INDEX

REGISTER TRANSFER LANGUAGE

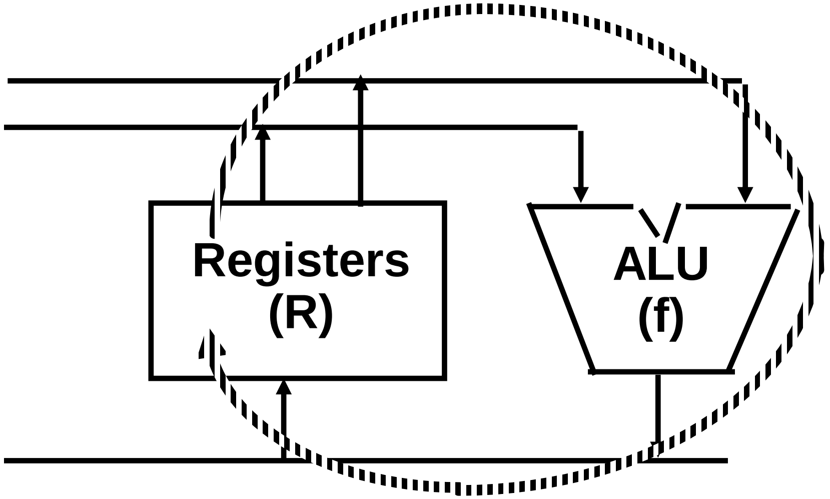
- Combinational and sequential circuits can be used to create simple digital systems.
- These are the low-level building blocks of a digital computer.
- Simple digital systems are frequently characterized in terms of
  - the registers they contain, and
  - the operations that they perform.
- Typically,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

MICROOPERATIONS (1)

- The operations executed on data stored in registers are called microoperations.
- Examples of microoperations
  - Shift
  - Load
  - Clear
  - Increment
  - Count

MICROOPERATION (2)

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



1 clock cycle

R ← f(R, R)

f: shift, load, clear, increment, add, subtract, complement, and, or, xor, …

INTERNAL HARDWAREORGANIZATION OF A DIGITAL SYSTEM

- **Definition of the internal hardware organization of a computer**

  - Set of registers it contains and their function

  - The sequence of microoperations performed on the binary information stored in the registers

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

REGISTER TRANSFER LANGUAGE

- *The symbolic notation used to describe the microoperation transfers among registers is called a Register transfer language.*
- 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.

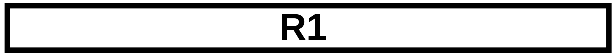
Register Transfer

- 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
- Information transfer from one register to another is designated in symbolic form by means of a replacement operator.

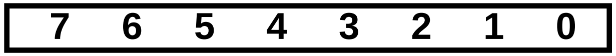
R2 ← R1

  - In this case the contents of register R2 are copied (loaded) into register R1 and contents of R1 remains same.

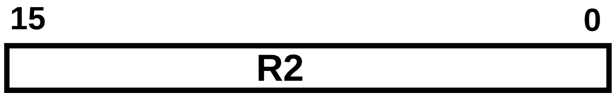
Block diagram of a register



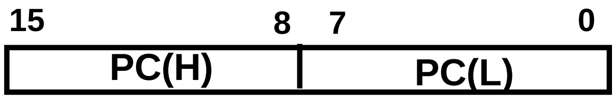
Register R



Showing individual bits



Numbering of bits



Subfields (Divided into two parts)

- Often we want the transfer to occur only under a predetermined control condition.

  if (p=1) then (R2 ← R1)

  where p is a control signal generated in the control section.

- 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

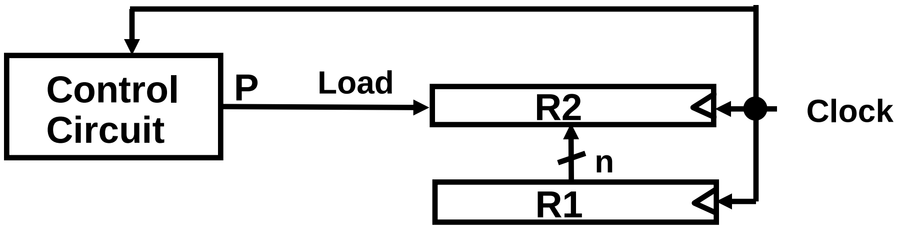
  Which means "if P = 1, then load the contents of register R1 into register R2", i.e., if (P = 1) then (R2 ← R1)

HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

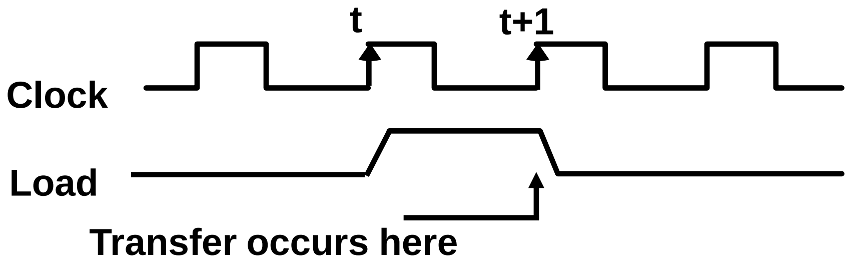
Implementation of controlled transfer

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

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

- 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

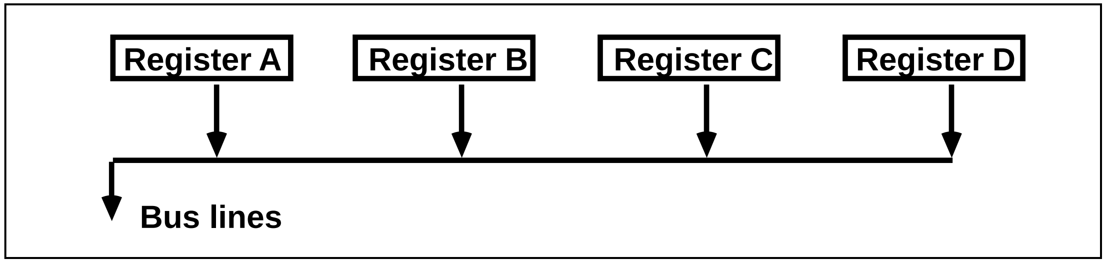
BASIC SYMBOLS FOR REGISTER TRANSFERS

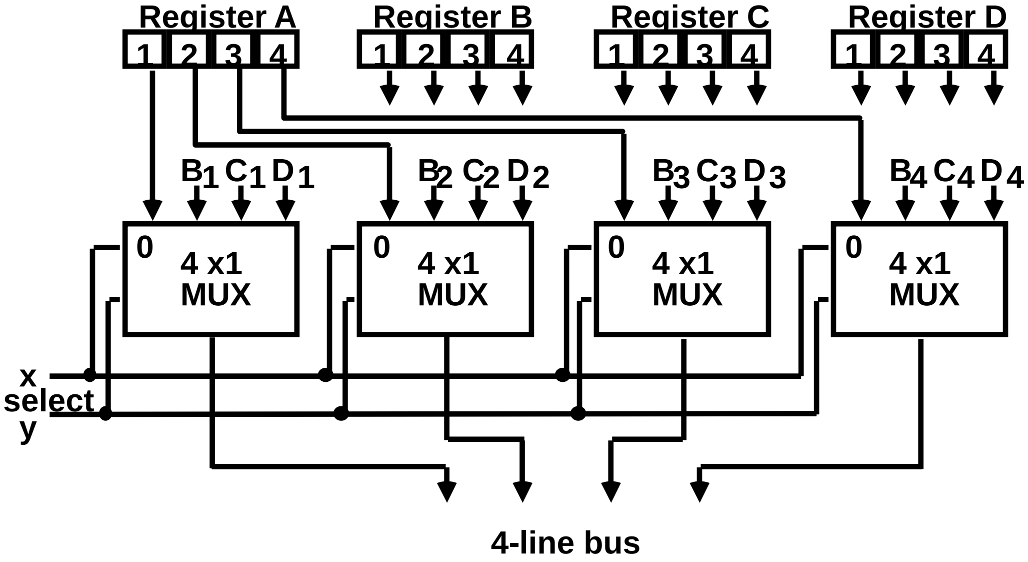
Symbols	Description	Examples
Capital letters MAR, R2 & numerals	Denotes a register	
Parentheses () R2(0-7), R2(L)	Denotes a part of a register	
Arrow ← R2 ← R1	Denotes transfer of information	
Colon : P:	Denotes termination of control function	
Comma , A ← B, B ← A	Separates two micro-operations	

BUS AND MEMORY TRANSFERS

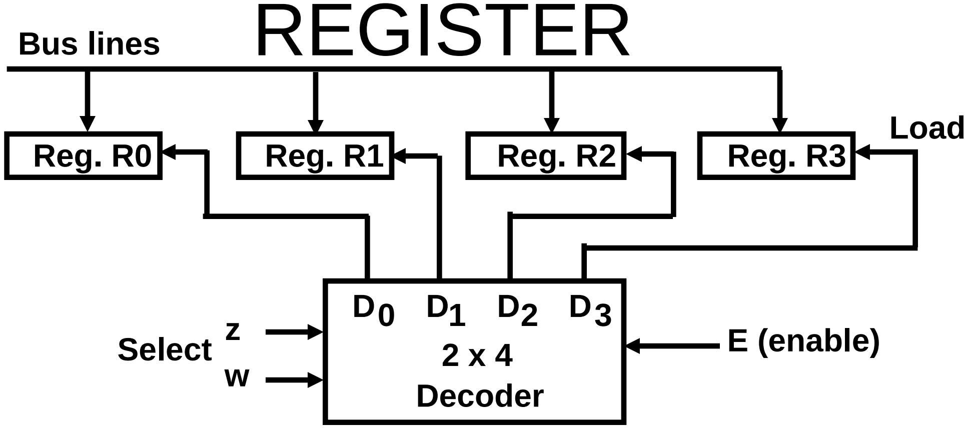
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

From a register to bus: BUS ← R

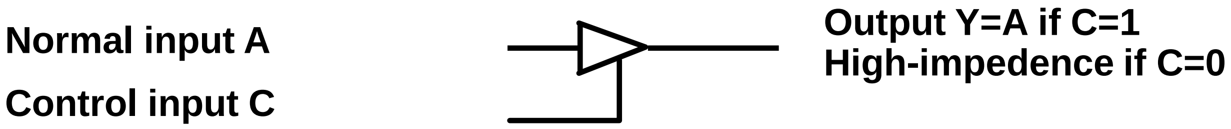




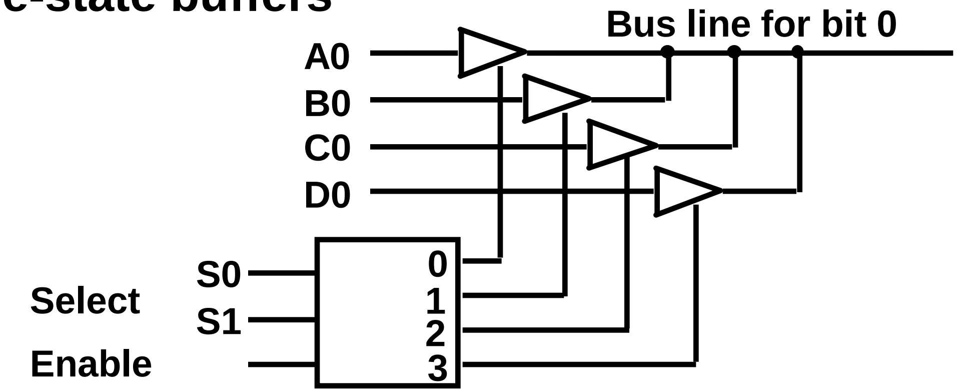
TRANSFER FROM BUS TO A DESTINATION REGISTER



Three-State Bus Buffers



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 ← R1

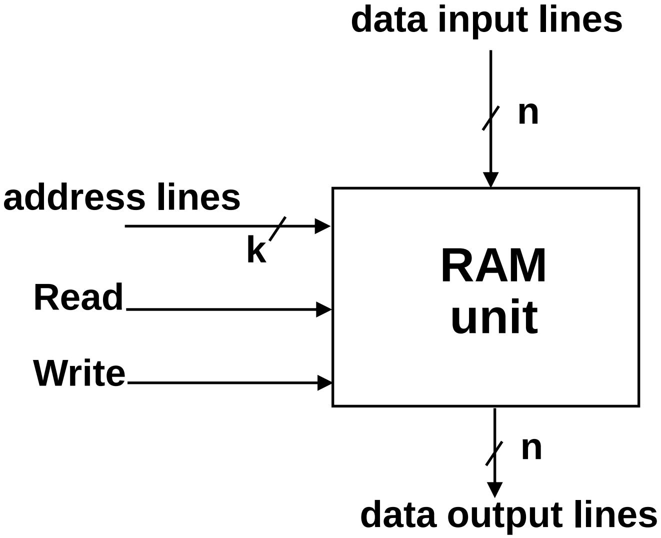
or

BUS ← R1, R2 ← BUS

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

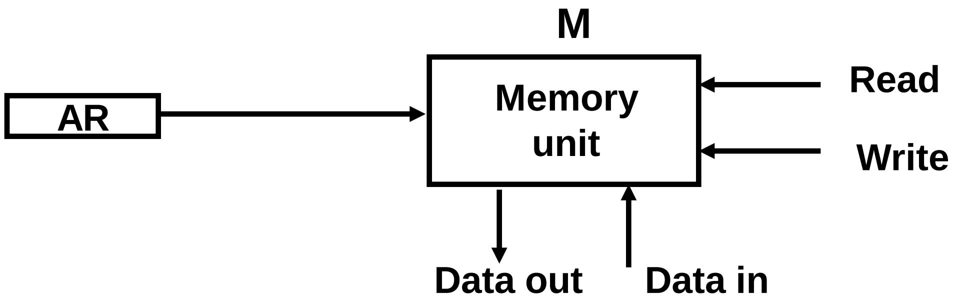
MEMORY (RAM)

- 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 ← 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] ← 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:

1. **Register transfer microoperations transfer binary information from one register to another**
2. **Arithmetic microoperations perform arithmetic operations on numeric data stored in registers.**
3. **Logic microoperations perform bit manipulation operations on non numeric data stored in registers.**
4. **Shift microoperations perform shift operations on data stored in registers.**

ARITHMETIC MICROOPERATIONS

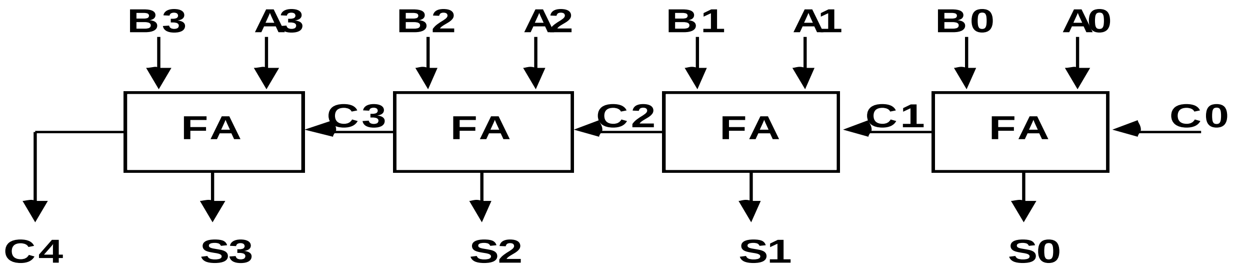
- The basic arithmetic microoperations are
  - Addition
  - Subtraction
  - Increment
  - Decrement
- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

Table: Arithmetic Micro-Operations

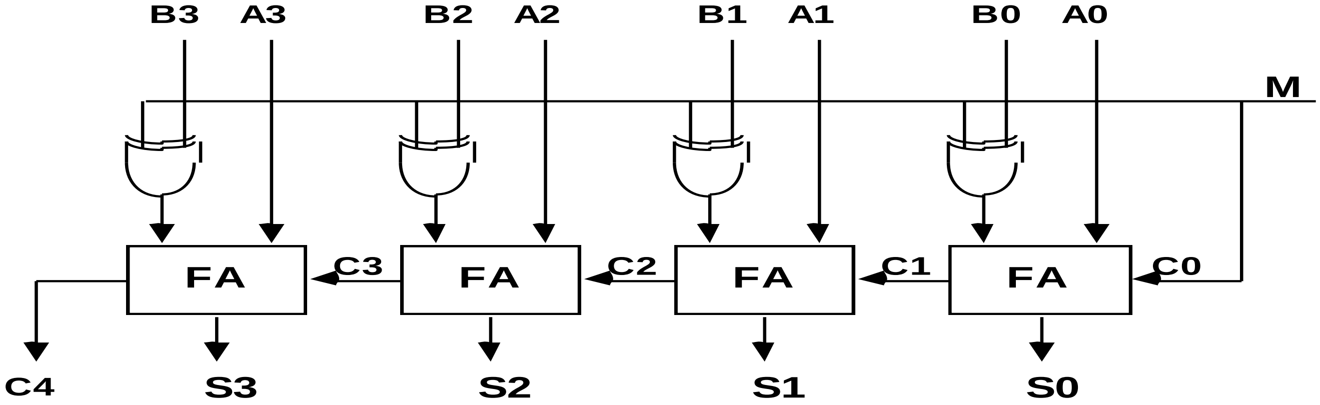
$R3 \leftarrow R1 + R2$	**Contents of R1 plus R2 transferred to R3**
$R3 \leftarrow R1 - R2$	**Contents of R1 minus R2 transferred to R3**
$R2 \leftarrow R2'$	**Complement the contents of R2**
$R2 \leftarrow R2' + 1$	**2's complement the contents of R2 (negate)**
$R3 \leftarrow R1 + R2' + 1$	**subtraction**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

BINARY ADDER / SUBTRACTOR / INCREMENTER

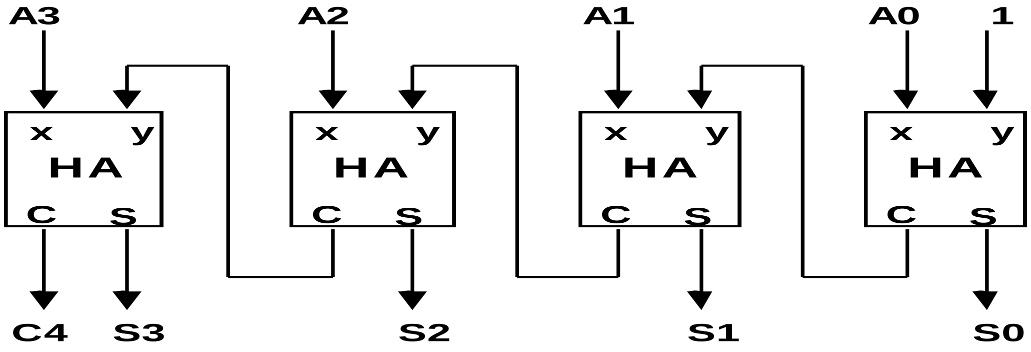
Binary Adder



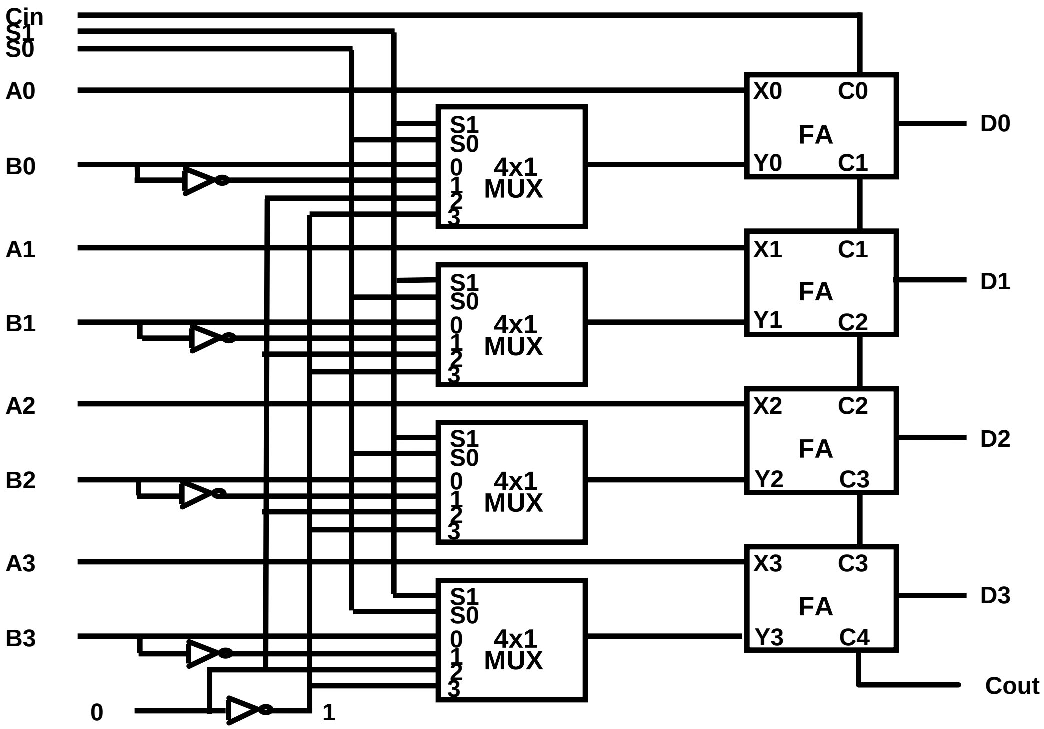
Binary Adder-Subtractor



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

- It specifies binary operations on the strings of bits stored 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 ($\wedge$), OR ($\vee$), XOR ($\oplus$), Complement/NOT
- The others can be created from combination of these

LIST OF LOGIC MICROOPERATIONS

- **List of Logic Microoperations**

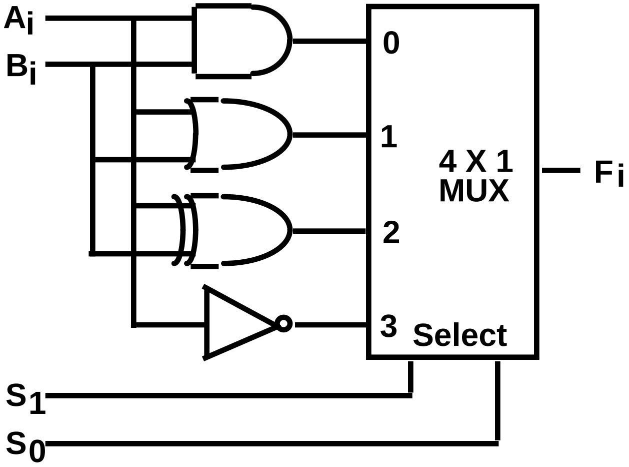
  - 16 different logic operations with 2 binary vars.

  - n binary vars → 2^{2^n} functions

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

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

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



Function table

S_1 S_0	Output	μ-operation
0 0	F = A ∧ B	AND
0 1	F = A ∨ B	OR
1 0	F = A ⊕ B	XOR
1 1	F = A’	Complement

APPLICATIONS OF LOGIC MICROOPERATIONS

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

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

$$\begin{array}{ll} 1\;1\;0\;0 & A_t \\ 1\;0\;1\;0 & B \\ \hline 1\;1\;1\;0 & A_{t+1} \end{array} \quad (A \leftarrow A + B)$$

- 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

$$\begin{array}{ll} 1\ 1\ 0\ 0 & A_t \\ 1\ 0\ 1\ 0 & B \\ \hline 0\ 1\ 1\ 0 & A_{t+1} \quad (A \leftarrow A \oplus B) \end{array}$$

- 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

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

- 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

$$\begin{array}{ll} 1\ 1\ 0\ 0 & A_t \\ 1\ 0\ 1\ 0 & B \\ \hline 1\ 0\ 0\ 0 & A_{t+1} \qquad (A \leftarrow A \cdot B) \end{array}$$

- 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

$$\begin{array}{lll} 1\;1\;0\;0 & A_t & \\ 1\;0\;1\;0 & B & \\ \hline 0\;1\;1\;0 & A_{t+1} & (A \leftarrow A \oplus B) \end{array}$$

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

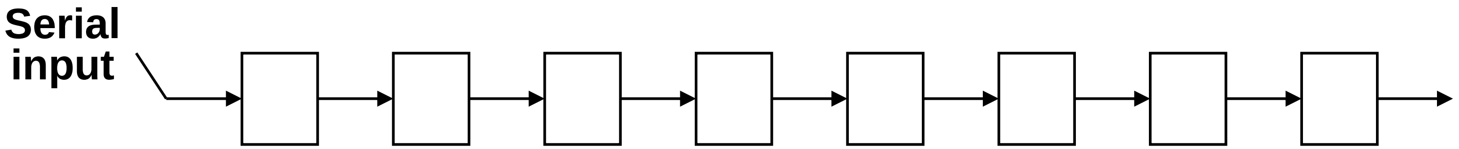
- Suppose you wanted to introduce 1010 into the low order
four bits of A: 1101 1000 1011 0001   A (Original)
1101 1000 1011 1010   A (Desired)

-
```
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)
_________________________________________
```

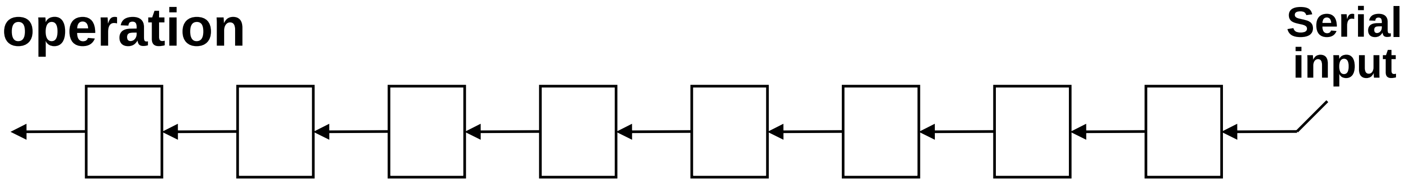
SHIFT MICROOPERATIONS

- Shift microoperations are used for serial transfer of data.
- The information transferred through the serial input determines the type of shift. There are three types of shifts
  - *Logical shift*
  - *Circular shift*
  - *Arithmetic shift*

- **A right shift operation**

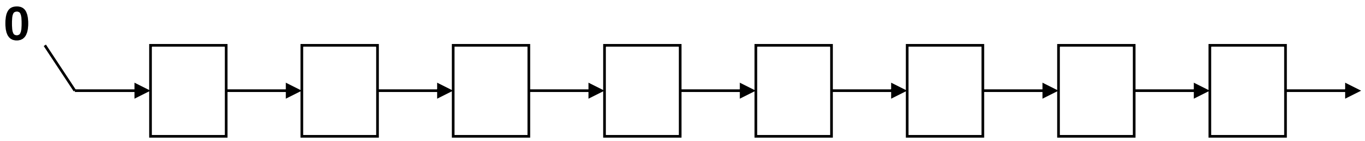


- **A left shift operation**

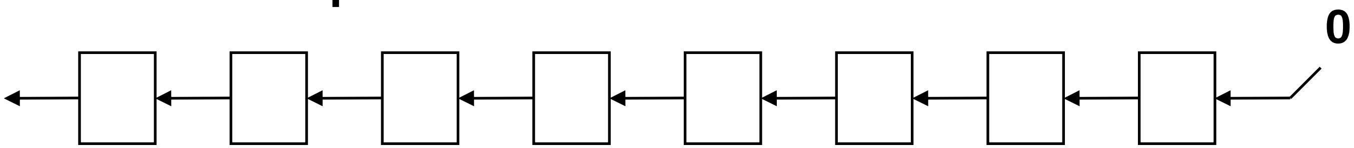


LOGICAL SHIFT

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



- A left logical shift operation:

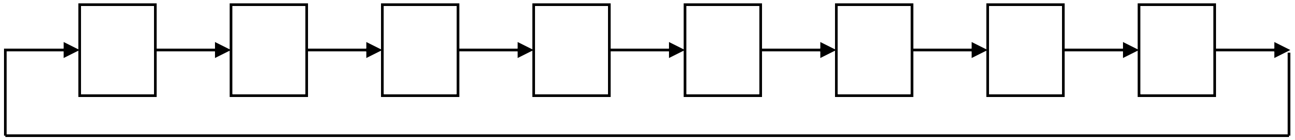


- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right
  - Examples:
    - R2 ← *shr* R2
    - R3 ← *shl* R3

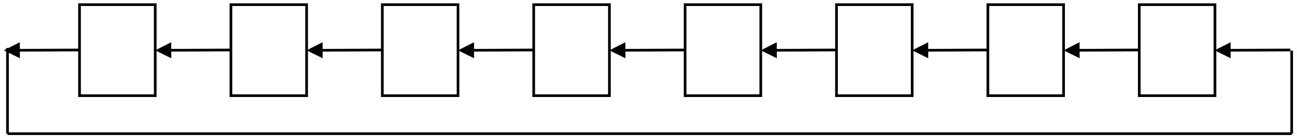
CIRCULAR SHIFT

- 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:



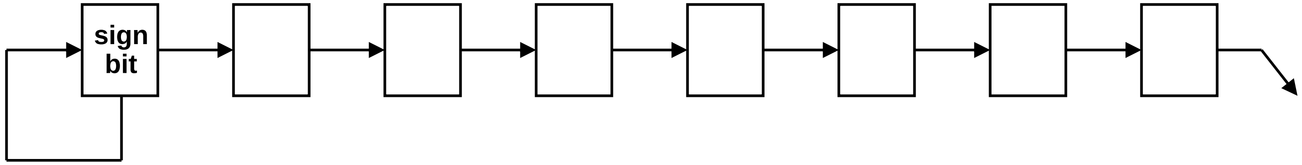
- A left circular shift operation:



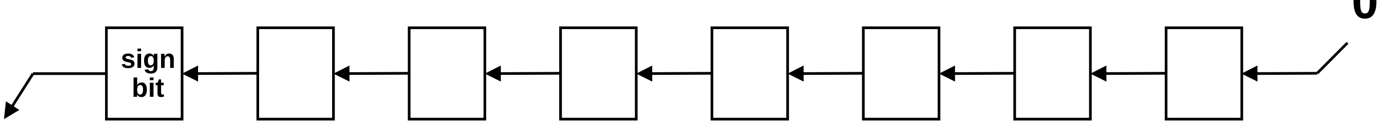
- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - R2 ← *cir* R2
    - R3 ← *cil* R3

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



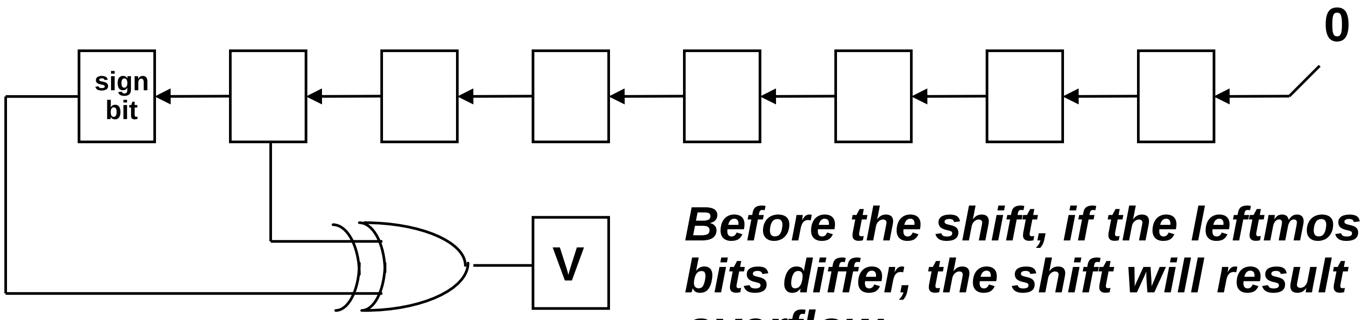
- A right arithmetic shift operation:



- A left arithmetic shift operation:

ARITHMETIC SHIFT

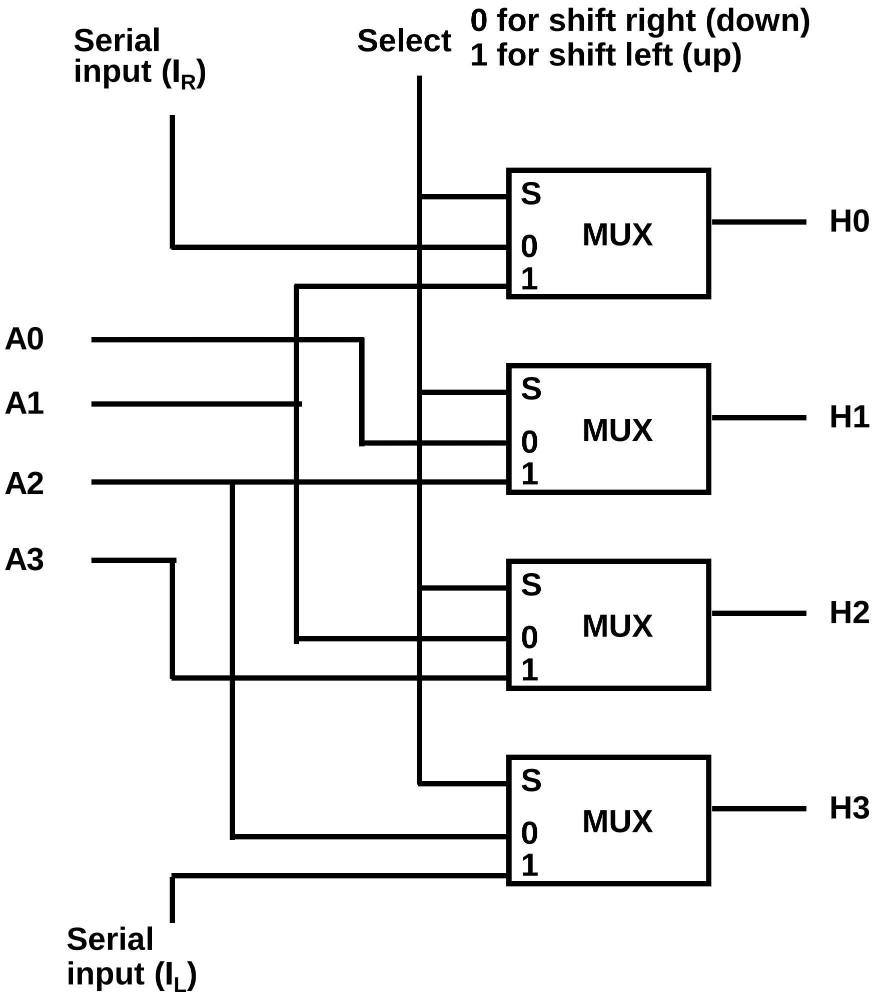
- An left arithmetic shift operation must be checked for the overflow



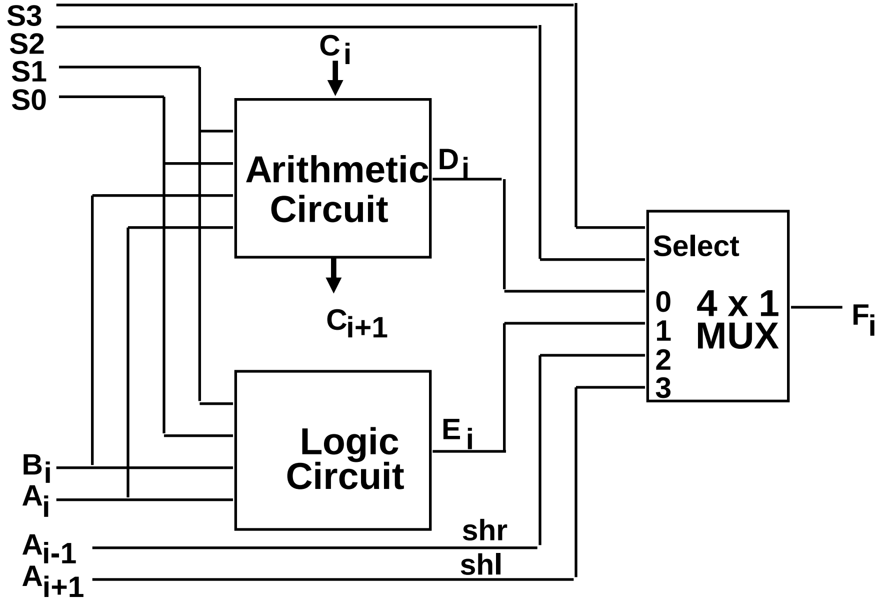
Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- 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	$F = A \wedge B$	AND
0	1	0	1	X	$F = A \vee B$	OR
0	1	1	0	X	$F = A \oplus B$	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

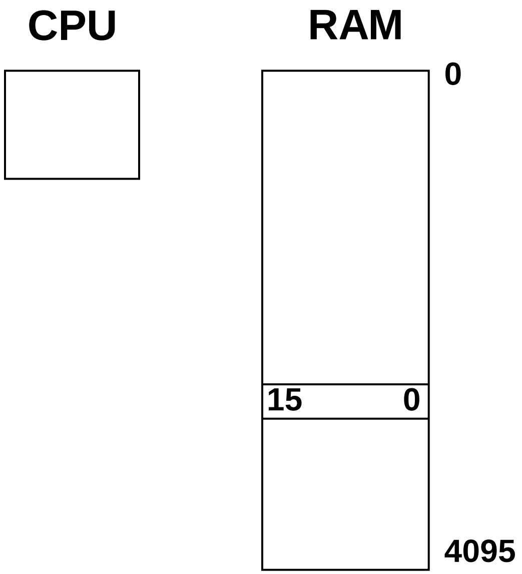
- **Instruction Codes**
- **Computer Registers**
- **Computer Instructions**
- **Timing and Control**
- **Instruction Cycle**
- **Memory Reference Instructions**
- **Input-Output and Interrupt**
- **Complete Computer Description**
- **Design of Basic Computer**
- **Design of Accumulator Logic**

Instruction Codes

- 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

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

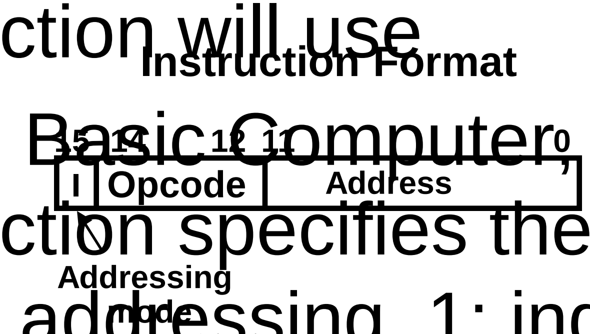


INSTRUCTIONS

- 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

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

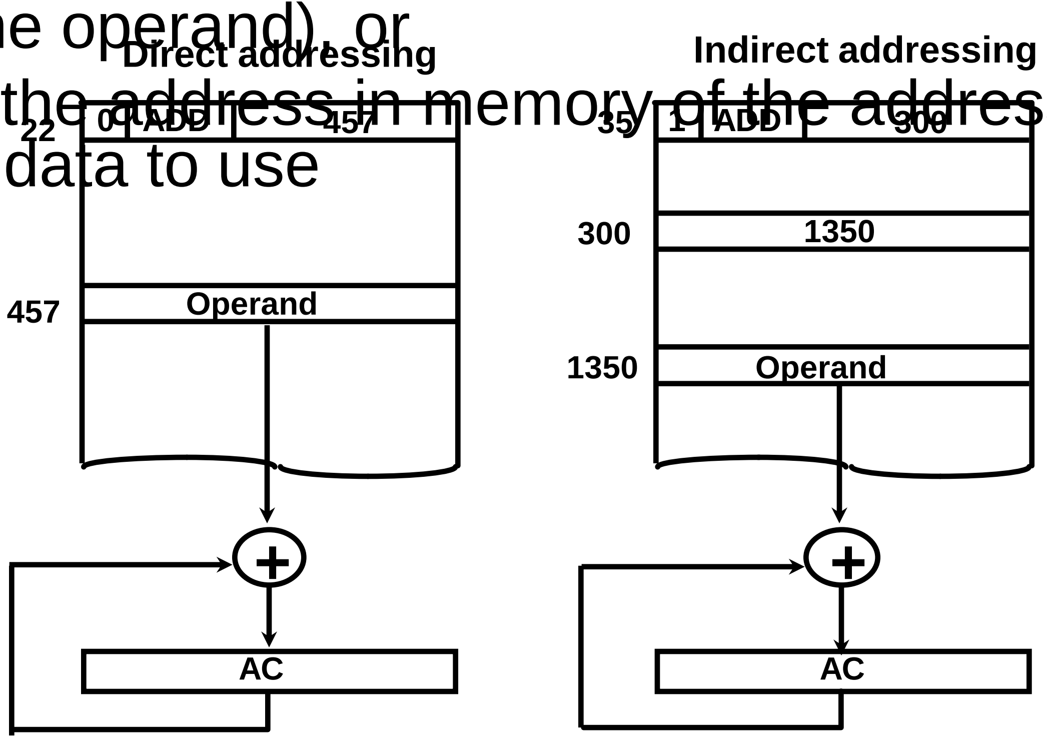


Instruction Format

- 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

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



PROCESSOR REGISTERS

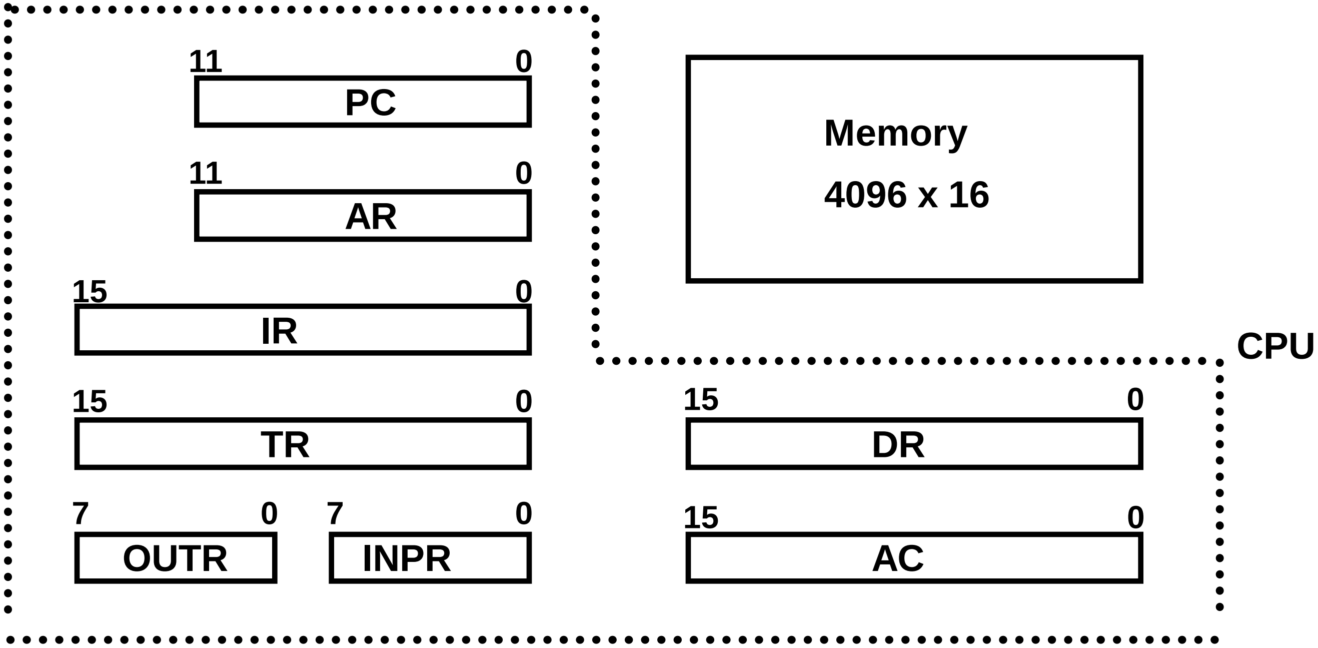
- A processor has many registers to hold instructions, addresses, data, etc
- The processor has a register, the *Program Counter* (PC) that holds the memory address of the next instruction to get
  - Since the memory in the Basic Computer only has 4096 locations, the PC only needs 12 bits
- In a direct 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

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

COMPUTER REGISTERS

Registers in the Basic Computer



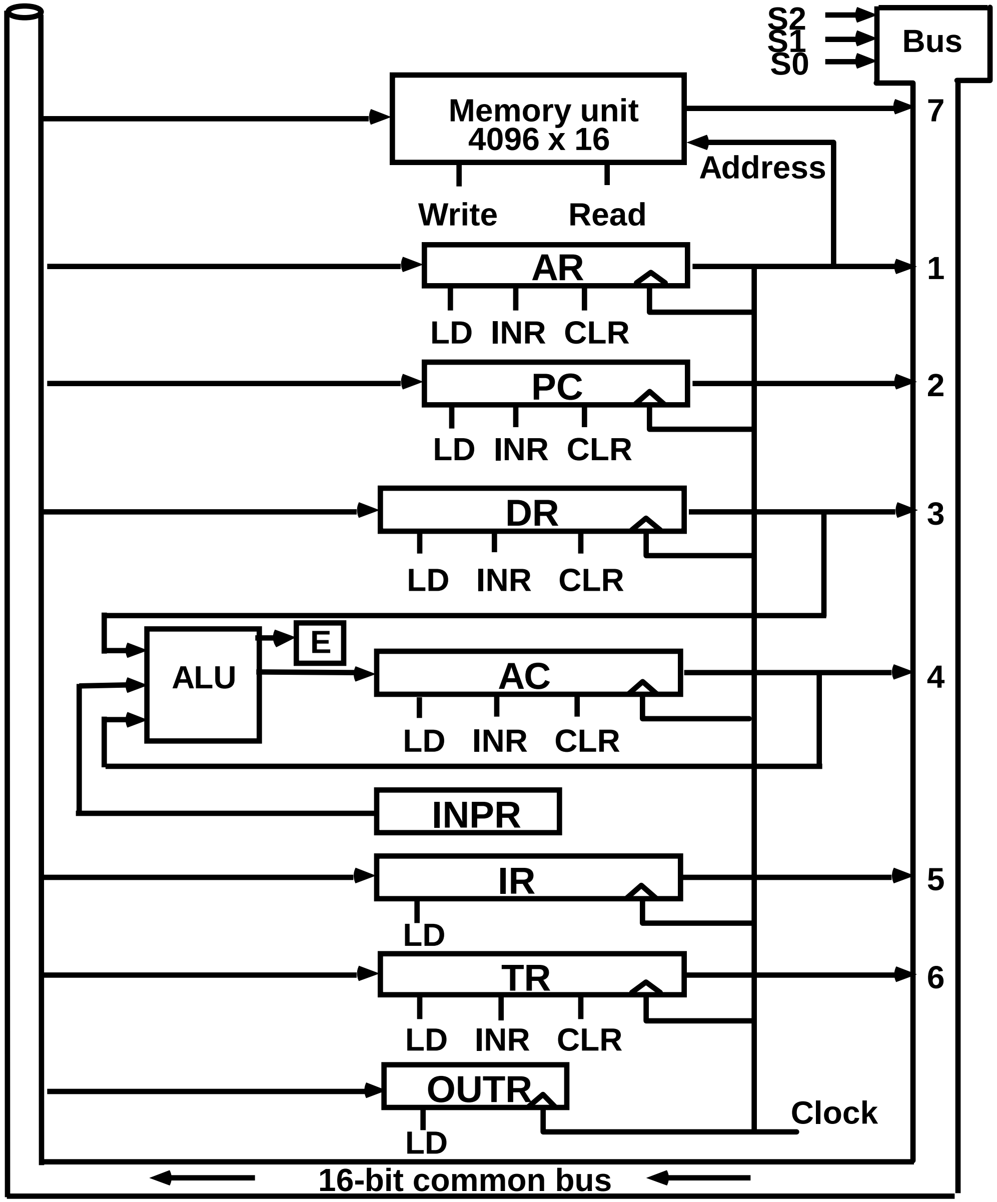
List of BC Registers

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

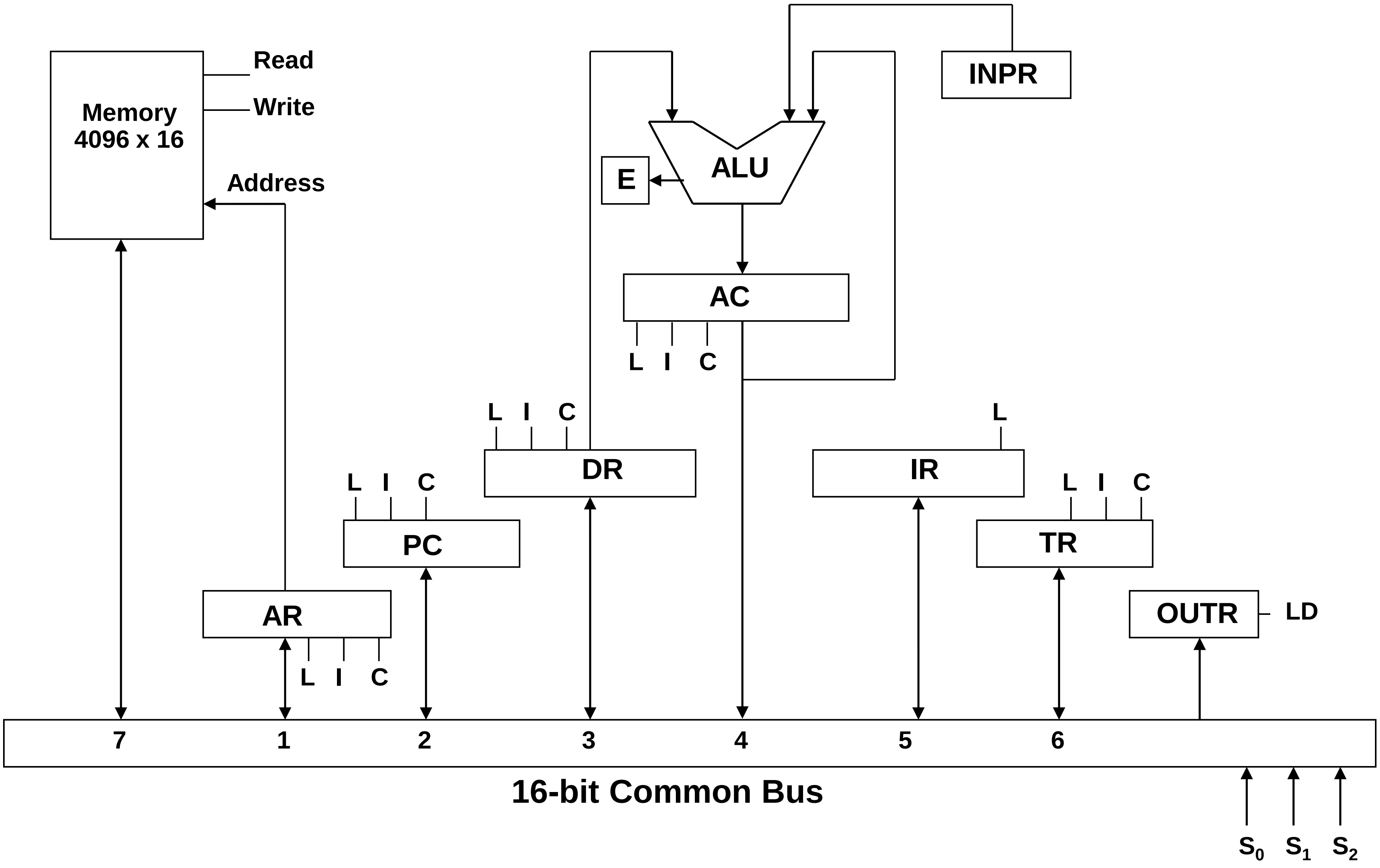
COMMON BUS SYSTEM

- 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

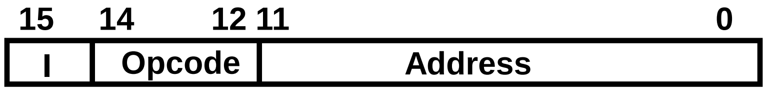
S_2	S_1	S_0	Register
0	0	0	x
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

- 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

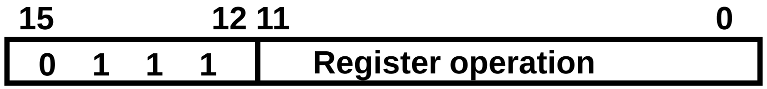
COMPUTER INSTRUCTIONS

- **Basic Computer Instruction Format**

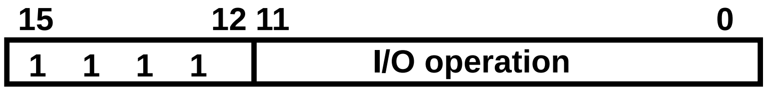
Memory-Reference Instructions (OP-code = 000 ~ 110)



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



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



BASIC COMPUTER INSTRUCTIONS

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

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

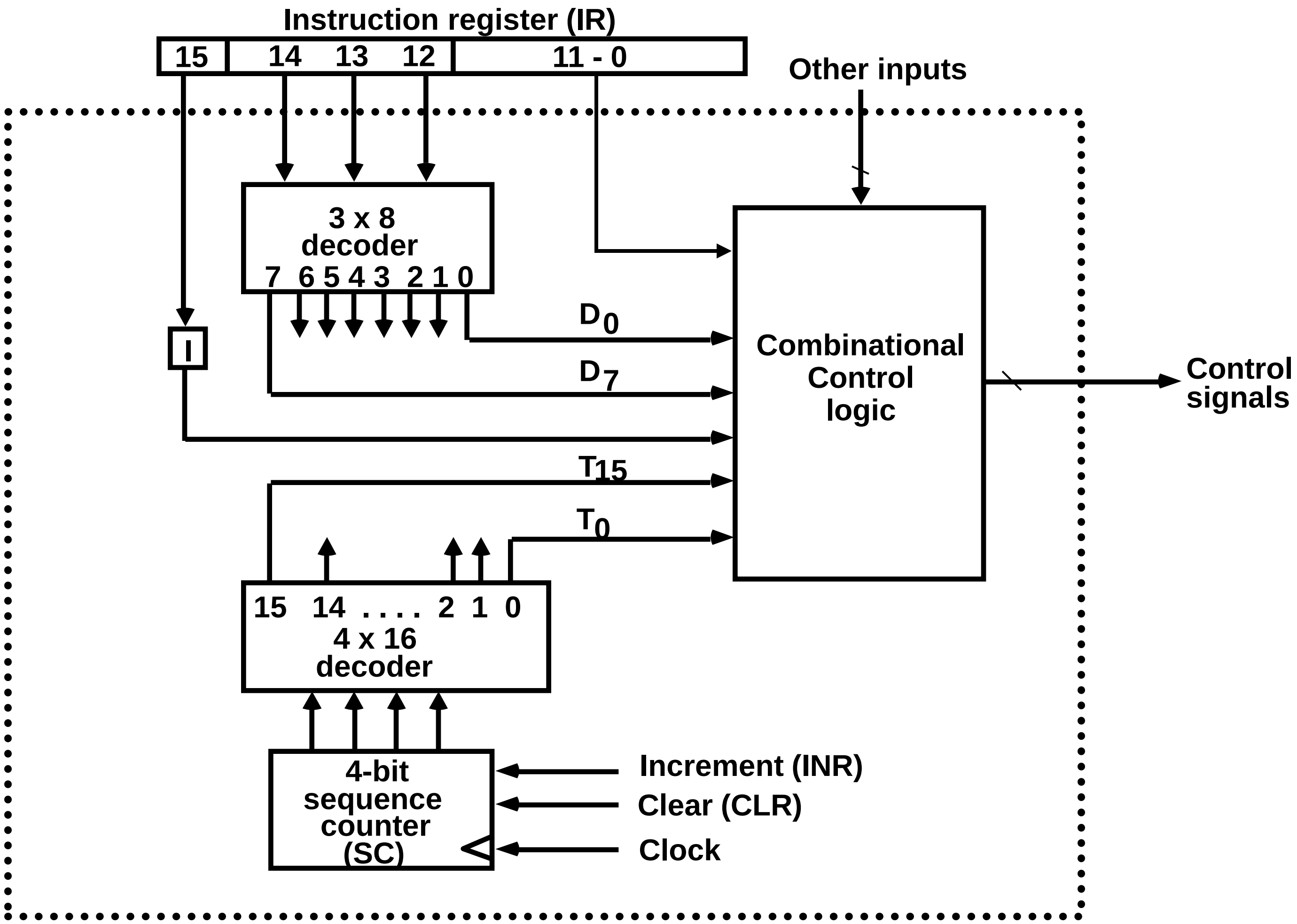
CONTROL UNIT

- 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

TIMING AND CONTROL

Control unit of Basic Computer

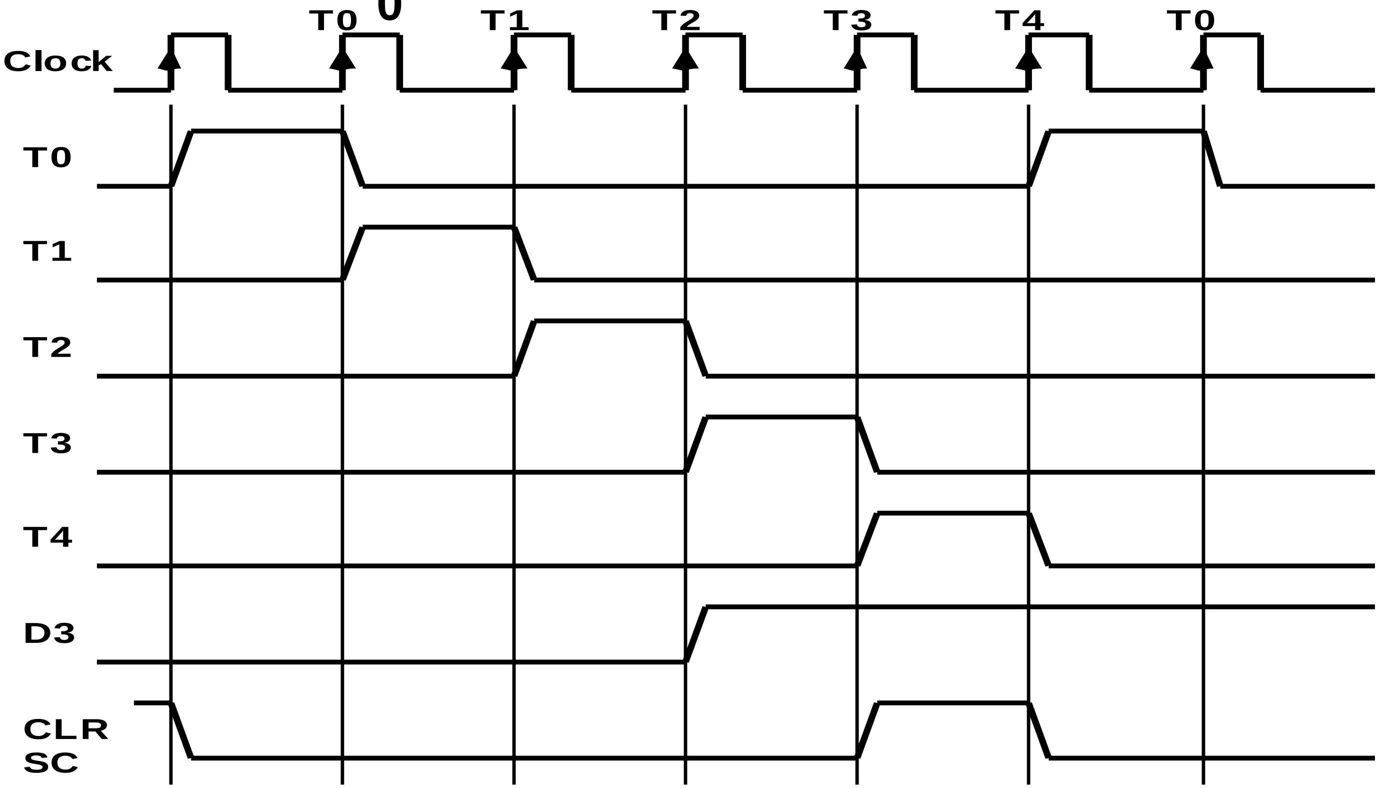


TIMING SIGNALS

- **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, . . .**
  Assume: At time T_4, SC is cleared to 0 if decoder output D3 is active.

D_3T_4: SC ← 0



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

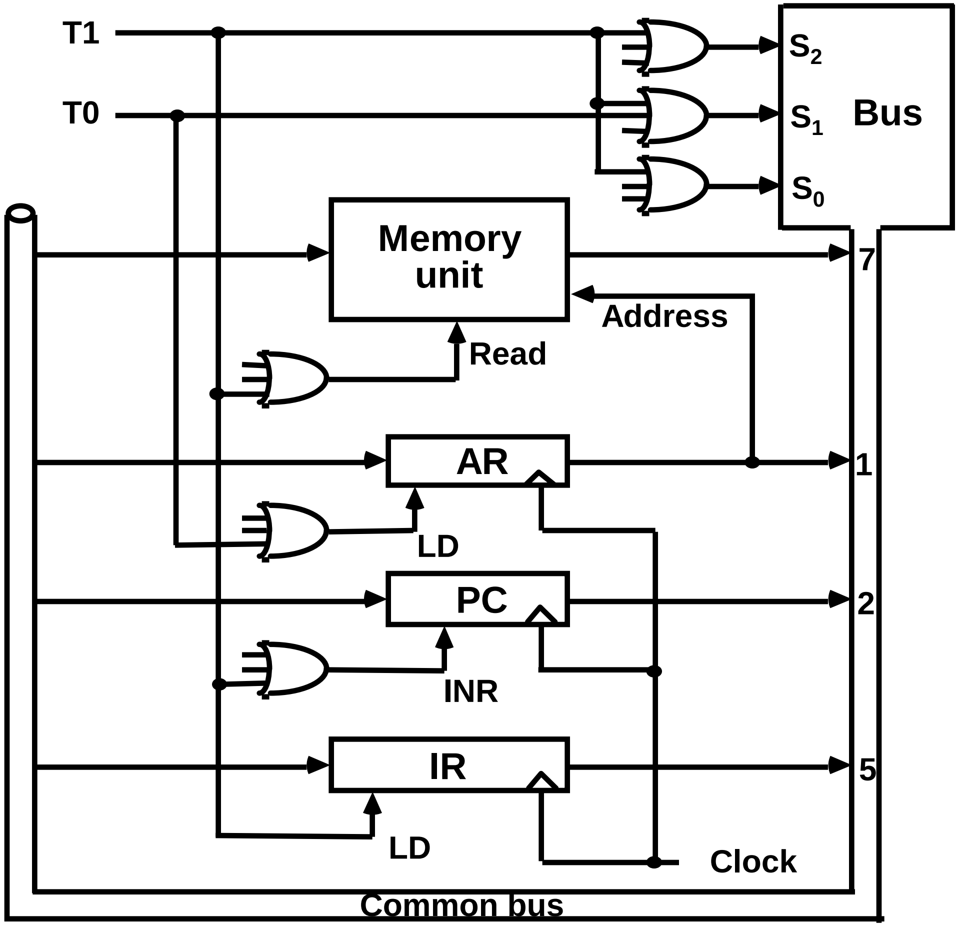
- After an instruction is executed, the cycle starts again at step 1, for the next instruction

- *Note*: Every different processor has its own (different) instruction cycle

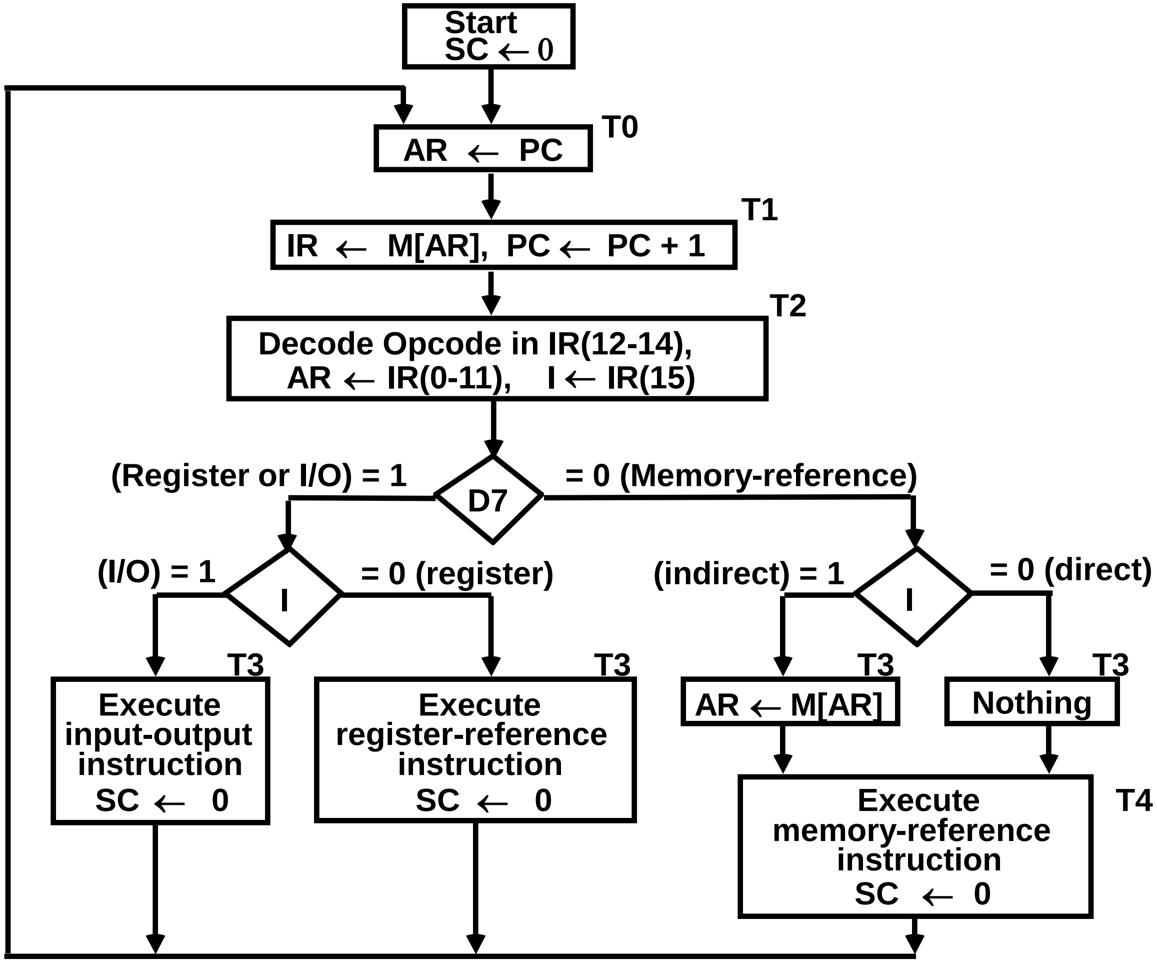
FETCH and DECODE

- **Fetch and Decode**

T0: AR ← PC (S$_0$S$_1$S$_2$=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


Start
SC ← 0
T0
AR ← PC
T1
IR ← M[AR], PC ← PC + 1
T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
T4
Execute memory-reference instruction SC ← 0

$D'_7 I T_3$:	AR ← M[AR]
$D'_7 I' T_3$:	Nothing
$D_7 I' T_3$:	Execute a register-reference instr.
$D_7 I T_3$:	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

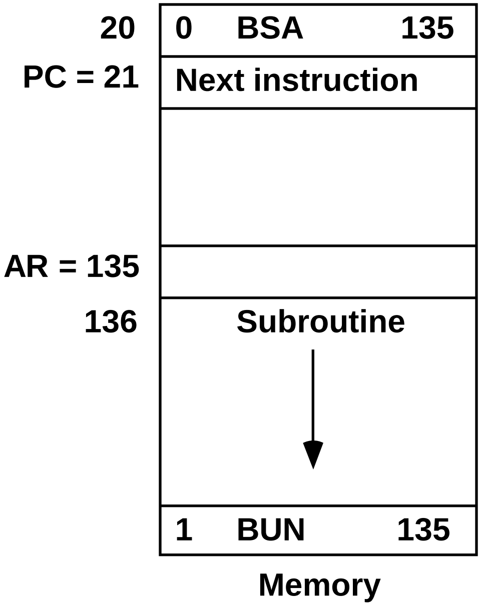
BUN: Branch Unconditionally

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

M[AR] ← PC, PC ← AR + 1

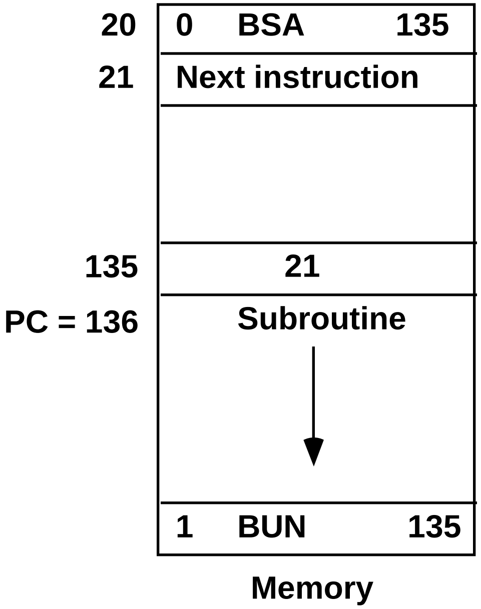
Memory, PC, AR at time T4



20	0 BSA 135
PC = 21	Next instruction
AR = 135	
136	Subroutine
	1 BUN 135

Memory

Memory, PC after execution



20	0 BSA 135
21	Next instruction
135	21
PC = 136	Subroutine
	1 BUN 135

Memory

MEMORY REFERENCE INSTRUCTIONS

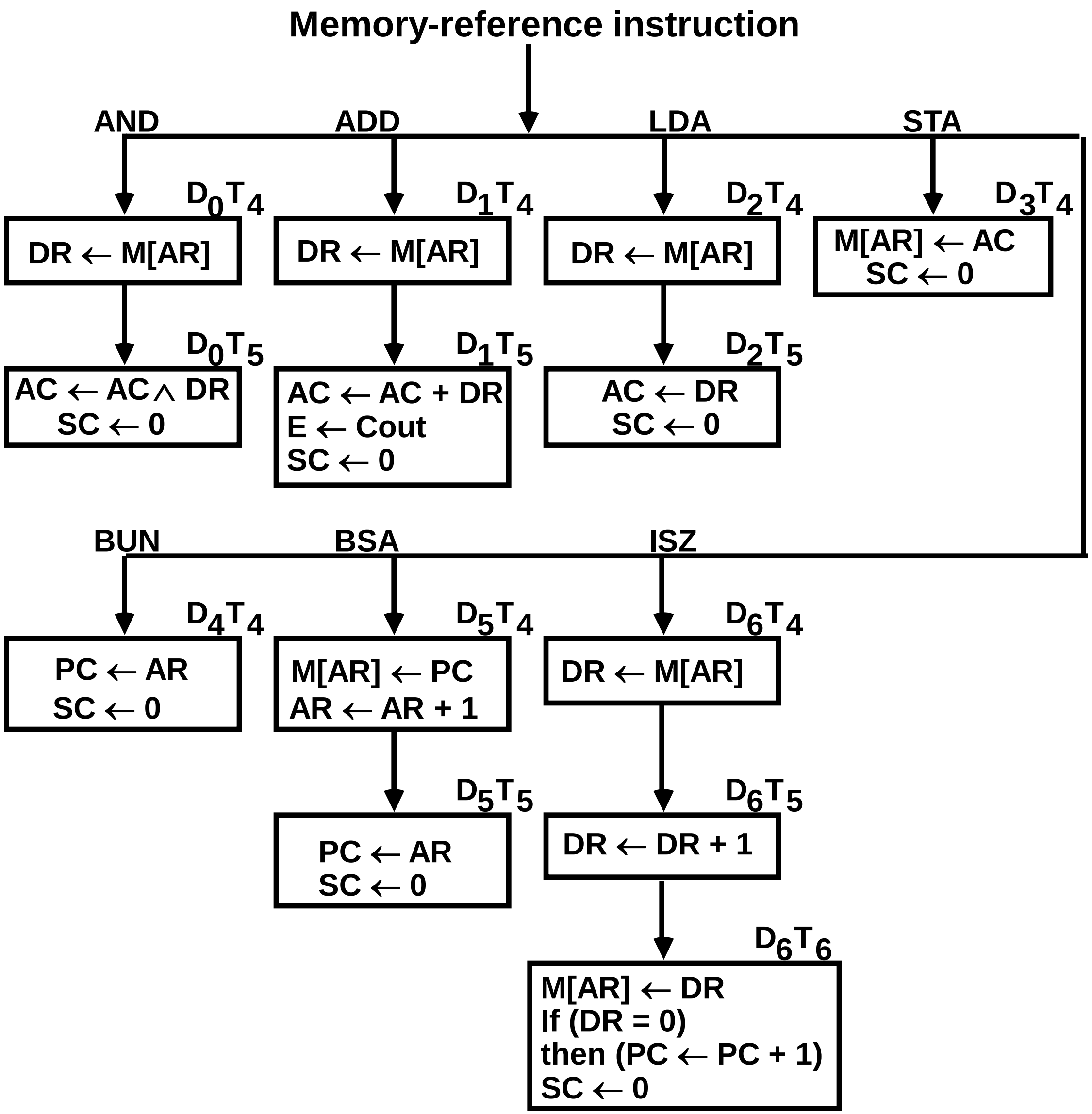
BSA:

D_5T_4: M[AR] ← PC, AR ← AR + 1
D_5T_5: PC ← AR, SC ← 0

ISZ: Increment and Skip-if-Zero

D_6T_4: DR ← M[AR]
D_6T_5: DR ← DR + 1
D_6T_4: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

LOWCHART FOR MEMORY REFERENCE INSTRUCTION



Memory-reference instruction

AND

D_0T_4 DR ← M[AR]

D_0T_5 AC ← AC ∧ DR
SC ← 0

ADD

D_1T_4 DR ← M[AR]

D_1T_5 AC ← AC + DR
E ← Cout
SC ← 0

LDA

D_2T_4 DR ← M[AR]

D_2T_5 AC ← DR
SC ← 0

STA

D_3T_4 M[AR] ← AC
SC ← 0

BUN

D_4T_4 PC ← AR
SC ← 0

BSA

D_5T_4 M[AR] ← PC
AR ← AR + 1

D_5T_5 PC ← AR
SC ← 0

ISZ

D_6T_4 DR ← M[AR]

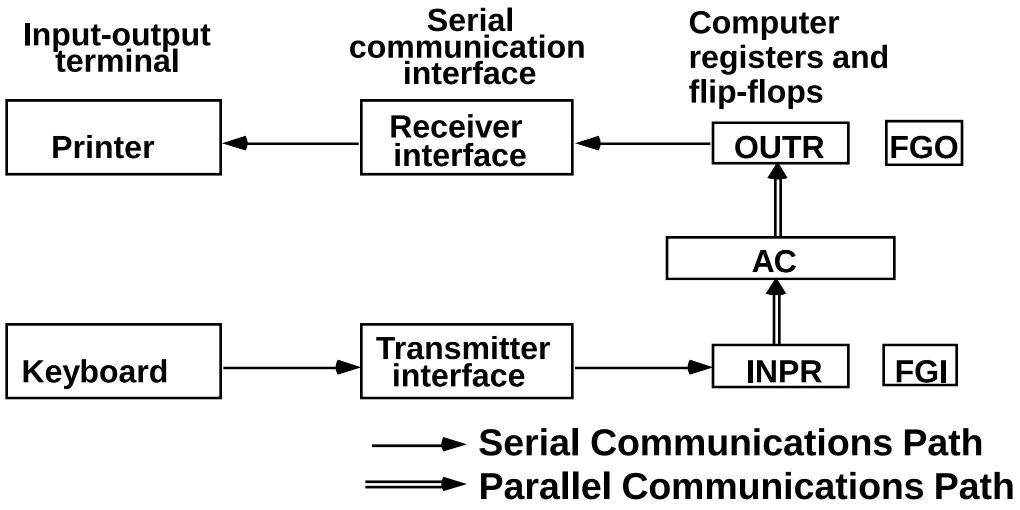
D_6T_5 DR ← DR + 1

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

INPUT-OUTPUT AND INTERRUPT

A Terminal with a keyboard and a Printer

- **Input-Output Configuration**


Input-output terminal
Serial communication interface
Computer registers and flip-flops
Printer
Receiver interface
OUTR
FGO
AC
Keyboard
Transmitter interface
INPR
FGI
Serial Communications Path
Parallel Communications Path

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

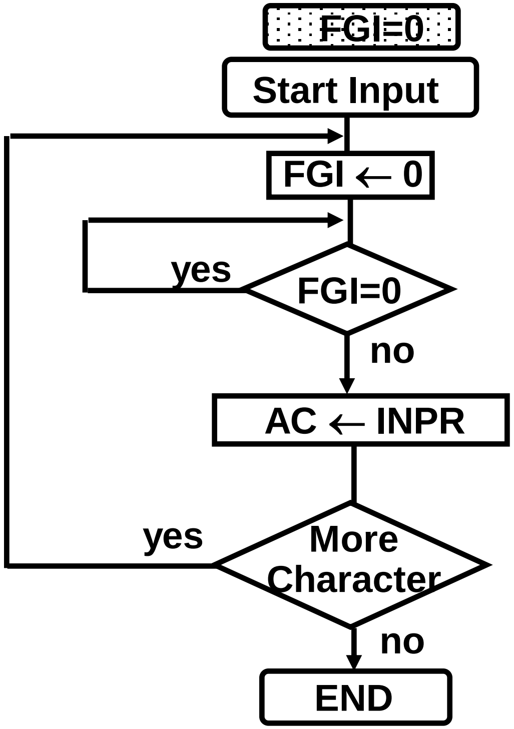
PROGRAM CONTROLLED DATA TRANSFER

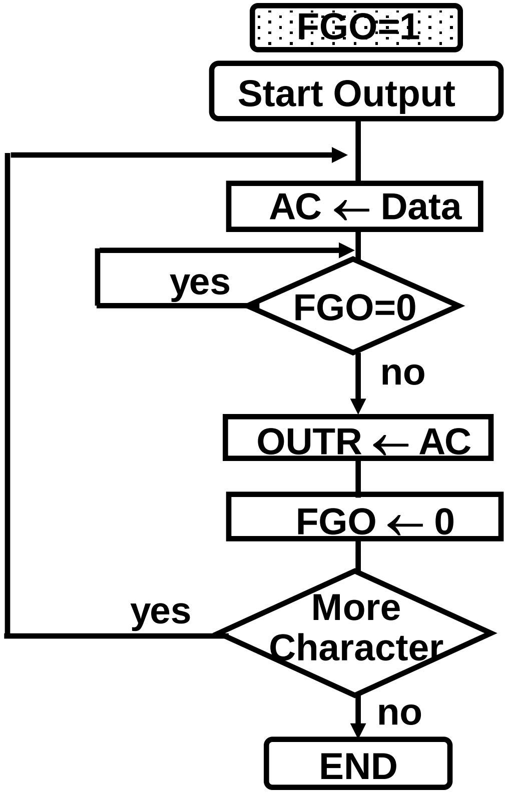
-- CPU --

```
/* 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.

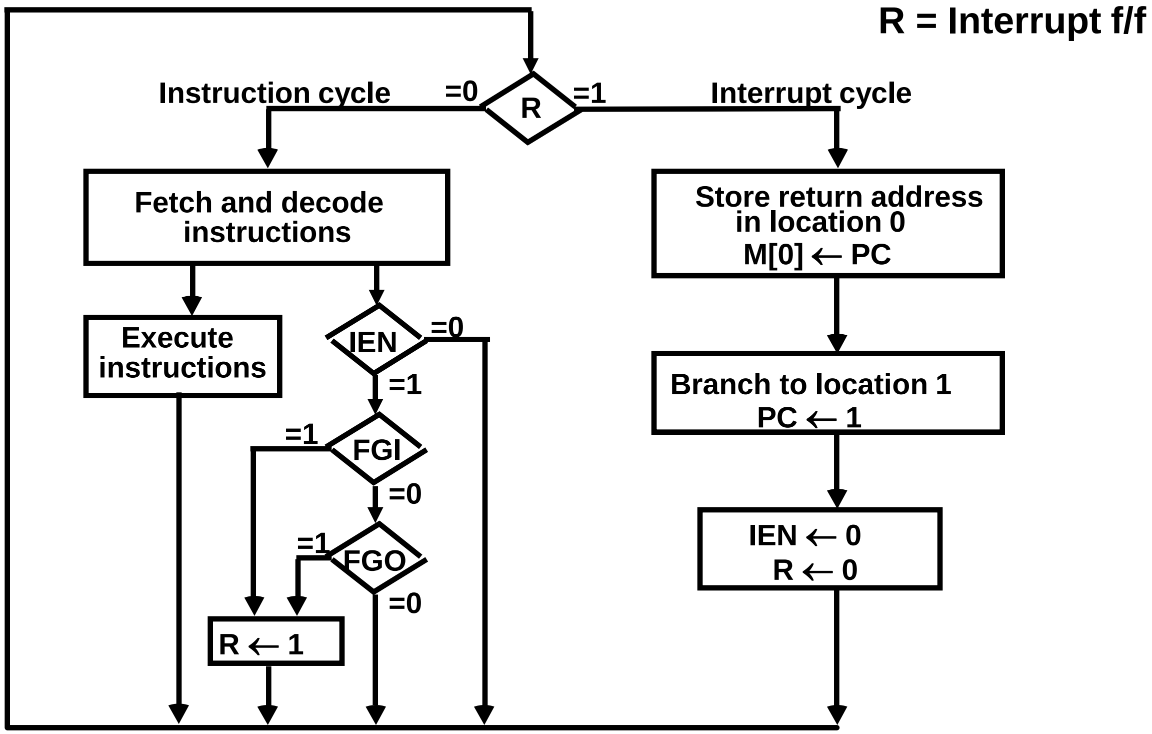
- When the interface founds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU

- 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)

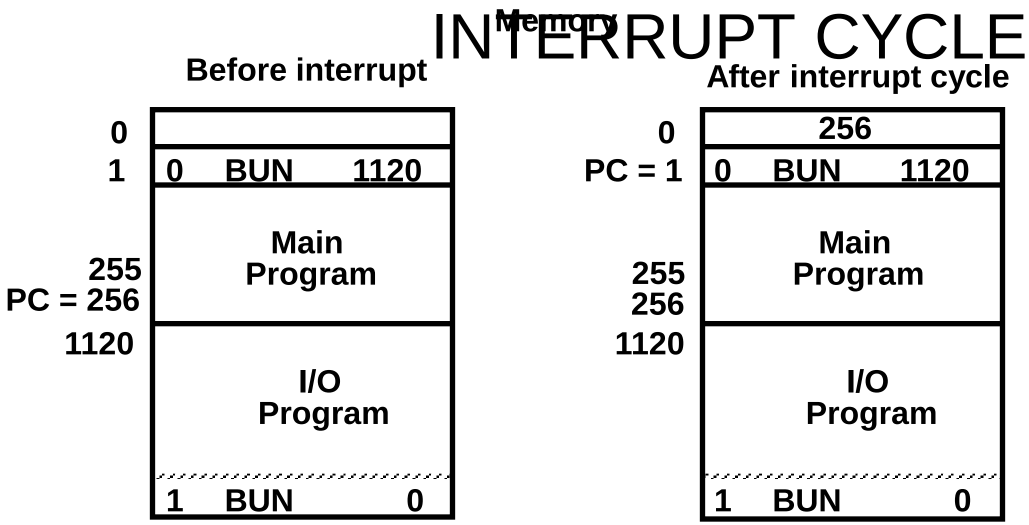
- can be set and cleared by instructions
- when cleared, the computer cannot be interrupted

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)$T_0'T_1'T_2'$

$\Leftrightarrow T_0'T_1'T_2'$ (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 ?