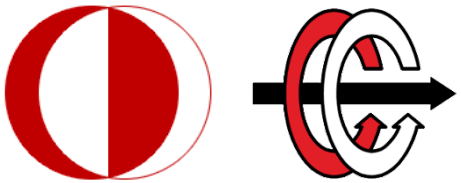


Register Transfer Language (RTL)

Microoperations

- An elementary operation performed on the information stored in one or more registers
- Performed during one clock pulse time
- The functions built into registers are examples of microoperations
  - Shift
  - Load
  - Clear
  - Increment

Register Transfer Level

- Register Transfer Level definition of a computer:
  - Set of registers
  - Data transfers and transformations in the registers
  - Set of allowable microoperations provided by the organization of the computer
  - Control signals that initiate the sequence of microoperations (to perform the functions)

Register Transfer Language (RTL)

- A symbolic language
- Describes the sequences of microoperations among the registers
- There is no standard form
- A convenient tool for describing the internal organization of digital computers
- Can also be used to facilitate the design process of any digital system.

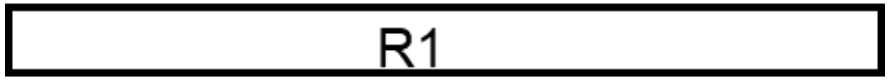
Register Designation

- 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

Register Designation

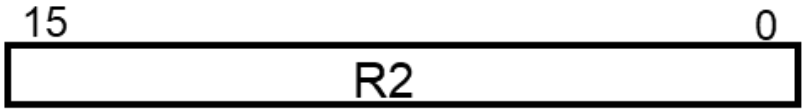
- Showing Registers

Single Diagram

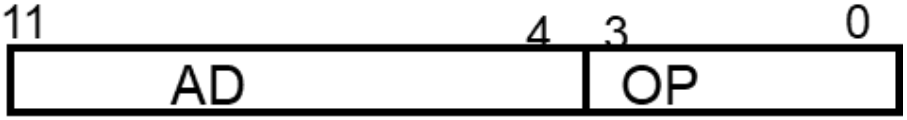


Showing individual bits

Numbering of bits



Subfields



MBR

MBR (OP) or MBR(0-3) refers to the first 4 bits of MBR

Similarly: MBR (AD) or MBR(4-11)

Basic Symbols Register Transfer

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

Register Transfer: Parallel Transfer

- R2 ←R1
- Microoperation, in one clock
- Load/Copy the contents of register R1 into register R2
- Contents of R1 are not altered by copying (loading) them to R2

Register Transfer: Control Function

- Control signal
- If the signal is 1, the action takes place
- This is represented as:

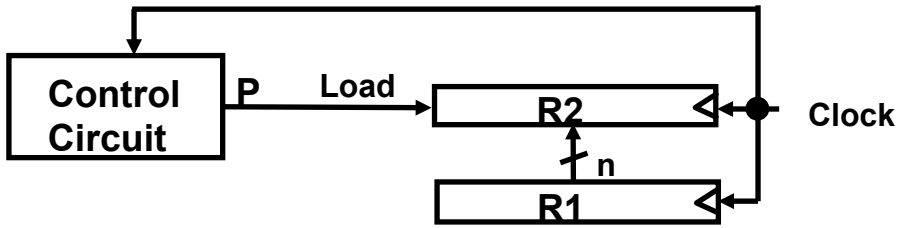
  P: R2 ←R1

  - if (P = 1) then (R2 ← R1)
- Q': A←B
  - if (Q = 0) then (A ← B)

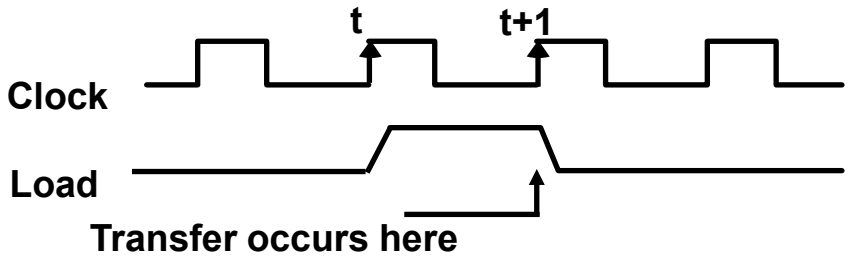
Register Transfer: Control Function

- P: R2 ←R1

Block diagram



Timing diagram



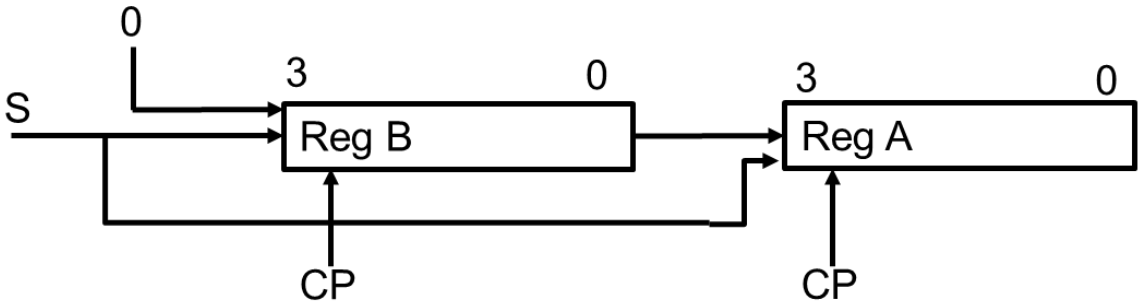
The same clock controls the circuits that generate the control function and the destination register

Register Transfer: Control Function

- If two or more operations are to occur simultaneously, they are separated with commas
- P: R3 ← R5, MAR ← IR
- Both microoperations are in the same clock pulse

Register Transfer: Serial Transfer

- Both source and destination registers are shift registers

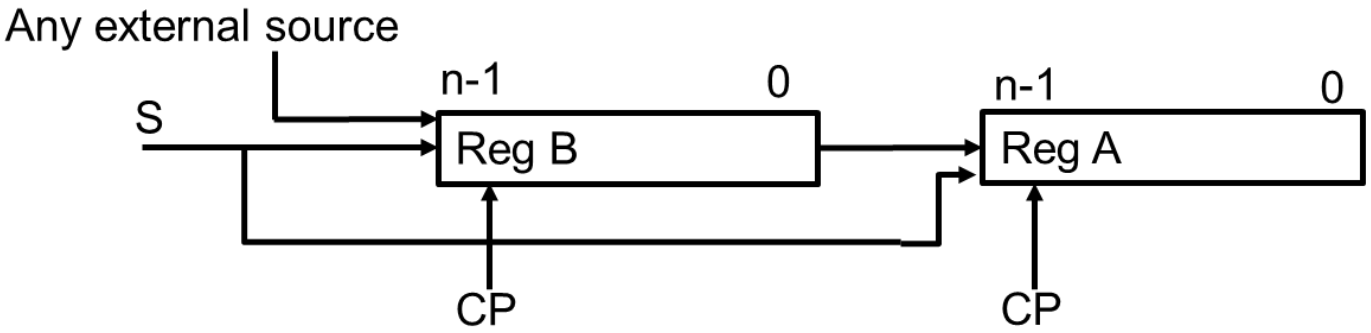


- S: $A_3 \leftarrow B_0$, $B_3 \leftarrow 0$, $A_{i-1} \leftarrow A_i$, $B_{i-1} \leftarrow B_i$, i=1,2,3
- For a complete transfer S must remain 1 for 4 clock pulses➔ Word time
- Serial Systems: a microoperation can be defined as an operation that takes a word-time for execution.
- S: A←B, B←0 can be the microoperation to define serial transfer

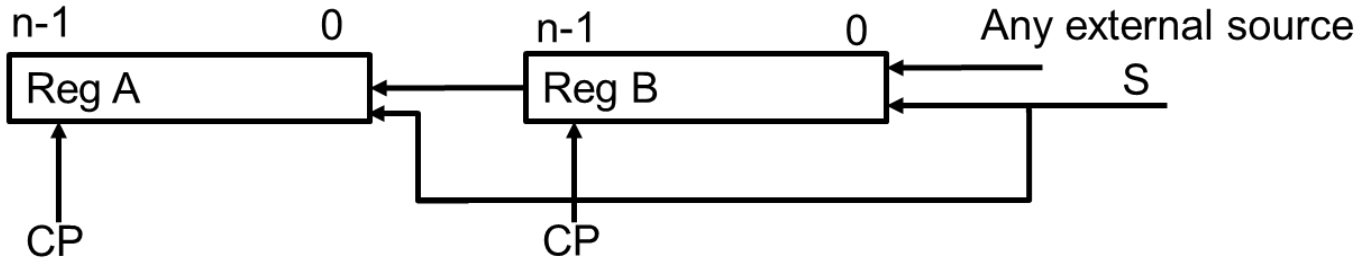
Shift Microoperations

1. Serial Transfer shift with an external source

- S: shrA, shrB, $B_{n-1} \leftarrow$ external_source, $A_{n-1} \leftarrow B_0$



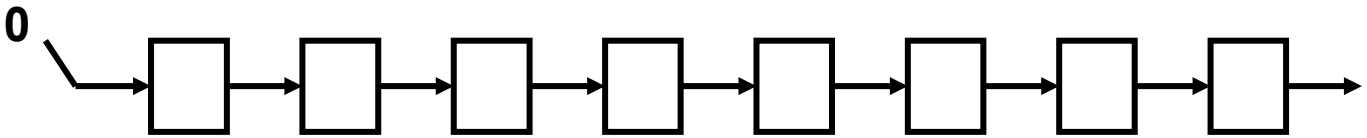
- S: shlA, shlB, $B_0 \leftarrow$ external_source, $A_0 \leftarrow B_{n-1}$



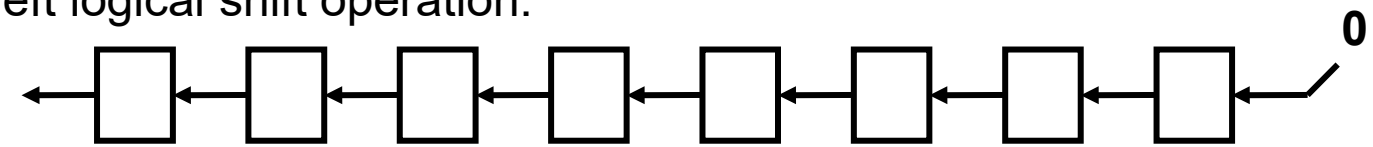
Shift Microoperations

2. Logical shift: the serial input is a 0.

A right logical shift operation:



A left logical shift operation:

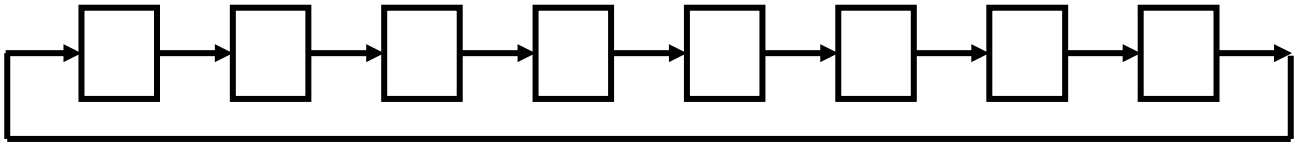


R2 ← shr R2
R3 ← shl R3
E: shl A

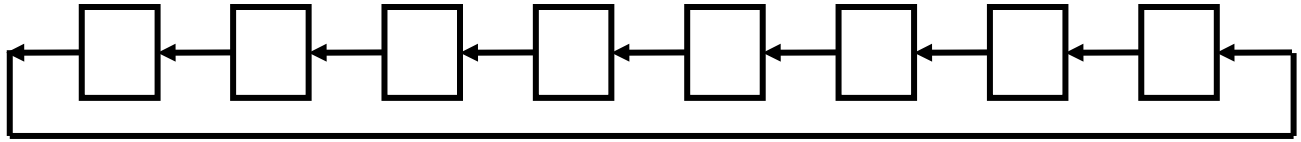
Shift Microoperations

3. Circular shift

A right circular shift operation:



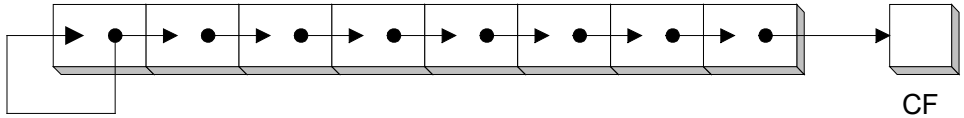
A left circular shift operation:



Examples:
R2 ← cir R2 (circular shift right)
R3 ← cil R3 (circular shift left)
P: cil A, cir B

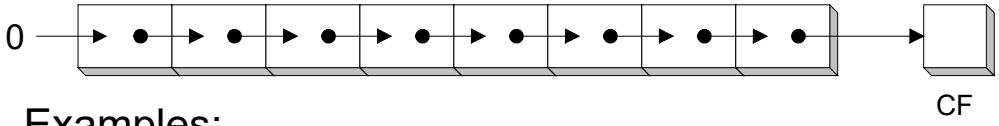
Shift Microoperations

4. Arithmetic shift:
   - Signed numbers
   - Sign bit is unchanged after the shift



Arithmetic right shift: divides the signed number by 2

Compare to logical shift: fills the newly created bit position with zero



Examples:

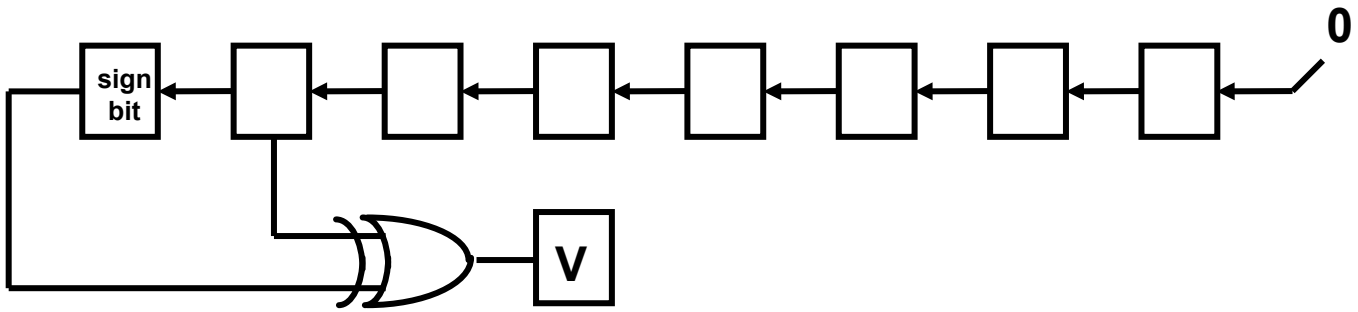
R2 ← *ashr* R2 (arithmetic shift right)

R3 ← *ashl* R3 (arithmetic shift left)

Shift Microoperations

- Arithmetic left shift:
  - multiplies the signed number by 2
  - must be checked for the overflow

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



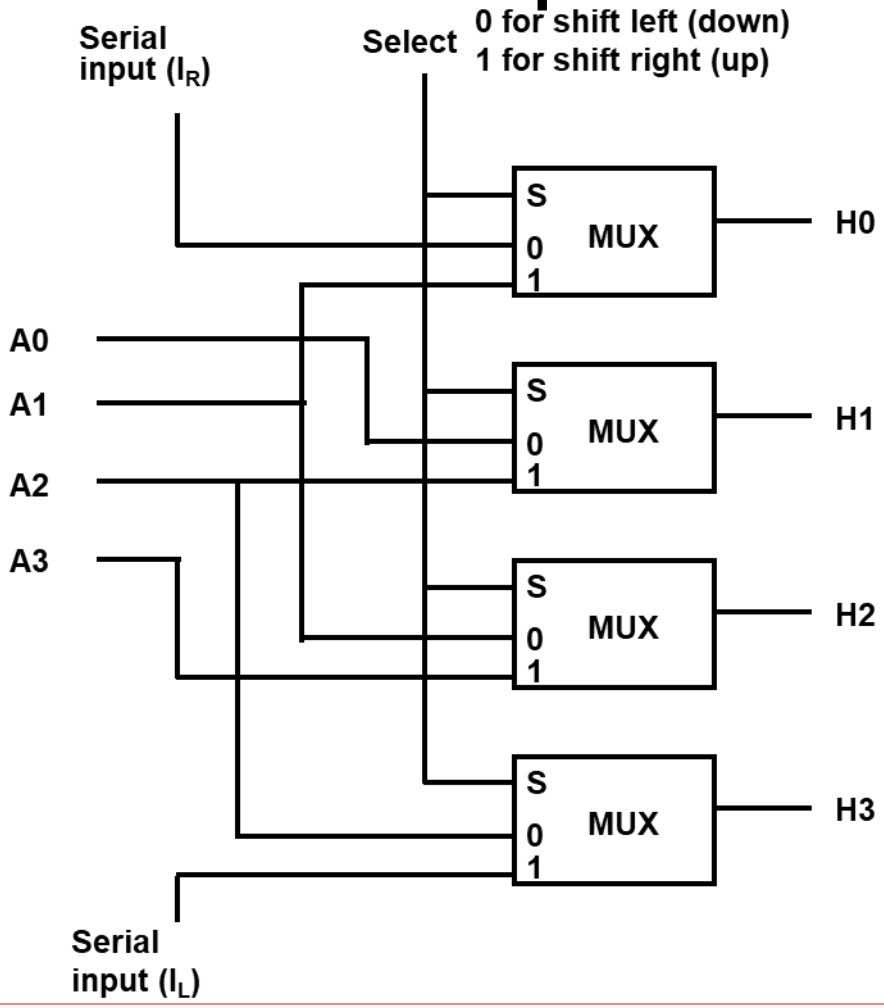
Rules for Arithmetic Shift

- Sign must be the same
- For positive numbers: all added bits are 0.
- For negative numbers:
  - (Signed magnitude) All added bits are 0.
  - (2's complement) AShl: Added bits are 0.
  - (2's complement) AShr: Added bits are 1.
  - (1's complement) All added bits are 1.

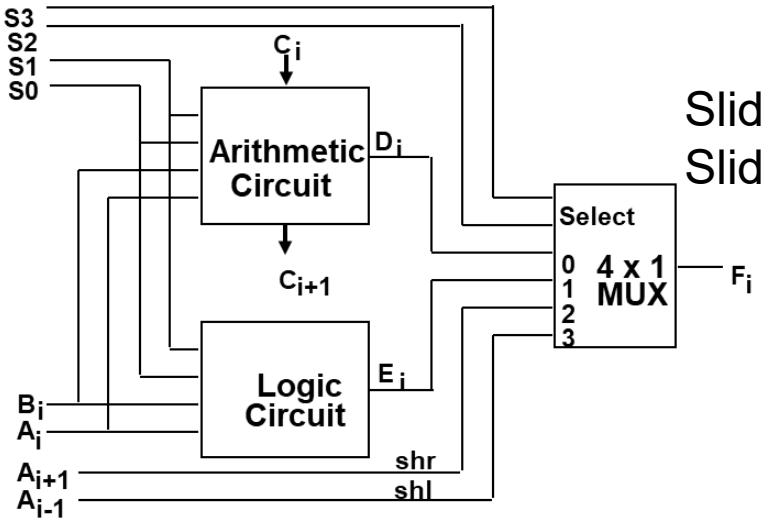
Examples for Arithmetic Shift

- Signed magnitude: Sign bit does not change, added bits are 0
  - A:1100 (-4) ashrA:1010 (-2)
  - A:1010 (-2) ashlA:1100 (-4)
- Signed 2's complement:
  - A:1010 (-6) ashrA:1101 (-3) Added bits are 1.
  - A:1101 (-3) ashlA:1010 (-6) Added bits are 0.
  - B:1010 (-6) ashlB:1100 (-4) OVERFLOW.

Hardware Implementation of Shift Microops



Arithmetic Logic and Shift Unit



Slide 43: Arithmetic Circuit
Slide 46: Logic Circuit

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

Connecting Registers

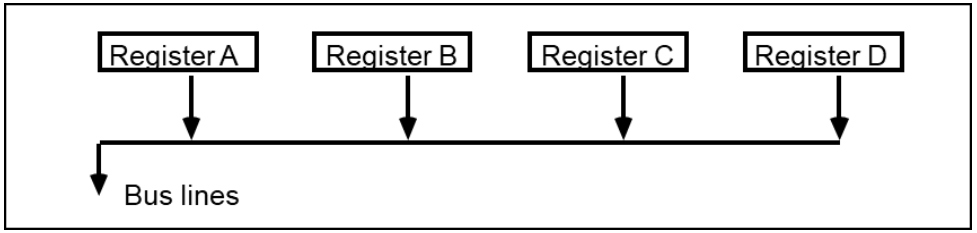
- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers
- To completely connect n registers:
  - n(n-1) lines
  - $O(n^2)$ cost

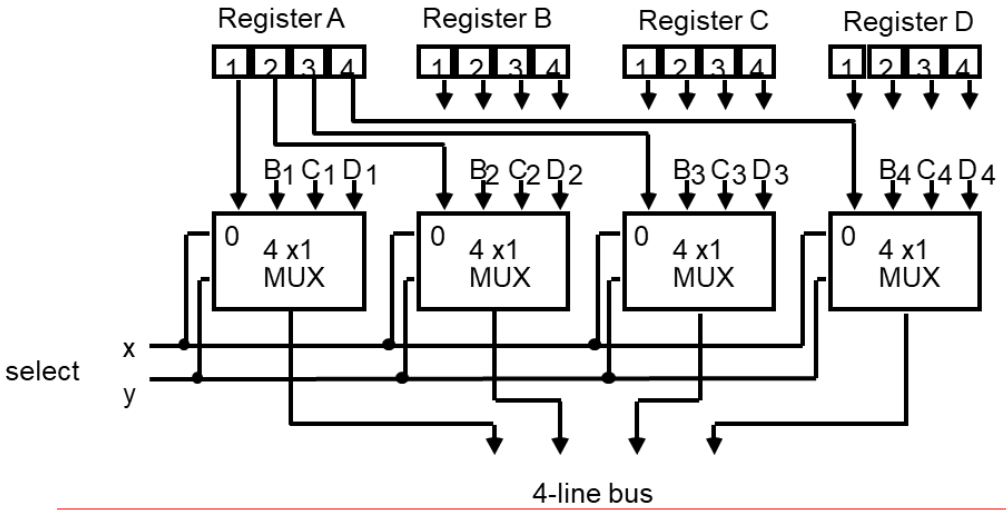
Connecting Registers

- Solution:
  - Have one centralized set of circuits for data transfer
  - Have control circuits to select which register is the source, and which is the destination
- Bus is a path (of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

Register to Bus Using MUX

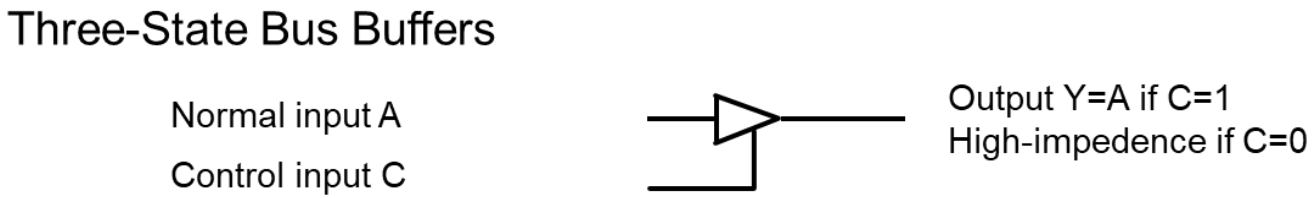
- BUS←RegisterA, RegisterB←BUS
- Equivalent: RegisterB←RegisterA





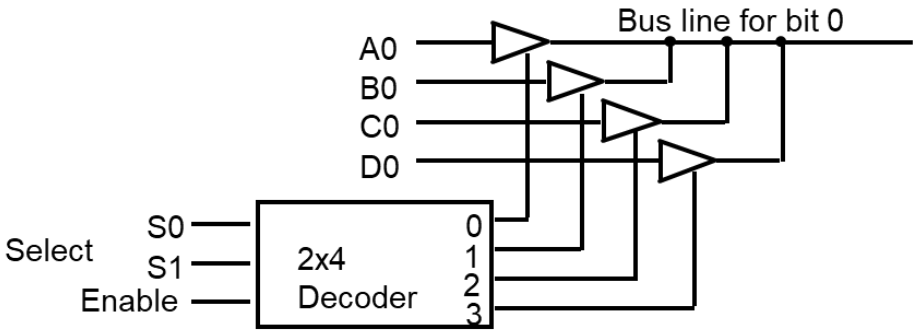
- k registers of n bits:
  - n MUX
  - Each MUX: kx1
  - MUX selects determine which register's content is on the bus
- Activate the load control input of destination register

Register to Bus Using Three-state Bus Buffers



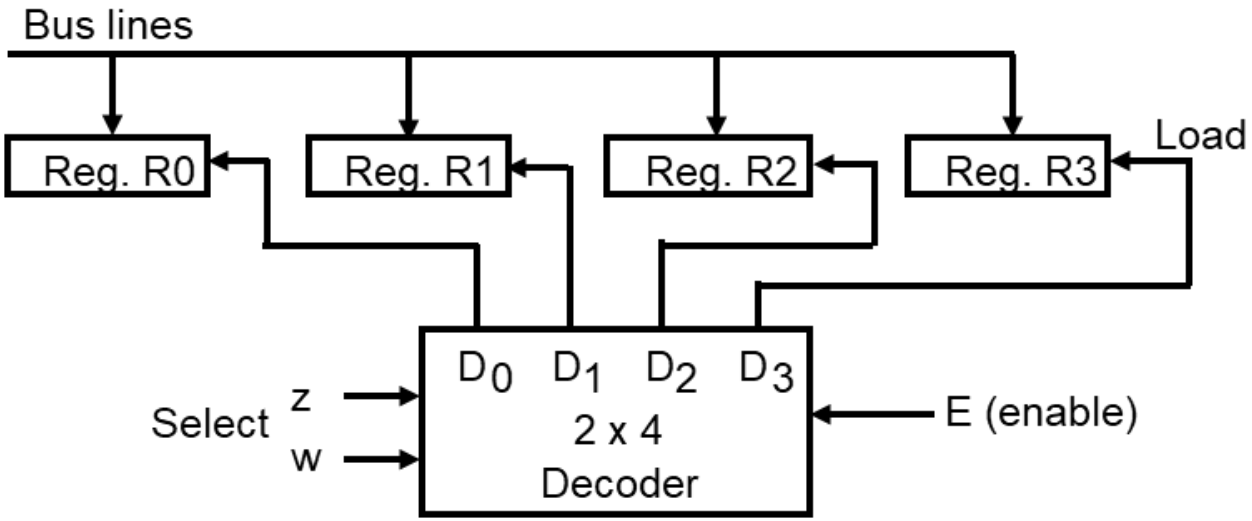
- High impedance state enables:
  - Connecting a large number of three-state gates with wires to form a common bus line

- Enable=0: All outputs are high impedance
- Enable=1: Only selected buffer is active
- This is actually a 4x1 MUX!



Bus line with three-state buffers

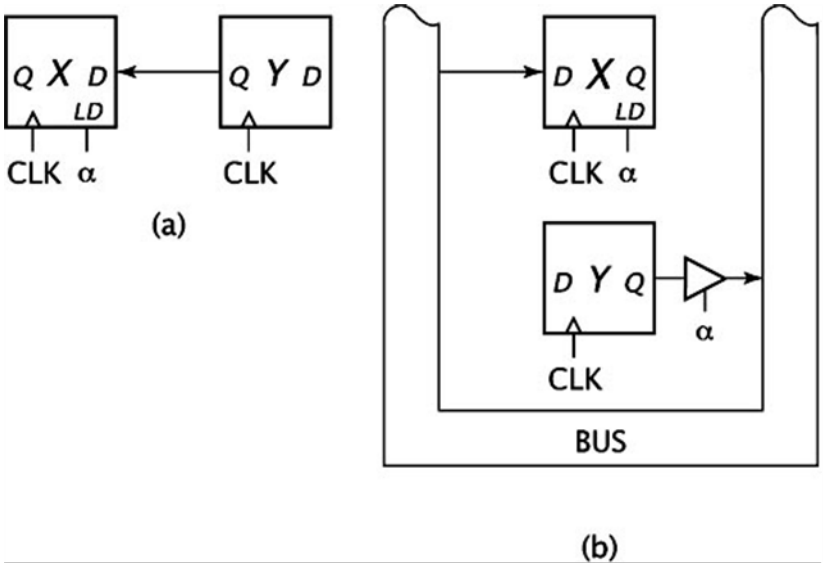
Bus to Register



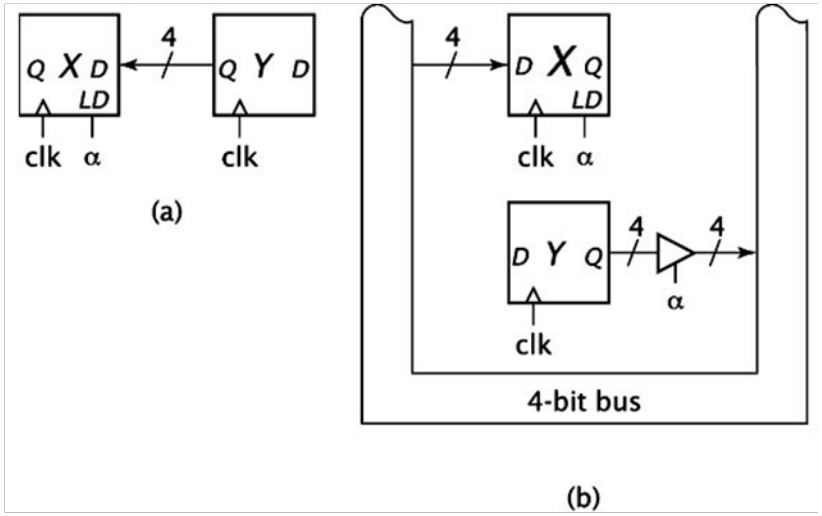
- One decoder is needed to select the destination register

Examples

Single bit

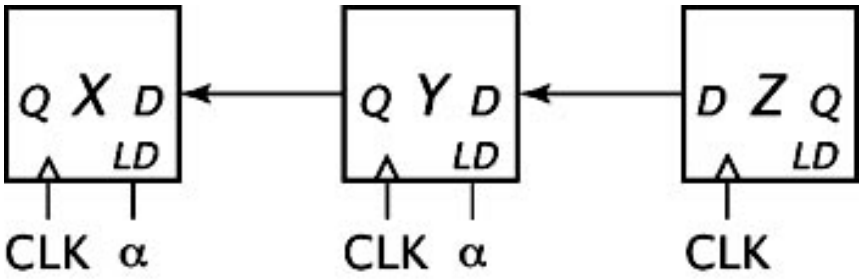


Multi bit

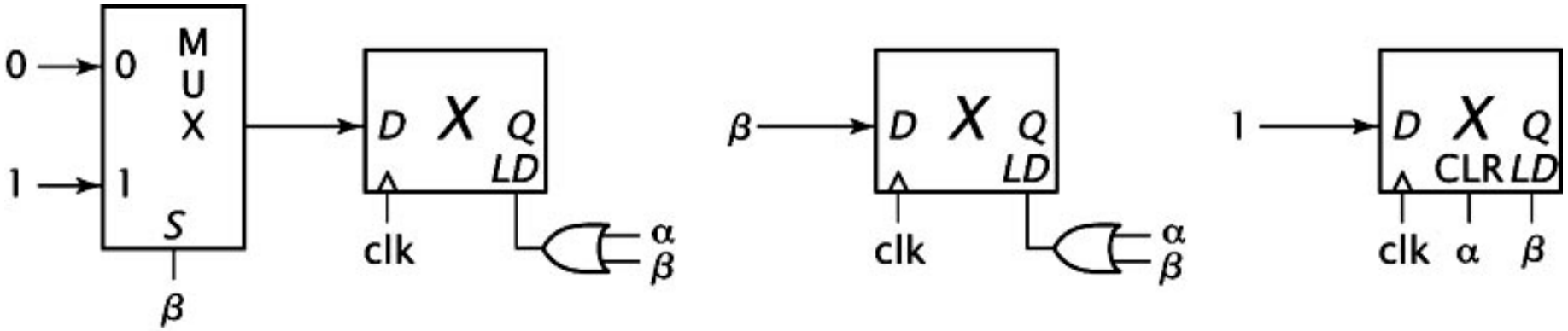


α: X ← Y

Examples



α: X ← Y, Y ← Z Simultaneous Data Transfers



(a) (b) (c)

α: X ← 0

β: X ← 1

Loading Constant Values into Registers

Memory

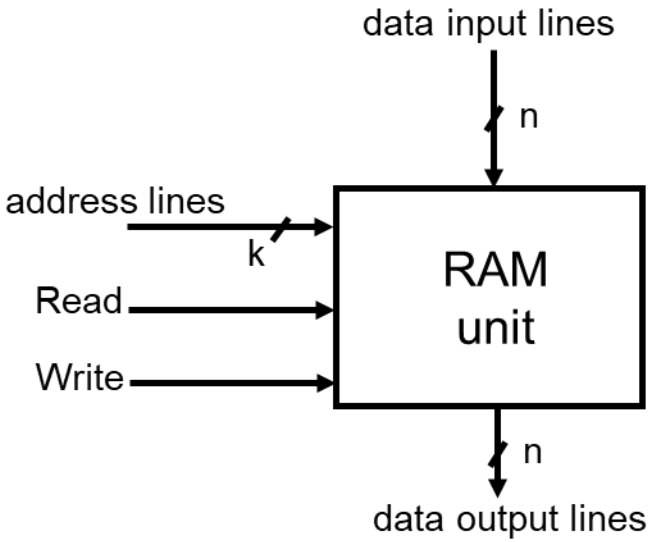
- A Memory Unit:
  - Collection of storage cells
  - Associated circuits to transfer the information in and out of storage
  - Stores binary information in groups of bits called words

Memory

- Each of the words is indicated by an address
- These addresses range from 0 to r-1 (for r word memory)
- Each word can hold n bits of data
- Random Access Memory (RAM): Locating any word in the memory requires the same time

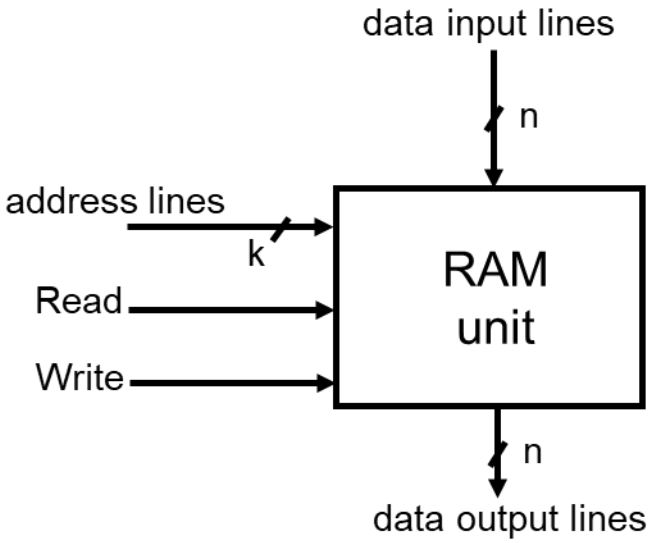
Memory Organization

- Assume that a RAM unit with n bit words contains r = 2^k words. Then it needs the following connections:
  - n data input lines
  - n data output lines
  - k address lines
  - a read control line
  - a write control line



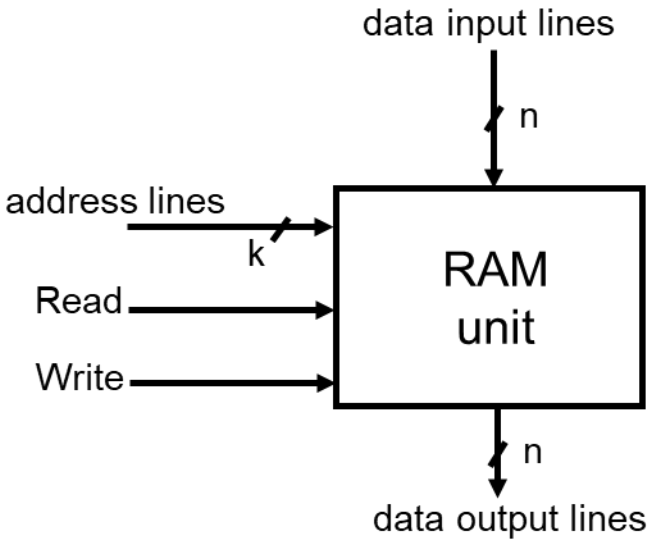
Memory Organization

- 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 locations (referencing).



Memory Organization

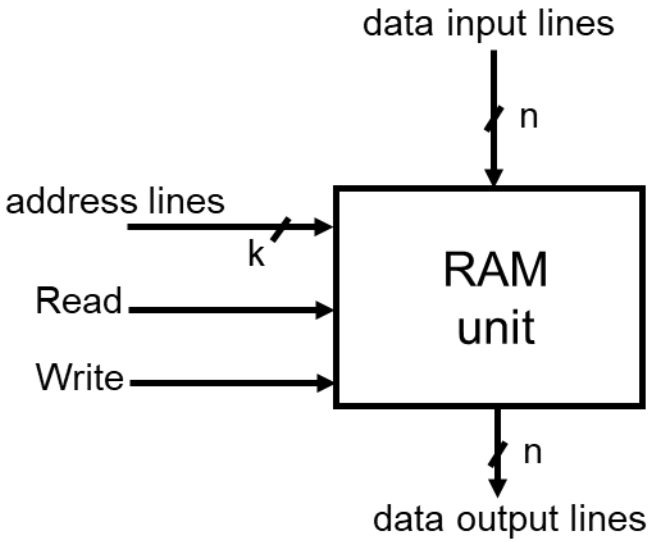
- Selection of a specific word:
  - Apply k-bit binary address to the address lines
  - The address applied outside is decoded by a decoder inside the memory and the corresponding word in the memory is *enabled*



Memory Write

- Transfer-in operation

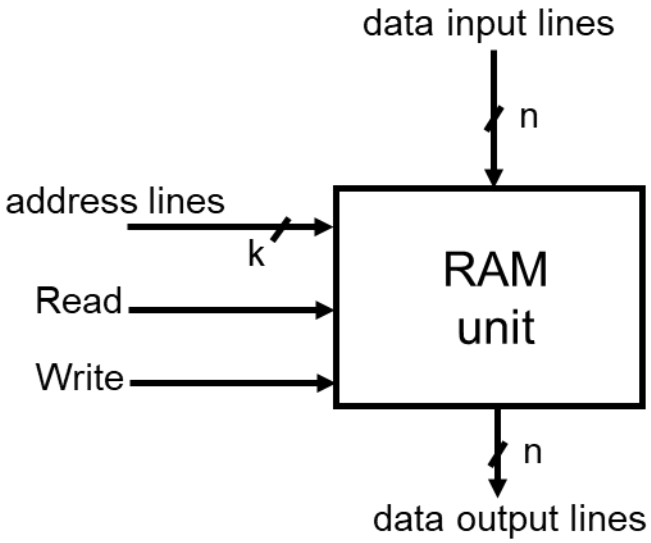
1. Apply the binary address of the desired word into the address lines
2. Apply the data bits that must be stored in memory into the data input lines
3. Activate the Write input



Memory Read

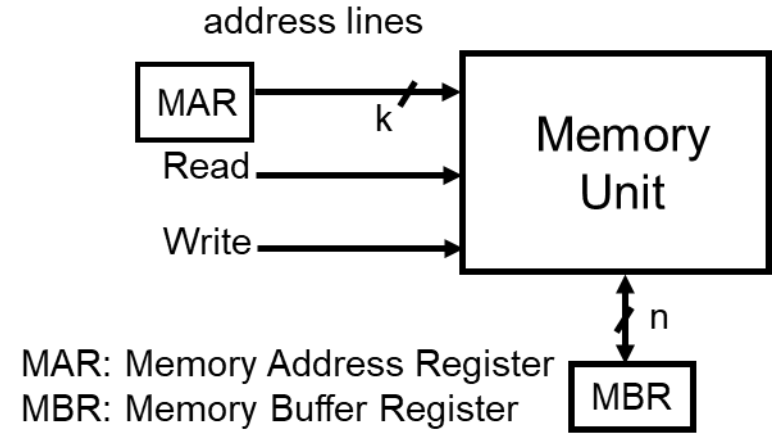
- Transfer-out operation

1. Apply the binary address of the desired word into the address lines
2. Activate the Read input
3. The output lines have the content of the selected word, the selected word is not changed after read



Memory Read in RTL

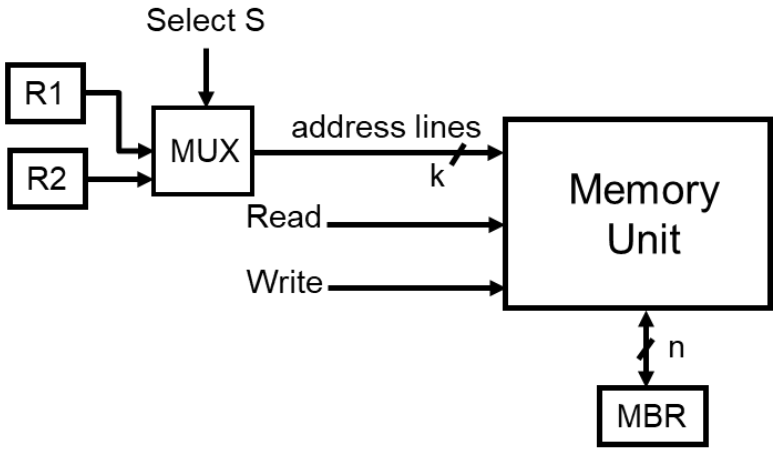
- Read Operation:
  - MBR←M or in long form: MBR←M[MAR]
- The contents of MAR get sent to the memory address lines
- A read (= 1) signal gets sent to the memory unit
- The contents of the specified address are put on the memory's output data lines
- These get sent over the bus to be loaded into MBR



Other namings:
AR (instead of MAR): Address Register
DR (instead of MBR): Data Register

Memory Transfer in RTL

- Read:
- S: MBR←M[R1]
- S': MBR←M[R2]



Summary of Operations

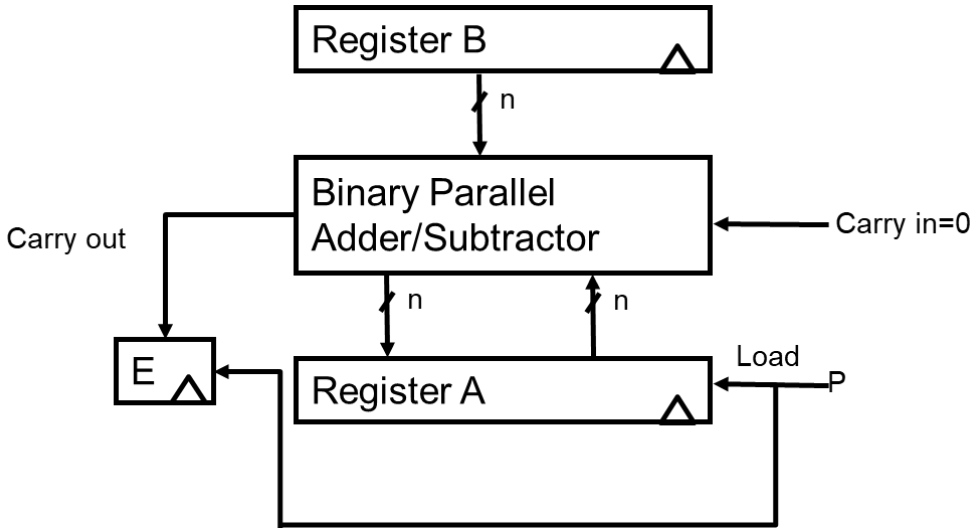
A ← B	Transfer content of reg. B into reg. A
AR ← DR(AD)	Transfer content of AD portion of reg. DR into reg. AR
A ← constant	Transfer a binary constant into reg. A
ABUS ← R1, R2 ← ABUS	Transfer content of R1 into bus A and, at the same time, transfer content of bus A into R2
AR	Address register
DR	Data 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

Arithmetic Microoperations

- R1←R2+R3
- R1←R1+R2
- Uses two or three registers
- Content of the destination register is modified according to the operation

Arithmetic Microoperations

- P: EA←A+B



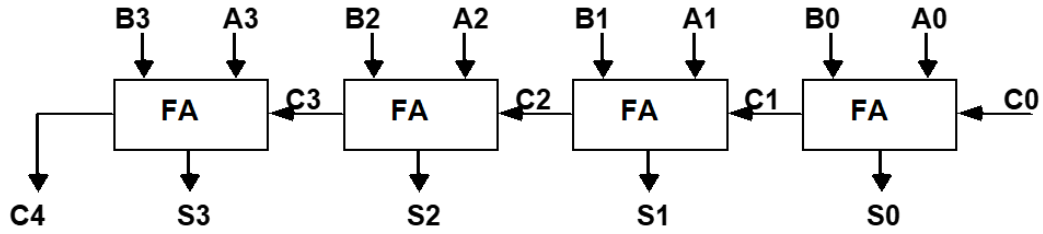
- When P=1 the transfer happens in the next clock pulse

Arithmetic Microoperations Summary

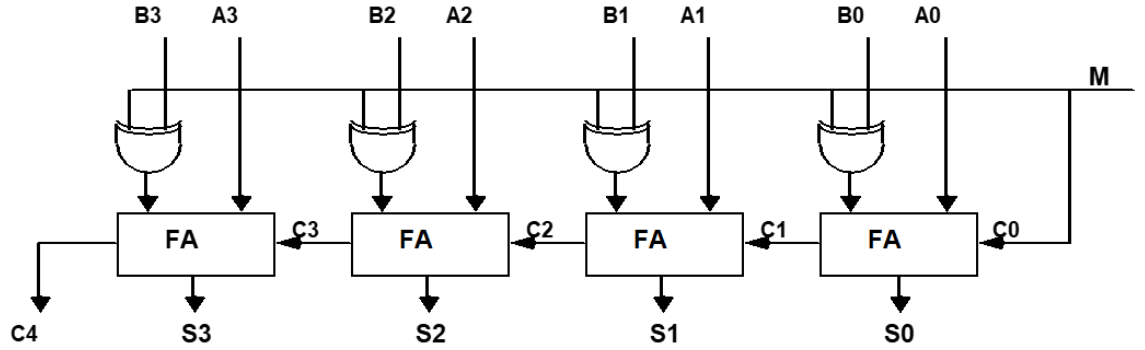
R3 ← R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ← R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ← R2’	Complement the contents of R2
R2 ← R2’+ 1	2's complement the contents of R2 (negate)
R3 ← R1 + R2’+ 1	Subtraction
R1 ← R1 + 1	Increment
R1 ← R1 - 1	Decrement

Binary Adder/Subtractor/Incrementer

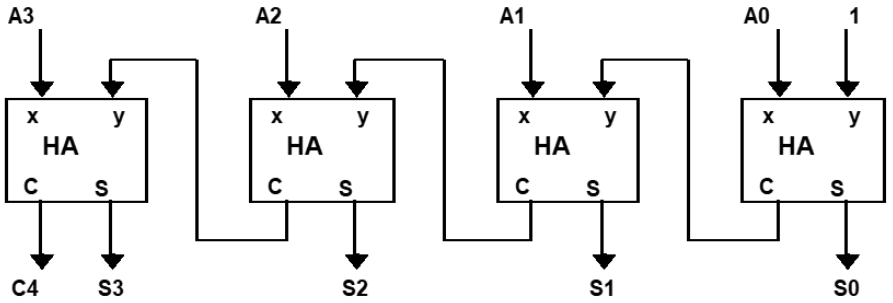
Binary Adder



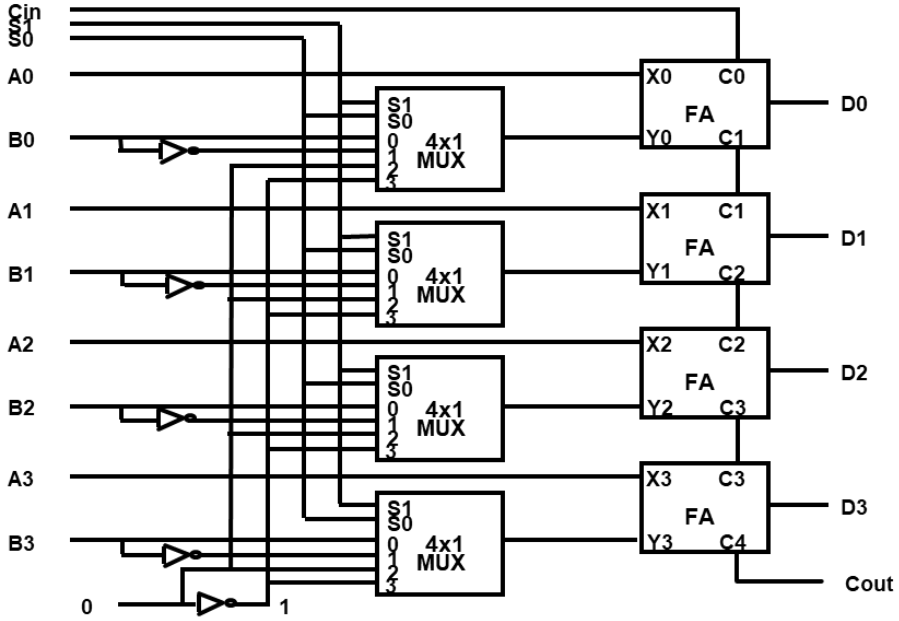
Binary Adder-Subtractor



Binary Incrementer



Example Arithmetic Circuit



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

Logic Microoperations

- Specify binary logic operations on the strings of bits in registers
  - Logic microoperations are bit-wise operations, i.e., they work on 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

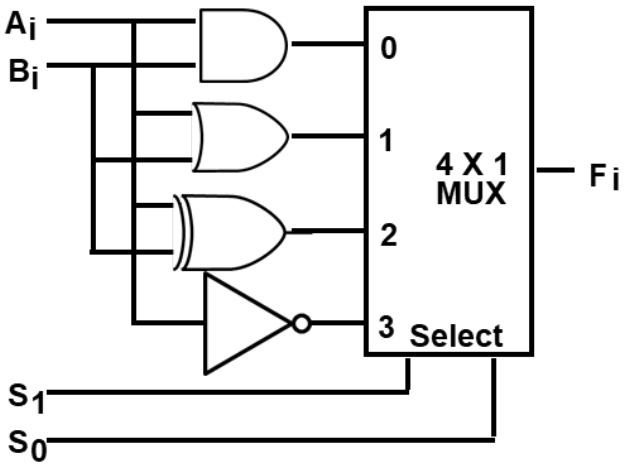
Logic Microoperations

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

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

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

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

Example

Assume a register A is defined

P_1: $A \leftarrow 0$

P_2: $A \leftarrow \bar{A}$

P_3: $A \leftarrow A \vee B$

P_4: $A \leftarrow A \wedge B$

P_5: $A \leftarrow A \oplus B$

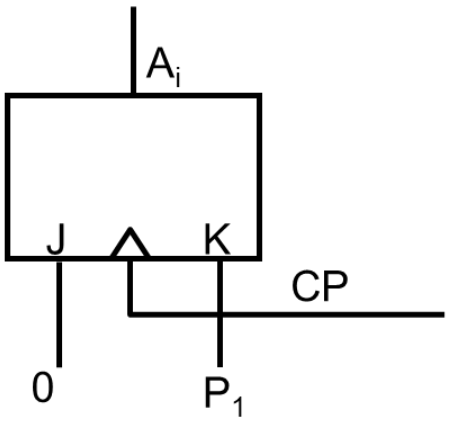
Design the register A using JK FF's

Design a typical cell A_i

Separate Design

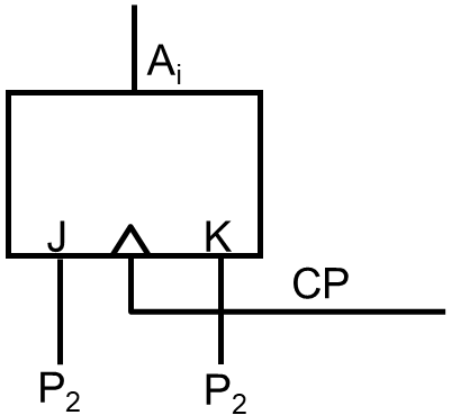
PS		NS		
A_i	B_i	$\bar{A}_i$	J	K
0	0	0	0	X
0	1	1	1	X
1	0	1	X	0
1	1	1	X	0

P_1: $A \leftarrow 0$



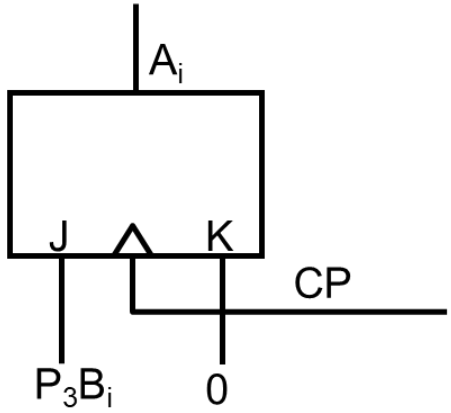
$JA_i=0$, $KA_i=P_1$

P_2: $A \leftarrow \bar{A}$



$JA_i=KA_i=P_2$

P_3: $A \leftarrow A \vee B$

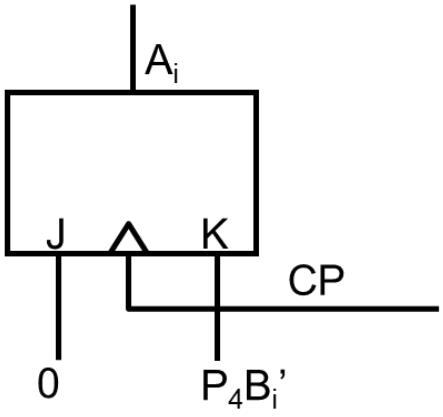


$JA_i=P_3B_i$, $KA_i=0$

Separate Design

PS		NS		
A_i	B_i	$\overline{A_i}$	J	K
0	0	0	0	X
0	1	0	0	X
1	0	0	X	1
1	1	1	X	0

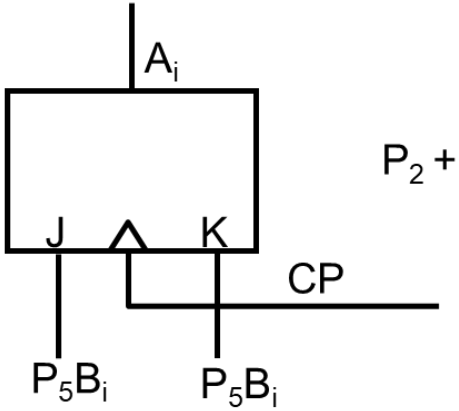
P_4: $A \leftarrow A \wedge B$



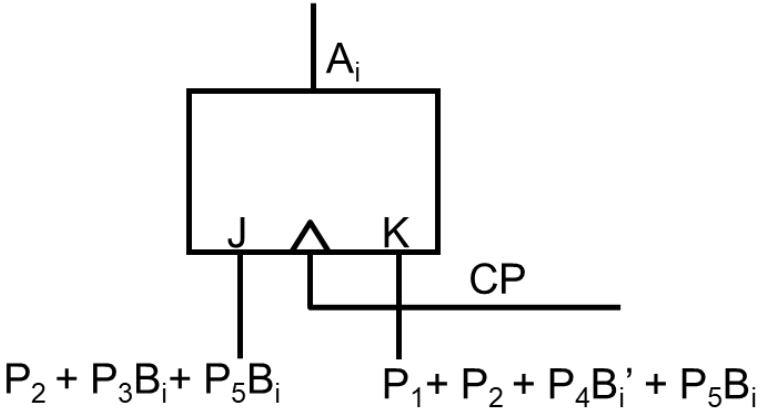
$JA_i=0$, $KA_i=P_4B_i'$

PS		NS		
A_i	B_i	$\overline{A_i}$	J	K
0	0	0	0	X
0	1	1	1	X
1	0	1	X	0
1	1	0	X	1

P_5: $A \leftarrow A \oplus B$



$JA_i=P_5B_i$, $KA_i=P_5B_i$

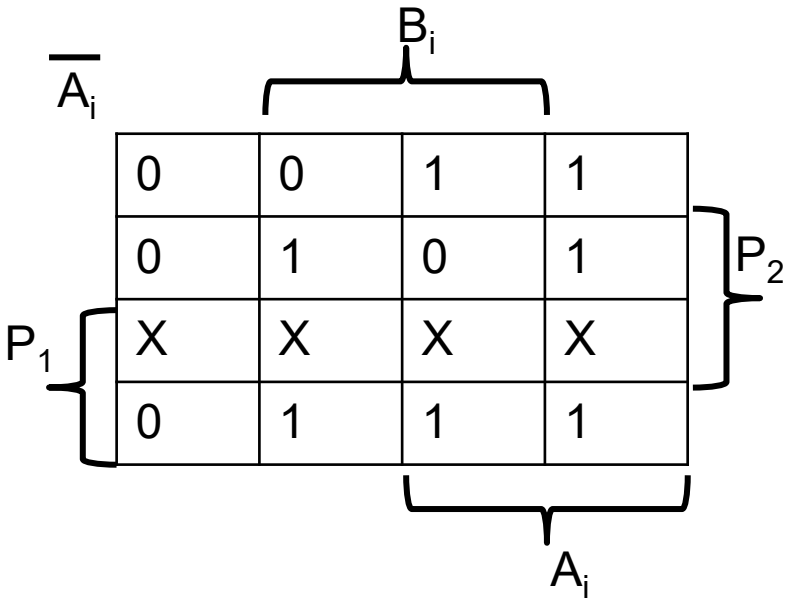


Example

P_1: A←A∨B

P_2: A←A⊕B

P_1	P_2		A_i	B_i		$\overline{A}_i$	JA_i	KA_i
0	0		X	X		A_i	0	0
0	1		0	0		0	0	X
0	1		0	1		1	1	X
0	1		1	0		1	X	0
0	1		1	1		0	X	1
1	0		0	0		0	0	X
1	0		0	1		1	1	X
1	0		1	0		1	X	0
1	0		1	1		1	X	0
1	1		X	X		X	X	X



$JA_i=P_1B_i+ P_2B_i$, $KA_i=P_2B_i$

Formal Register Design

Application of Logic Microoperations

- Logic microoperations can be used to manipulate individual bits or a portion of a word in a register
- Consider the data in a register A. Assume that in another register, B, there 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

$$\begin{array}{llll} 1\ 1\ 0\ 0 & A_t & & \\ 1\ 0\ 1\ 0 & B & & \\ 1\ 1\ 1\ 0 & A_{t+1} & (A \leftarrow A + B) & \end{array}$$

- 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

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

- 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

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

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

Mask

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

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

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

Clear

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

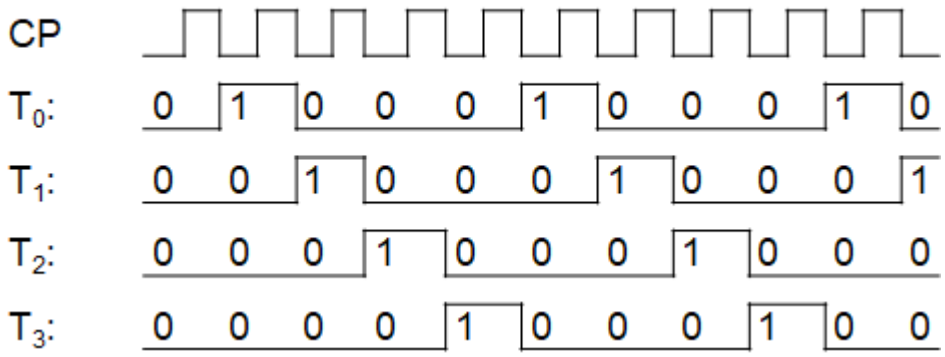
1 1 0 0 A_t

1 0 1 0 B

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

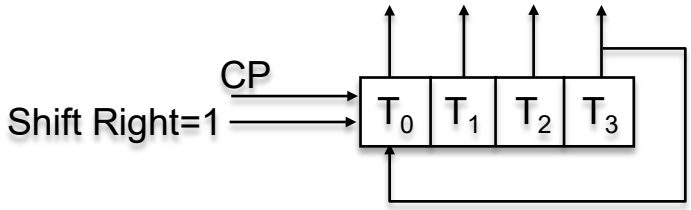
Control Functions and Timing

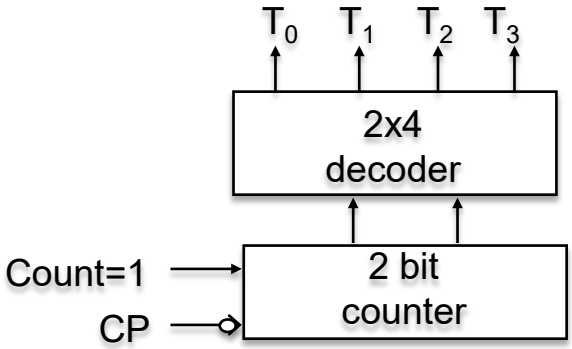
- We have to define non-overlapping timing signals T_i synch. with the CP.



If a computer needs to read a word from the memory at time T1
T_1: MBR←M

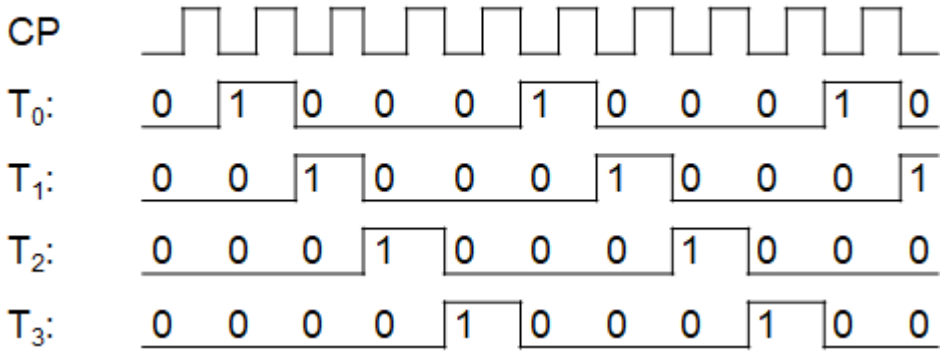
At time T_1 if P=1 or at time T_3 if R=0 the content of B is loaded into A
$PT_1+R'T_3$: A←B



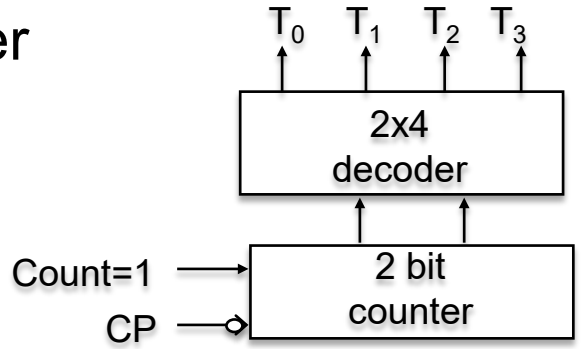


Timing Signals: Multiphase

- Multiphase clock pulses
  - Clock pulse is generated in multiples of the actual clock pulse



- Realization 2: 2 bit counter and 2x4 decoder
  - Decoder outputs are turned on sequentially based on the counter value



Control Functions and Timing Signals

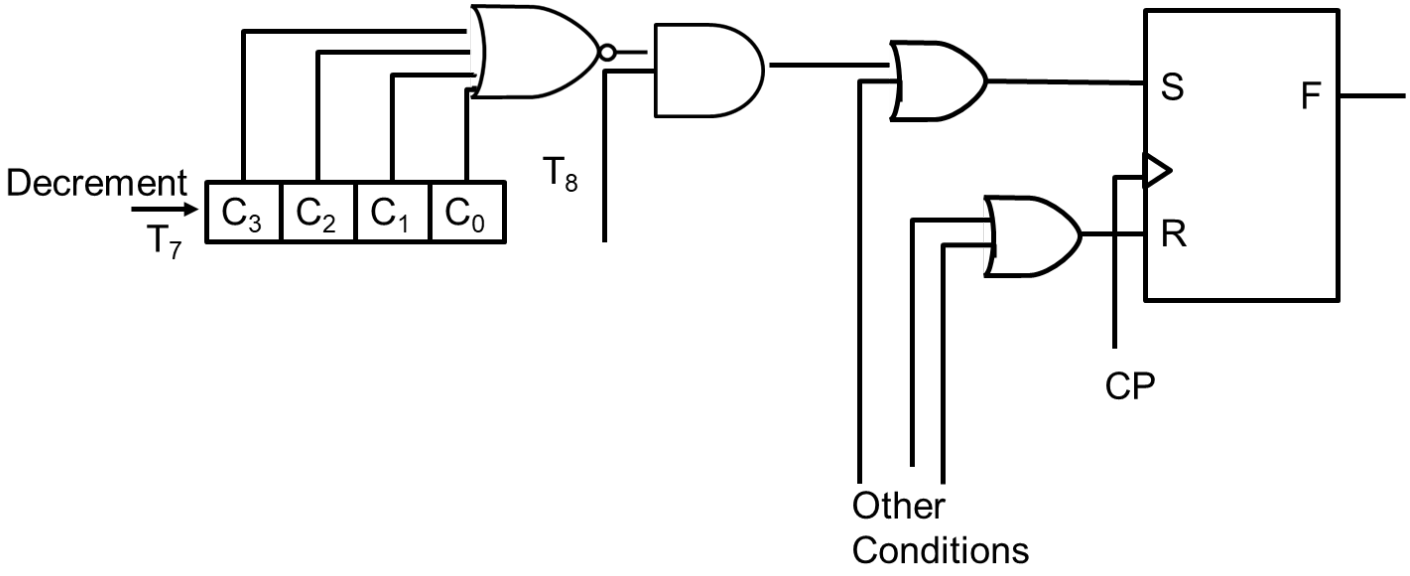
Symbolized as

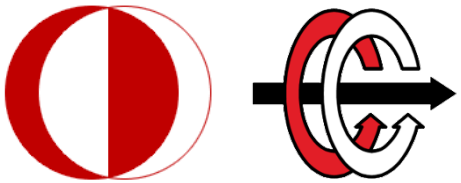
PT_3: if (condition) then (microoperation)

Example:

T_7: C←C-1

T_8: if (C=0) then F←1





Register Transfer Language (RTL)