Register Transfer Language and Microoperations

Unit-2

Contents:

1. Register Transfer Language

SIMPLE DIGITAL SYSTEMS

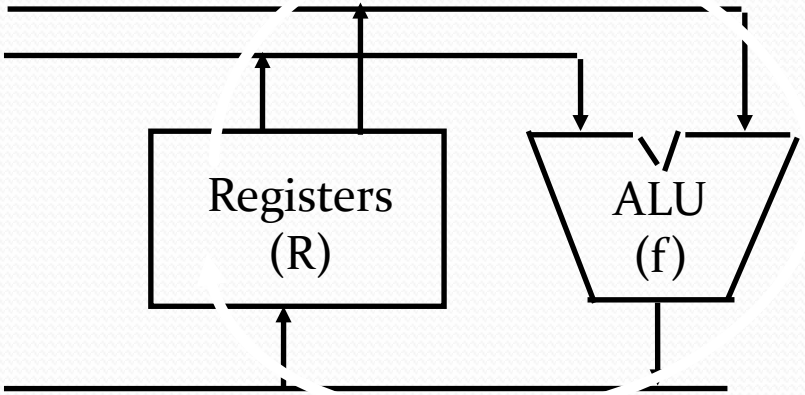
- Combinational and sequential circuits can be used to create simple digital systems.
- These are the low-level building blocks of a digital computer.
- Simple digital systems are frequently characterized in terms of
  - the registers they contain, and
  - the operations that they perform.
- Typically,
  - Operations are performed on the data in the registers.
  - Information is passed between registers.

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
  - …

Microoperations (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

The internal organization of a computer is defined as

1. Set of registers and their functions.
2. Microoperations (Set of allowable microoperations provided by the organization of the computer).
3. 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 called *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.

2. Register Transfer

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

Register Transfer contd..

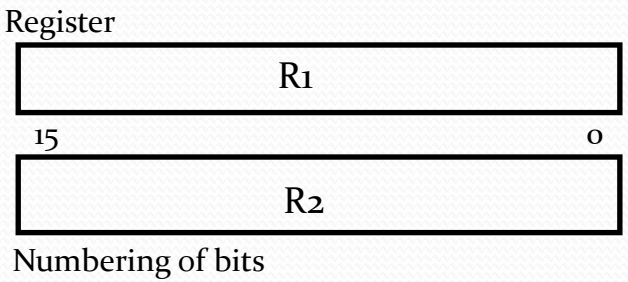
- Designation of a register

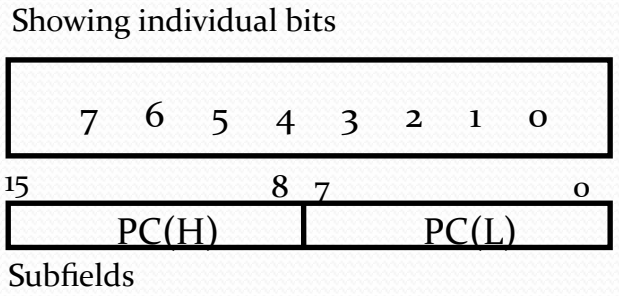
  - a register

  - portion of a register

  - a bit of a register

- Common ways of drawing the block diagram of a register





Register Transfer contd..

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

Register Transfer contd..

- A register transfer such as

  $R3 \leftarrow R5$

  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.

Register Transfer contd..

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 P is 1, the action takes place
- This is represented as:

  P: R2 ← R1

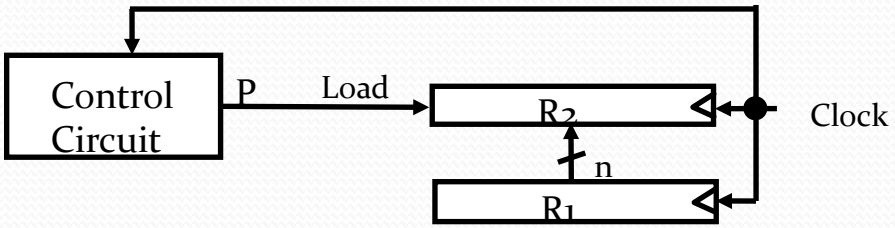
  Which means "if P = 1, then load the contents of register R1 into register R2", i.e., if (P = 1) then (R2 ← R1).

Hardware Implementation Of Controlled Transfers

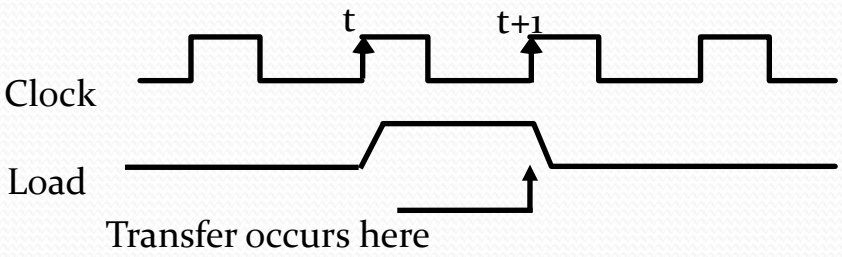
Implementation of controlled transfer

P: $R2 \leftarrow R1$

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: $R3 \leftarrow R5$, $MAR \leftarrow IR$

- 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

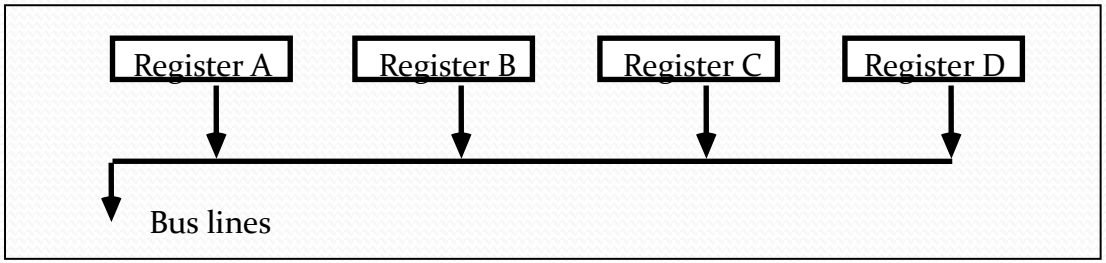
Connecting Registers

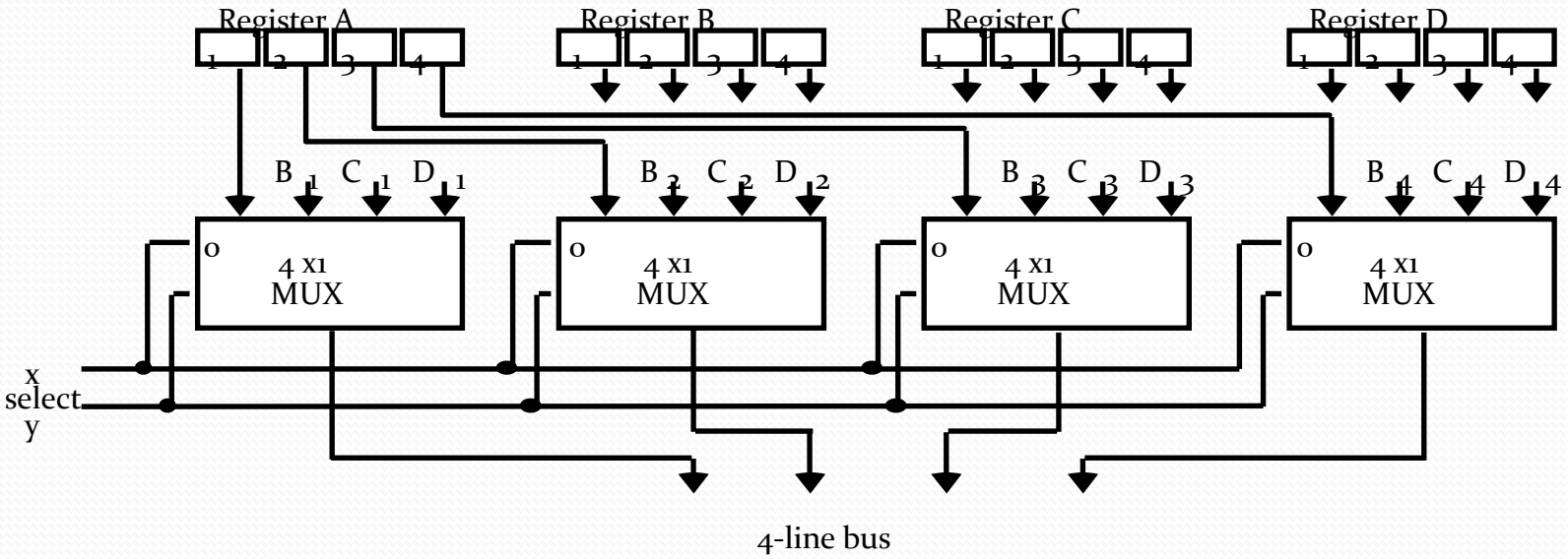
- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers.
- To completely connect n registers → n(n-1) lines.
  - This is not a realistic approach to use in a large digital system
- Instead, take a different approach
- Have one centralized set of circuits for data transfer .bus
- Have control circuits to select which register is the source, and which is the destination.

3. Bus and Bus Transfer

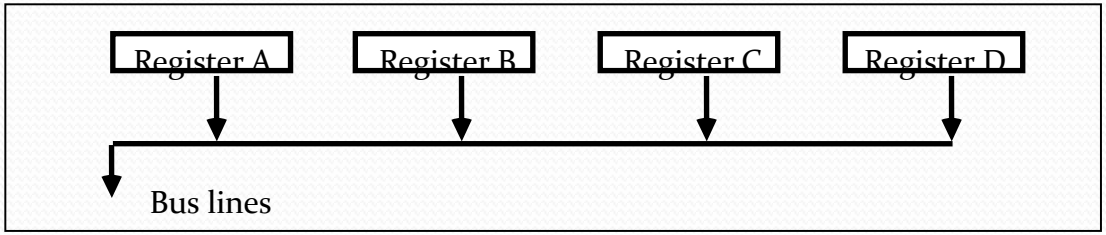
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

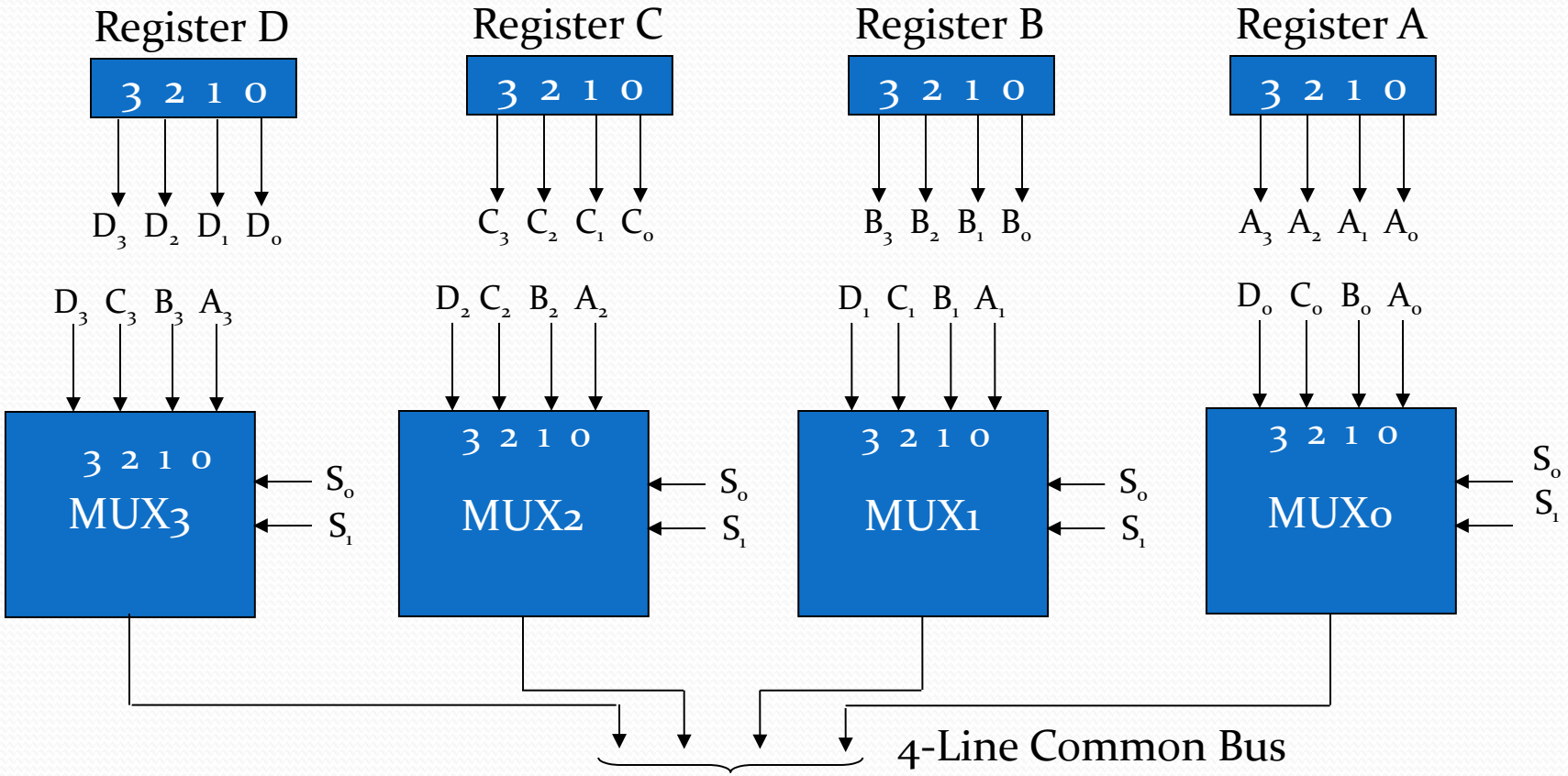
From a register to bus: BUS ← R





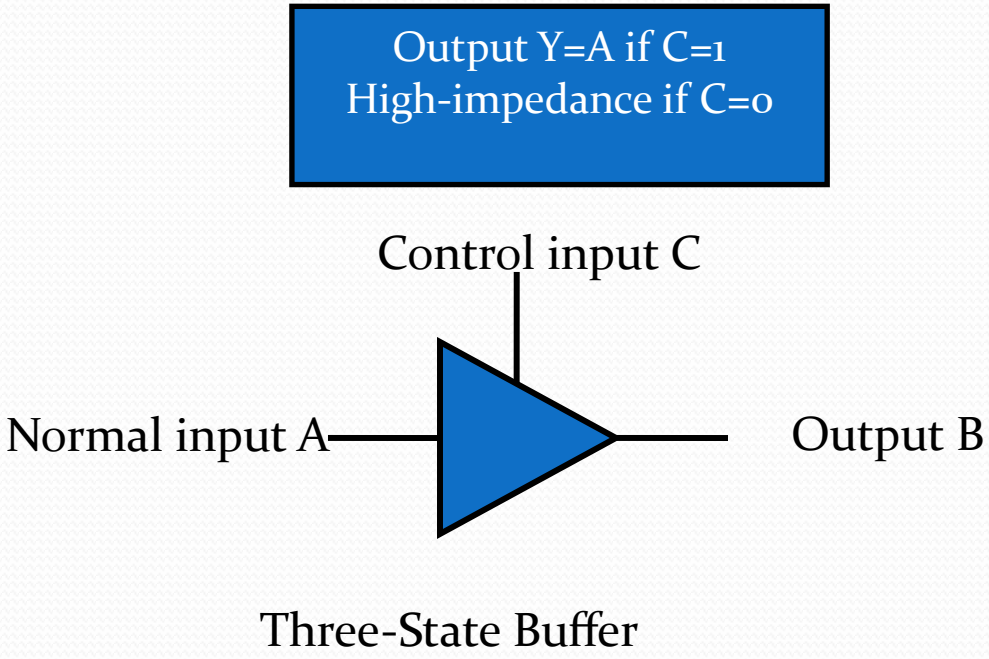
Bus and Memory Transfers





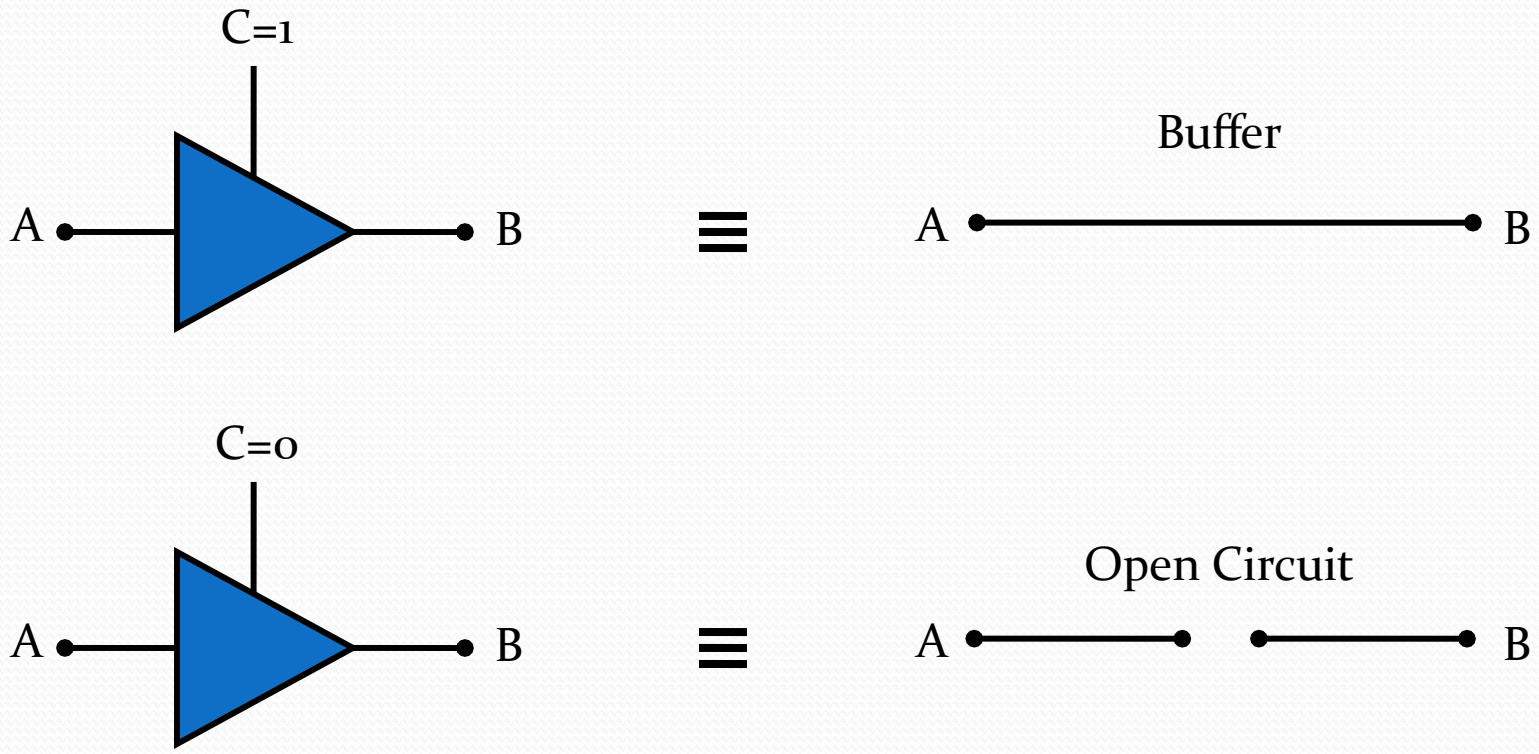
Bus and Memory Transfers: Three-State Bus Buffers

- A bus system can be constructed with three-state buffer gates instead of multiplexers.
- A three-state buffer is a digital circuit that exhibits three states: logic-0, logic-1, and high-impedance (Hi-Z)



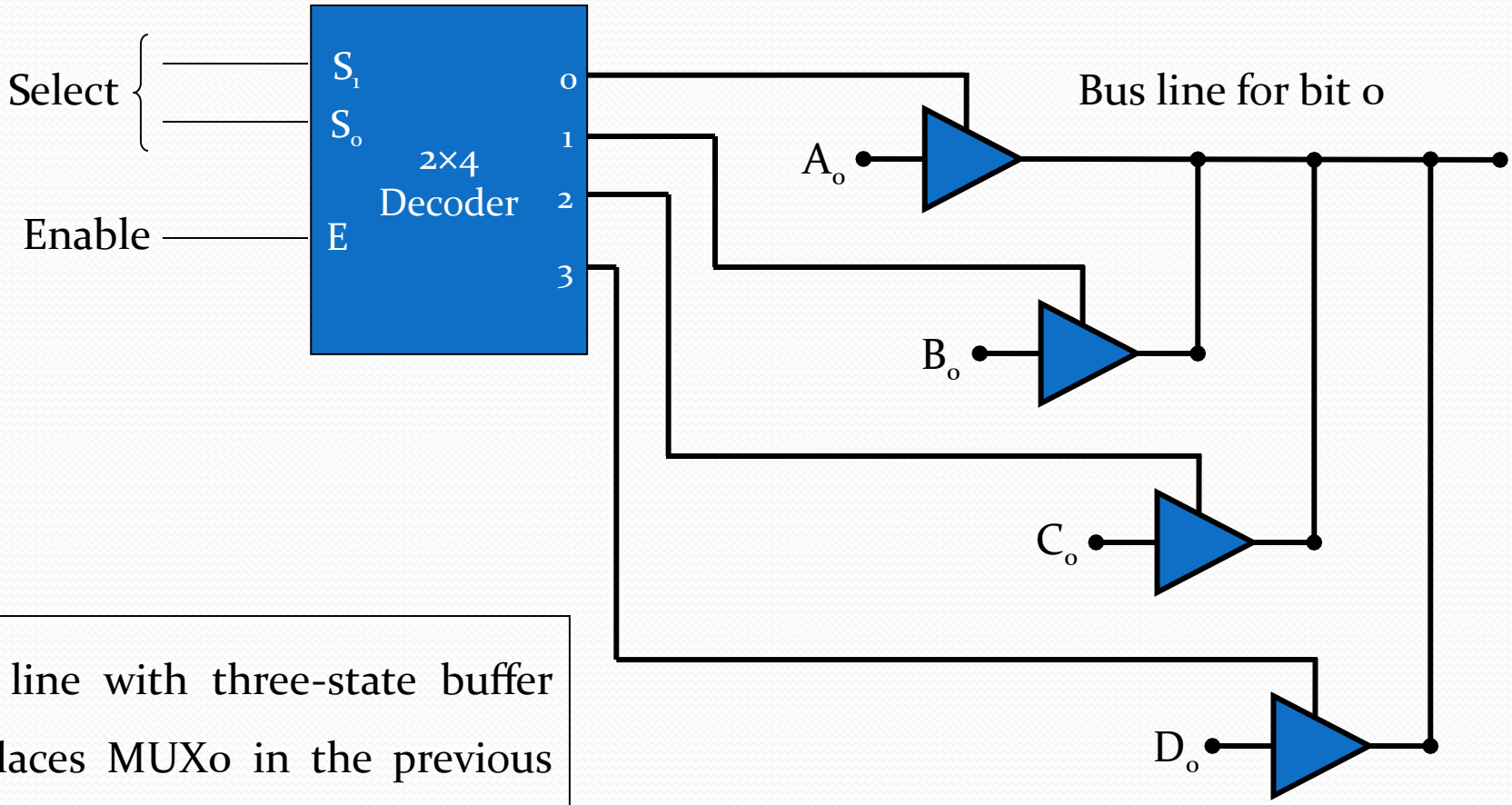
Bus and Memory Transfers: Three-State Bus Buffers

cont.



Bus and Memory Transfers: Three-State Bus Buffers

cont.



Bus line with three-state buffer (replaces MUXo in the previous diagram)

Bus Transfer In RTL

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

$$R2 \leftarrow R1$$

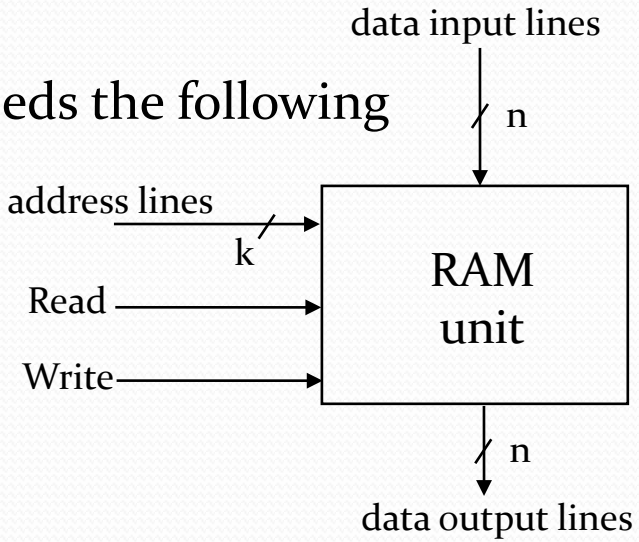
or

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

- 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



Bus and Memory Transfers: Memory Transfer

- Memory read : Transfer from memory
- Memory write : Transfer to memory
- Data being read or written is called a memory word (called M).
- It is necessary to specify the address of M when writing /reading memory.
- This is done by enclosing the address in square brackets following the letter M.

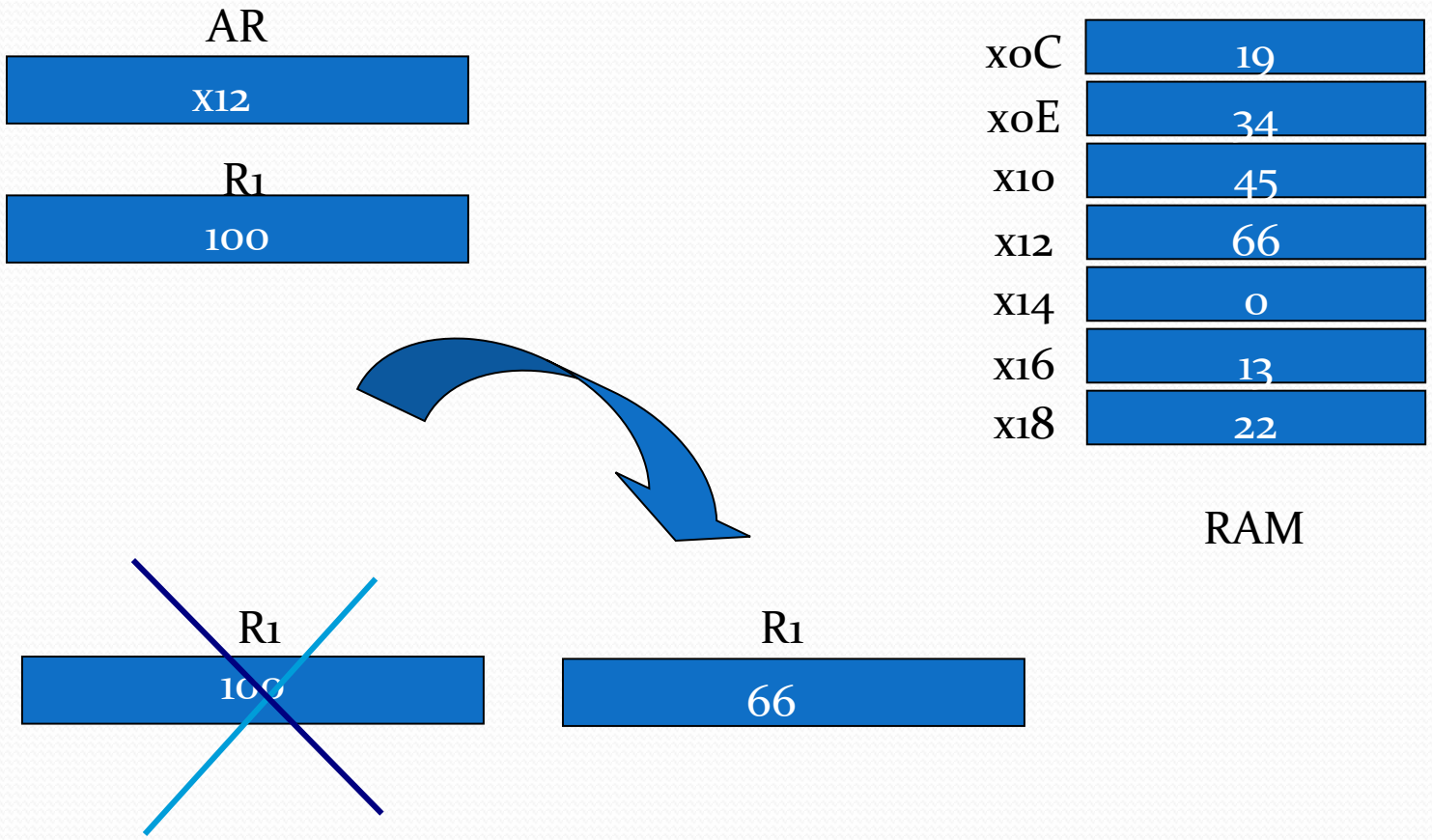
Memory Read

- 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 values are sent over the bus to be loaded into register R1.

Bus and Memory Transfers: Memory Transfer cont.



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

4. Arithmetic Microoperations

- The microoperations most often encountered in digital computers are classified into four categories:
  - *Register transfer* microoperations
  - *Arithmetic* microoperations (on numeric data stored in the registers)
  - *Logic microoperations* (bit manipulations on non-numeric data)
  - *Shift microoperations*

Arithmetic Microoperations cont.

- The basic arithmetic microoperations are:
  - addition
  - subtraction
  - increment
  - decrement
  - shift
- Addition Microoperation:

$$R3 \leftarrow R1+R2$$

- Subtraction Microoperation:

$$R3 \quad R1-R2 \text{ (or)} \leftarrow$$

1's complement

$$R3 \leftarrow R1+ \overline{R2} +1$$

Arithmetic Microoperations cont

- One's Complement Microoperation:

$$R2 \leftarrow \overline{R2}$$

- Two's Complement Microoperation:

$$R2 \leftarrow \overline{R2}+1$$

- Increment Microoperation:

$$R2 \leftarrow R2+1$$

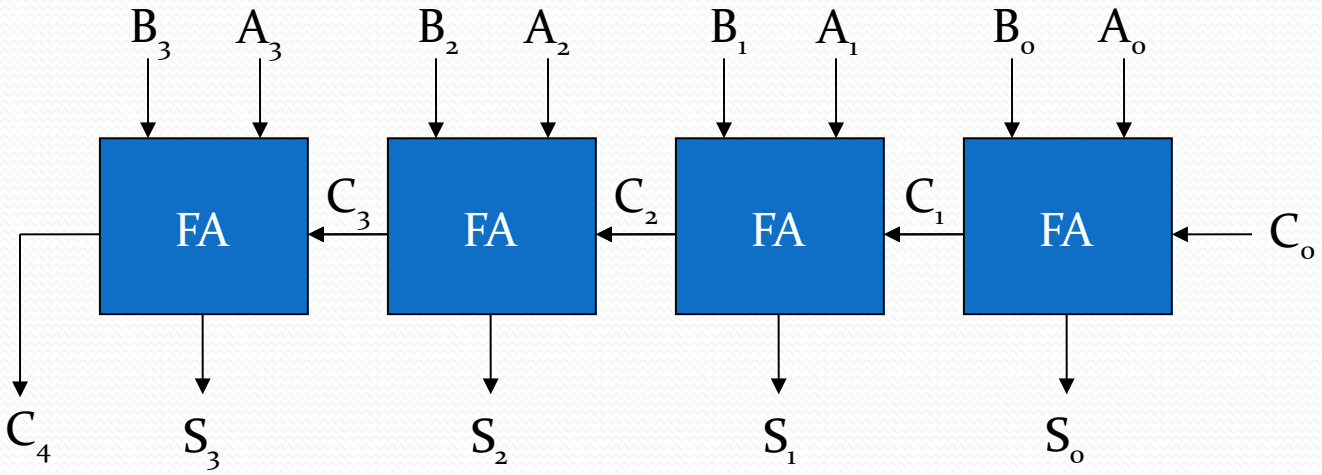
- Decrement Microoperation:

$$R2 \leftarrow R2-1$$

Summary of Typical Arithmetic Micro-Operations

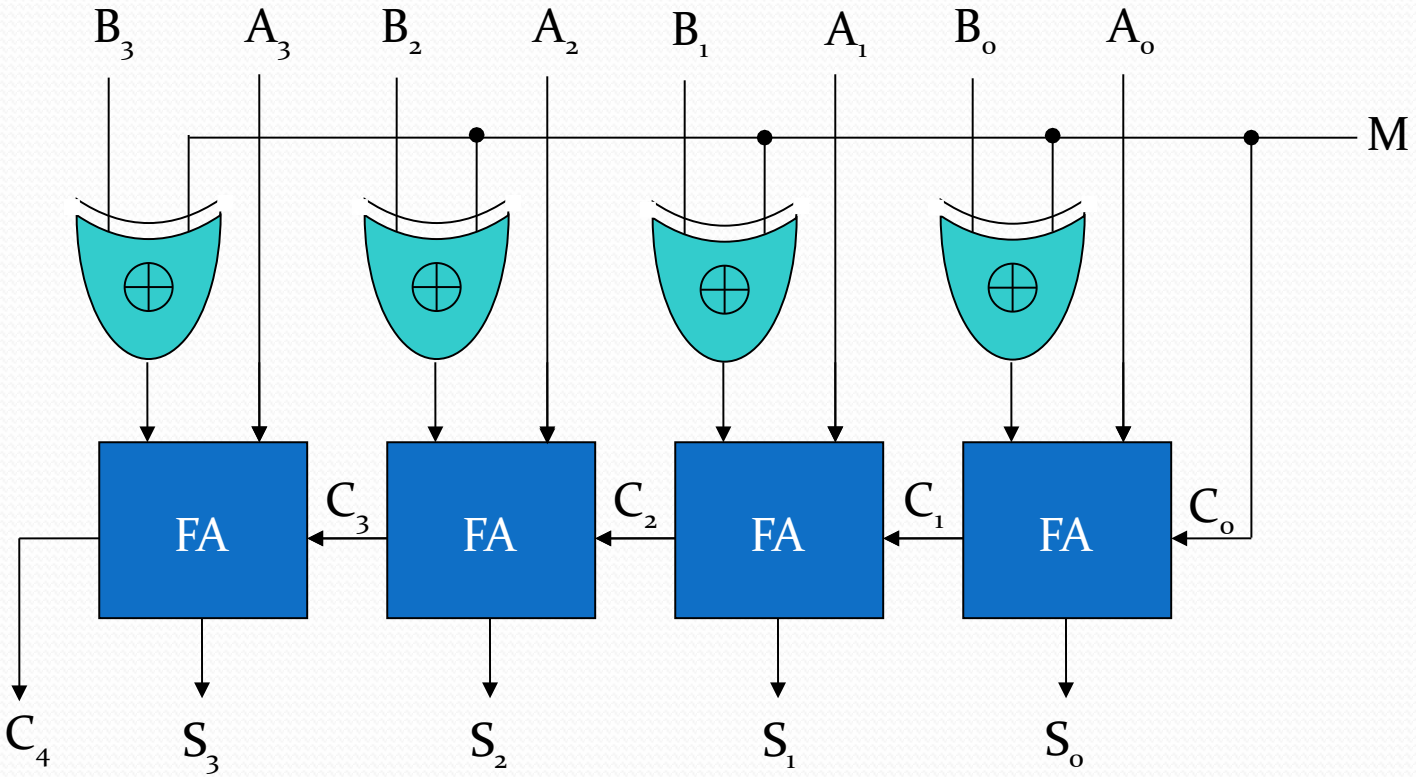
Operations	Description
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow \overline{R2}$	Complement the contents of R2
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of R2 (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	subtraction
$R1 \leftarrow R1 + 1$	Increment
$R1 \leftarrow R1 - 1$	Decrement

Arithmetic Microoperations: Binary Adder



4-bit binary adder (connection of FAs)

Arithmetic Microoperations: Binary Adder-Subtractor



4-bit adder-subtractor

Arithmetic Microoperations : Binary Adder-Subtractor

- For unsigned numbers, this gives A – B if A≥B or the 2's complement of (B – A) if A < B.

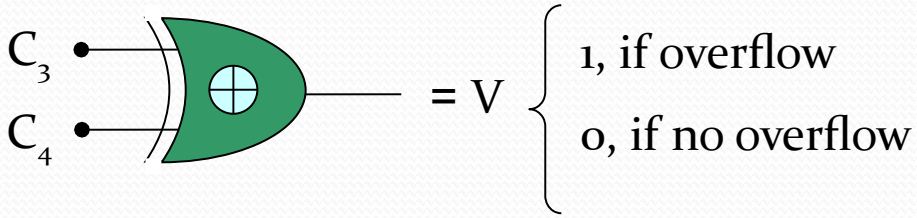
  (example: 3 – 5 = -2= 1110)

- For signed numbers, the result is A – B provided that there is no overflow. (example : -3 – 5= -8)

  1101

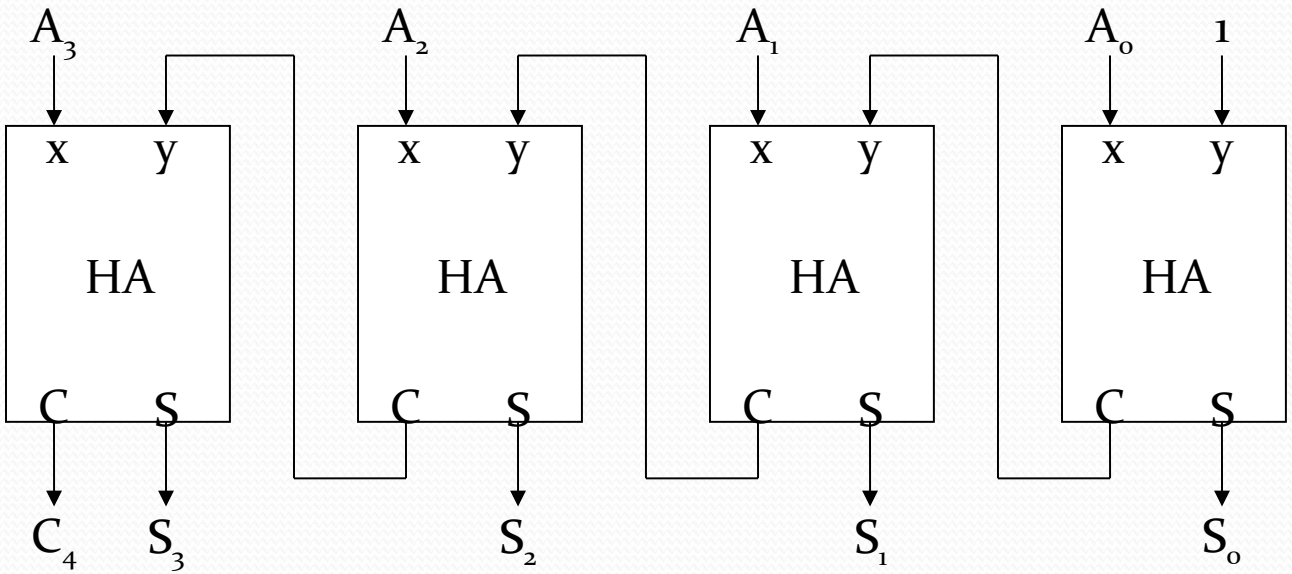
  1011 +

  1000



Overflow detector for signed numbers

Arithmetic Microoperations Binary Incrementer



4-bit Binary Incrementer

- Binary Incrementer can also be implemented using a counter.
- A binary decrementer can be implemented by adding 1111 to the desired register each time.

Arithmetic Microoperations Arithmetic Circuit

- This circuit performs seven distinct arithmetic operations and the basic component of it is the parallel adder.
- The output of the binary adder is calculated from the following arithmetic sum:
  - $D = A + Y + C_{in}$

Arithmetic Microoperations : Arithmetic Circuit cont.

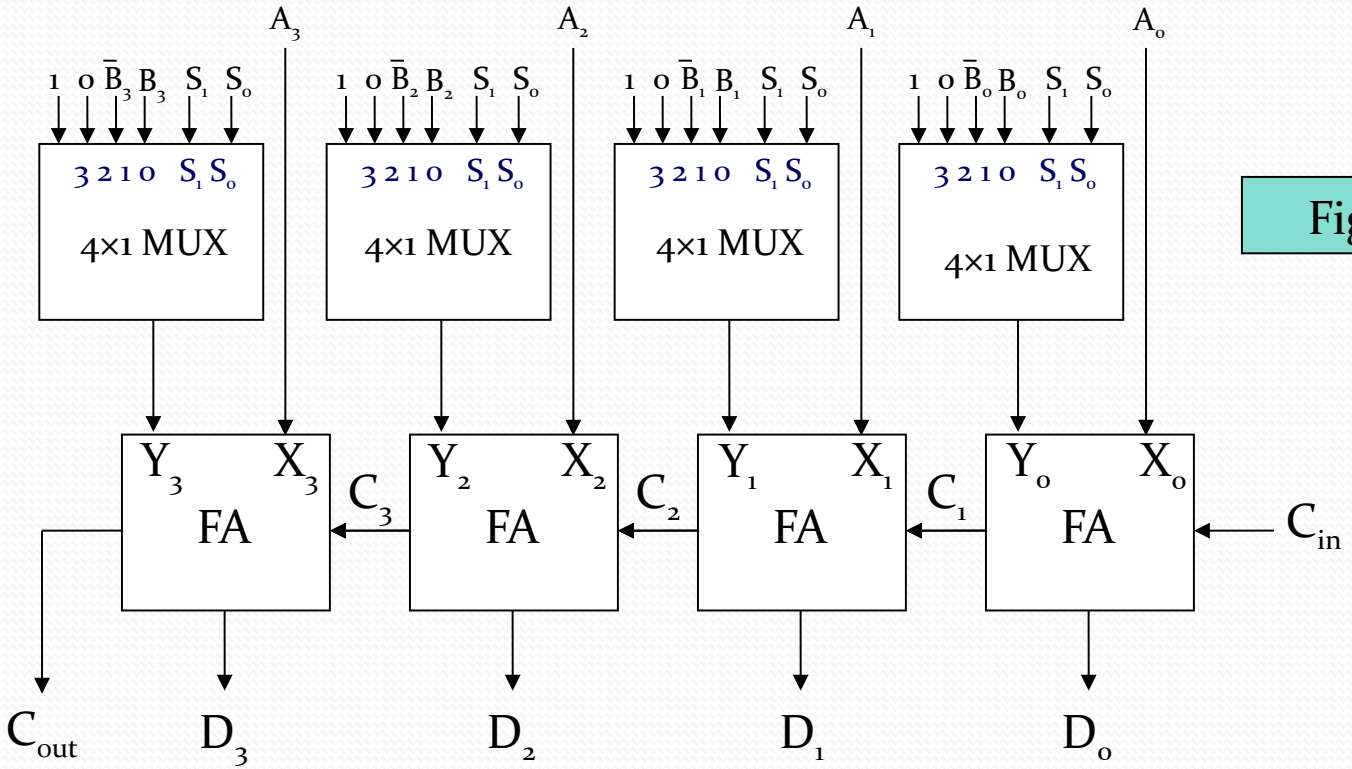
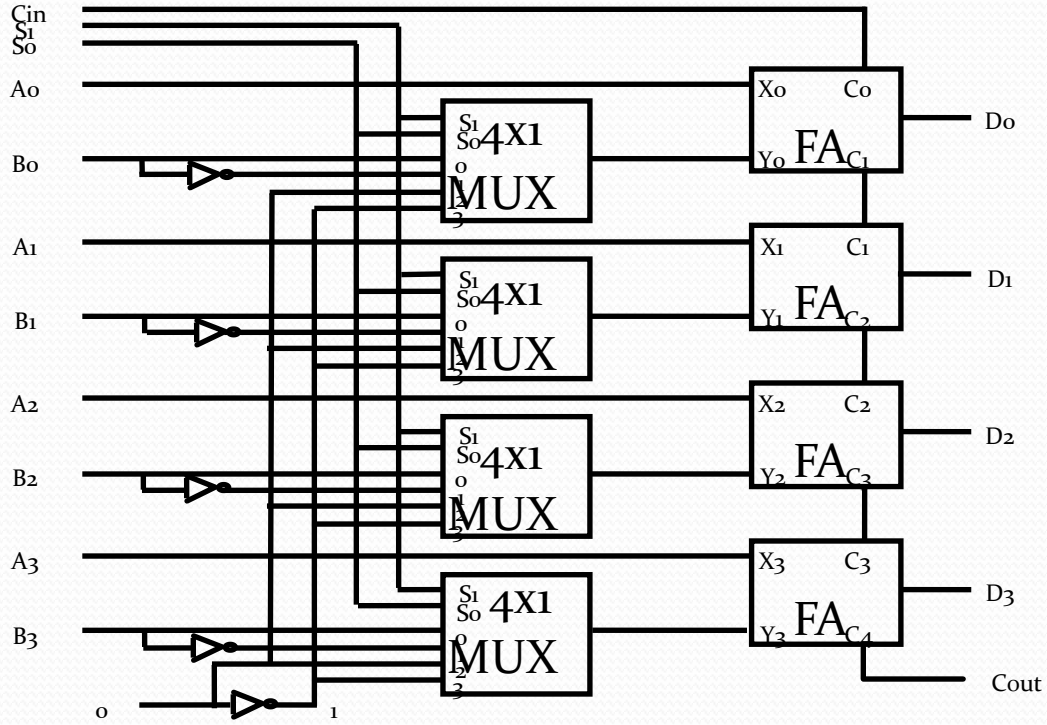


Figure A

4-bit Arithmetic Circuit

Arithmetic Circuit



S1	So	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

5. 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,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 ($\wedge$), OR ($\vee$), XOR ($\oplus$), Complement/NOT
- The others can be created from combination of these.

List of Logic Microoperations

- 16 different logic operations with 2 binary variables.

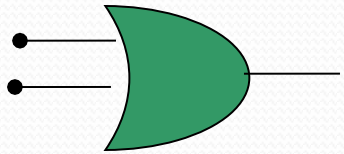
x y	0 0 1 1 0 1 0 1	*Boolean Function*	*Micro-Operations*	*Name*
	0 0 0 0	F0 = 0	F ← 0	Clear
	0 0 0 1	F1 = xy	F ← A ∧ B	AND
	0 0 1 0	F2 = xy'	F ← A ∧ B'	
	0 0 1 1	F3 = x	F ← A	Transfer A
	0 1 0 0	F4 = x'y	F ← A'∧ B	
	0 1 0 1	F5 = y	F ← B	Transfer B
	0 1 1 0	F6 = x ⊕ y	F ← A ⊕ B	Exclusive-OR
	0 1 1 1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

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

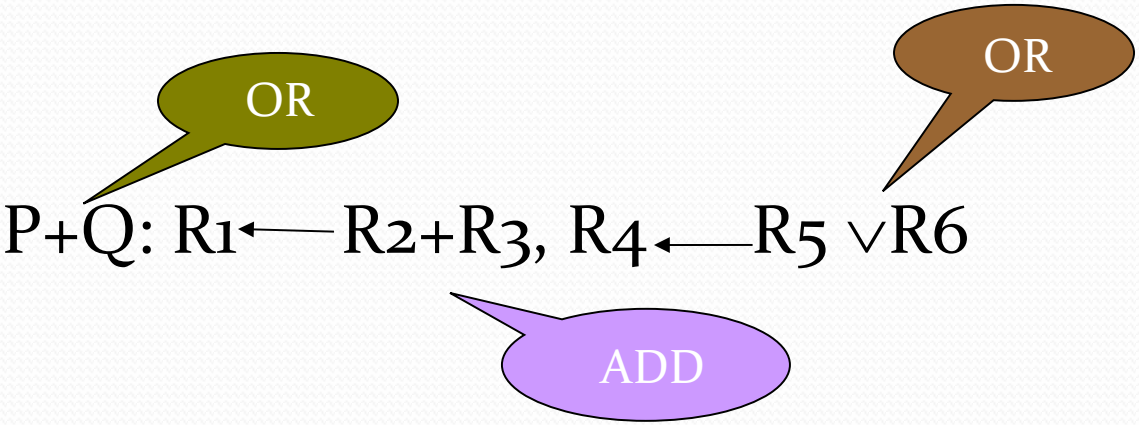
Logic Microoperations

OR Microoperation

- Symbol: $\vee$, +

Gate: 

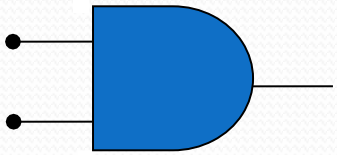
- Example: $100110_2 \vee 1010110_2 = 1110110_2$



Logic Microoperations

AND Microoperation

- Symbol: $\wedge$
- Gate:

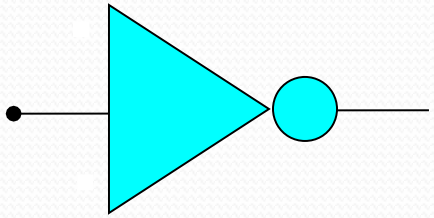


- Example: $100110_2 \wedge 1010110_2 = 0000110_2$

Logic Microoperations

Complement (NOT) Microoperation

- Symbol: $\overline{}$
- Gate:

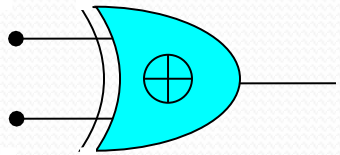


- Example: $1010110_2 = 0101001_2$

Logic Microoperations

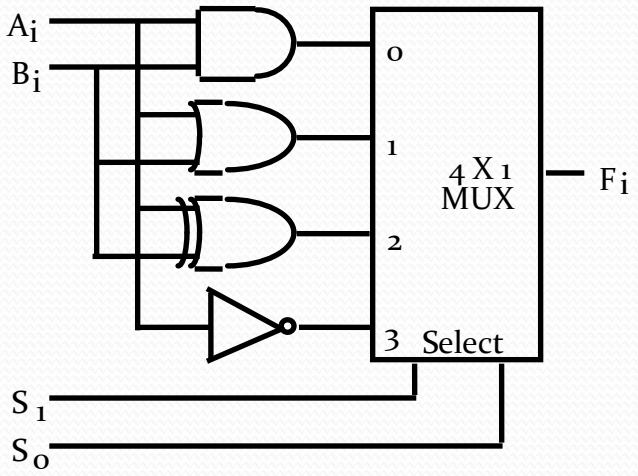
XOR (Exclusive-OR) Microoperation

- Symbol: $\oplus$
- Gate:



- Example: $100110_2 \oplus 1010110_2 = 1110000_2$

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.

$$
\begin{array}{ll}
1\,1\,0\,0 & A_t \\
1\,0\,1\,0 & B \\
\hline
0\,1\,0\,0 & A_{t+1} \qquad (A \leftarrow A \cdot B')
\end{array}
$$

- 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.

$$\begin{array}{ll} 1100 & A_t \\ 1010 & B \\ \hline 1000 & A_{t+1} \end{array} \qquad (A \leftarrow A \cdot B)$$

- 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

$$
\begin{array}{ll}
1100 & A_t \\
1010 & B \\
\hline
0110 & A_{t+1} \qquad (A \leftarrow A \oplus B)
\end{array}
$$

Insert Operation

- Step1: mask the desired bits
- Step2: OR them with the desired value

Example: suppose R1 = 0110 1010, and we desire to replace the leftmost 4 bits (0110) with 1001 then:

- Step1:

```
  0110 1010   A before
  0000 1111   B (mask)
  0000 1010   A After
```

Step2:

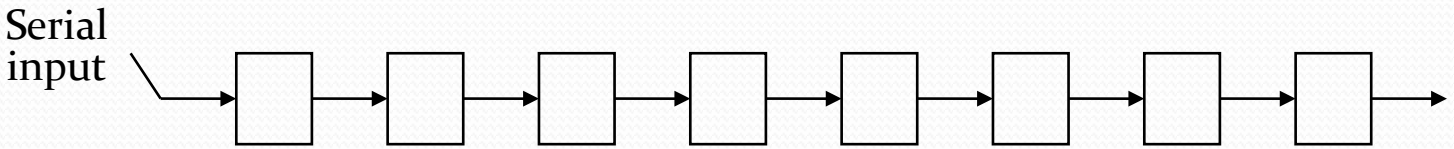
```
  0000 1010   A before
  1001 0000   B (insert)
  1001 1010
```

→ R1 = 1001 1010

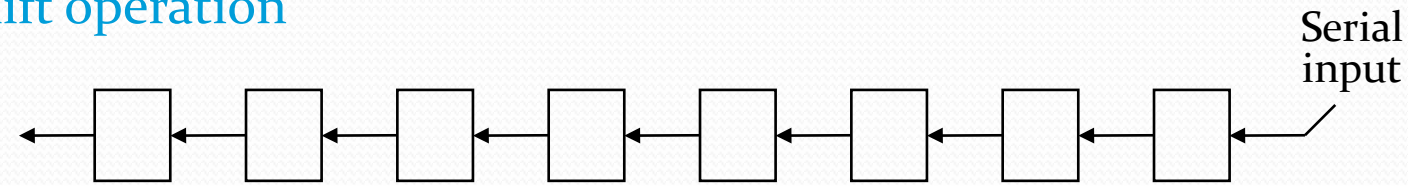
6. Shift Microoperations

- 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

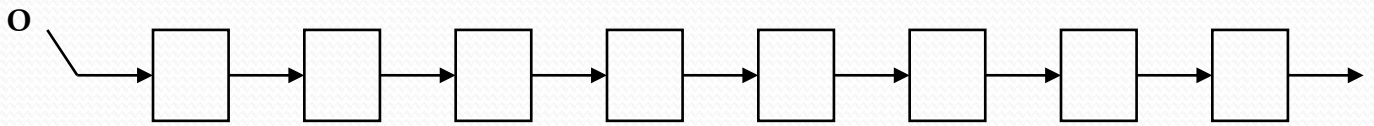


• A left shift operation

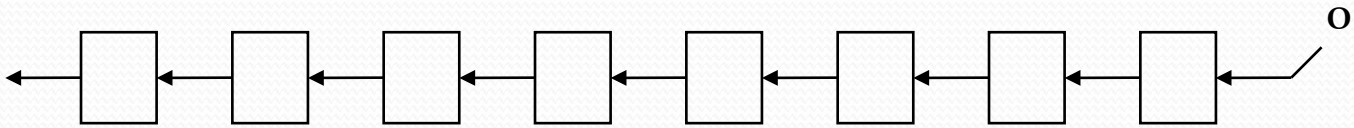


Logical Shift

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



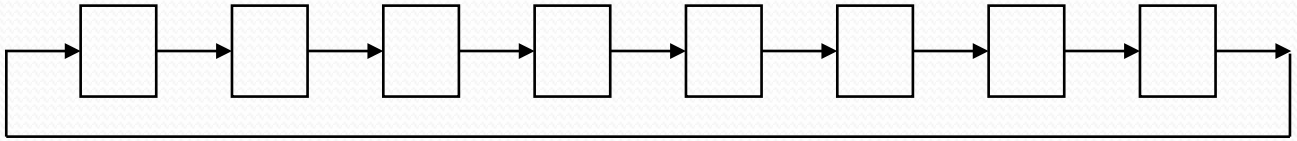
- A left logical shift operation:



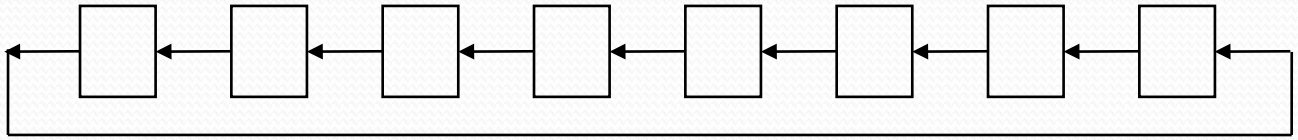
- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right
  - Examples:
    - R2 ← *shr* R2
    - R3 ← *shl* R3

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:

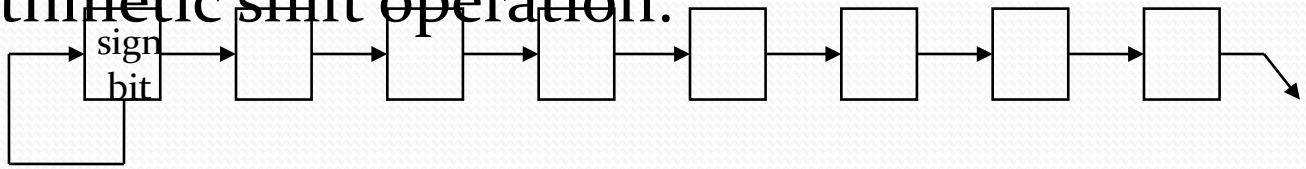


- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - R2 ← *cir* R2
    - R3 ← *cil* R3

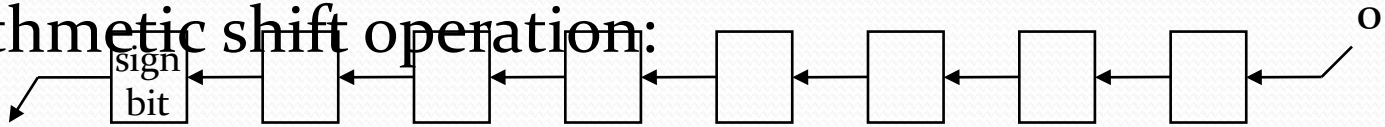
Arithmetic Shift

- 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.

- A right arithmetic shift operation:



- A left arithmetic shift operation:



Arithmetic Shift

- 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

Arithmetic Shifts

- Shifts a signed binary number to the left or right
- An arithmetic shift-left multiplies a signed binary number by 2:

  ashl (00100): 01000

- An arithmetic shift-right divides the number by 2.

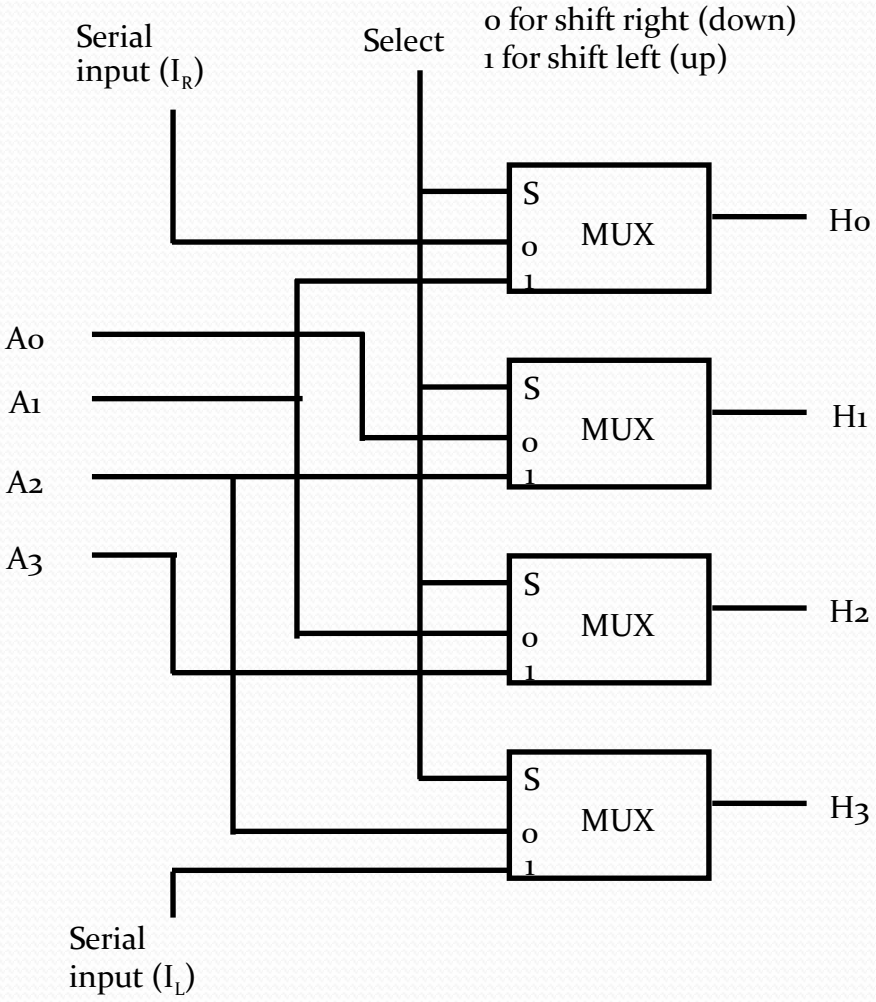
  ashr (00100) : 00010

- An overflow may occur in arithmetic shift-left, and occurs when the sign bit is changed (sign reversal).

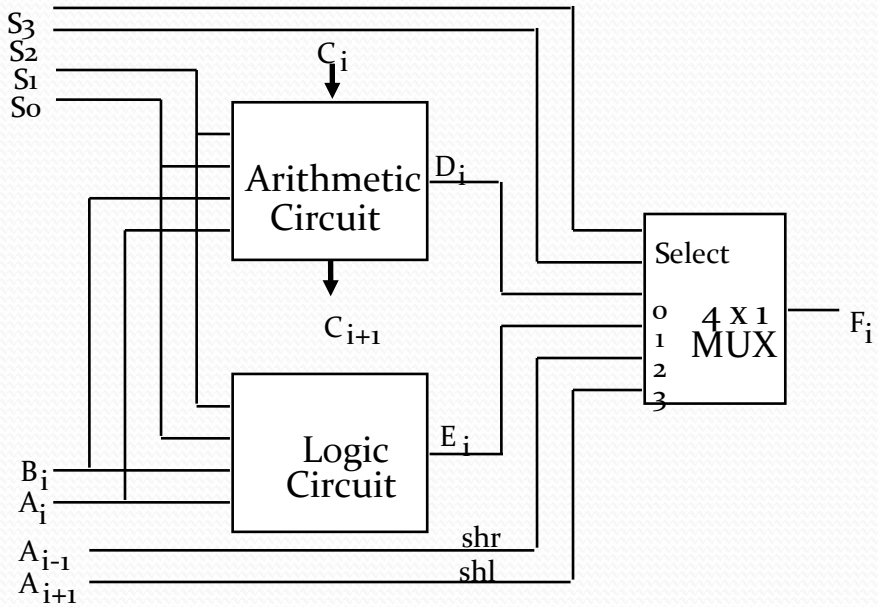
Shift Microoperations

- Example: Assume R1=11001110, then:
  - Arithmetic shift right once : R1 = 11100111
  - Arithmetic shift right twice : R1 = 11110011
  - Arithmetic shift left once : R1 = 10011100
  - Arithmetic shift left twice : R1 = 00111000
  - Logical shift right once : R1 = 01100111
  - Logical shift left once : R1 = 10011100
  - Circular shift right once : R1 = 01100111
  - Circular shift left once : R1 = 10011101

Hardware Implementation Of Shift Microoperations



Arithmetic Logic Shift Unit



S3	S2	S1	So	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
- Instruction Cycle
- Memory Reference Instructions
- Input-Output and Interrupt

8. Instruction Codes

- *Instruction Code*:

It is a group of bits that instruct the computer to perform a specific operation.

Instruction Format

Operation code :

The operation code of an instruction is a group of bits that define such as

add

subtract

multiply

shift

complement.

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

Instruction Format

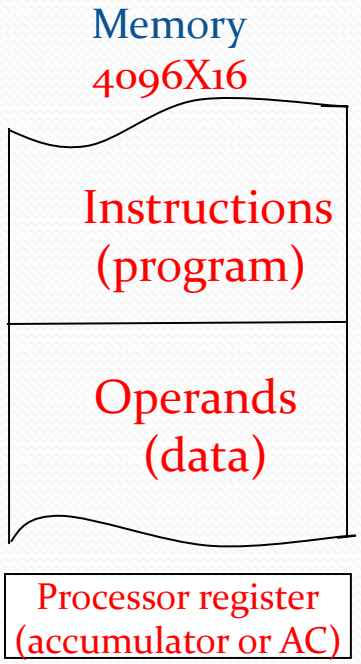


Fig: Stored program organization

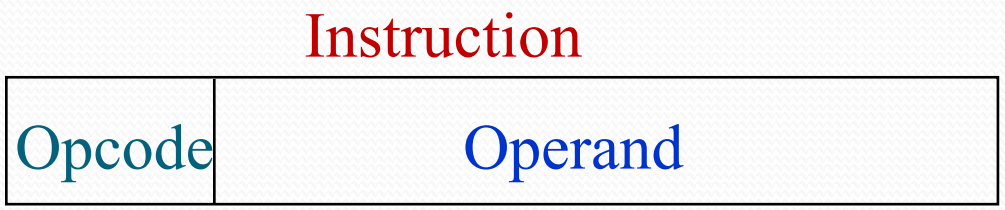
Addressing Modes

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement (Indexed)
- Stack

Immediate Addressing

- Operand is part of instruction
- Operand = address field
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

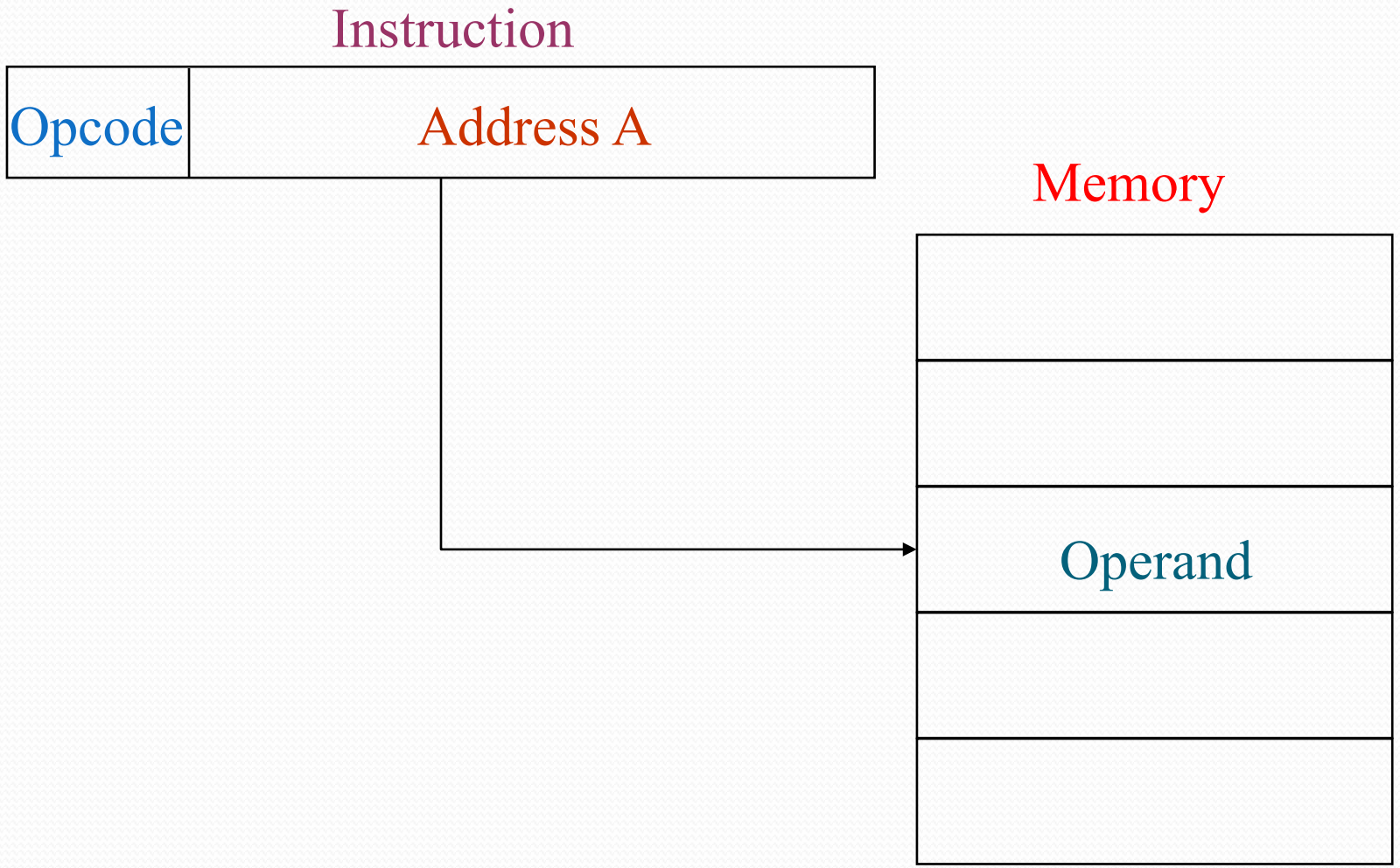
Immediate Addressing Diagram



Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space

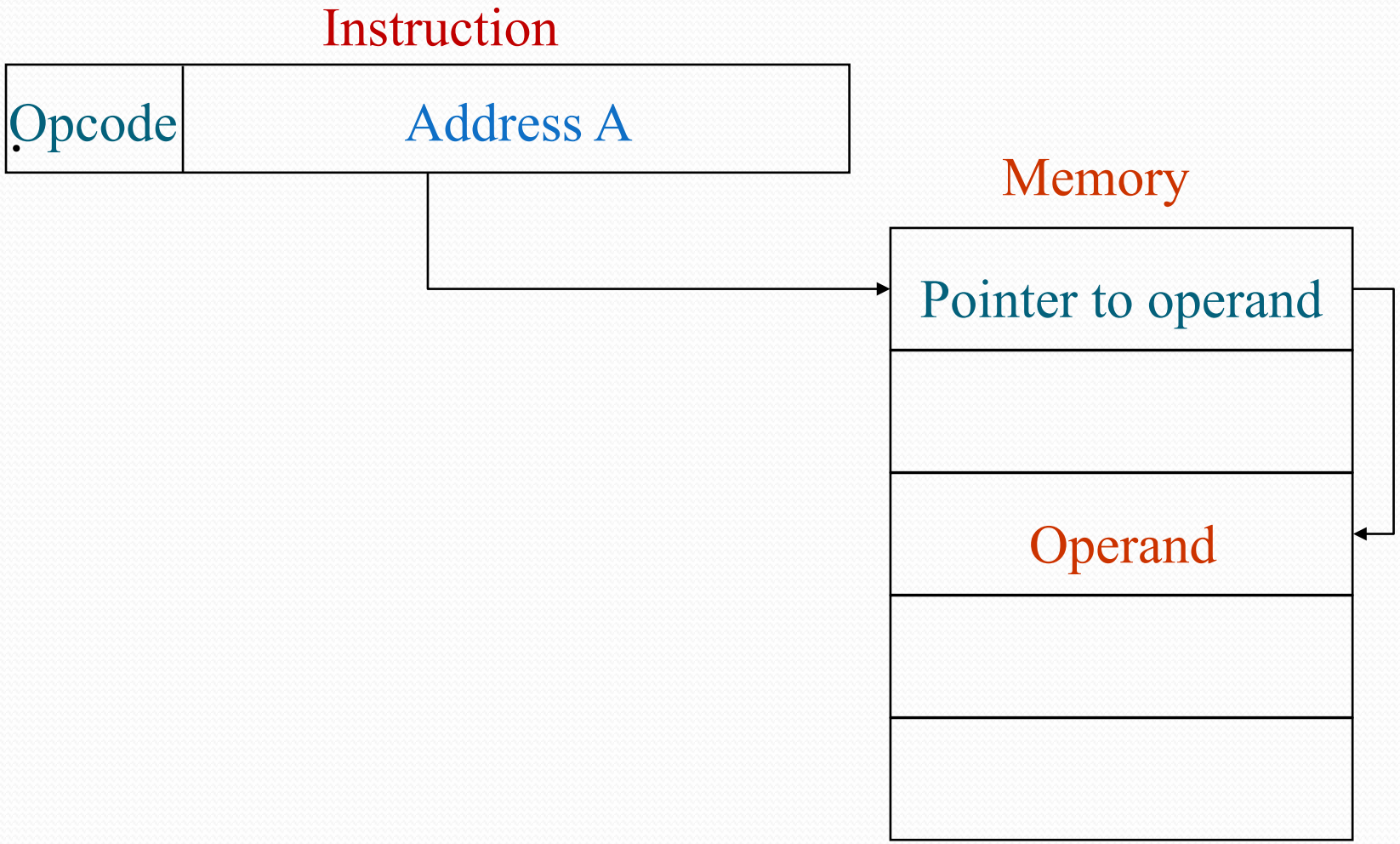
Direct Addressing Diagram



Indirect Addressing

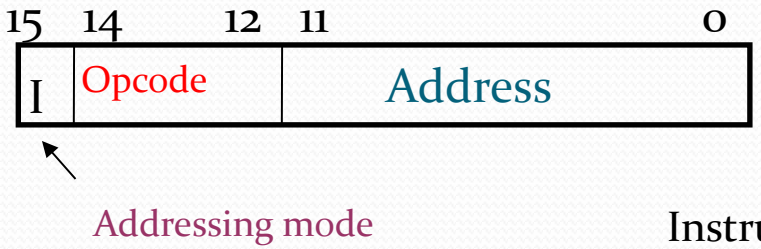
- Memory cell pointed to by address field contains the address of (pointer to) the operand
- EA = (A)
  - Look in A, find address (A) and look there for operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator

Indirect Addressing Diagram



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

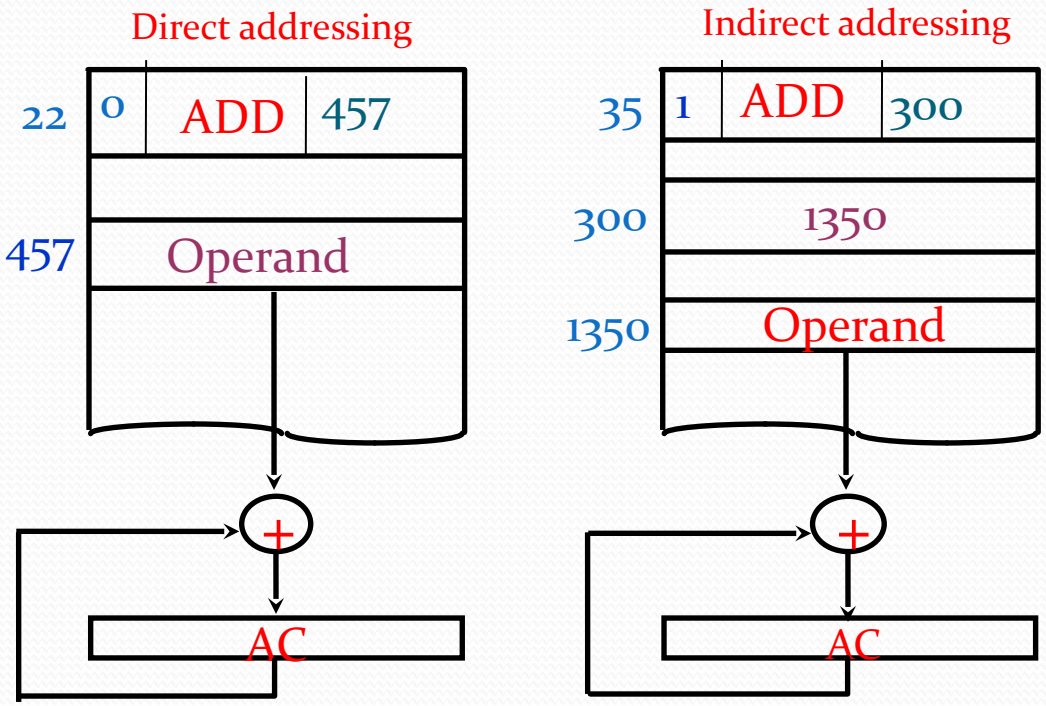


Instruction Format

- 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.

Addressing Modes

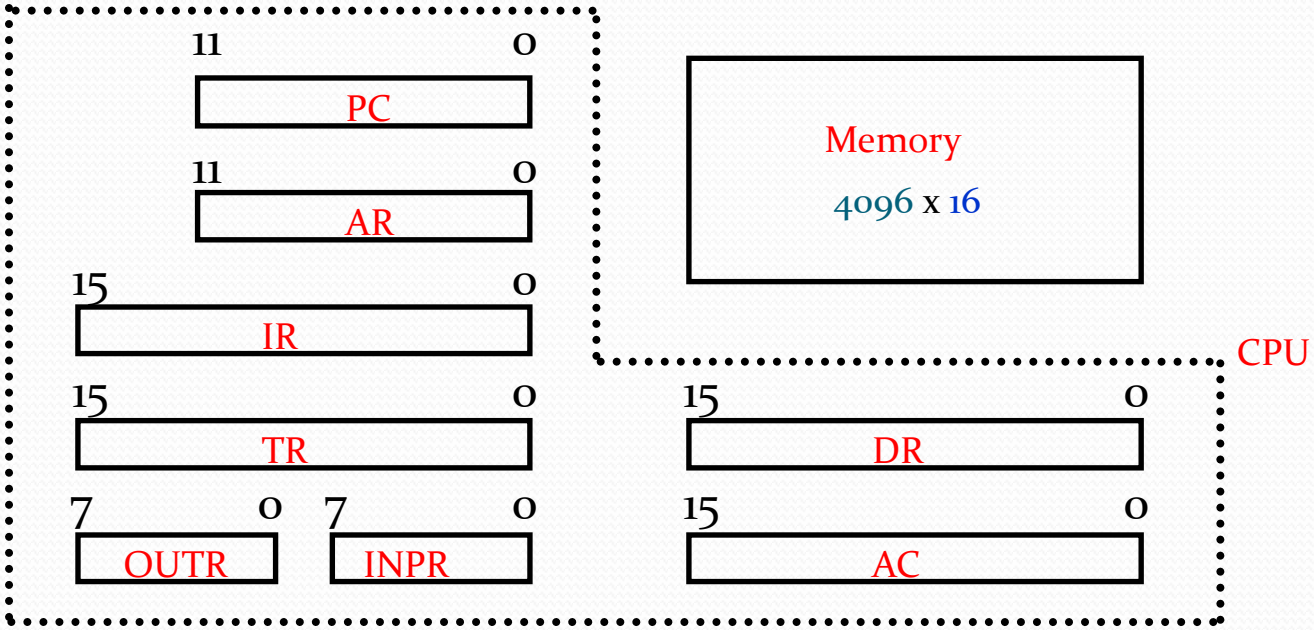
- The address field of an instruction can represent either
  - Direct address: the address in memory of the data to use (the address of the operand), or
  - Indirect address: the address in memory of the address in memory of the data to use



- 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

9. Basic Computer Registers

- Registers in the Basic Computer



Basic Computer Registers

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.

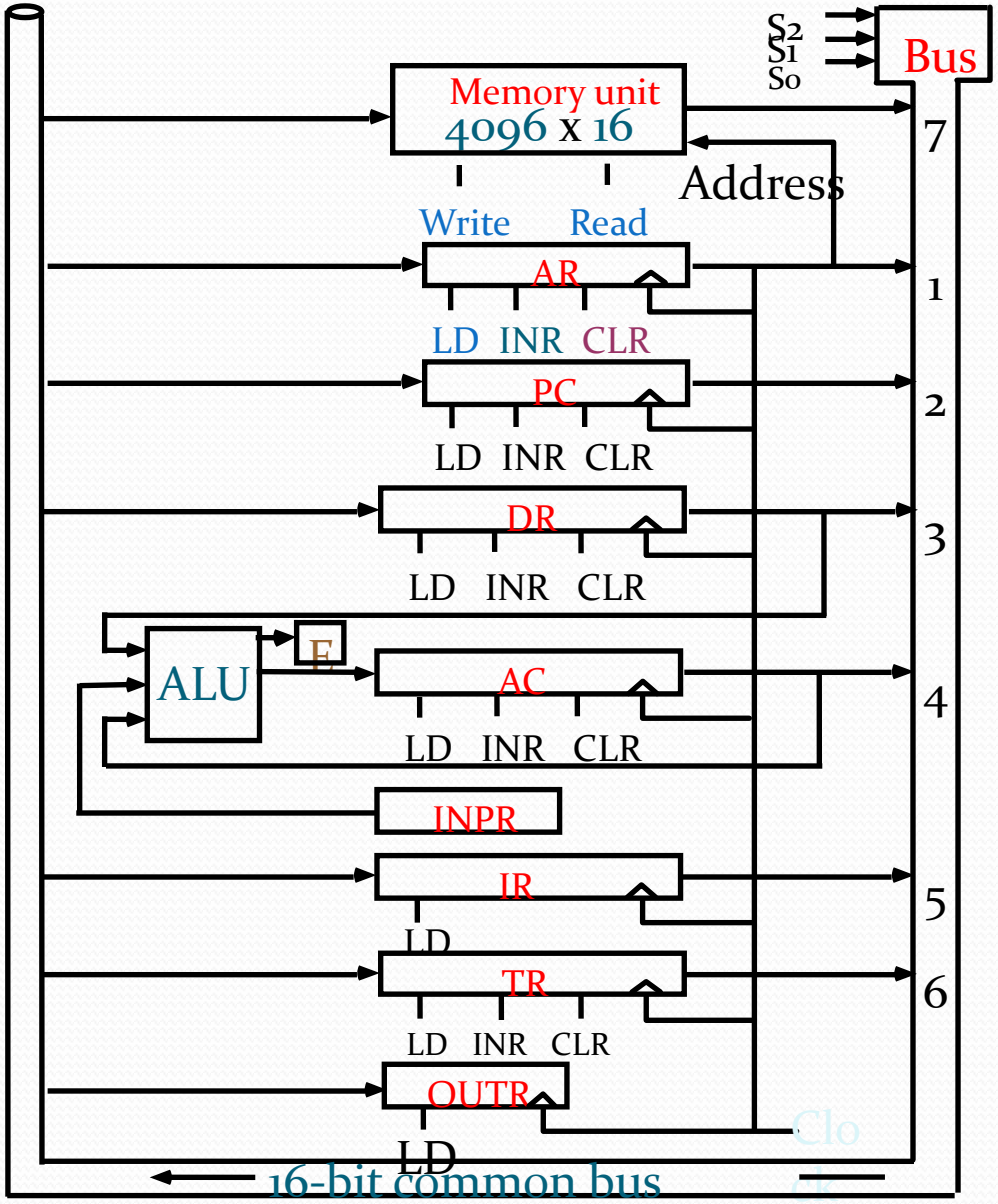
Processor Registers

- 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 got from an input device.
- The *Output Register* (OUTR) holds an 8 bit character to be sent to an output device.

Common Bus System



Common Bus System

- Three control lines, S_2, S_1, and S_0 control the specific output that is selected for the bus lines at any given time.

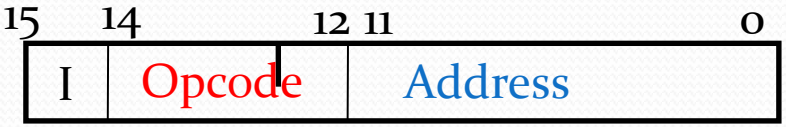
	S_2 S_1 S_0	Register
0	0 0	x
0	0 1	AR
0	1 0	PC
0	1 1	DR
1	0 0	AC
1	0 1	IR
1	1 0	TR
1	1 1	Memory

- 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.

10. Basic Computer Instructions

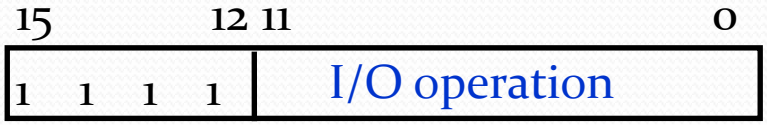
- Basic Computer Instruction Format

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

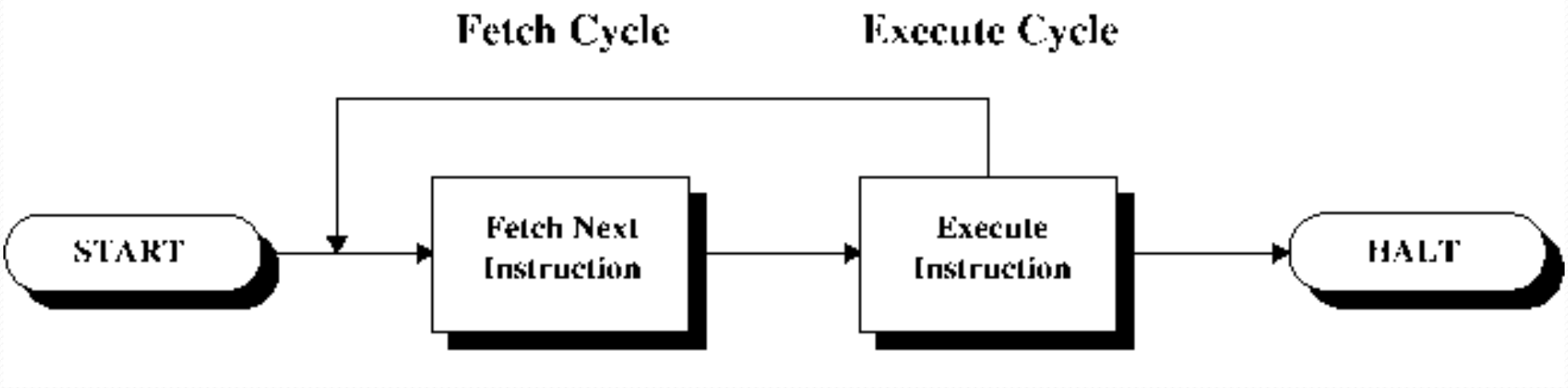
Transfer Instructions
- Data transfers between the main memory and the processor registers
- LDA, STA

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

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

11. 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

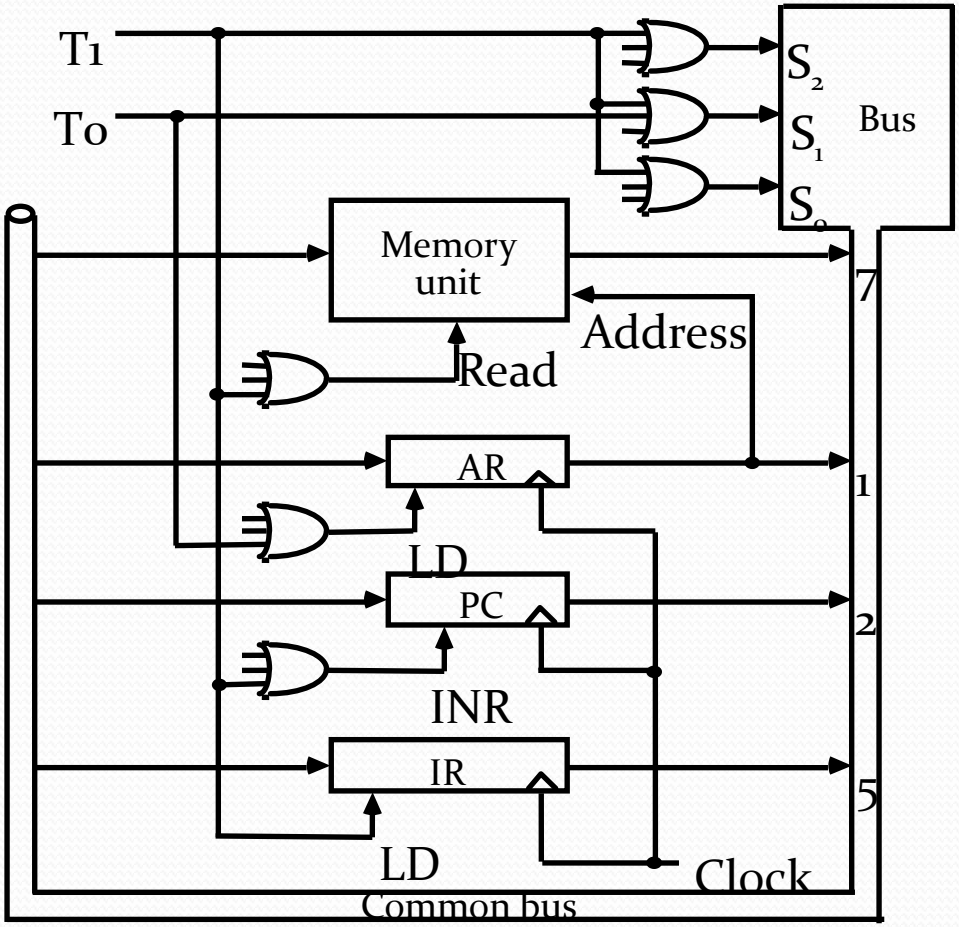


- After an instruction is executed, the cycle starts again at step 1, for the next instruction.
- *Note*: Every different processor has its own (different) instruction cycle .

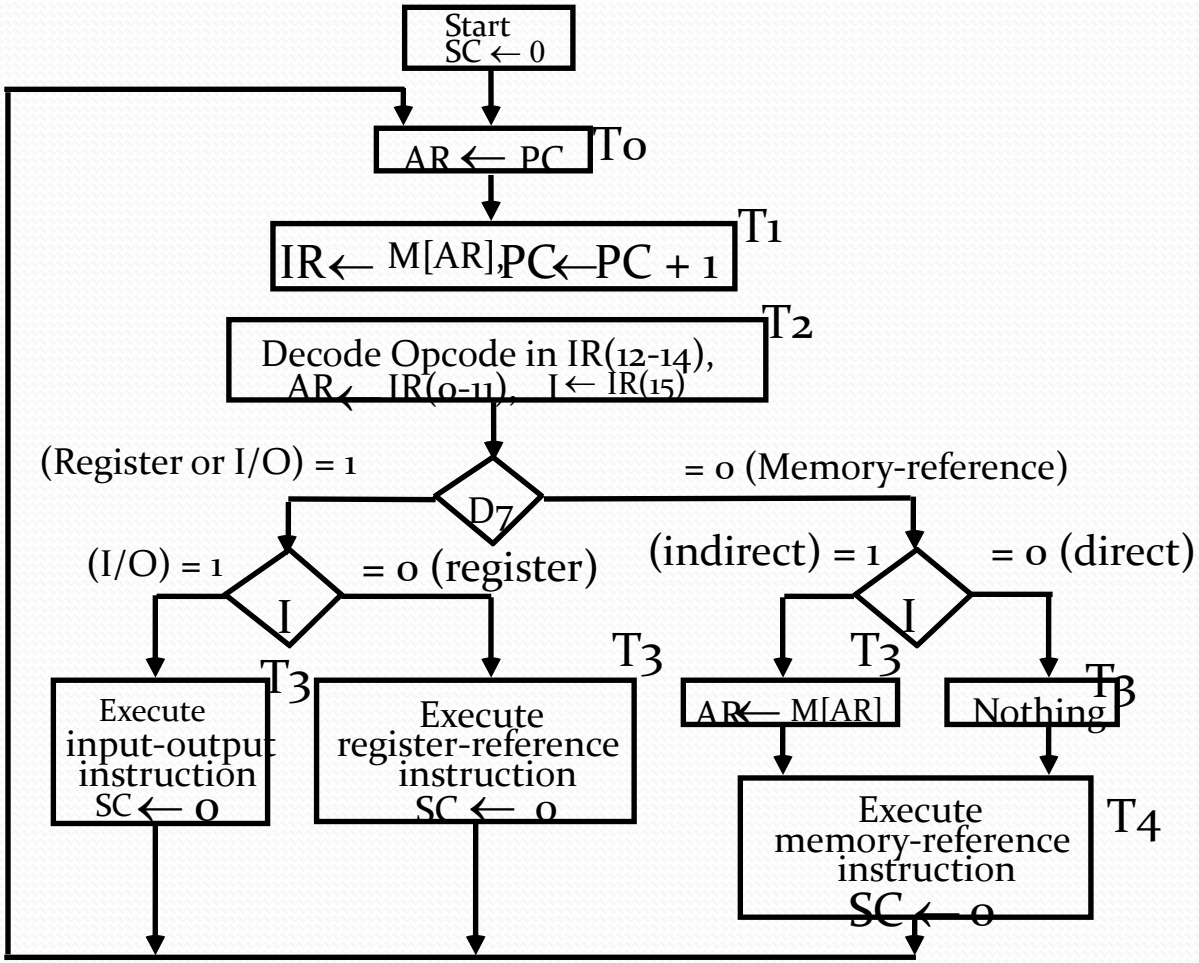
Fetch and Decode

- Fetch and Decode

To: AR ← PC ($S_0S_1S_2$=010, To=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)

12. 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

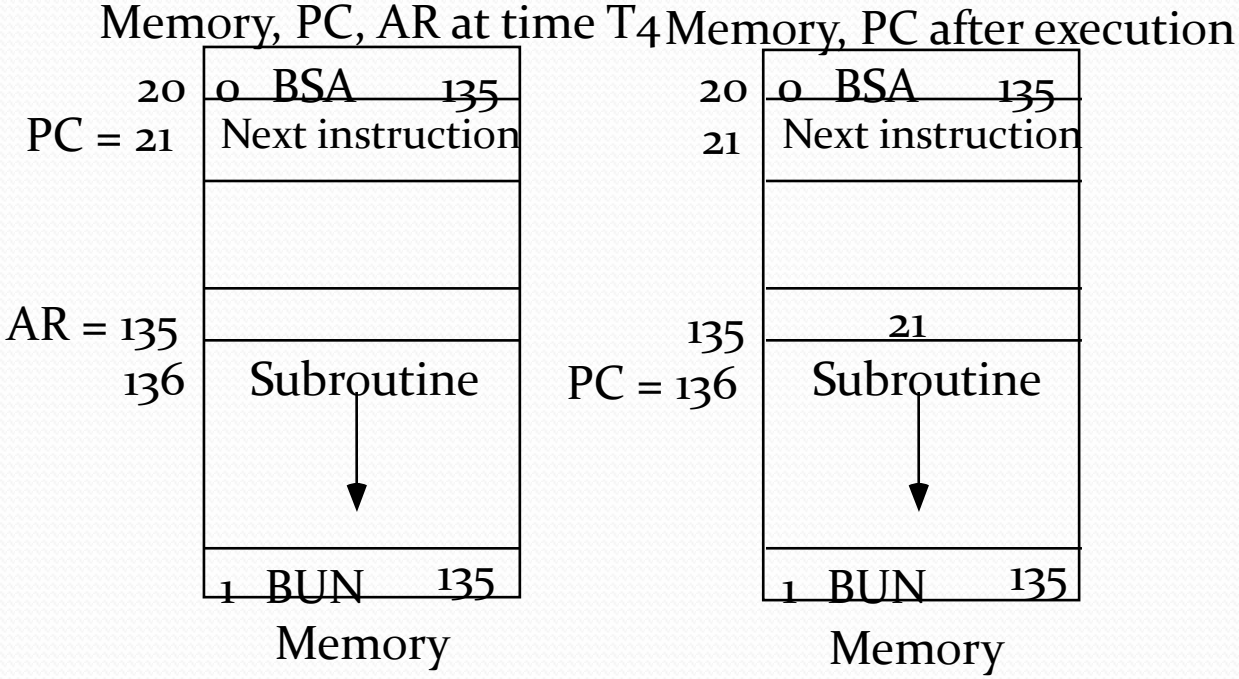
D_3T_4: M[AR] ← AC, SC ← 0

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

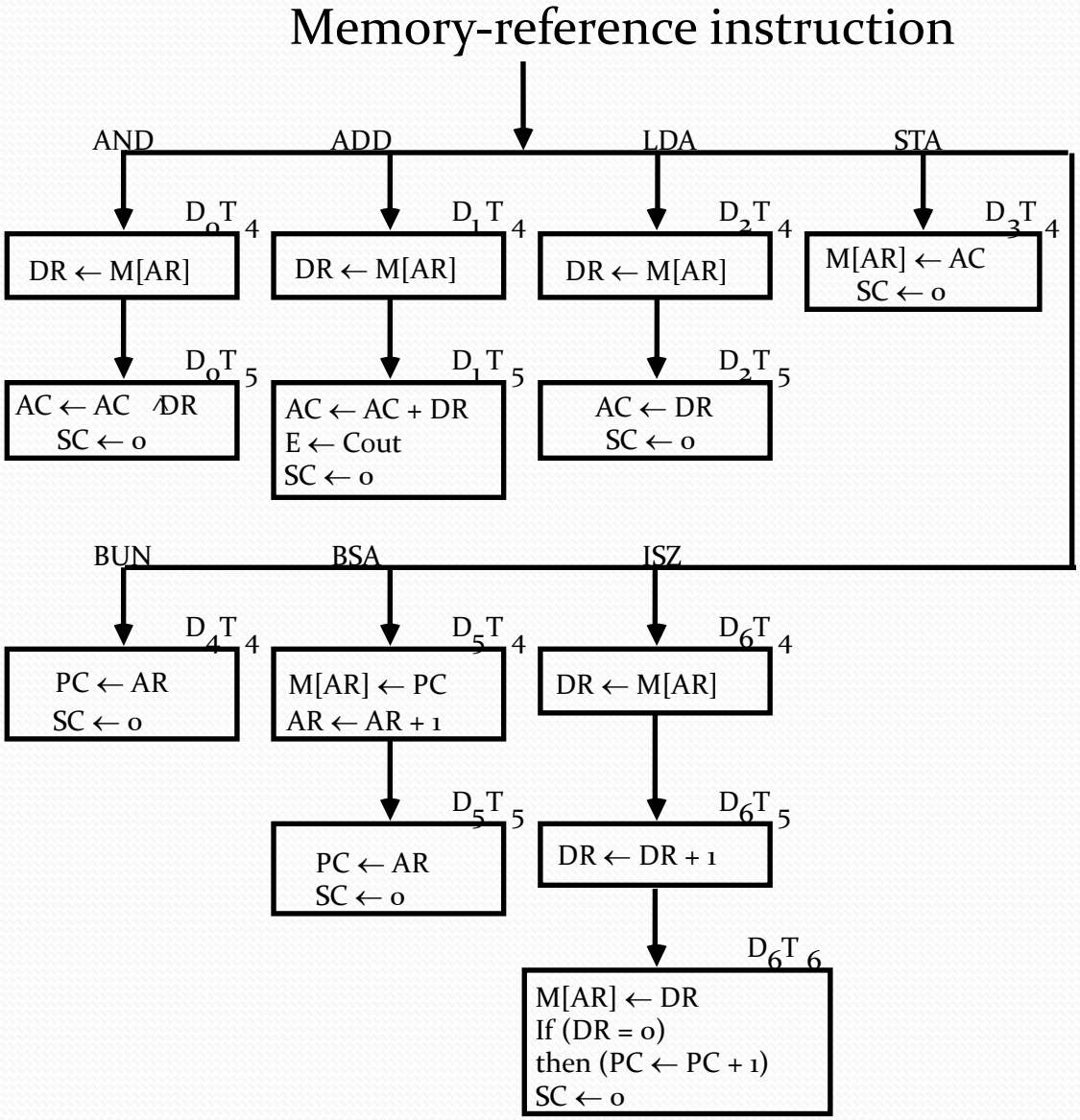
ISZ: Increment and Skip-if-Zero

D_6T_4: $DR \leftarrow M[AR]$

D_6T_5: $DR \leftarrow DR + 1$

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

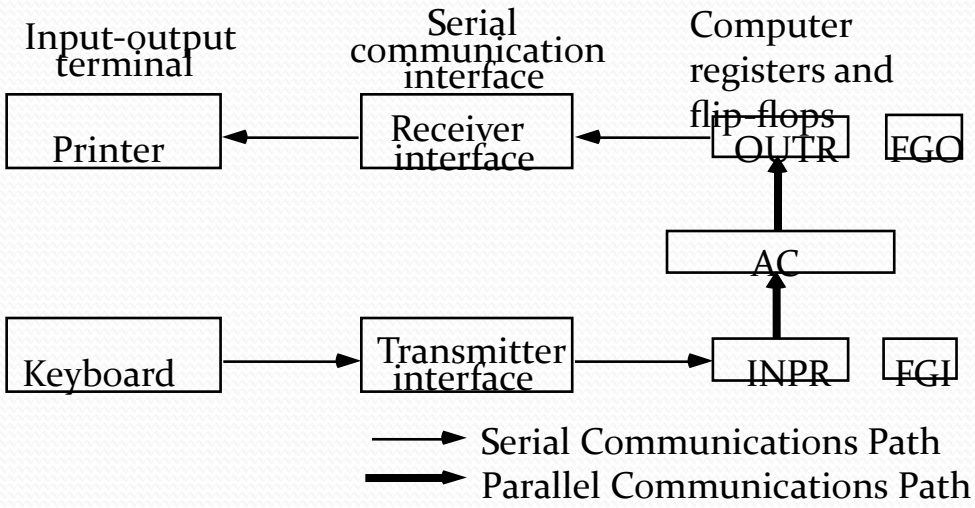
Flowchart For Memory Reference Instructions



13. 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

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

Interrupt Initiated Input/Output

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

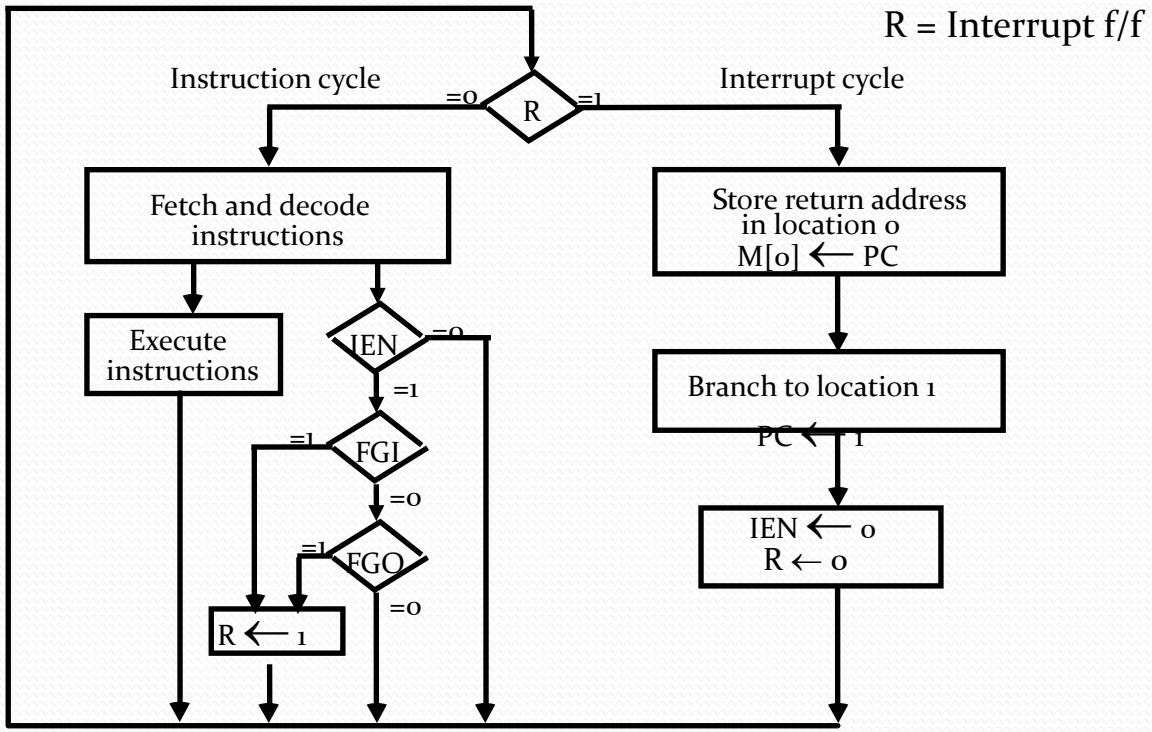
-- The I/O interface, instead of the CPU, monitors the I/O device.

- 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



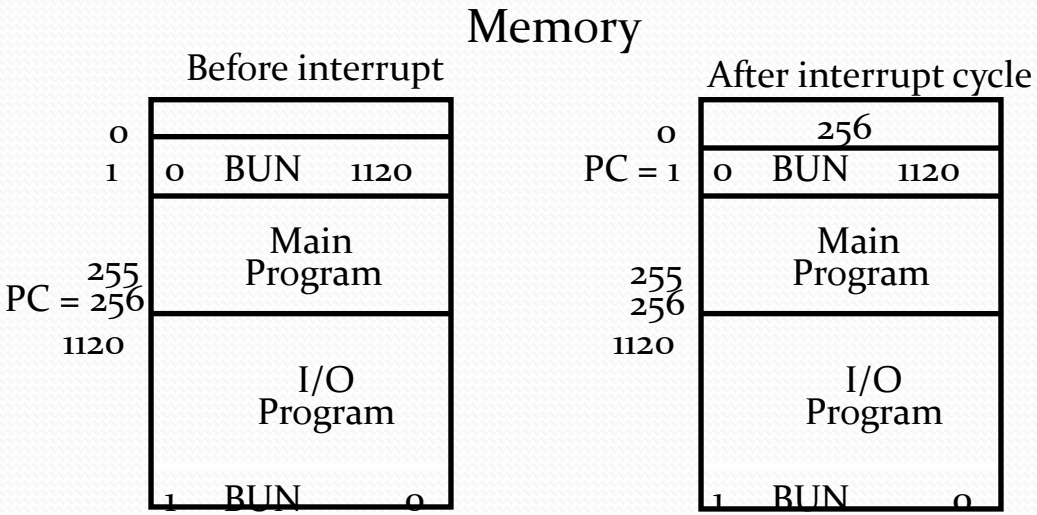
- The interrupt cycle is a HW implementation of a branch and save return address operation.

- 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 $\leftarrow 1$ if IEN (FGI + FGO)$T_0'T_1'T_2'$

$\Leftrightarrow T_0'T_1'T_2'$ (IEN)(FGI + FGO): $R \leftarrow 1$

- 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: $AR \leftarrow 0$, $TR \leftarrow PC$

RT_1: $M[AR] \leftarrow TR$, $PC \leftarrow 0$

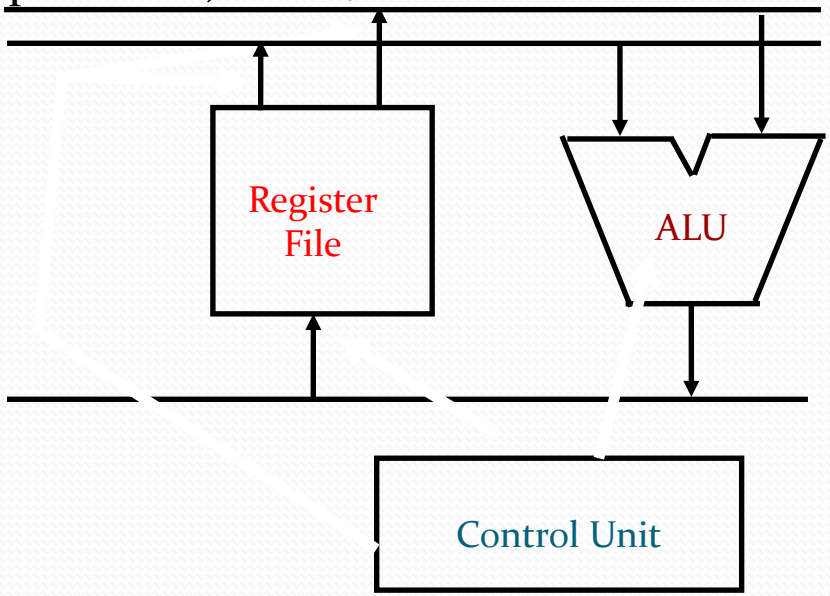
RT_2: $PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$

Central Processing Unit

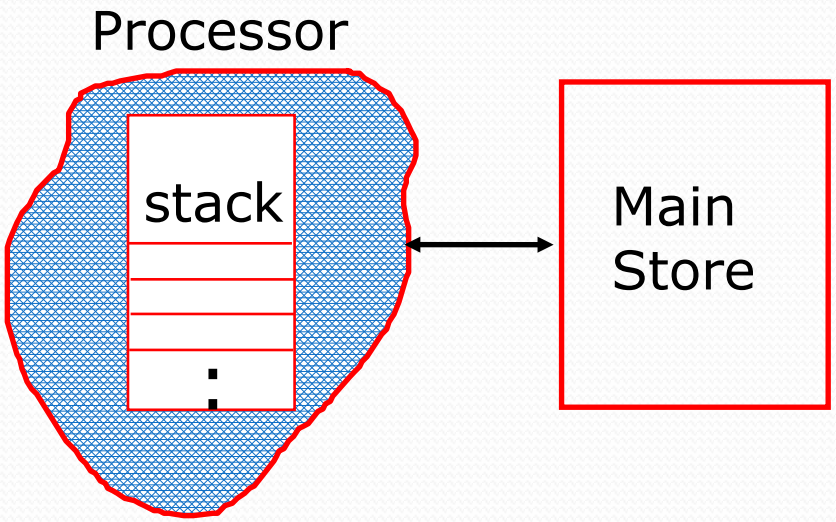
- Stack Organization
- Instruction Formats
- Addressing Modes
- Data Transfer and Manipulation
- Program Control
- Reduced Instruction Set Computer

Major Components of CPU

- Storage Components
  - Registers
  - Flags
- Execution (Processing) Components
  - Arithmetic Logic Unit(ALU)
  - Arithmetic calculations, Logical computations, Shifts/Rotates
- Transfer Components
  - Bus
- Control Components
  - Control Unit



14. A Stack Machine

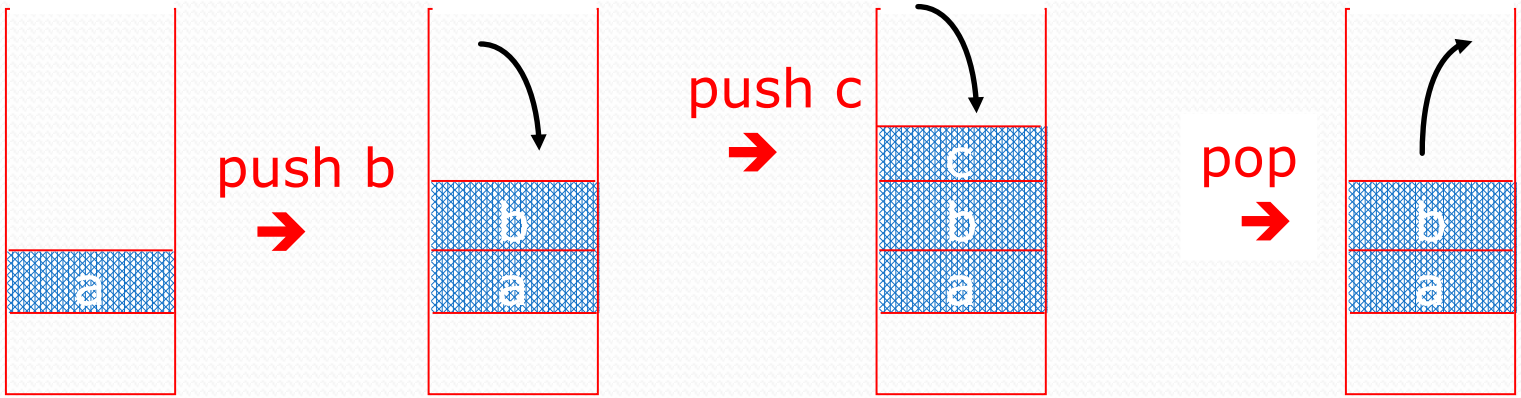


A Stack machine has a stack as a part of the processor state

typical operations:

push, pop, +, *, ...

Instructions like + implicitly specify the top 2 elements of the stack as operands.

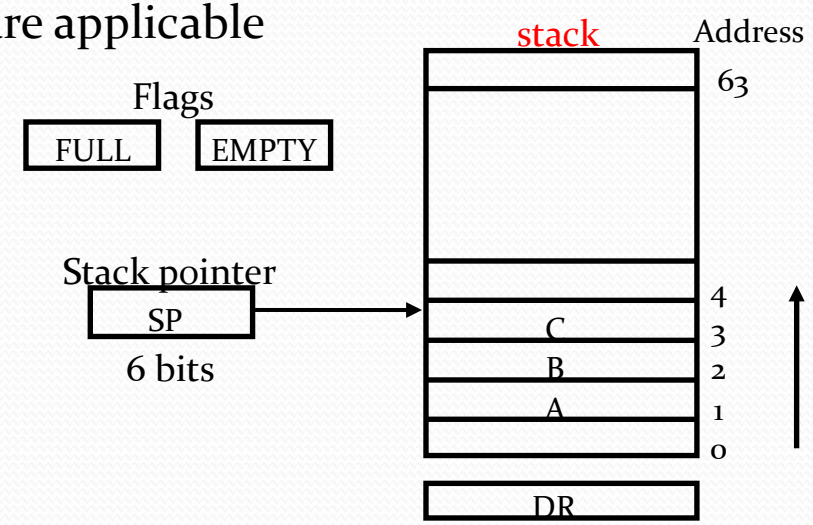


Register Stack Organization

Stack

- Very useful feature for nested subroutines, nested interrupt services
- Also efficient for arithmetic expression evaluation
- Storage which can be accessed in LIFO
- Pointer: SP
- Only PUSH and POP operations are applicable

Register Stack



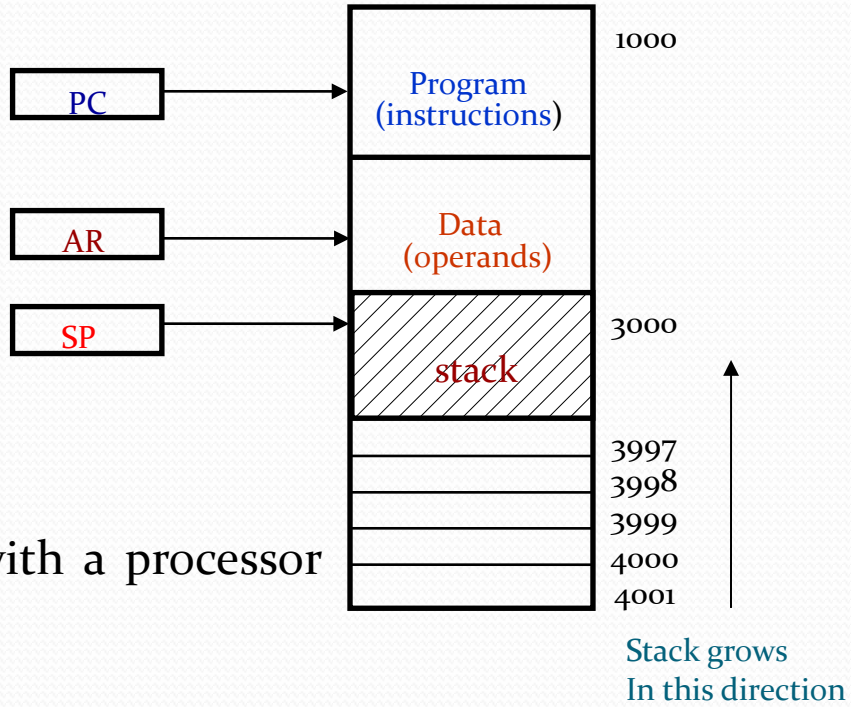
Push, Pop operations

/* Initially, SP = 0, EMPTY = 1, FULL = 0 */

PUSH	POP
SP ← SP + 1	DR ← M[SP]
M[SP] ← DR	SP ← SP – 1
If (SP = 0) then (FULL ← 1)	If (SP = 0) then (EMPTY ← 1)
EMPTY ← 0	FULL ← 0

Memory Stack Organization

Memory with Program, Data, and Stack Segments



- A portion of memory is used as a stack with a processor register as a stack pointer.

- PUSH: $SP \leftarrow SP - 1$

  $M[SP] \leftarrow DR$

- POP: $DR \leftarrow M[SP]$

  $SP \leftarrow SP + 1$

- Most computers do not provide hardware to check stack overflow (full stack) or underflow (empty stack) → must be done in software

Reverse Polish Notation

- Arithmetic Expressions: A + B

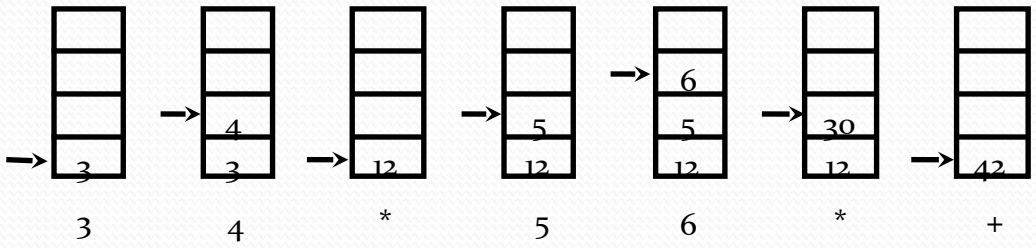
A + B	Infix notation
+ A B	Prefix or Polish notation
A B +	Postfix or reverse Polish notation

- The reverse Polish notation is very suitable for stack manipulation

- Evaluation of Arithmetic Expressions

Any arithmetic expression can be expressed in parenthesis-free Polish notation, including reverse Polish notation

$(3 * 4) + (5 * 6) \Rightarrow 3\ 4\ *\ 5\ 6\ *\ +$



Processor Organization

- In general, most processors are organized in one of 3 ways
  - Single register (Accumulator) organization
    - Basic Computer is a good example.
    - Accumulator is the only general purpose register.
  - General register organization
    - Used by most modern computer processors.
    - Any of the registers can be used as the source or destination for computer operations.
  - Stack organization
    - All operations are done using the hardware stack.
    - For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack.

15. Instruction Format

Instruction Fields

OP-code field - specifies the operation to be performed

Address field - designates memory address(es) or a processor register(s)

Mode field - determines how the address field is to be interpreted (to get effective address or the operand)

- The number of address fields in the instruction format depends on the internal organization of CPU.
- The three most common CPU organizations:

Single accumulator organization:

```
        ADD     X                   /* AC ← AC + M[X]  */
```

General register organization:

```
        ADD     R1, R2, R3      /* R1 ← R2 + R3  */
ADD     R1, R2                  /* R1 ← R1 + R2  */
        MOV     R1, R2                  /* R1 ← R2  */
ADD     R1, X                   /* R1 ← R1 + M[X]  */
```

Stack organization:

```
        PUSH    X                   /* TOS ← M[X]  */
ADD
```

Three, and Two-Address Instructions

- Three-Address Instructions

Program to evaluate X = (A + B) * (C + D) :

```
ADD     R1, A, B     /*  R1 ← M[A] + M[B]      */
ADD     R2, C, D     /*  R2 ← M[C] + M[D]      */
MUL     X, R1, R2    /*  M[X] ← R1 * R2                */
```

- Results in short programs
- Instruction becomes long (many bits)

- Two-Address Instructions

Program to evaluate X = (A + B) * (C + D) :

```
MOV    R1, A          /* R1 ← M[A]          */
ADD    R1, B          /* R1 ← R1 + M[A]  */
MOV    R2, C          /* R2 ← M[C]          */
ADD    R2, D          /* R2 ← R2 + M[D]  */
MUL    R1, R2         /* R1 ← R1 * R2       */
MOV    X, R1          /* M[X] ← R1          */
```

One and Zero-Address Instructions

- One-Address Instructions

  - Use an implied AC register for all data manipulation
  - Program to evaluate X = (A + B) * (C + D) :

```
LOAD    A       /*  AC ← M[A]          */
ADD     B       /*  AC ← AC + M[B]     */
STORE   T       /*  M[T] ← AC          */
LOAD    C       /*  AC ← M[C]          */
ADD     D       /*  AC ← AC + M[D]     */
MUL     T       /*  AC ← AC * M[T]     */
STORE   X       /*  M[X] ← AC          */
```

- Zero-Address Instructions

  - Can be found in a stack-organized computer
  - Program to evaluate X = (A + B) * (C + D) :

```
PUSH    A       /*  TOS ← A            */
PUSH    B       /*  TOS ← B            */
ADD             /*  TOS ← (A + B)      */
PUSH    C       /*  TOS ← C            */
PUSH    D       /*  TOS ← D            */
ADD             /*  TOS ← (C + D)      */
MUL             /*  TOS ← (C + D) * (A + B)  */
POP     X       /*  M[X] ← TOS         */
```

16. Addressing Modes

- Addressing Modes

* Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)

* Variety of addressing modes

  - to give programming flexibility to the user

  - to use the bits in the address field of the instruction efficiently

Types of Addressing Modes

- Implied
- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Autoincrement or AutoDecrement
- Relative
- Indexed
- Base Register

Implied Addressing Mode

- Implied Mode

  Address of the operands are specified implicitly in the definition of the instruction

  - No need to specify address in the instruction

  - EA = AC, or EA = Stack[SP]

  - Examples from Basic Computer

    CLA, CME, INP

Immediate Addressing

- Immediate Mode

  Instead of specifying the address of the operand, operand itself is specified.

  - No need to specify address in the instruction
  - However, operand itself needs to be specified
  - Sometimes, require more bits than the address
  - Fast to acquire an operand

- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand

Instruction

Direct Addressing

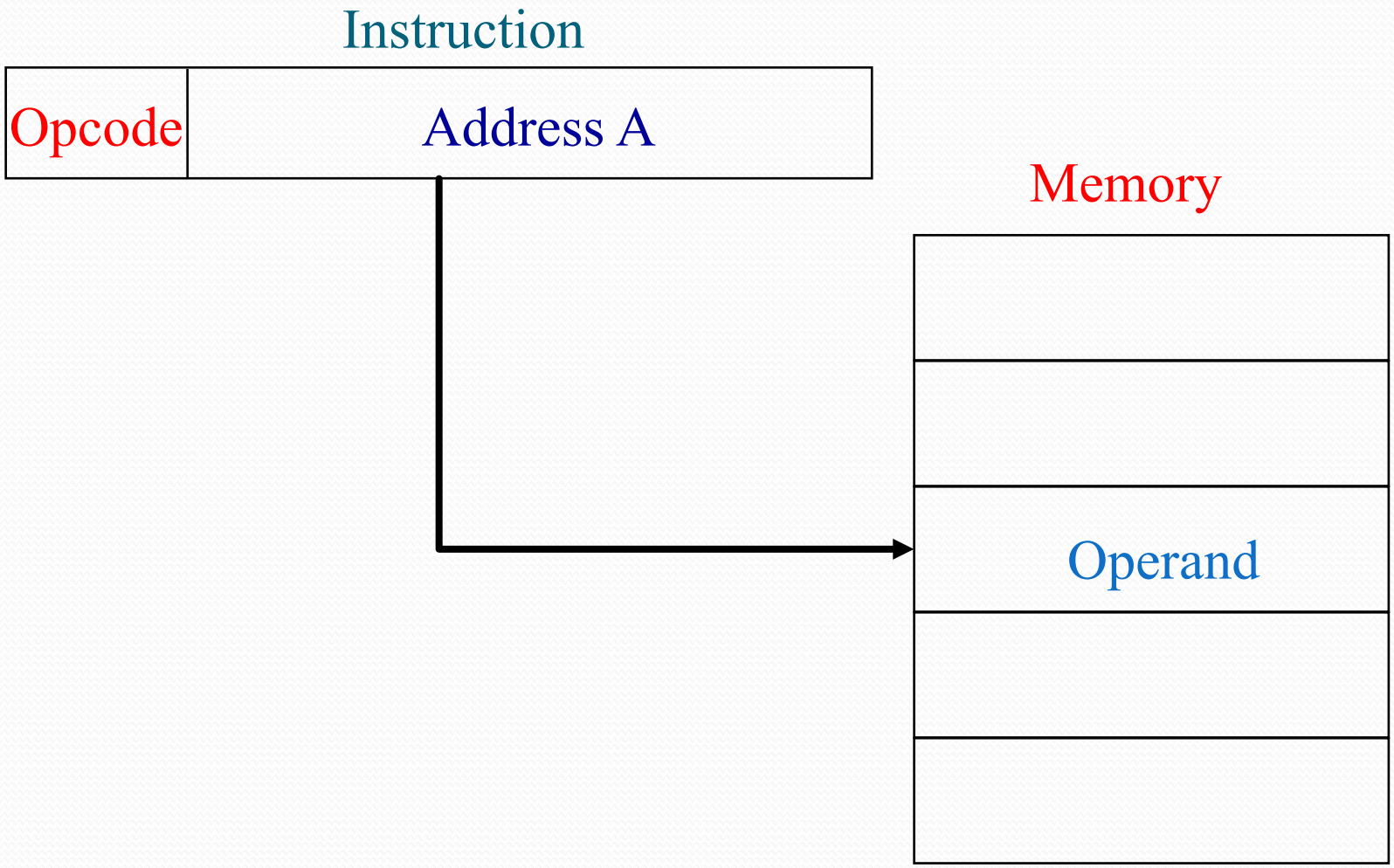
- Direct Address Mode

  Instruction specifies the memory address which can be used directly to access the memory

  - Faster than the other memory addressing modes
  - Too many bits are needed to specify the address for a large physical memory space
  - EA = IR(addr) (IR(addr): address field of IR)

- e.g. ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand

Direct Addressing Diagram



Indirect Addressing

- Indirect Addressing Mode

  The address field of an instruction specifies the address of a memory location that contains the address of the operand.

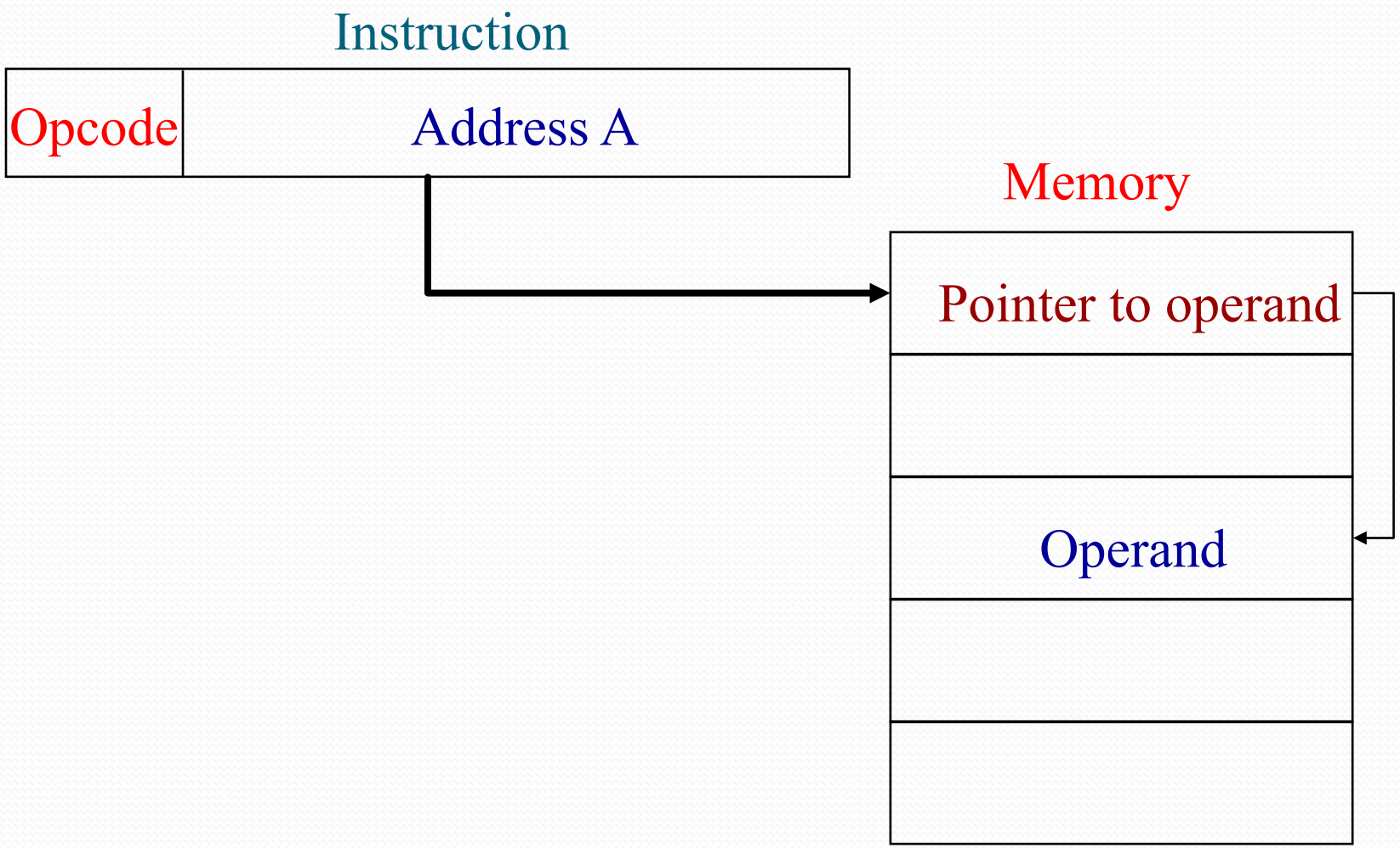
  - When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits.

  - Slow to acquire an operand because of an additional memory access.

  - EA = M[IR(address)]

- EA = (A)
  - Look in A, find address (A) and look there for operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator

Indirect Addressing Diagram



Register Addressing

- Register Mode

  Address specified in the instruction is the register address

  - Designated operand need to be in a register

  - Shorter address than the memory address

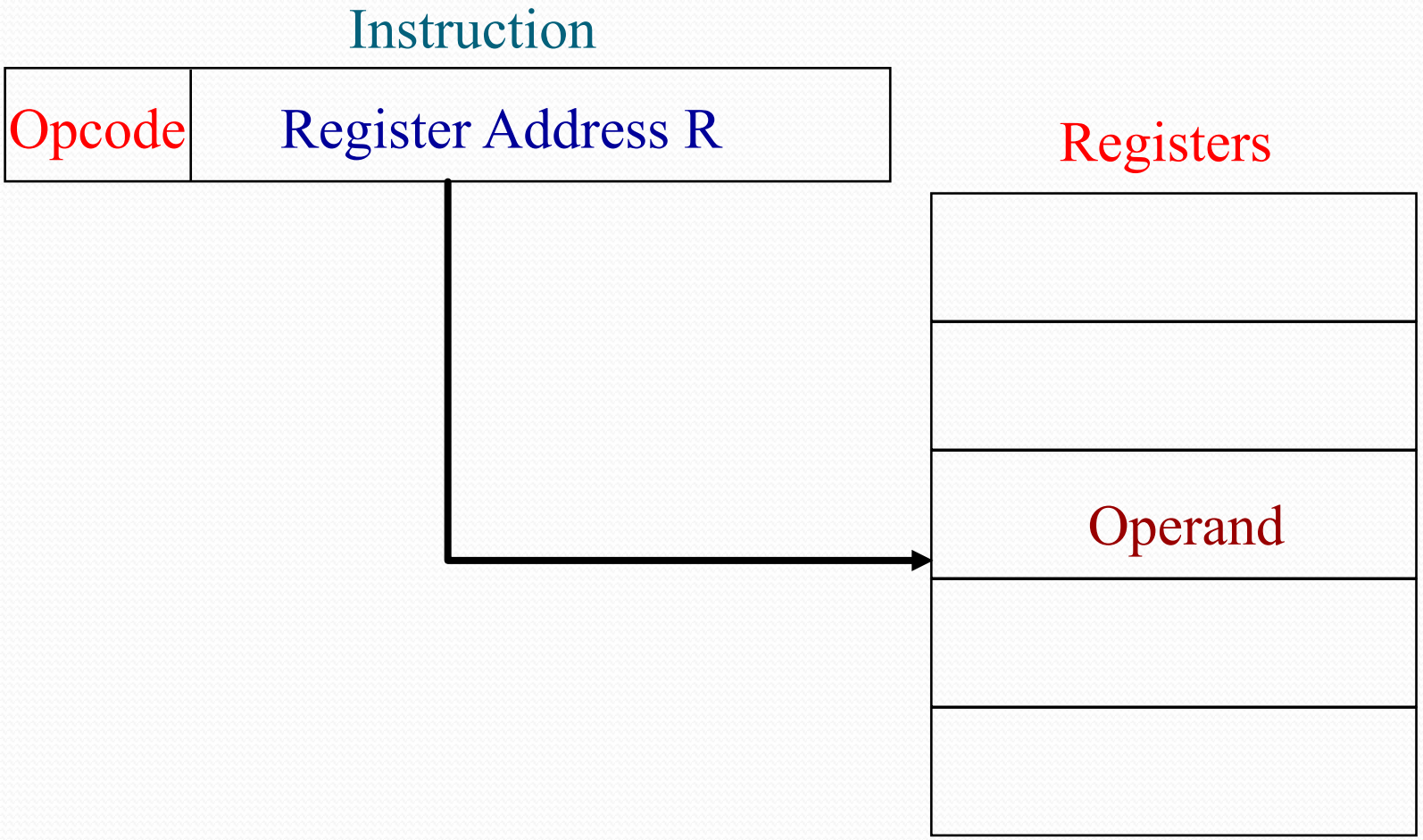
  - Saving address field in the instruction

  - Faster to acquire an operand than the memory addressing

  - EA = IR(R) (IR(R): Register field of IR)

- EA = R

Register Addressing Diagram



Register Indirect Addressing

- Register Indirect Mode

  Instruction specifies a register which contains the memory address of the operand.

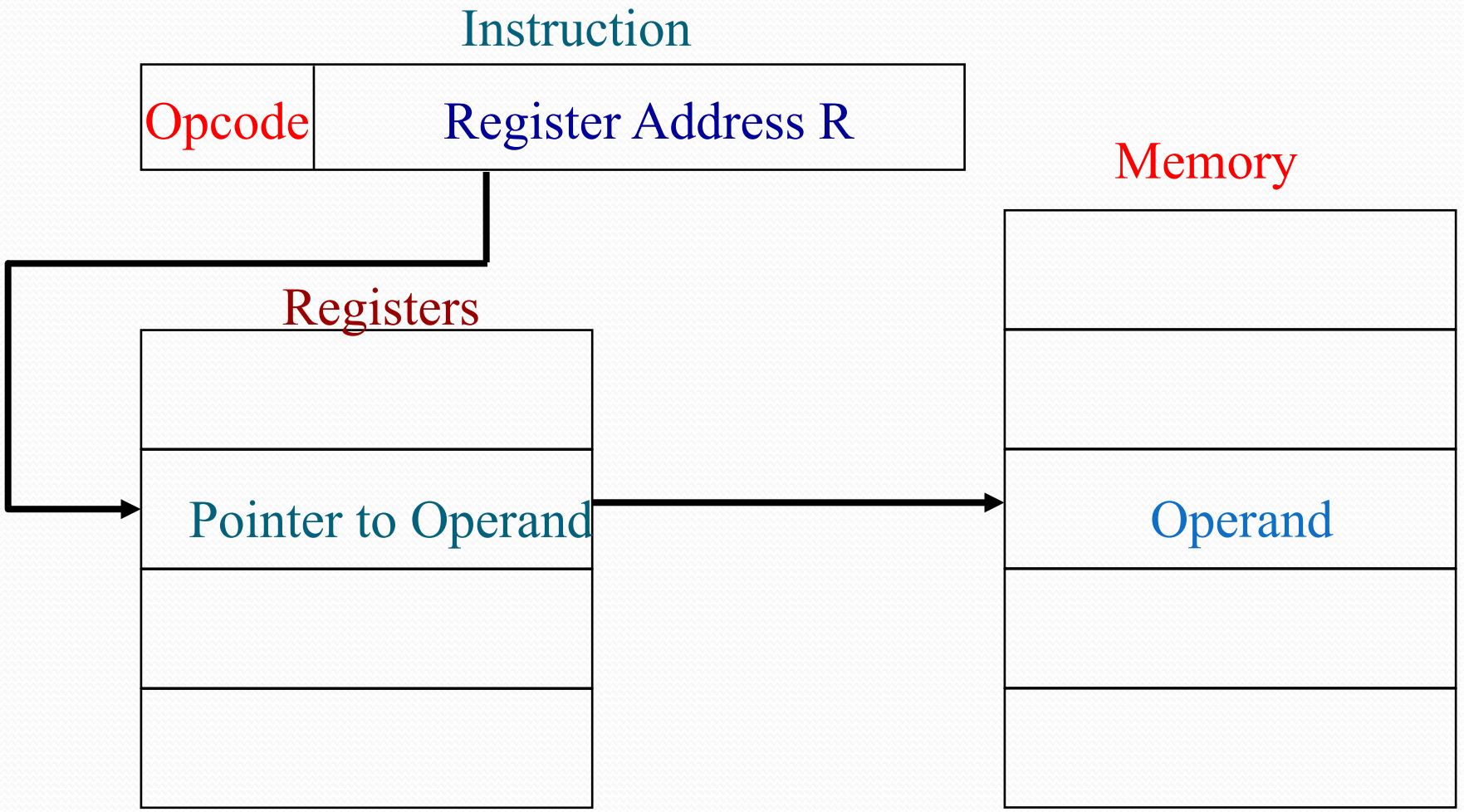
  - Saving instruction bits since register address is shorter than the memory address.

  - Slower to acquire an operand than both the register addressing or memory addressing.

  - EA = [IR(R)] ([x]: Content of x)

- EA = (R)

Register Indirect Addressing Diagram



Types Of Addressing Modes

- Relative Addressing Modes

  - Address field of the instruction is short.

  - Large physical memory can be accessed with a small number of address bits.

  - EA = f(IR(address), R), R is sometimes implied.

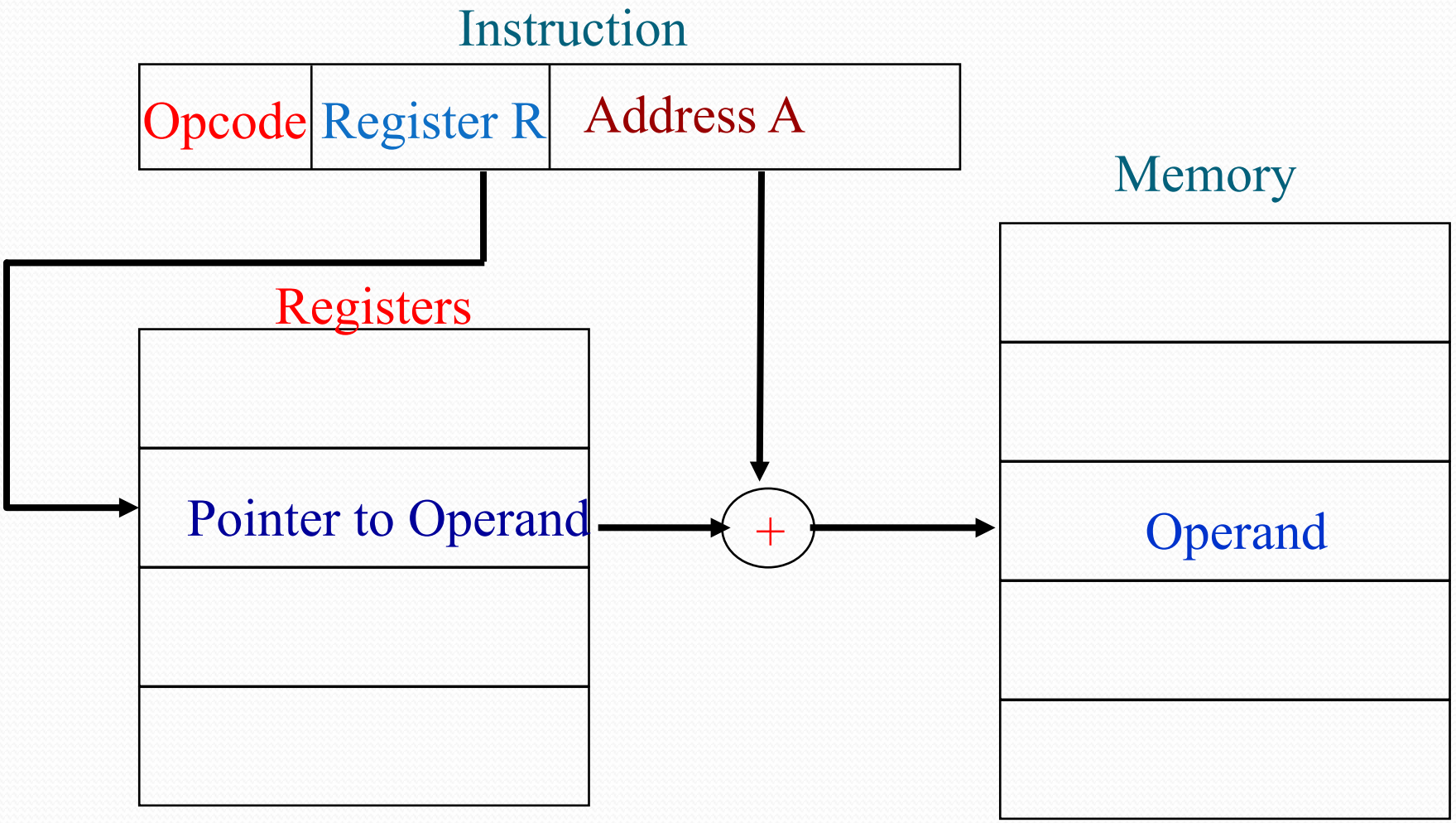
  3 different Relative Addressing Modes depending on R;

  * PC Relative Addressing Mode (R = PC)
    - EA = PC + IR(address)
  * Indexed Addressing Mode (R = IX, where IX: Index Register)
    - EA = IX + IR(address)
  * Base Register Addressing Mode
    (R = BAR, where BAR: Base Address Register)
    - EA = BAR + IR(address)

- Autoincrement or Autodecrement Mode

  - When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically.

Index Addressing Diagram



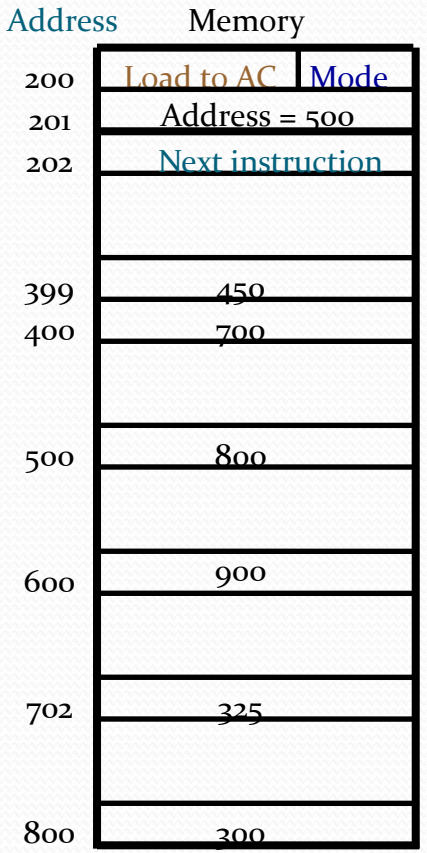
Addressing Modes - Examples

PC = 200

R1 = 400

XR = 100

AC



Addressing Mode	Effective Address		Content of AC
Direct address	500	/* AC ← (500) */	800
Immediate operand	-	/* AC ← 500 */	500
Indirect address	800	/* AC ← ((500)) */	300
Relative address	702	/* AC ← (PC+500) */	325
Indexed address	600	/* AC ← (RX+500) */	900
Register	-	/* AC ← R1 */	400
Register indirect	400	/* AC ← (R1) */	700
Autoincrement	400	/* AC ← (R1)+ */	700
Autodecrement	399	/* AC ← -(R) */	450

17. Data Transfer Instructions and Manipulation

- Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

- Data Transfer Instructions with Different Addressing Modes

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ADR	AC ← M[PC + ADR]
Immediate operand	LD #NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[ADR + XR]
Register	LD R1	AC ← R1
Register indirect	LD (R1)	AC ← M[R1]
Autoincrement	LD (R1)+	AC ← M[R1], R1 ← R1 + 1
Autodecrement	LD -(R1)	R1 ← R1 - 1, AC ← M[R1]

Data Manipulation Instructions

- Three Basic Types: Arithmetic instructions
  Logical and bit manipulation instructions
  Shift instructions

- Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Borrow	SUBB
Negate(2's Complement)	NEG

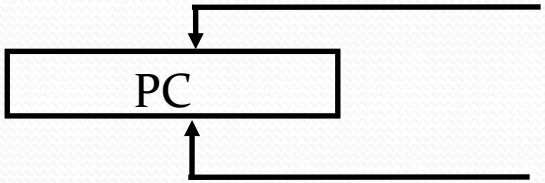
- Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

- Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right thru carry	RORC
Rotate left thru carry	ROLC

18. Program Control Instructions



+1

In-Line Sequencing (Next instruction is fetched from the next adjacent location in the memory)

Address from other source; Current Instruction, Stack, etc; Branch, Conditional Branch, Subroutine, etc

- Program Control Instructions

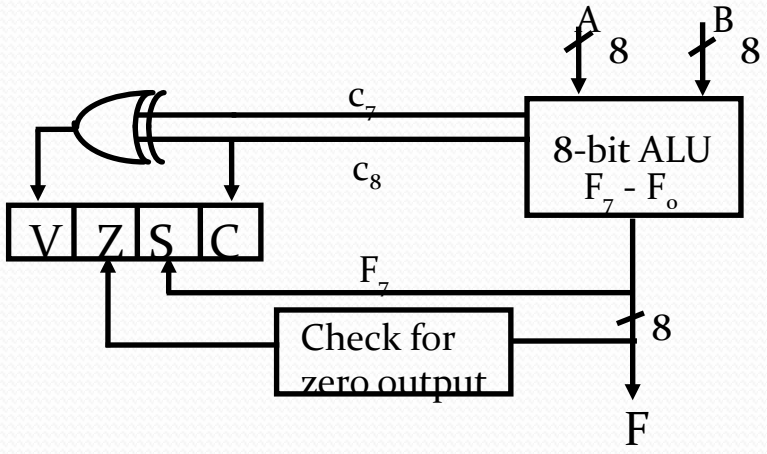
Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RTN
Compare(by –)	CMP
Test(by AND)	TST

* CMP and TST instructions do not retain their results of operations (– and AND, respectively). They only set or clear certain Flags.

Flag, Processor Status Word

- In Basic Computer, the processor had several (status) flags – 1 bit value that indicate various information about the processor's state – E, FGI, FGO, I, IEN, R.
- In some processors, flags like these are often combined into a register – the processor status register (PSR); sometimes called a processor status word (PSW).
- Common flags in PSW are
  - C (Carry): Set to 1 if the carry out of the ALU is 1
  - S (Sign): The MSB bit of the ALU's output
  - Z (Zero): Set to 1 if the ALU's output is all 0's
  - V (Overflow): Set to 1 if there is an overflow

Status Flag Circuit



Conditional Branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned compare conditions (A - B)	
BHI	Branch if higher	A > B
BHE	Branch if higher or equal	A ≥ B
BLO	Branch if lower	A < B
BLOE	Branch if lower or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B
	Signed compare conditions (A - B)	
BGT	Branch if greater than	A > B
BGE	Branch if greater or equal	A ≥ B
BLT	Branch if less than	A < B
BLE	Branch if less or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B

Subroutine Call And Return

- Subroutine Call

  Call subroutine
  Jump to subroutine
  Branch to subroutine
  Branch and save return address

- Two Most Important Operations are Implied;

  * Branch to the beginning of the Subroutine
    - Same as the Branch or Conditional Branch

  * Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine

- Locations for storing Return Address
  - Fixed Location in the subroutine (Memory)
  - Fixed Location in memory
  - In a processor Register
  - In memory *stack*
    - most efficient way

CALL

$SP \leftarrow SP - 1$
$M[SP] \leftarrow PC$
$PC \leftarrow EA$

RTN

$PC \leftarrow M[SP]$
$SP \leftarrow SP + 1$

Program Interrupt

Types of Interrupts

External interrupts

External Interrupts initiated from the outside of CPU and Memory
- I/O Device → Data transfer request or Data transfer complete
- Timing Device → Timeout
- Power Failure
- Operator

Internal interrupts (traps)

Internal Interrupts are caused by the currently running program
- Register, Stack Overflow
- Divide by zero
- OP-code Violation
- Protection Violation

Software Interrupts

Both External and Internal Interrupts are initiated by the computer HW.
Software Interrupts are initiated by the executing an instruction.
- Supervisor Call → Switching from a user mode to the supervisor mode
  → Allows to execute a certain class of operations which are not allowed in the user mode

Interrupt Procedure

Interrupt Procedure and Subroutine Call

- The interrupt is usually initiated by an internal or an external signal rather than from the execution of an instruction (except for the software interrupt).
- - The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction.
- An interrupt procedure usually stores all the information necessary to define the state of CPU rather than storing only the PC.
- The state of the CPU is determined from;

  Content of the PC

  Content of all processor registers

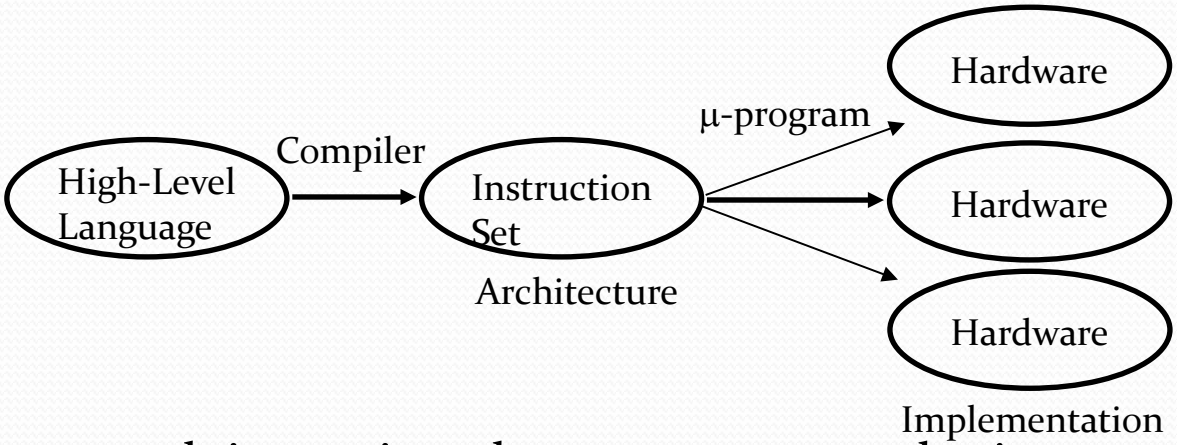
  Content of status bits

  Many ways of saving the CPU state depends on the CPU architectures.

19. RISC: Historical Background

IBM System/360, 1964

- The real beginning of modern computer architecture
- Distinction between *Architecture* and *Implementation*
- Architecture: The abstract structure of a computer
  seen by an assembly-language programmer



- Continuing growth in semiconductor memory and microprogramming
  $\Rightarrow$ A much richer and complicated instruction sets
  $\Rightarrow$ CISC(Complex Instruction Set Computer)

Complex Instruction Set Computer

- Another characteristic of CISC computers is that they have instructions that act directly on memory addresses
  - For example,

    ADD L1, L2, L3

    that takes the contents of $M[L1]$ adds it to the contents of $M[L2]$ and stores the result in location $M[L3]$
- An instruction like this takes three memory access cycles to execute
- That makes for a potentially very long instruction execution cycle
- The problems with CISC computers are
  - The complexity of the design may slow down the processor.
  - The complexity of the design may result in costly errors in the processor design and implementation.
  - Many of the instructions and addressing modes are used rarely.

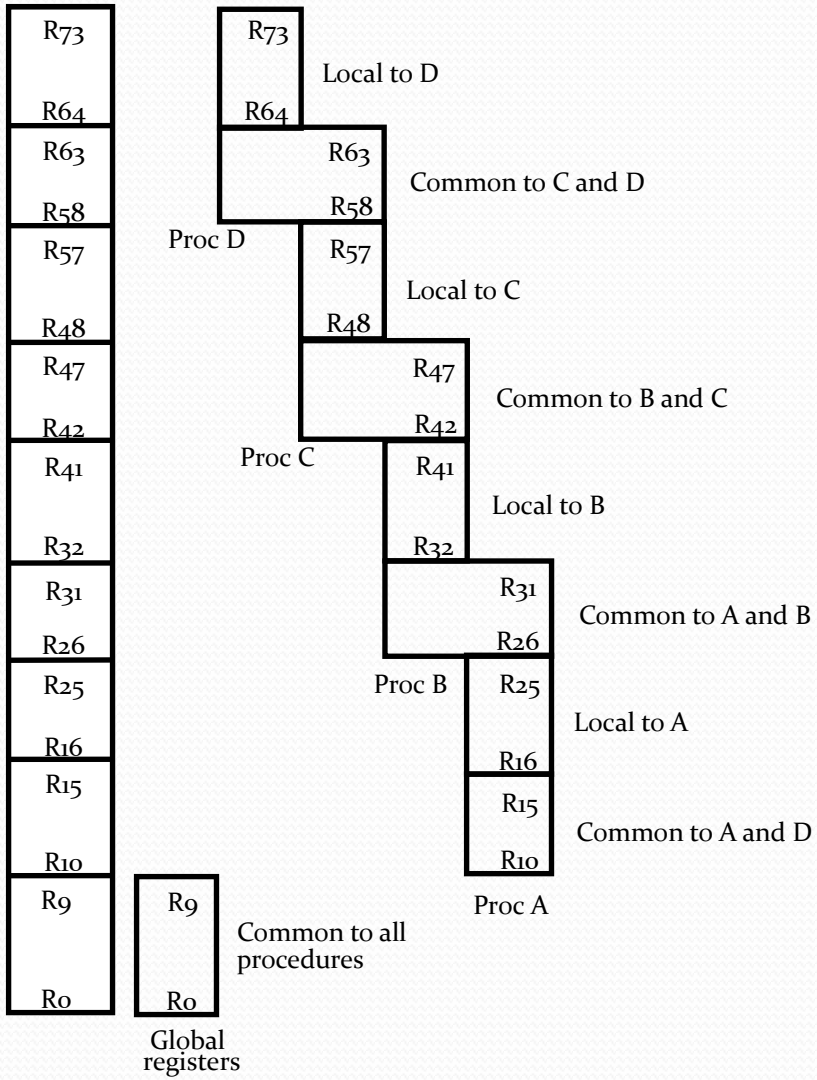
Reduced Instruction Set Computers

- In the late '70s and early '80s there was a reaction to the shortcomings of the CISC style of processors
- Reduced Instruction Set Computers (RISC) were proposed as an alternative
- The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time

- RISC processors often feature:

  - Few instructions
  - Few addressing modes
  - Only load and store instructions access memory
  - All other operations are done using on-processor registers
  - Fixed length instructions
  - Single cycle execution of instructions
  - The control unit is hardwired, not microprogrammed

Overlapped Register Windows



Overlapped Register Windows

- There are three classes of registers:
  - Global Registers
    - Available to all functions
  - Window local registers
    - Variables local to the function
  - Window shared registers
    - Permit data to be shared without actually needing to copy it
- Only one register window is active at a time
  - The active register window is indicated by a pointer
- When a function is called, a new register window is activated
  - This is done by incrementing the pointer
- When a function calls a new function, the high numbered registers of the calling function window are shared with the called function as the low numbered registers in its register window
- This way the caller's high and the called function's low registers overlap and can be used to pass parameters and results

Overlapped Register Windows

- In addition to the overlapped register windows, the processor has some number of registers, *G*, that are global registers
  - This is, all functions can access the global registers.
- The advantage of overlapped register windows is that the processor does not have to push registers on a stack to save values and to pass parameters when there is a function call
  - Conversely, pop the stack on a function return
- This saves
  - Accesses to memory to access the stack.
  - The cost of copying the register contents at all
- And, since function calls and returns are so common, this results in a significant savings relative to a stack-based approach.

Characteristics Of RISC

- RISC Characteristics
  - Relatively few instructions
  - Relatively few addressing modes
  - Memory access limited to load and store instructions
  - All operations done within the registers of the CPU
  - Fixed-length, easily decoded instruction format
  - Single-cycle instruction format
  - Hardwired rather than microprogrammed control

- Advantages of RISC
  - VLSI Realization
  - Computing Speed
  - Design Costs and Reliability
  - High Level Language Support

Advantages Of RISC

- Design Costs and Reliability

  - Shorter time to design
    ⇒ reduction in the overall design cost and
    reduces the problem that the end product will
    be obsolete by the time the design is completed

  - Simpler, smaller control unit
    ⇒ higher reliability

  - Simple instruction format (of fixed length)
    ⇒ ease of virtual memory management

- High Level Language Support

  - A single choice of instruction
    ⇒ shorter, simpler compiler

  - A large number of CPU registers
    ⇒ more efficient code

  - Register window
    ⇒ Direct support of HLL

  - Reduced burden on compiler writer

The End