- Compiler detects the branch and rearranges the instruction sequence by inserting useful instructions that keep the pipeline busy in the presence of a branch instruction

RISC PIPELINE

RISC

- Machine with a very fast clock cycle that executes at the rate of one instruction per cycle

<- Simple Instruction Set

Fixed Length Instruction Format

Register-to-Register Operations

Instruction Cycles of Three-Stage Instruction Pipeline

Data Manipulation Instructions

I: Instruction Fetch

A: Decode, Read Registers, ALU Operations

E: Write a Register

Load and Store Instructions

I: Instruction Fetch

A: Decode, Evaluate Effective Address

E: Register-to-Memory or Memory-to-Register

Program Control Instructions

I: Instruction Fetch

A: Decode, Evaluate Branch Address

E: Write Register(PC)

DELAYED LOAD

LOAD: R1 ← M[address 1]
LOAD: R2 ← M[address 2]
ADD: R3 ← R1 + R2
STORE: M[address 3] ← R3

Three-segment pipeline timing

Pipeline timing with data conflict

clock cycle	1	2	3	4	5	6
Load R1	I	A	E			
Load R2		I	A	E		
Add R1+R2			I	A	E	
Store R3				I	A	E

Pipeline timing with delayed load

clock cycle	1	2	3	4	5	6	7
Load R1	I	A	E				
Load R2		I	A	E			
NOP			I	A	E		
Add R1+R2				I	A	E	
Store R3					I	A	E

The data dependency is taken care by the compiler rather than the hardware

DELAYED BRANCH

Compiler analyzes the instructions before and after the branch and rearranges the program sequence by inserting useful instructions in the delay steps

Using no-operation instructions

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. NOP						I	A	E		
7. NOP							I	A	E	
8. Instr. in X								I	A	E

Rearranging the instructions

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instr. in X						I	A	E

VECTOR PROCESSING

Vector Processing Applications

- **Problems that can be efficiently formulated in terms of vectors**
  - **Long-range weather forecasting**
  - **Petroleum explorations**
  - **Seismic data analysis**
  - **Medical diagnosis**
  - **Aerodynamics and space flight simulations**
  - **Artificial intelligence and expert systems**
  - **Mapping the human genome**
  - **Image processing**

Vector Processor (computer)

Ability to process vectors, and related data structures such as matrices and multi-dimensional arrays, much faster than conventional computers

Vector Processors may also be pipelined

VECTOR PROGRAMMING

```
      DO  20  I = 1, 100
20    C(I) = B(I) + A(I)
```

Conventional computer

```
      Initialize I = 0
20    Read A(I)
      Read B(I)
      Store C(I) = A(I) + B(I)
      Increment I = i + 1
      If I ≤ 100 goto 20
```

Vector computer

```
C(1:100) = A(1:100) + B(1:100)
```

VECTOR INSTRUCTIONS

f1: V ∗ V
f2: V ∗ S
f3: V x V ∗ V
f4: V x S ∗ V

V: Vector operand
S: Scalar operand

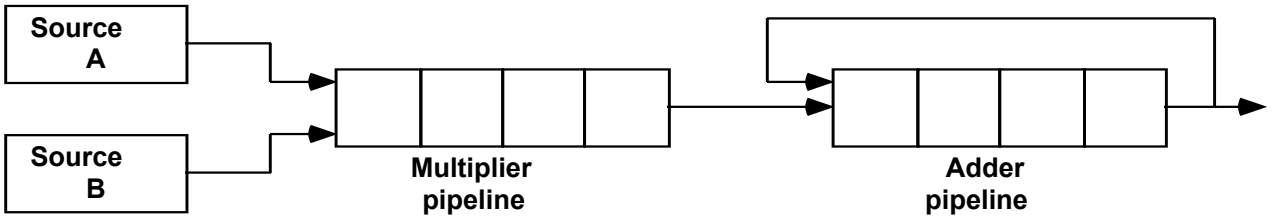
Type	Mnemonic	Description (I = 1, ..., n)	
f1	VSQR	Vector square root	B(I) ∗ SQR(A(I))
	VSIN	Vector sine	B(I) ∗ sin(A(I))
	VCOM	Vector complement	A(I) ∗ A(I)
f2	VSUM	Vector summation	S ∗ Σ A(I)
	VMAX	Vector maximum	S ∗ max{A(I)}
f3	VADD	Vector add	C(I) ∗ A(I) + B(I)
	VMPY	Vector multiply	C(I) ∗ A(I) * B(I)
	VAND	Vector AND	C(I) ∗ A(I) . B(I)
	VLAR	Vector larger	C(I) ∗ max(A(I),B(I))
	VTGE	Vector test >	C(I) ∗ 0 if A(I) < B(I)
			C(I) ∗ 1 if A(I) > B(I)
f4	SADD	Vector-scalar add	B(I) ∗ S + A(I)
	SDIV	Vector-scalar divide	B(I) ∗ A(I) / S

VECTOR INSTRUCTION FORMAT

Vector Instruction Format

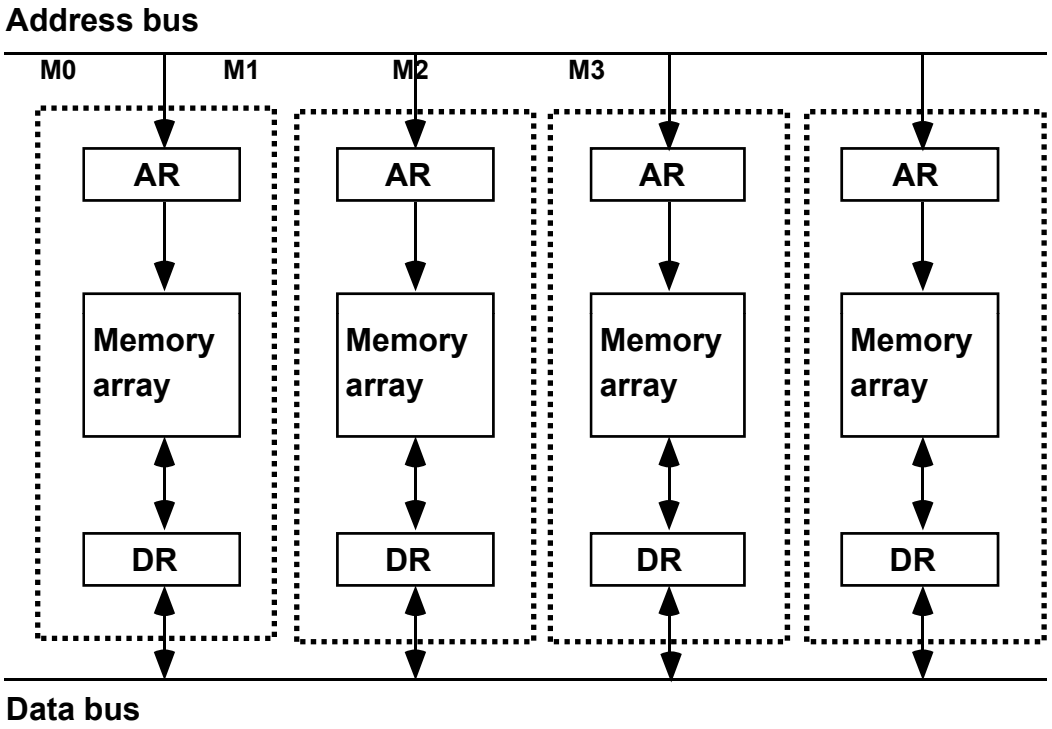
Operation code	Base address source 1	Base address source 2	Base address destination	Vector length

Pipeline for Inner Product



MULTIPLE MEMORY MODULE AND INTERLEAVING

Multiple Module Memory



Address Interleaving

Different sets of addresses are assigned to different memory modules

MULTIPROCESSORS

- Characteristics of Multiprocessors
- Interconnection Structures
- Interprocessor Arbitration
- Interprocessor Communication and Synchronization
- Cache Coherence

TERMINOLOGY

Parallel Computing

Simultaneous use of multiple processors, all components of a single architecture, to solve a task. Typically processors identical, single user (even if machine multiuser)

Distributed Computing

Use of a network of processors, each capable of being viewed as a computer in its own right, to solve a problem. Processors may be heterogeneous, multiuser, usually individual task is assigned to a single processors

Concurrent Computing

All of the above?

TERMINOLOGY

Supercomputing

Use of fastest, biggest machines to solve big, computationally intensive problems. Historically machines were vector computers, but parallel/vector or parallel becoming the norm

Pipelining

Breaking a task into steps performed by different units, and multiple inputs stream through the units, with next input starting in a unit when previous input done with the unit but not necessarily done with the task

Vector Computing

Use of vector processors, where operation such as multiply broken into several steps, and is applied to a stream of operands ("vectors"). Most common special case of pipelining

Systolic

Similar to pipelining, but units are not necessarily arranged linearly, steps are typically small and more numerous, performed in lockstep fashion. Often used in special-purpose hardware such as image or signal processors

SPEEDUP AND EFFICIENCY

A: Given problem

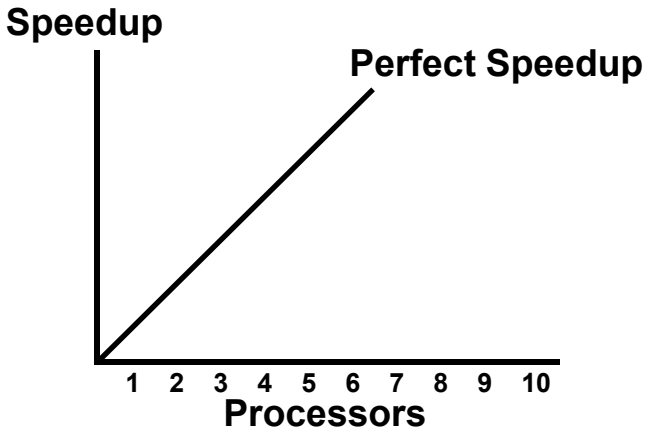
$T^*(n)$: Time of best sequential algorithm to solve an instance of A of size n on 1 processor

$T_p(n)$: Time needed by a given parallel algorithm and given parallel architecture to solve an instance of A of size n, using p processors

Note: $T^*(n) \leq T_1(n)$

Speedup: $T^*(n) / T_p(n)$

Efficiency: $T^*(n) / [pT_p(n)]$



Speedup should be between 0 and p, and Efficiency should be between 0 and 1
Speedup is *linear* if there is a constant c > 0 so that speedup is always at least cp.

AMDAHL'S LAW

Given a program

f **: Fraction of time that represents operations that must be performed serially**

Maximum Possible Speedup: *S*

$$S \leq \frac{1}{f + (1 - f) / p}$$ **, with p processors**

$$S < 1 / f$$ **, with unlimited number of processors**

- Ignores possibility of new algorithm, with much smaller *f*

- Ignores possibility that more of program is run from higher speed memory such as Registers, Cache, Main Memory

- Often problem is scaled with number of processors, and *f* is a function of size which may be decreasing (Serial code may take constant amount of time, independent of size)

FLYNN's HARDWARE TAXONOMY

I: Instruction Stream
D: Data Stream

$$\left[\begin{matrix}M\\S\end{matrix}\right] I \left[\begin{matrix}M\\S\end{matrix}\right] D$$

SI: Single Instruction Stream
- All processors are executing the same instruction in the same cycle
- Instruction may be conditional
- For Multiple processors, the control processor issues an instruction

MI: Multiple Instruction Stream
- Different processors may be simultaneously executing different instructions

SD: Single Data Stream
- All of the processors are operating on the same data items at any given time

MD: Multiple Data Stream
- Different processors may be simultaneously operating on different data items

SISD : standard serial computer
MISD : very rare
MIMD and *SIMD* : Parallel processing computers

COUPLING OF PROCESSORS

Tightly Coupled System

- Tasks and/or processors communicate in a highly synchronized fashion
- Communicates through a common shared memory
- Shared memory system

Loosely Coupled System

- Tasks or processors do not communicate in a synchronized fashion
- Communicates by message passing packets
- Overhead for data exchange is high
- Distributed memory system

GRANULARITY OF PARALLELISM

Granularity of Parallelism

Coarse-grain

- A task is broken into a handful of pieces, each of which is executed by a powerful processor
- Processors may be heterogeneous
- Computation/communication ratio is very high

Medium-grain

- Tens to few thousands of pieces
- Processors typically run the same code
- Computation/communication ratio is often hundreds or more

Fine-grain

- Thousands to perhaps millions of small pieces, executed by very small, simple processors or through pipelines
- Processors typically have instructions broadcasted to them
- Compute/communicate ratio often near unity

MEMORY

Shared (Global) Memory

- A Global Memory Space accessible by all processors
- Processors may also have some local memory

Distributed (Local, Message-Passing) Memory

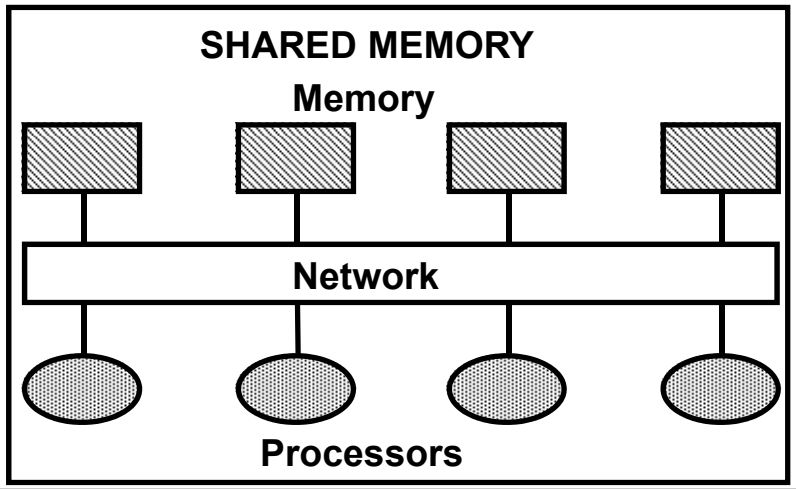
- All memory units are associated with processors
- To retrieve information from another processor's memory a message must be sent there

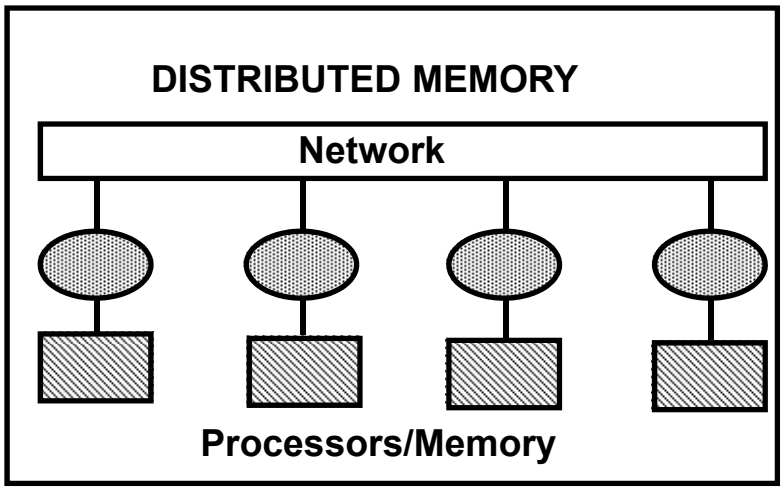
Uniform Memory

- All processors take the same time to reach all memory locations

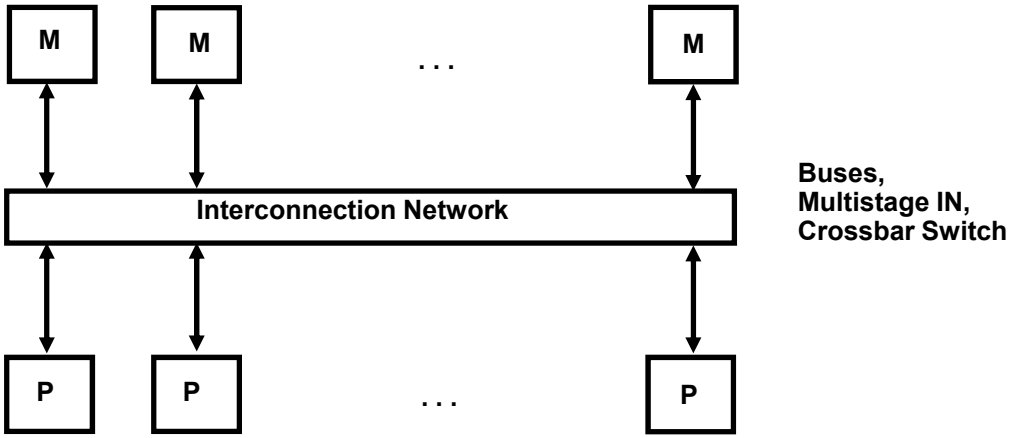
Nonuniform (NUMA) Memory

- Memory access is not uniform





SHARED MEMORY MULTIPROCESSORS



Characteristics

All processors have equally direct access to one large memory address space

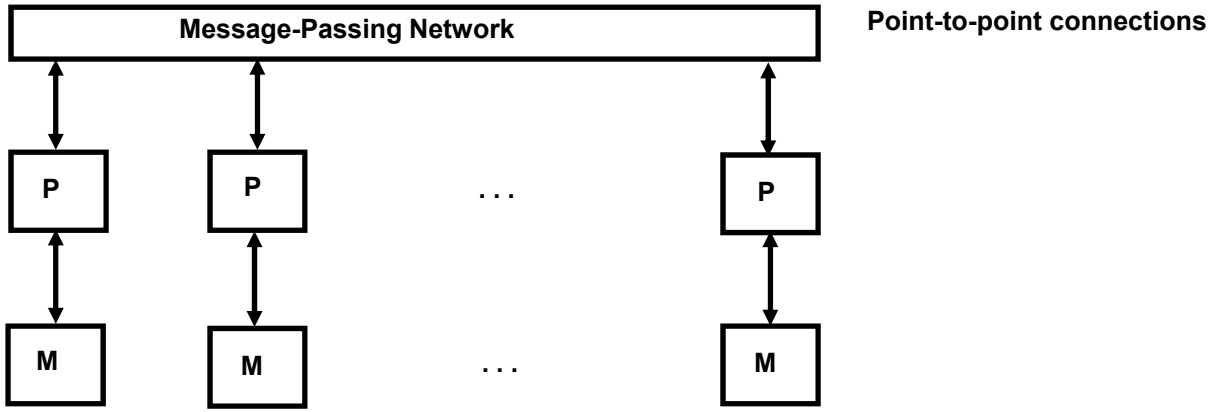
Example systems

- Bus and cache-based systems: Sequent Balance, Encore Multimax
- Multistage IN-based systems: Ultracomputer, Butterfly, RP3, HEP
- Crossbar switch-based systems: C.mmp, Alliant FX/8

Limitations

Memory access latency; Hot spot problem

MESSAGE-PASSING MULTIPROCESSORS



Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

Example systems

- Tree structure: Teradata, DADO
- Mesh-connected: Rediflow, Series 2010, J-Machine
- Hypercube: Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III

Limitations

- Communication overhead; Hard to programming

INTERCONNECTION STRUCTURES

* Time-Shared Common Bus
* Multiport Memory
* Crossbar Switch
* Multistage Switching Network
* Hypercube System

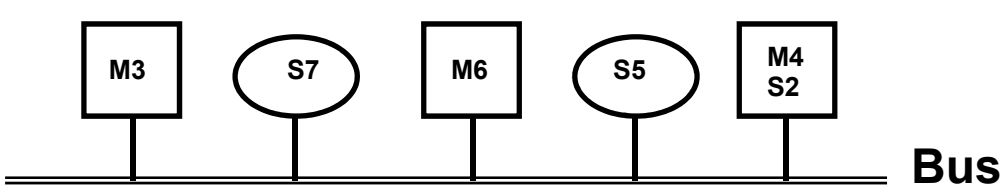
Bus

All processors (and memory) are connected to a common bus or busses
- Memory access is fairly uniform, but not very scalable

BUS

- A collection of signal lines that carry module-to-module communication
- Data highways connecting several digital system elements

Operations of Bus



M3 wishes to communicate with S5

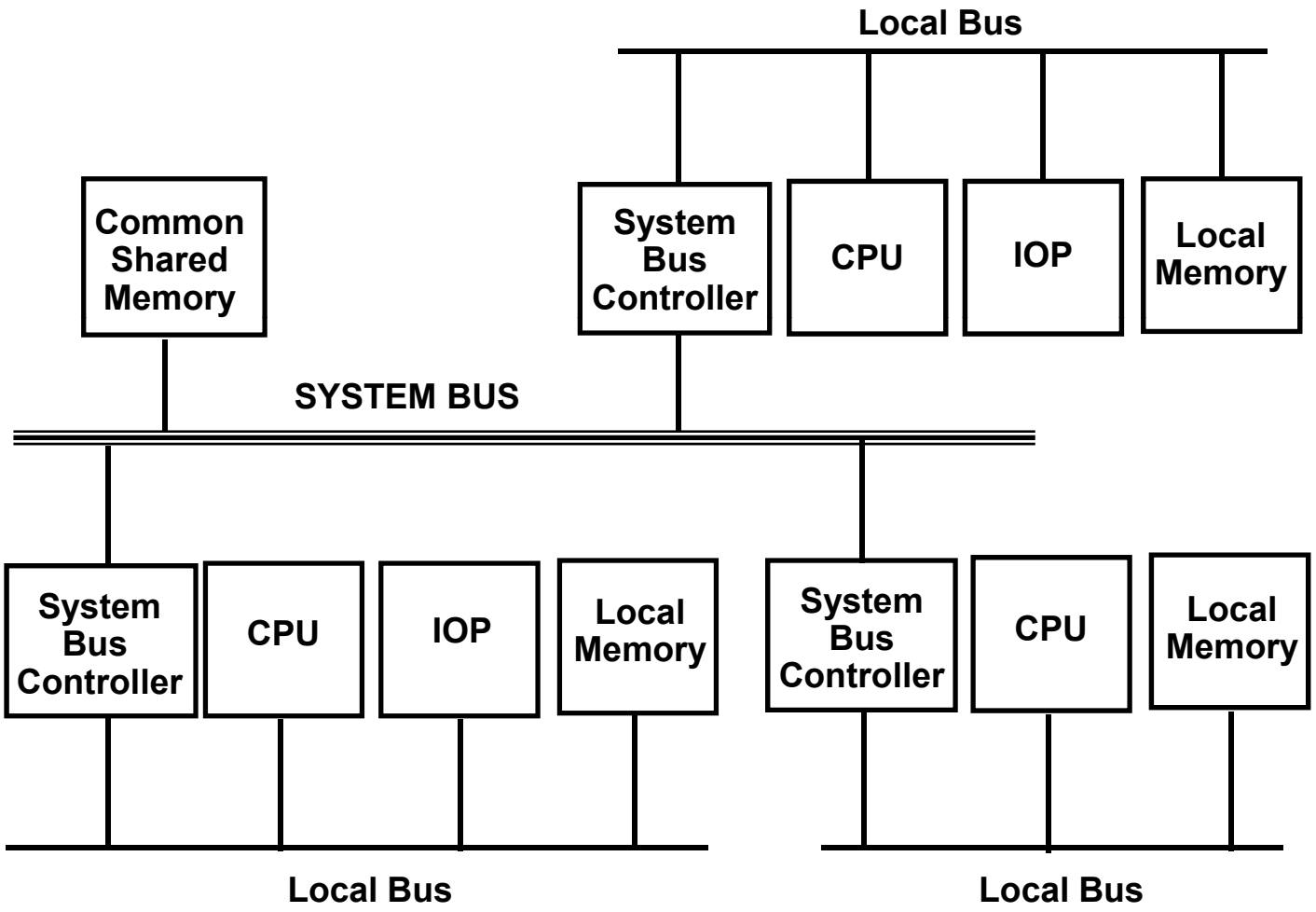
[1] M3 sends signals (address) on the bus that causes S5 to respond
[2] M3 sends data to S5 or S5 sends data to M3(determined by the command line)

Master Device: Device that initiates and controls the communication

Slave Device: Responding device

Multiple-master buses
-> Bus conflict
-> need bus arbitration

SYSTEM BUS STRUCTURE FOR MULTIPROCESSORS



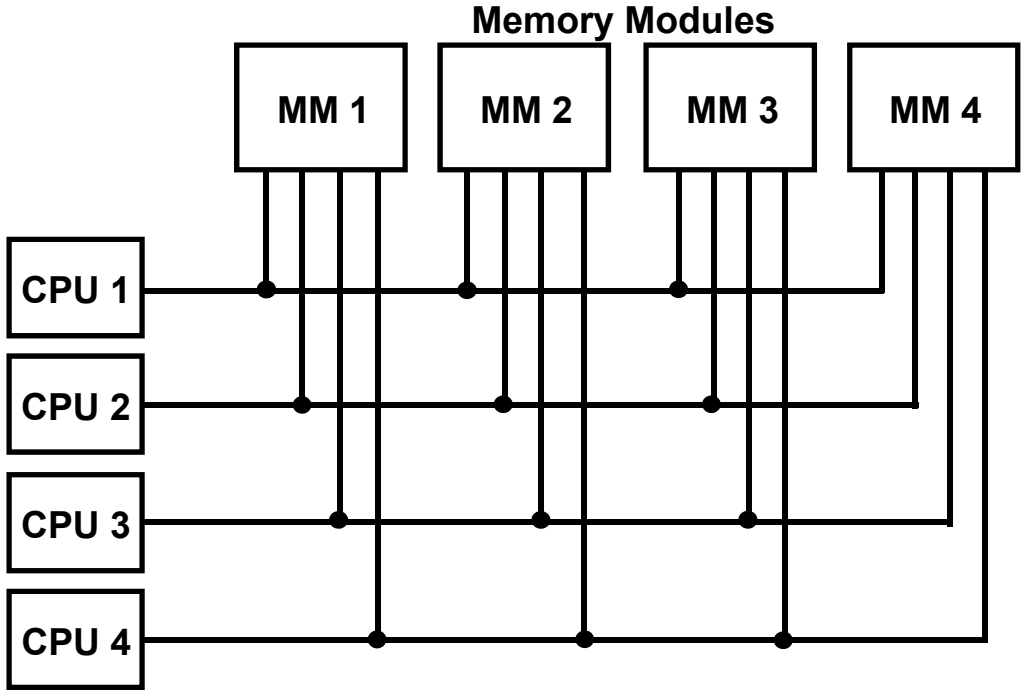
MULTIPORT MEMORY

Multiport Memory Module
- Each port serves a CPU

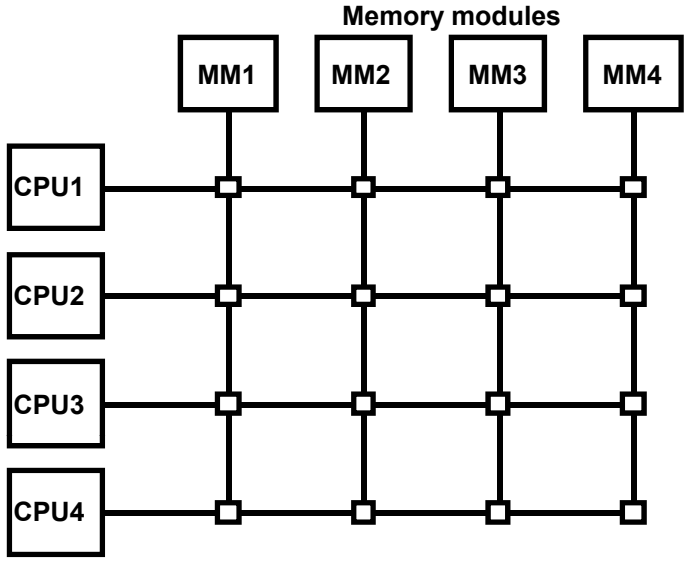
Memory Module Control Logic
- Each memory module has control logic
- Resolve memory module conflicts Fixed priority among CPUs

Advantages
- Multiple paths -> high transfer rate

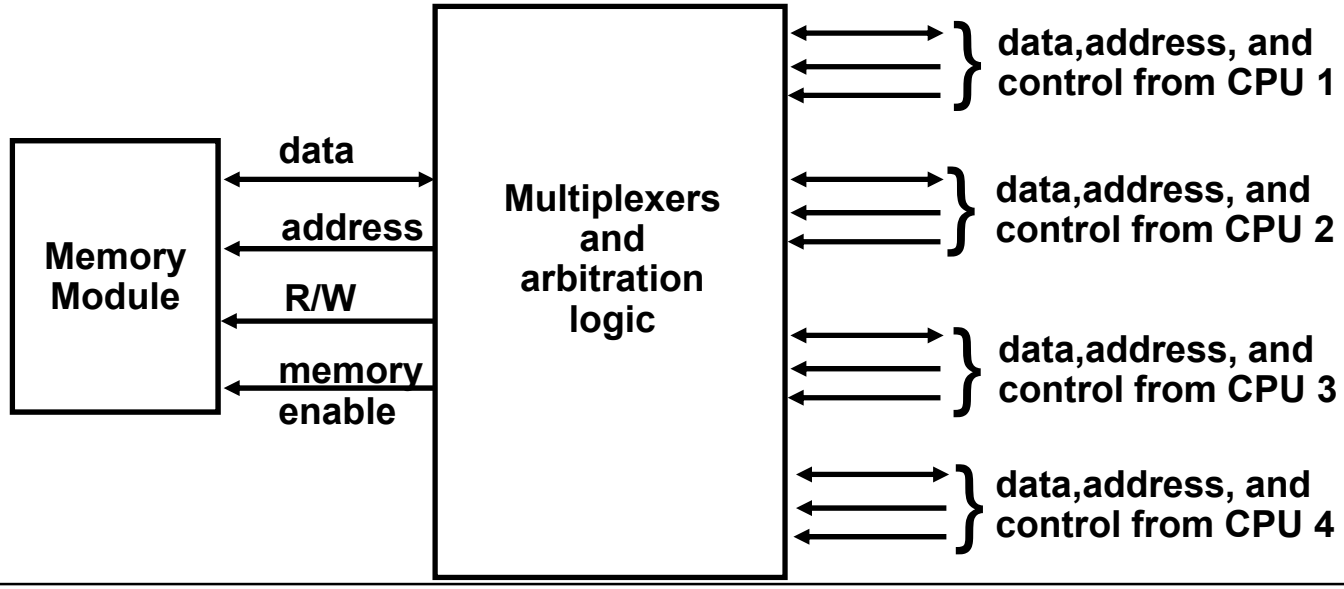
Disadvantages
- Memory control logic
- Large number of cables and connections



CROSSBAR SWITCH

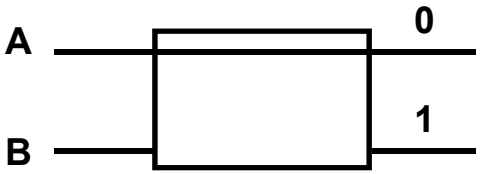


Block Diagram of Crossbar Switch

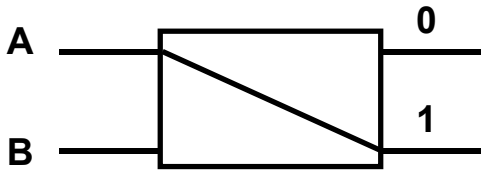


MULTISTAGE SWITCHING NETWORK

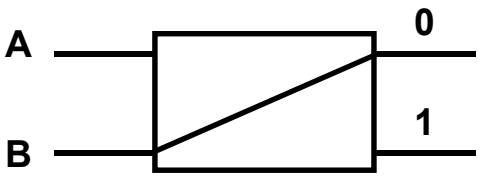
Interstage Switch



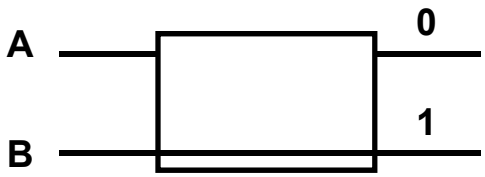
A connected to 0



A connected to 1



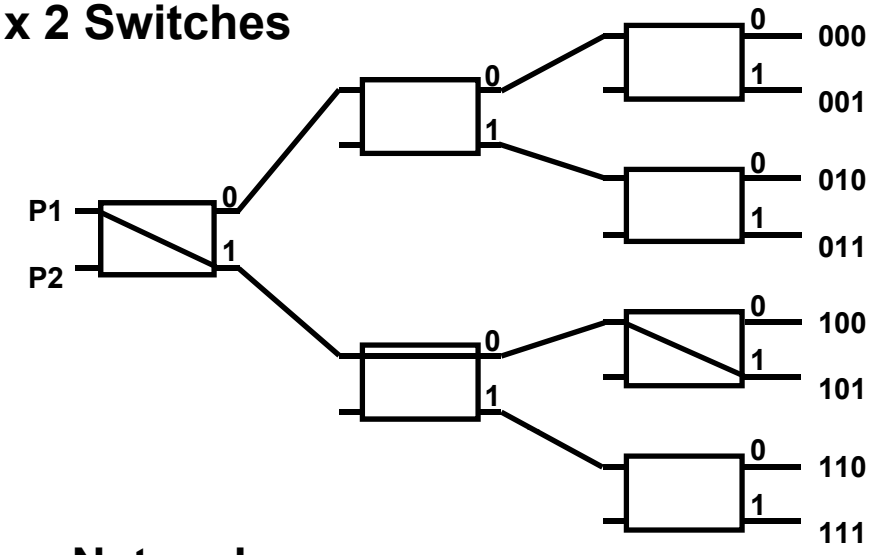
B connected to 0



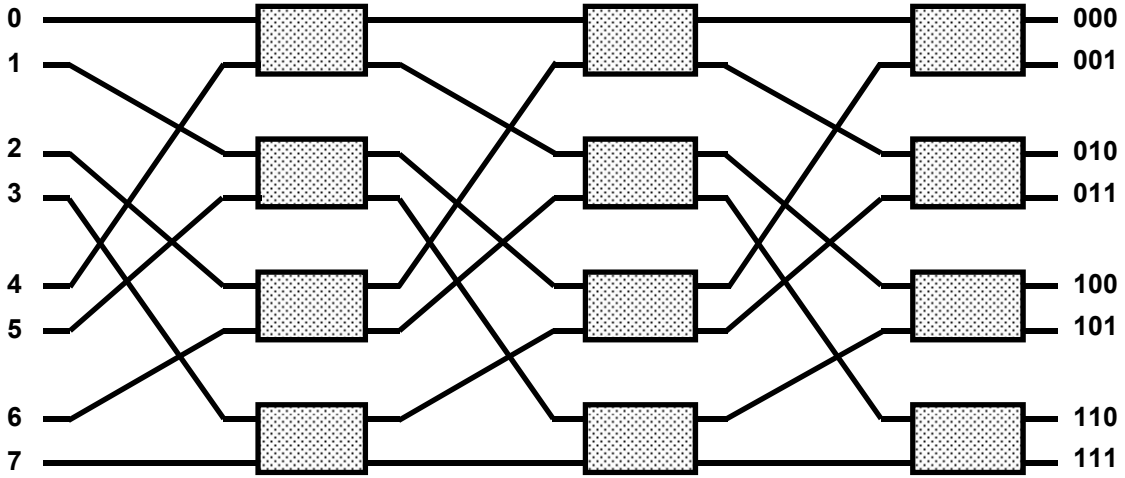
B connected to 1

MULTISTAGE INTERCONNECTION NETWORK

Binary Tree with 2 x 2 Switches



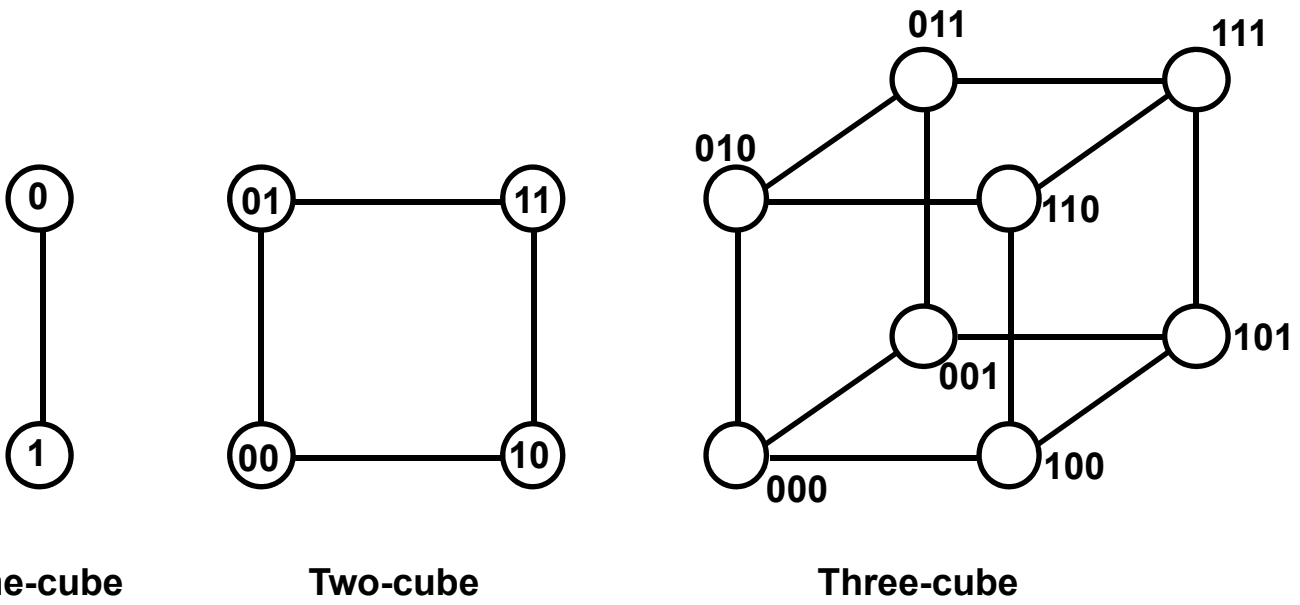
8x8 Omega Switching Network



HYPERCUBE INTERCONNECTION

n-dimensional hypercube (binary n-cube)

- $p = 2^n$
- processors are conceptually on the corners of a n-dimensional hypercube, and each is directly connected to the n neighboring nodes
- Degree = n



INTERPROCESSOR ARBITRATION

Bus

- **Board level bus**
- **Backplane level bus**
- **Interface level bus**

System Bus - A Backplane level bus

- **Printed Circuit Board**
- **Connects CPU, IOP, and Memory**
- **Each of CPU, IOP, and Memory board can be plugged into a slot in the backplane(system bus)**
- **Bus signals are grouped into 3 groups**

  Data, Address, and Control(plus power)

e.g. IEEE standard 796 bus - 86 lines	
Data:	16(multiple of 8)
Address:	24
Control:	26
Power:	20

- **Only one of CPU, IOP, and Memory can be granted to use the bus at a time**
- **Arbitration mechanism is needed to handle multiple requests**

SYNCHRONOUS & ASYNCHRONOUS DATA TRANSFER

Synchronous Bus

Each data item is transferred over a time slice known to both source and destination unit

- Common clock source
- Or separate clock and synchronization signal is transmitted periodically to synchronize the clocks in the system

Asynchronous Bus

* Each data item is transferred by *Handshake* mechanism
  - Unit that transmits the data transmits a control signal that indicates the presence of data
  - Unit that receiving the data responds with another control signal to acknowledge the receipt of the data

* Strobe pulse - supplied by one of the units to indicate to the other unit when the data transfer has to occur

BUS SIGNALS

Bus signal allocation

- address
- data
- control
- arbitration
- interrupt
- timing
- power, ground

IEEE Standard 796 Multibus Signals

Data and address	
Data lines (16 lines)	DATA0 - DATA15
Address lines (24 lines)	ADRS0 - ADRS23
Data transfer	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK (XACK)
Interrupt control	
Interrupt request	INT0 - INT7
interrupt acknowledge	INTA

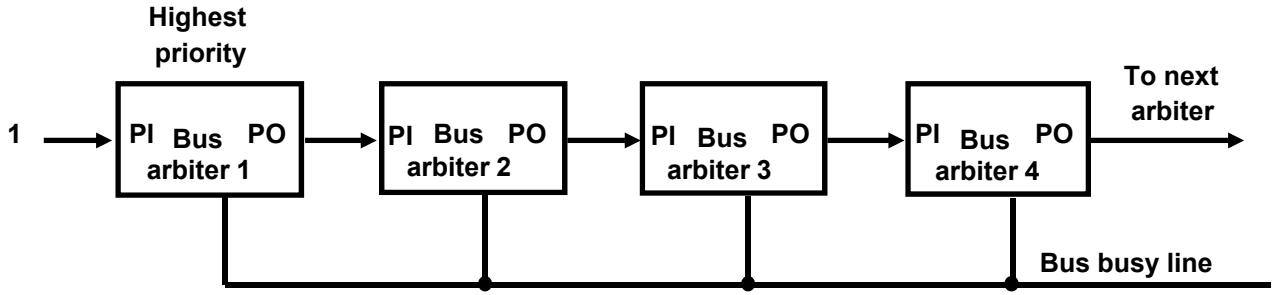
BUS SIGNALS

IEEE Standard 796 Multibus Signals (Cont'd)

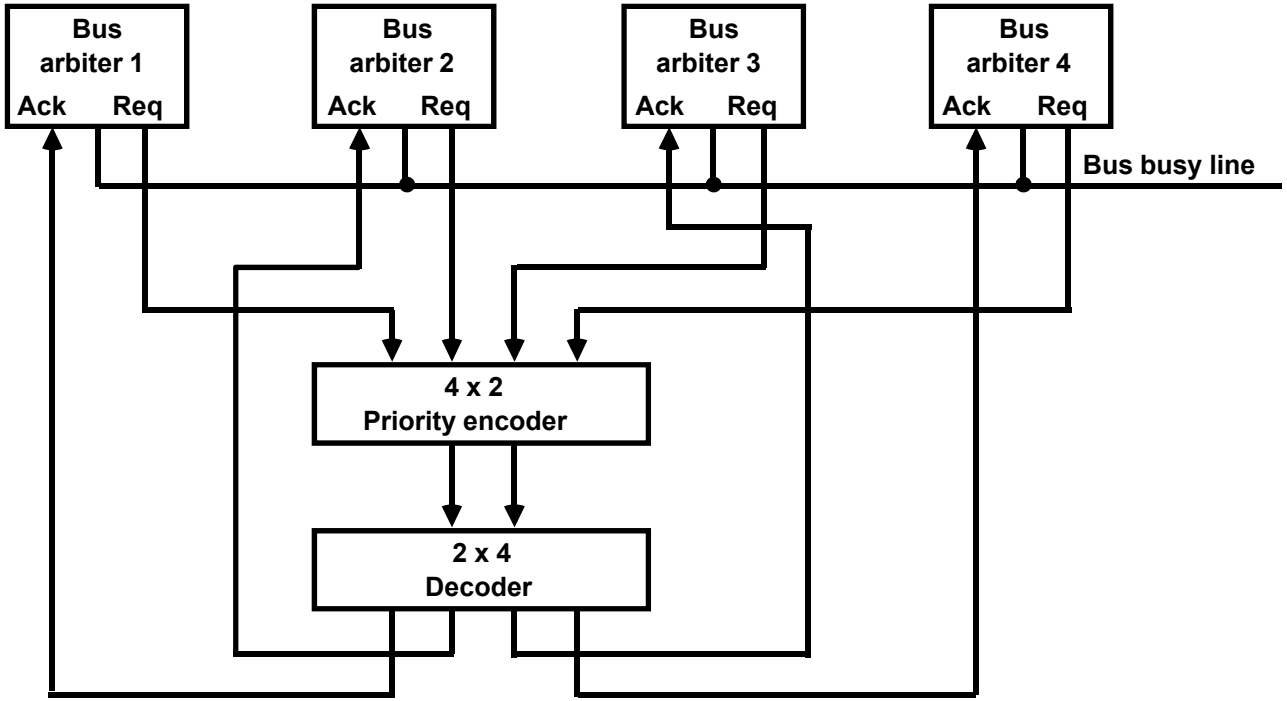
Miscellaneous control	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1 - INH2
Bus lock	LOCK
Bus arbitration	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
Power and ground (20 lines)	

INTERPROCESSOR ARBITRATION STATIC ARBITRATION

Serial Arbitration Procedure



Parallel Arbitration Procedure



INTERPROCESSOR ARBITRATION DYNAMIC ARBITRATION

Priorities of the units can be dynamically changeable while the system is in operation

Time Slice

- **Fixed length time slice is given sequentially to each processor, round-robin fashion**

Polling

- **Unit address polling - Bus controller advances the address to identify the requesting unit**

LRU

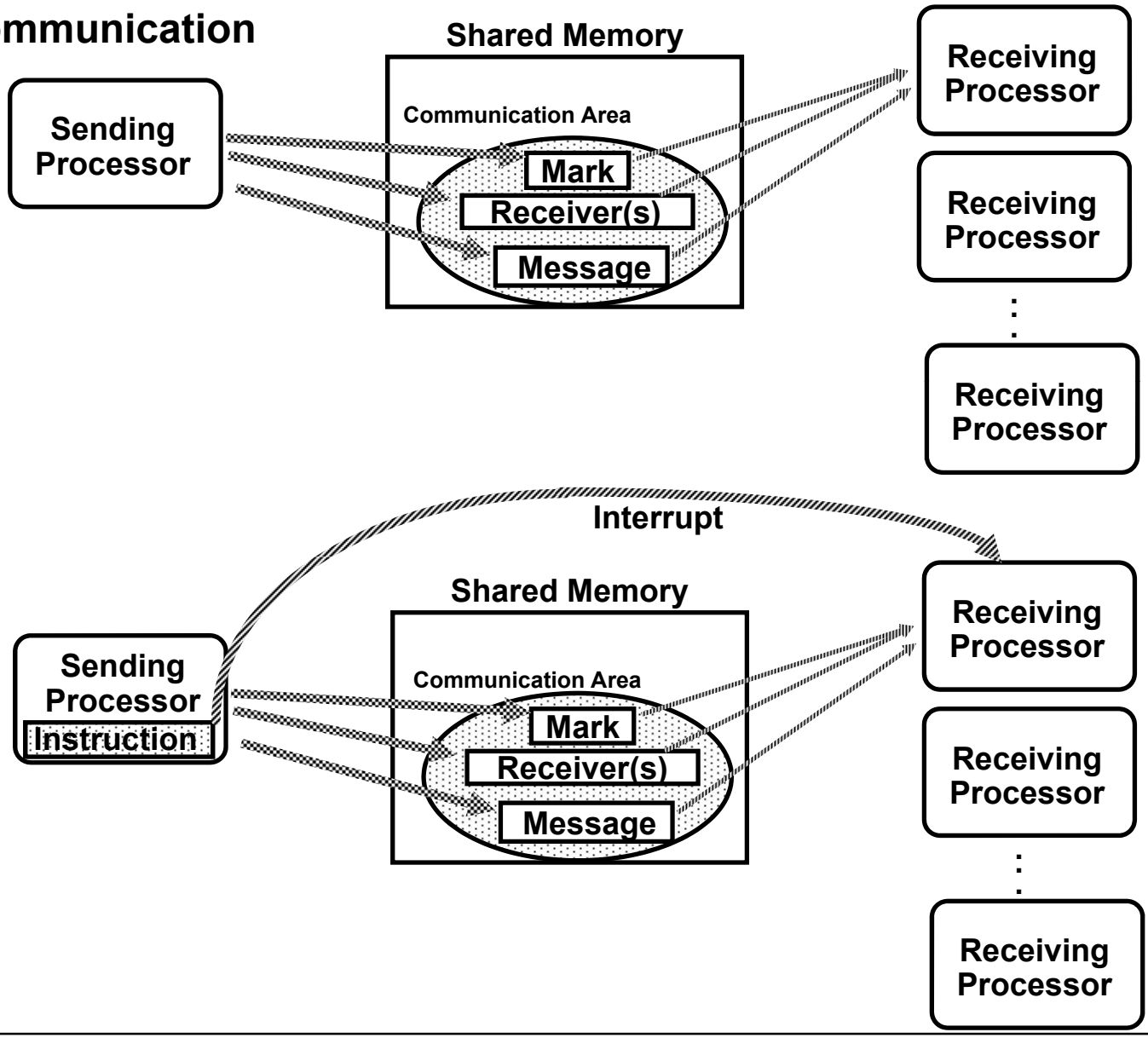
FIFO

Rotating Daisy Chain

- **Conventional Daisy Chain - Highest priority to the nearest unit to the bus controller**
- **Rotating Daisy Chain - Highest priority to the unit that is nearest to the unit that has most recently accessed the bus(it becomes the bus controller)**

INTERPROCESSOR COMMUNICATION

Interprocessor Communication



INTERPROCESSOR SYNCHRONIZATION

Synchronization

Communication of control information between processors

- To enforce the correct sequence of processes

- To ensure mutually exclusive access to shared writable data

Hardware Implementation

Mutual Exclusion with a *Semaphore*

Mutual Exclusion

- One processor to exclude or lock out access to shared resource by other processors when it is in a *Critical Section*

- Critical Section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource

Semaphore

- A binary variable

- 1: A processor is executing a critical section, that not available to other processors

0: Available to any requesting processor

- Software controlled Flag that is stored in memory that all processors can be access

SEMAPHORE

Testing and Setting the Semaphore

- Avoid two or more processors test or set the same semaphore
- May cause two or more processors enter the same critical section at the same time
- Must be implemented with an indivisible operation

```
R <- M[SEM]        / Test semaphore /
M[SEM] <- 1        / Set semaphore /
```

These are being done while *lock*ed, so that other processors cannot test and set while current processor is being executing these instructions

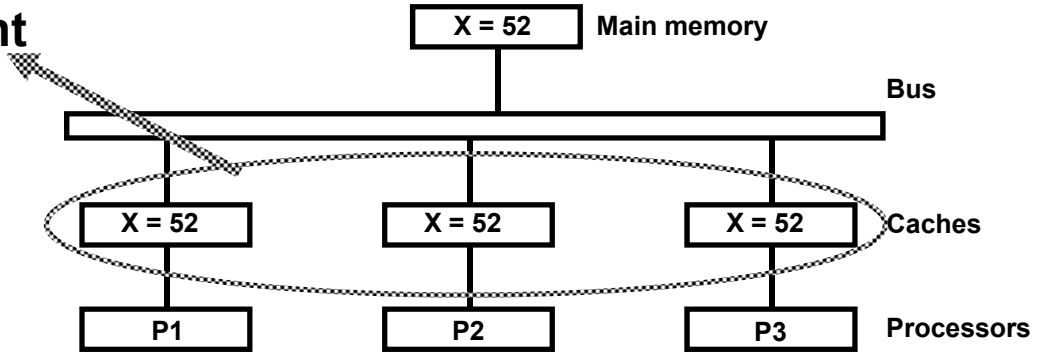
If R=1, another processor is executing the critical section, the processor executed this instruction does not access the shared memory

If R=0, available for access, set the semaphore to 1 and access

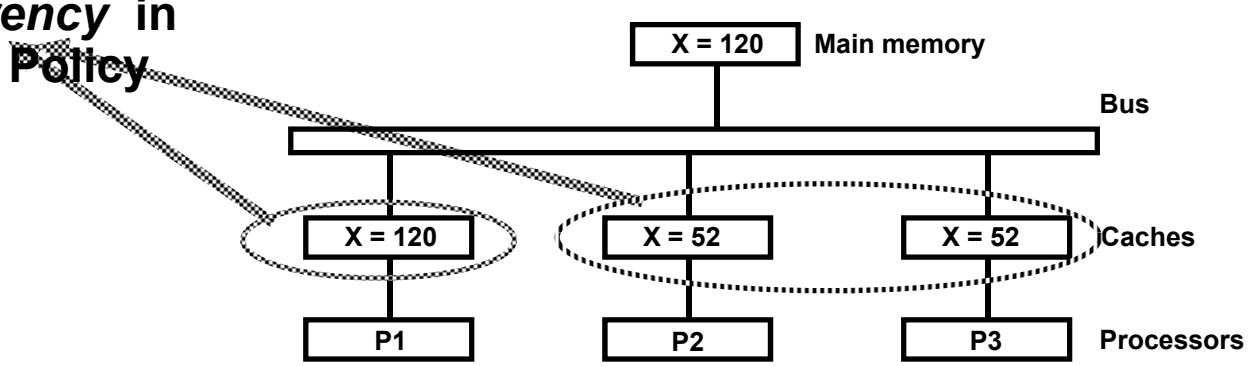
The last instruction in the program must clear the semaphore

CACHE COHERENCE

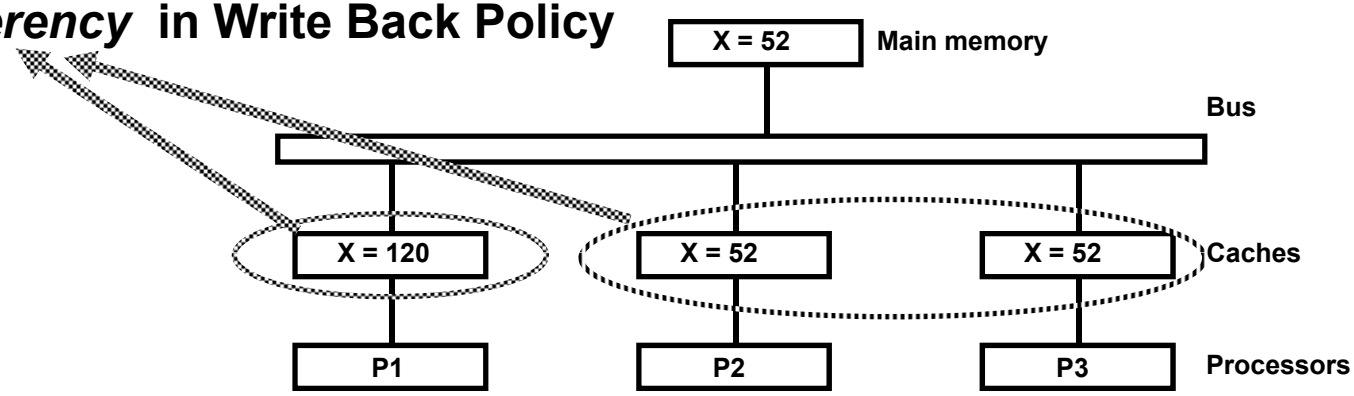
Caches are Coherent



Cache Incoherency in Write Through Policy



Cache Incoherency in Write Back Policy



MAINTAINING CACHE COHERENCY

Shared Cache

- Disallow private cache
- Access time delay

Software Approaches

* Read-Only Data are Cacheable

- Private Cache is for Read-Only data
- Shared Writable Data are not cacheable
- Compiler tags data as cacheable and noncacheable
- Degrade performance due to software overhead

* Centralized Global Table

- Status of each memory block is maintained in CGT: RO(Read-Only); RW(Read and Write)
- All caches can have copies of RO blocks
- Only one cache can have a copy of RW block

Hardware Approaches

* Snoopy Cache Controller

- Cache Controllers monitor all the bus requests from CPUs and IOPs
- All caches attached to the bus monitor the write operations
- When a word in a cache is written, memory is also updated (write through)
- Local snoopy controllers in all other caches check their memory to determine if they have a copy of that word; If they have, that location is marked invalid(future reference to this location causes cache miss)

PARALLEL COMPUTING

Grosche's Law

Grosch's Law states that the speed of computers is proportional to the square of their cost. Thus if you are looking for a fast computer, you are better off spending your money buying one large computer than two small computers and connecting them.

Grosch's Law is true within classes of computers, but not true between classes. Computers may be priced according to Groach's Law, but the Law cannot be true asymptotically.

Minsky's Conjecture

Minsky's conjecture states that the speedup achievable by a parallel computer increases as the logarithm of the number of processing elements,thus making large-scale parallelism unproductive.

Many experimental results have shown linear speedup for over 100 processors.

PARALLEL COMPUTING

History

History tells us that the speed of traditional single CPU Computers has increased 10 folds every 5 years.
Why should great effort be expended to devise a parallel computer that will perform tasks 10 times faster when, by the time the new architecture is developed and implemented, single CPU computers will be just as fast.

Utilizing parallelism is better than waiting.

Amdahl's Law

A small number of sequential operations can effectively limit the speedup of a parallel algorithm.

Let f be the fraction of operations in a computation that must be performed sequentially, where 0 < f < 1. Then the maximum speedup S achievable by a parallel computer with p processors performing the computation is $S < 1 / [f + (1 - f) / p]$. For example, if 10% of the computation must be performed sequentially, then the maximum speedup achievable is 10, no matter how many processors a parallel computer has.

There exist some parallel algorithms with almost no sequential operations. As the problem size(n) increases, f becomes smaller (f -> 0 as n->∞). In this case, $\lim_{n \to \infty} S = p$.

PARALLEL COMPUTING

Pipelined Computers are Sufficient

Most supercomputers are vector computers, and most of the successes attributed to supercomputers have accomplished on pipelined vector processors, especially Cray=1 and Cyber-205.

If only vector operations can be executed at high speed, supercomputers will not be able to tackle a large number of important problems. The latest supercomputers incorporate both pipelining and high level parallelism (e.g., Cray-2)

Software Inertia

Billions of dollars worth of FORTRAN software exists. Who will rewrite them? Virtually no programmers have any experience with a machine other than a single CPU computer. Who will retrain them ?

INTERCONNECTION NETWORKS

Switching Network (Dynamic Network)

Processors (and Memory) are connected to routing switches like in telephone system

- Switches might have queues(combining logic), which improve functionality but increase latency
- Switch settings may be determined by message headers or preset by controller
- Connections can be packet-switched or circuit-switched(remain connected as long as it is needed)
- Usually NUMA, blocking, often scalable and upgradable

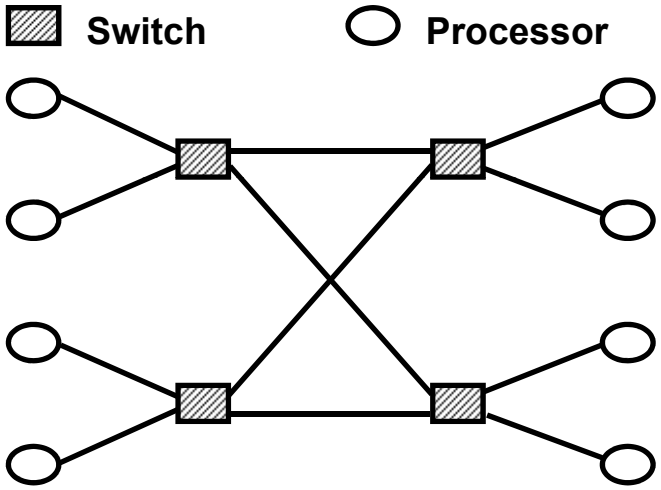
Point-Point (Static Network)

Processors are directly connected to only certain other processors and must go multiple hops to get to additional processors

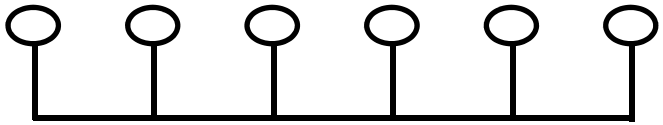
- Usually distributed memory
- Hardware may handle only single hops, or multiple hops
- Software may mask hardware limitations
- Latency is related to graph diameter, among many other factors
- Usually NUMA, nonblocking, scalable, upgradable
- Ring, Mesh, Torus, Hypercube, Binary Tree

INTERCONNECTION NETWORKS

Multistage Interconnect



Bus



INTERCONNECTION NETWORKS

Static Topology - Direct Connection

- Provide a direct inter-processor communication path
- Usually for distributed-memory multiprocessor

Dynamic Topology - Indirect Connection

- Provide a physically separate switching network for inter-processor communication
- Usually for shared-memory multiprocessor

Direct Connection

Interconnection Network

A graph G(V,E)

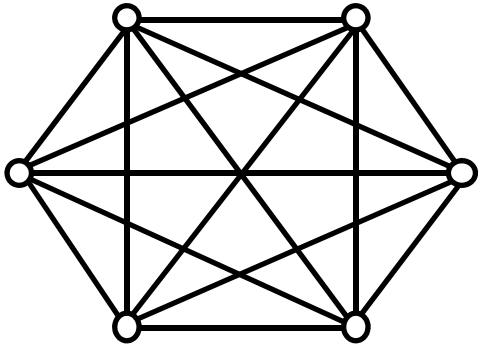
V: a set of processors (nodes)

E: a set of wires (edges)

Performance Measures: - degree, diameter, etc

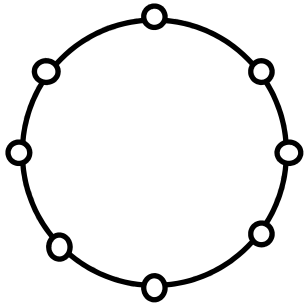
INTERCONNECTION NETWORKS

Complete connection



- Every processor is directly connected to every other processors
- Diameter = 1, Degree = p - 1
- # of wires = p (p - 1) / 2; dominant cost
- Fan-in/fanout limitation makes it impractical for large p
- Interesting as a theoretical model because algorithm bounds for this model are automatically lower bounds for all direct connection machines

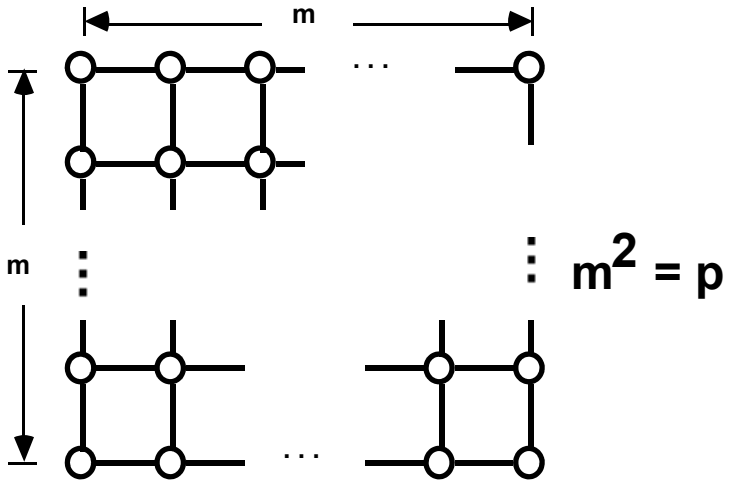
Ring



- Degree = 2, (not a function of p)
- Diameter = $\lfloor p/2 \rfloor$

INTERCONNECTION NETWORKS

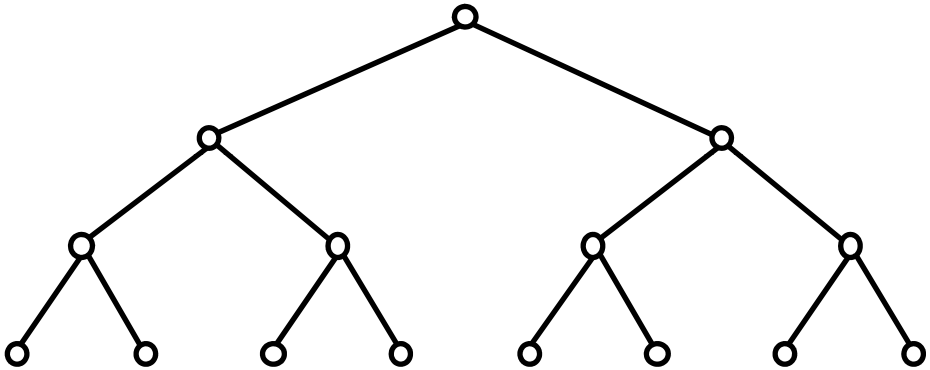
• 2-Mesh



- Degree = 4
- Diameter = 2(m - 1)
- In general, an n-dimensional mesh has
  diameter = d ($p^{1/n}$ - 1)
- Diameter can be halved by having wrap-around connections (-> Torus)
- Ring is a 1-dimensional mesh with wrap-around connection

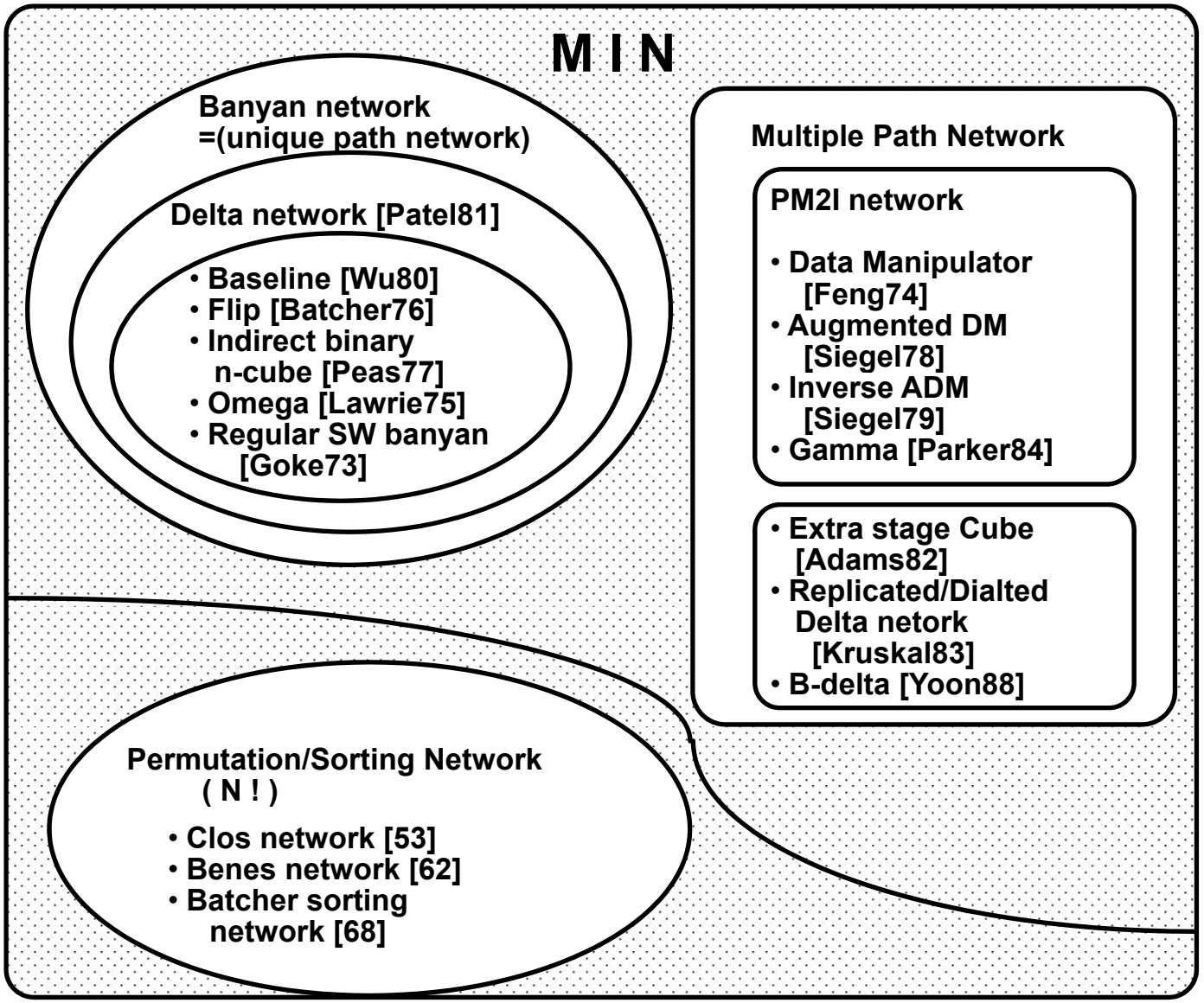
INTERCONNECTION NETWORK

Binary Tree



- Degree = 3
- Diameter = $2 \log \frac{p+1}{2}$

MIN SPACE


MIN
Banyan network
=(unique path network)
Delta network [Patel81]
• Baseline [Wu80]
• Flip [Batcher76]
• Indirect binary
n-cube [Peas77]
• Omega [Lawrie75]
• Regular SW banyan
[Goke73]
Multiple Path Network
PM2I network
• Data Manipulator
[Feng74]
• Augmented DM
[Siegel78]
• Inverse ADM
[Siegel79]
• Gamma [Parker84]
• Extra stage Cube
[Adams82]
• Replicated/Dialted
Delta netork
[Kruskal83]
• B-delta [Yoon88]
Permutation/Sorting Network
(N !)
• Clos network [53]
• Benes network [62]
• Batcher sorting
network [68]

SOME CURRENT PARALLEL COMPUTERS

DM-SIMD

- AMT DAP
- Goodyear MPP
- Thinking Machines CM series
- MasPar MP1
- IBM GF11

SM-MIMD

- Alliant FX
- BBN Butterfly
- Encore Multimax
- Sequent Balance/Symmetry
- CRAY 2, X-MP, Y-MP
- IBM RP3
- U. Illinois CEDAR

DM-MIMD

- Intel iPSC series, Delta machine
- NCUBE series
- Meiko Computing Surface
- Carnegie-Mellon/ Intel iWarp