CSM51A Discussion #8

Hyunjin Kim

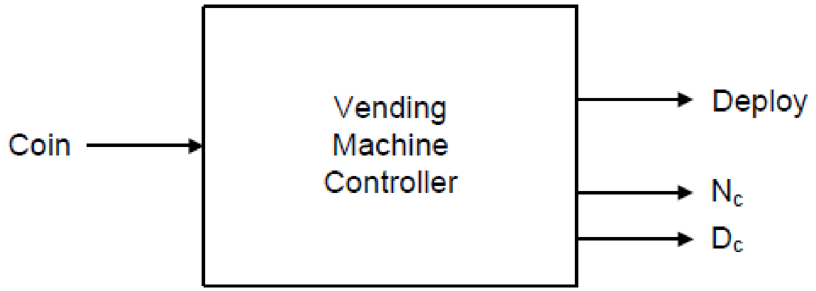
Outline

- Review HW 6 & 7
- Vending Machine Example
- Registers / Shift Registers
- Memory System
  - Memory with FFs
  - SRAM
  - DRAM
- Instruction Set Architecture

Our goal is to design a vending machine which sells stamps. The price of a stamp is 35 cents (for the sake of the problem). The machine accepts only nickels (5 cents), dimes (10 cents), and quarters (25 cents).

When the total value of coins is equal to or larger than the price of the stamp, the machine deploys the stamp, and returns any change as necessary. The change is given in nickels or dimes, and is returned in a way such that the total number of coins returned is the smallest possible. For example, if the amount of change is 30 cents, the machine returns 3 dimes, and not a mixture of dimes and nickels which will result in a higher coin count.

The machine's control module looks like the diagram below. It has an input *Coin* which denotes the type of coin deposited, and has three outputs: *Deploy*, which is 1 when the machine needs to deploy a stamp and 0 otherwise, N_c, number of nickels to return as change, and D_c, number of dimes to return as change.



1. What is the minimum number of states necessary for the control module? What would each state represent? (*Hint: To find the minimum number of states, first write any state machine which has the functionality that you want, and try to reduce the number of states afterwards.*)

2. Show the state transition table. The output should be written as a three-digit number, where each digit corresponds to the value of *Deploy*, N_c and D_c, in that order.

Solution One way of representing states would be to have a state for each possible total value of coins that have been inserted into the vending machine up to that time point. As the coin with the minimum value is a nickel, we can go up in multiples of 5. Any input which makes the current total of coins go over the price of the stamp makes the machine output the stamp and appropriate change, and sends the machine to the initial state.

For this we will need the following states: $S_{init}, S_5, S_{10}, S_{15}, S_{20}, S_{25}$ and S_{30}.

To make sure that this is the minimum number of necessary states, we apply the state minimization method. First, we need to write the state transition table as shown. The value of the input $Coin$ is equal to N for nickel, D for dime, and Q for quarter. The three output digits are the values of $Deploy, N_c$ and D_c, in the same order.

	$Coin = N$	$Coin = D$	$Coin = Q$
S_{init}	$S_5, 000$	$S_{10}, 000$	$S_{25}, 000$
S_5	$S_{10}, 000$	$S_{15}, 000$	$S_{30}, 000$
S_{10}	$S_{15}, 000$	$S_{20}, 000$	$S_{init}, 100$
S_{15}	$S_{20}, 000$	$S_{25}, 000$	$S_{init}, 110$
S_{20}	$S_{25}, 000$	$S_{30}, 000$	$S_{init}, 101$
S_{25}	$S_{30}, 000$	$S_{init}, 100$	$S_{init}, 111$
S_{30}	$S_{init}, 100$	$S_{init}, 110$	$S_{init}, 102$

With this table, we shall try to minimize the states.

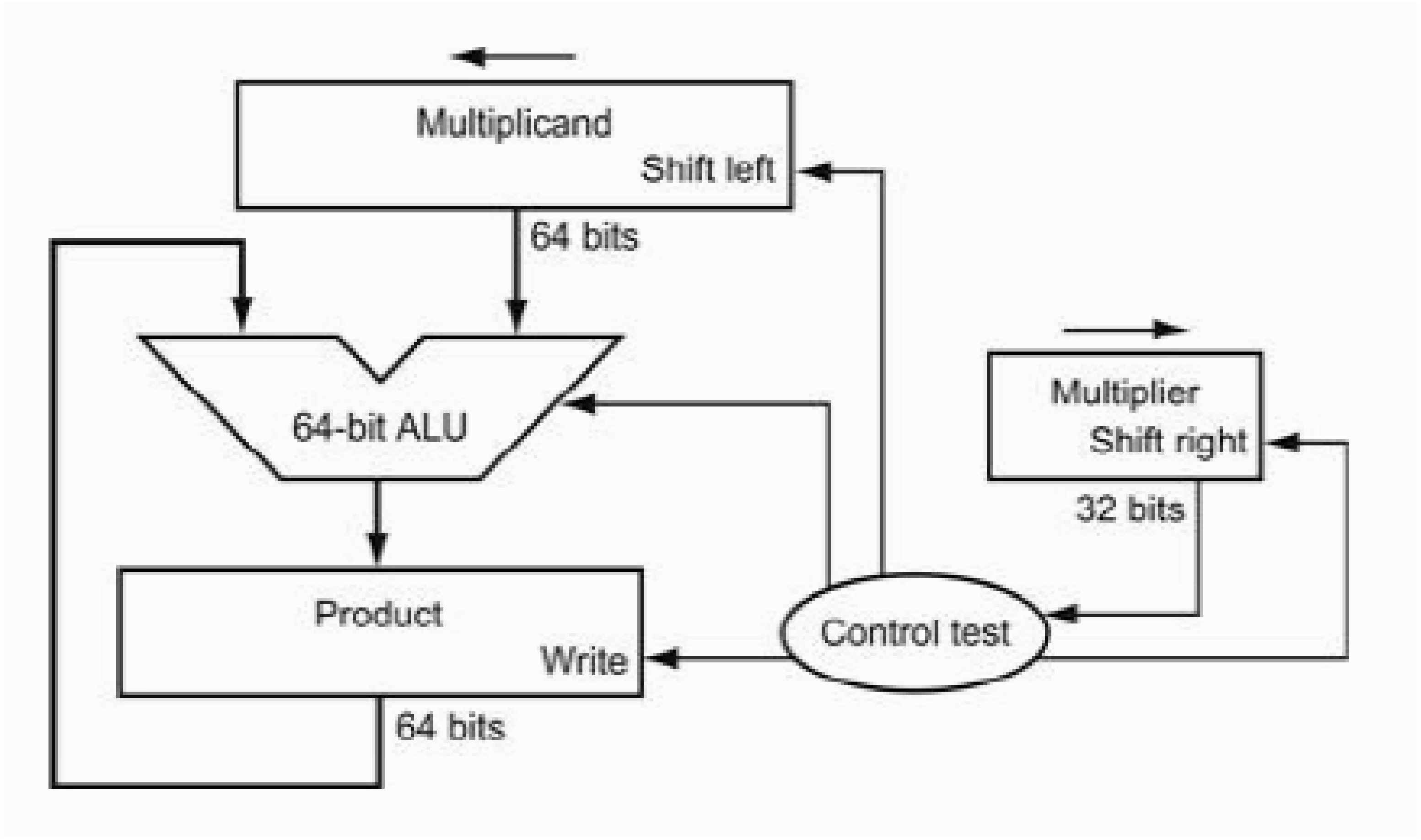
$P_1 = \{S_{init}, S_5\}, \{S_{10}\}, \{S_{15}\}, \{S_{20}\}, \{S_{25}\}, \{S_{30}\}$

	group 1		group 2	group 3	group 4	group 5	group 6
	S_{init}	S_5	S_{10}	S_{15}	S_{20}	S_{25}	S_{30}
N	1	2	3	4	5	6	1
D	2	3	4	5	6	1	1
Q	5	6	1	1	1	1	1

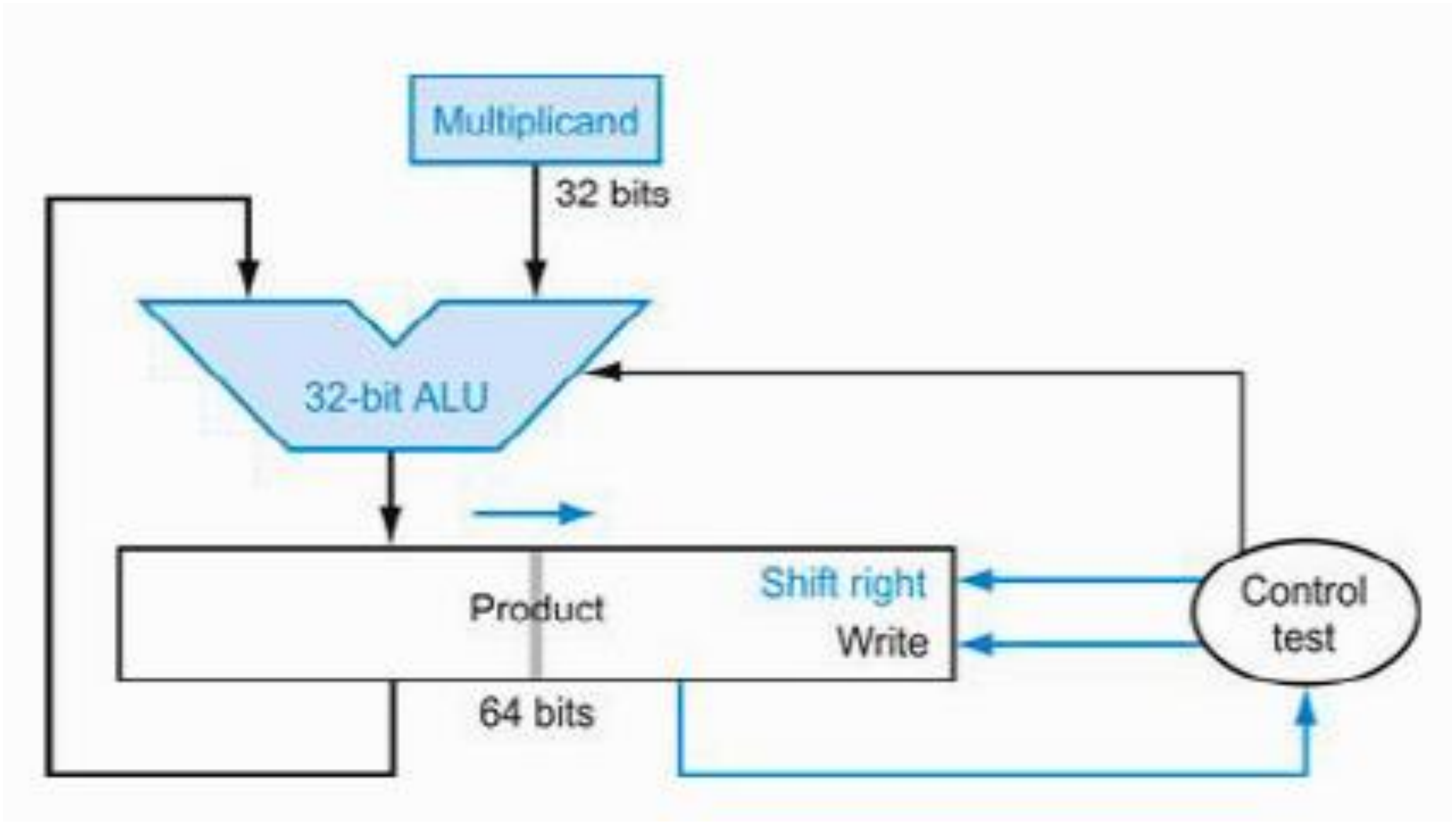
Now we see that $P_2 = \{S_{init}\}, \{S_5\}, \{S_{10}\}, \{S_{15}\}, \{S_{20}\}, \{S_{25}\}, \{S_{30}\}$. Since all states are partitioned, it is not possible to reduce any states and we know that our state set is the minimum set.

The minimum number of states for the vending machine is 7. Each state represents the total value of coins that have been inserted into the vending machine up to that time point.

32-bit Multiplication

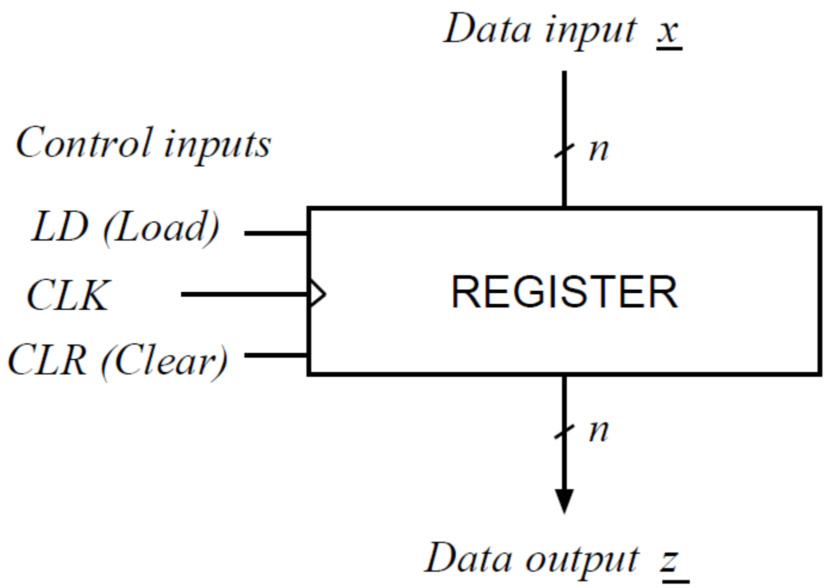


Revised Version



The Product register is actually **65 bit** to hold the **carry-out**.

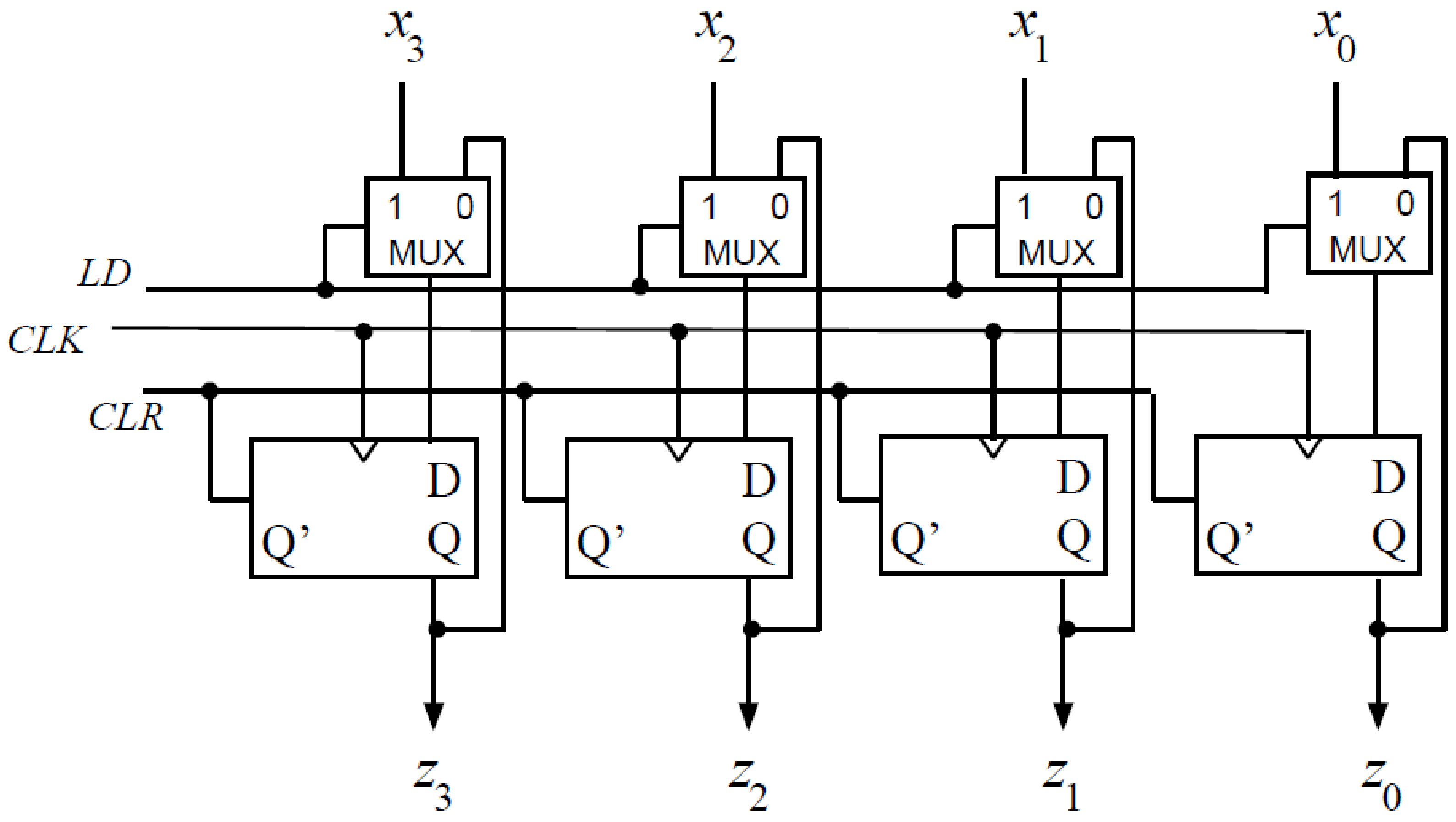
Registers



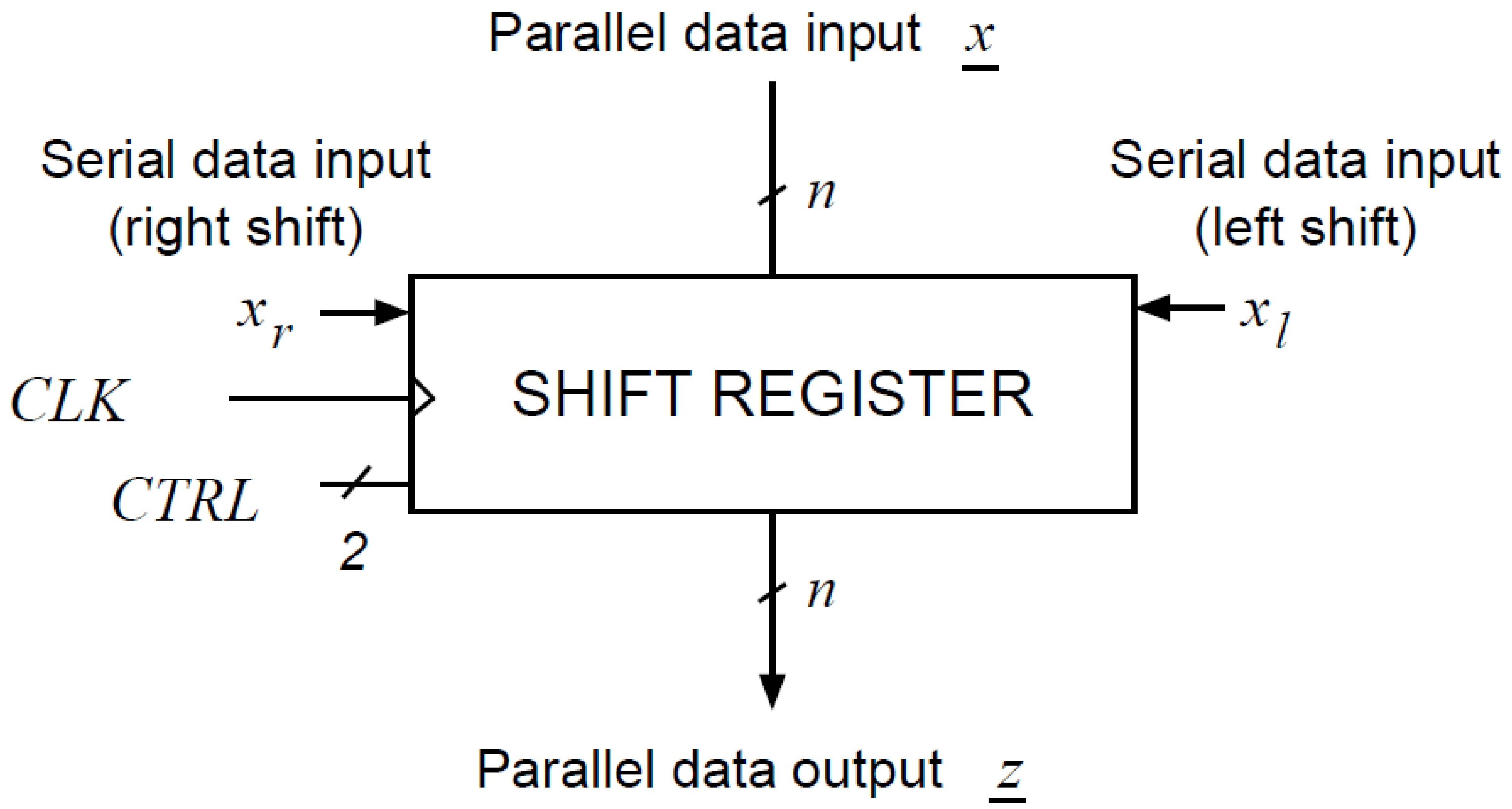
$$\underline{s}(t+1) = \begin{cases} \underline{x}(t) & \textbf{if} \quad LD(t) = 1 \textbf{ and } CLR(t) = 0 \\ \underline{s}(t) & \textbf{if} \quad LD(t) = 0 \textbf{ and } CLR(t) = 0 \\ (0\ldots0) & \textbf{if} \quad CLR(t) = 1 \end{cases}$$

$$\underline{z}(t) = \underline{s}(t)$$

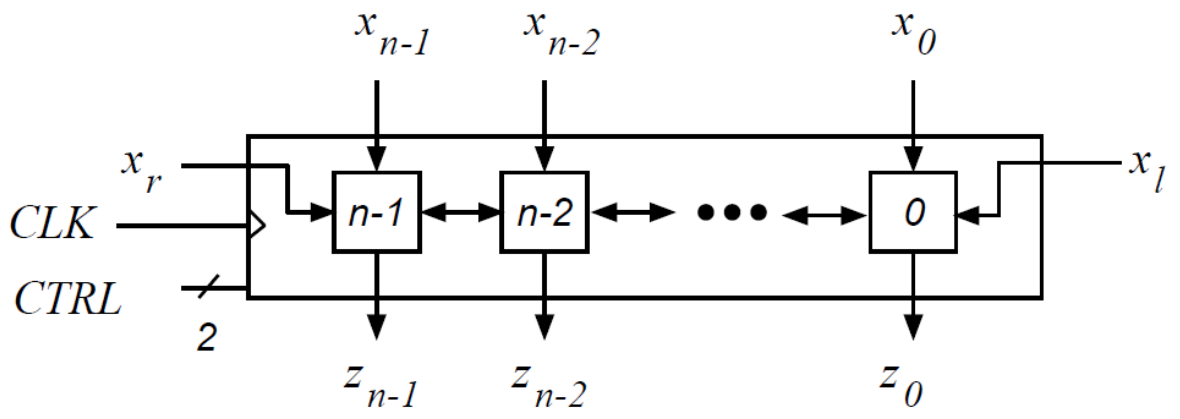
4-BIT Register



Shift Registers


Parallel data input $\underline{x}$
n
Serial data input (right shift)
Serial data input (left shift)
x_r
x_l
SHIFT REGISTER
CLK
$CTRL$
2
n
Parallel data output $\underline{z}$

Parallel In/Out Bidirectional Shift Register



$$\underline{s}(t+1) = \begin{cases} \underline{s}(t) & \textbf{if} \quad CTRL = NONE \\ \underline{x}(t) & \textbf{if} \quad CTRL = LOAD \\ (s_{n-2}, \ldots, s_0, x_l) & \textbf{if} \quad CTRL = LEFT \\ (x_r, s_{n-1}, \ldots, s_1) & \textbf{if} \quad CTRL = RIGHT \end{cases}$$

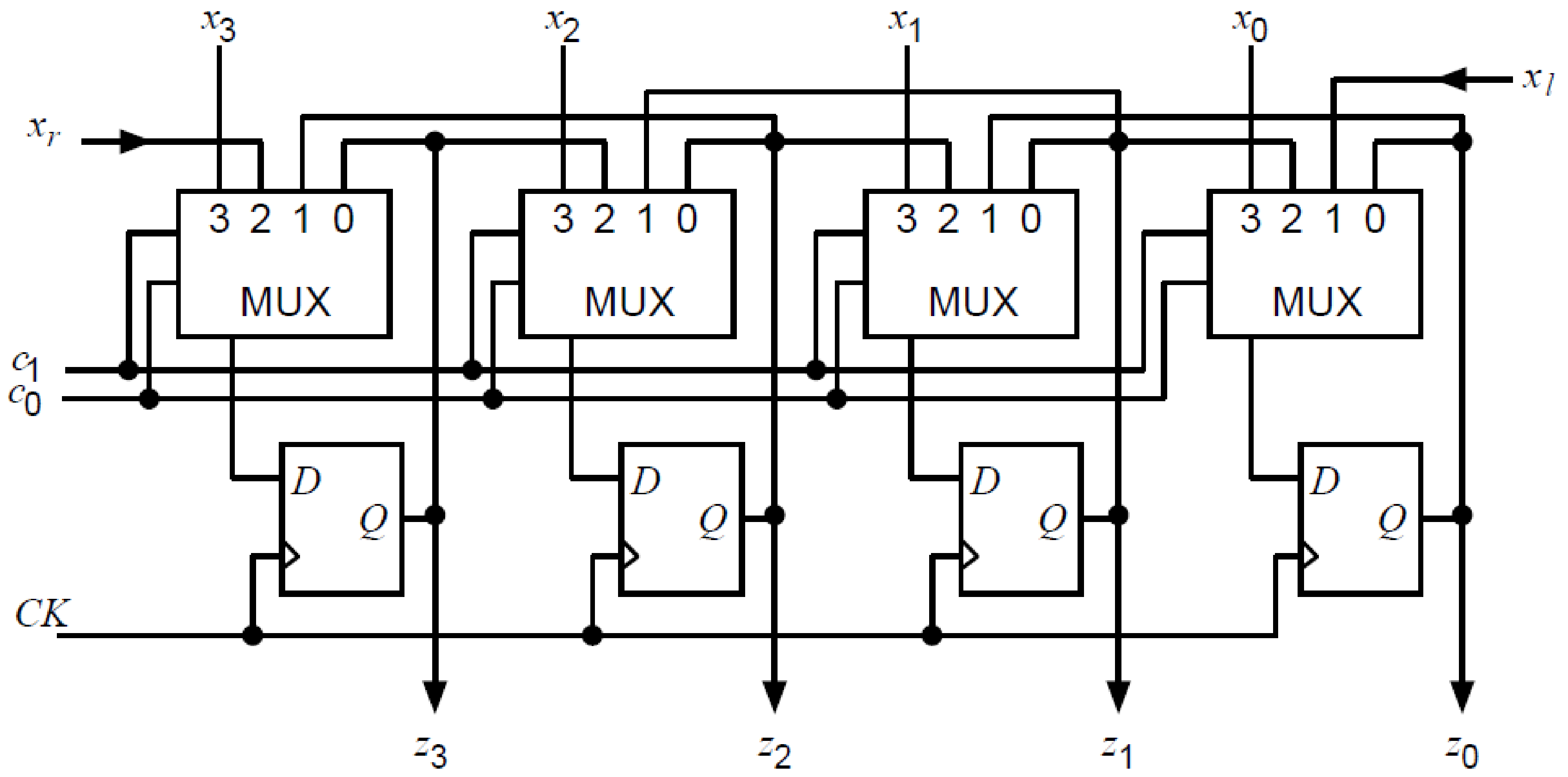
$$\underline{z} = \underline{s}$$

Shift Register Control

$CTRL$	c_1	c_0
$NONE$	0	0
$LEFT$	0	1
$RIGHT$	1	0
$LOAD$	1	1

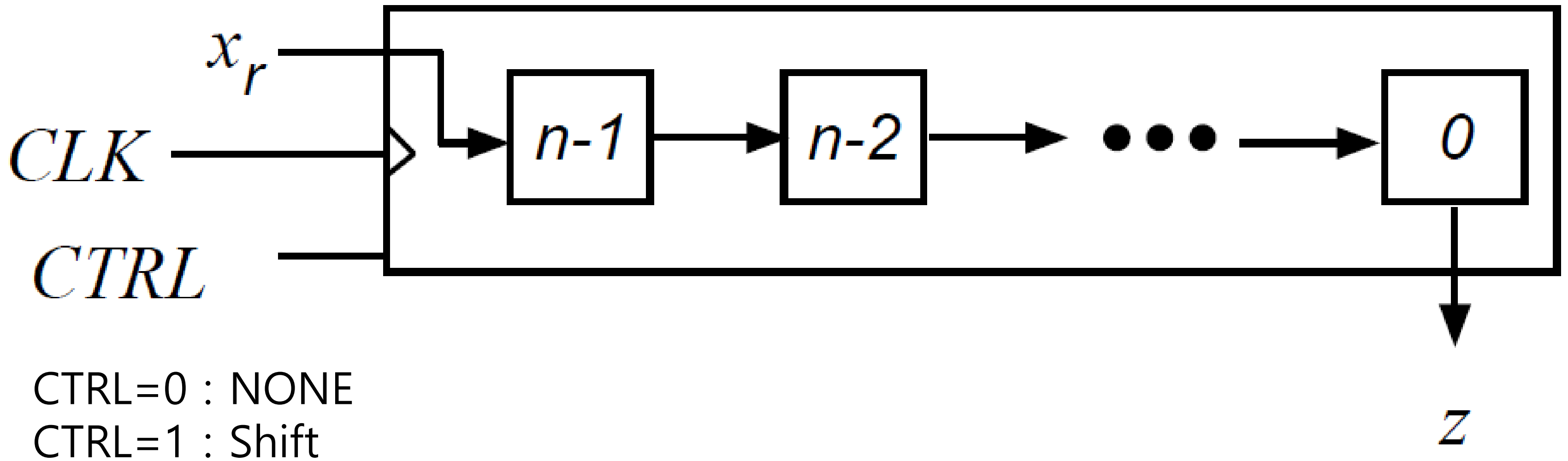
Control		$s(t+1) = z(t+1)$
$NONE$		0101 Default
$LOAD$		1110
$LEFT$	$x_l = 0$	1010
$LEFT$	$x_l = 1$	1011
$RIGHT$	$x_r = 0$	0010
$RIGHT$	$x_r = 1$	1010

4-Bit P-in/out B-d Shift Register



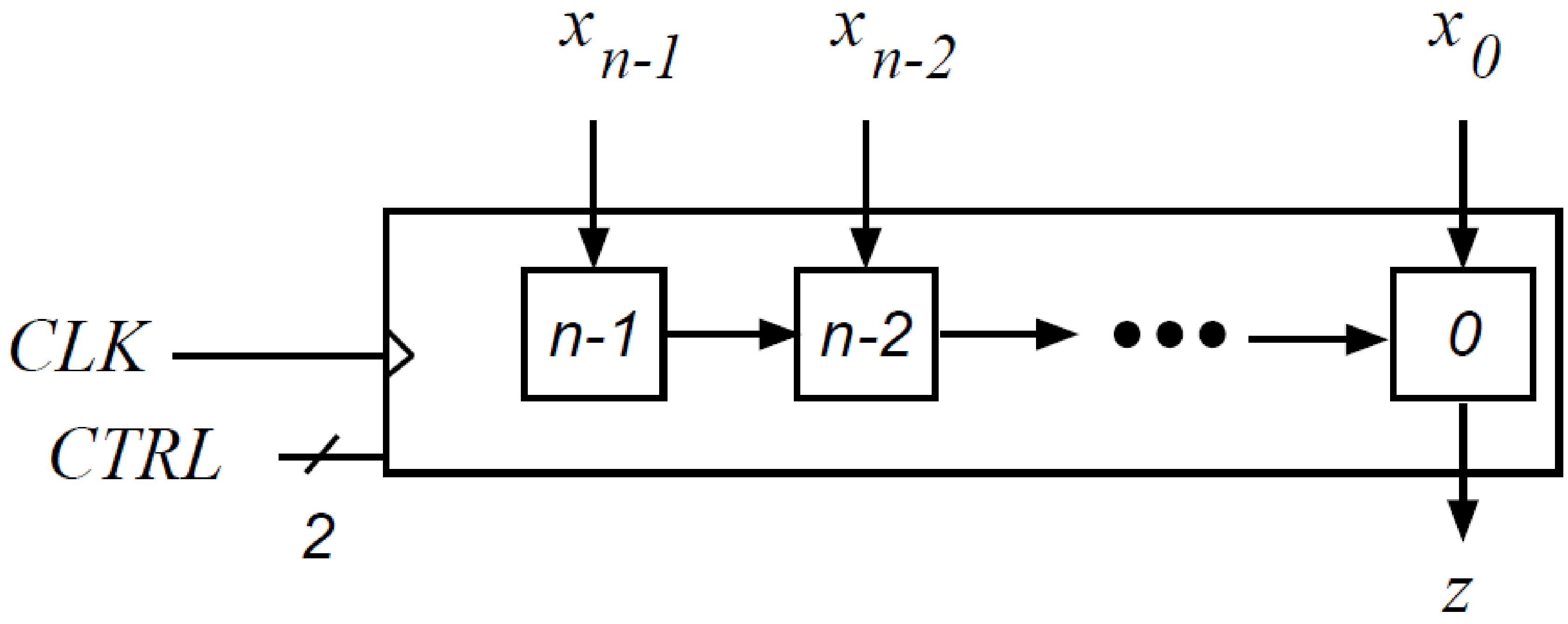
Serial In/Out Uni-D Shift Register

$$z(t) = x(t-n)$$



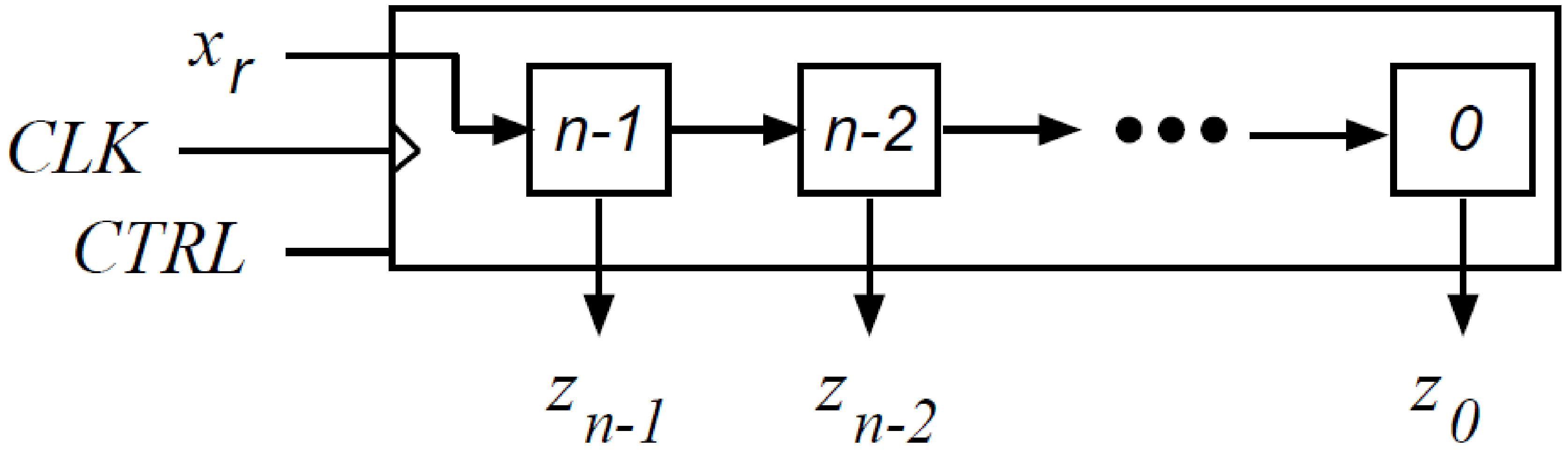
CTRL=0 : NONE
CTRL=1 : Shift

P-in/S-out Uni-D Shift Register



CTRL=00 : NONE
CTRL=01 : Shift
CTRL=10 : Load
CTRL=11 : Undefined

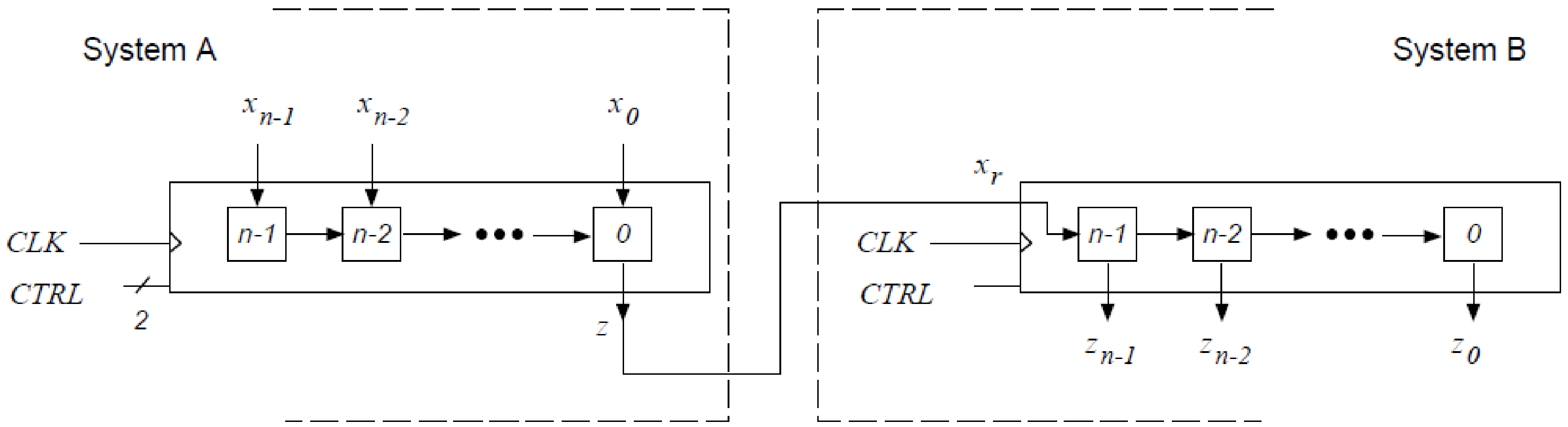
P-In/S-Out Uni-D Shift Register



CTRL=0 : NONE
CTRL=1 : Shift

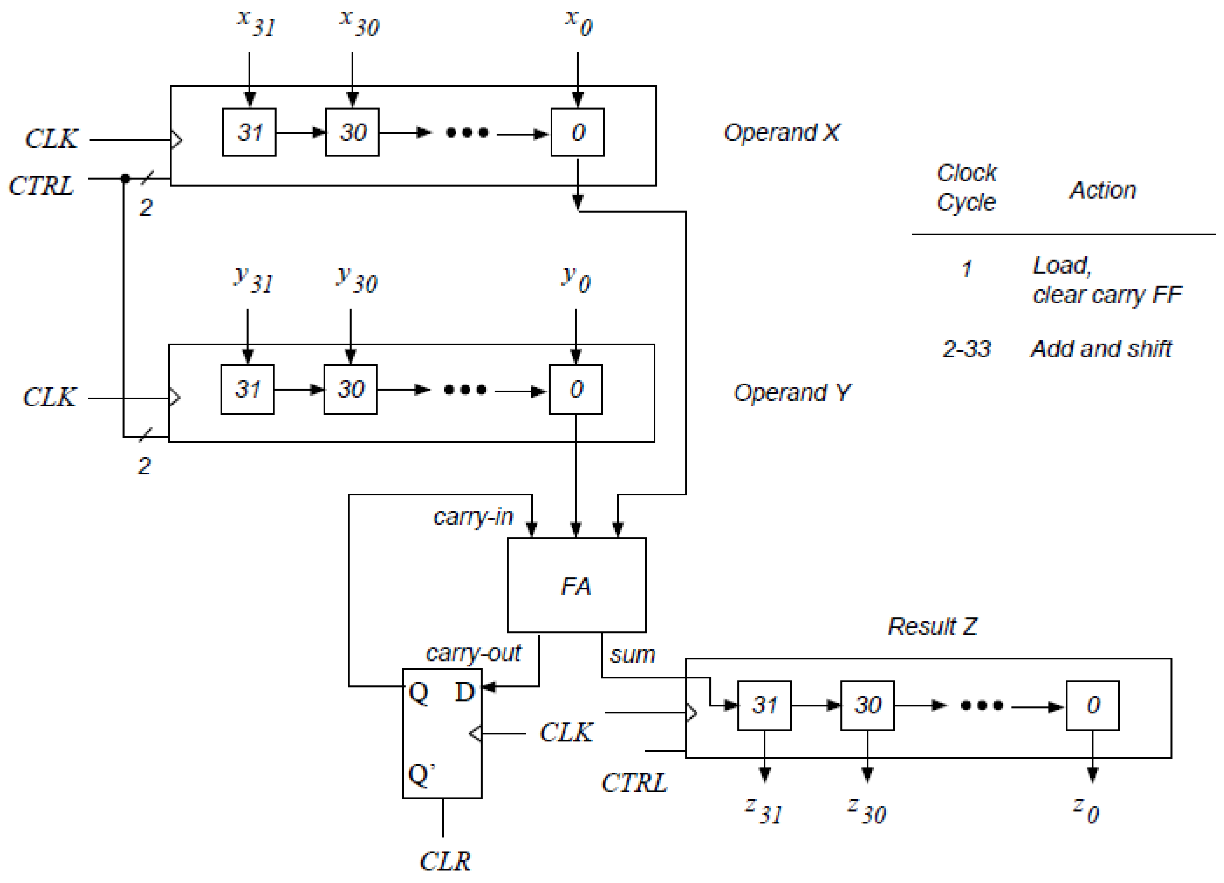
Example (1/2)

- SERIAL INTERCONNECTION OF SYSTEMS



Example (2/2)

- BIT-SERIAL OPERATIONS



Clock Cycle	Action
1	Load, clear carry FF
2-33	Add and shift

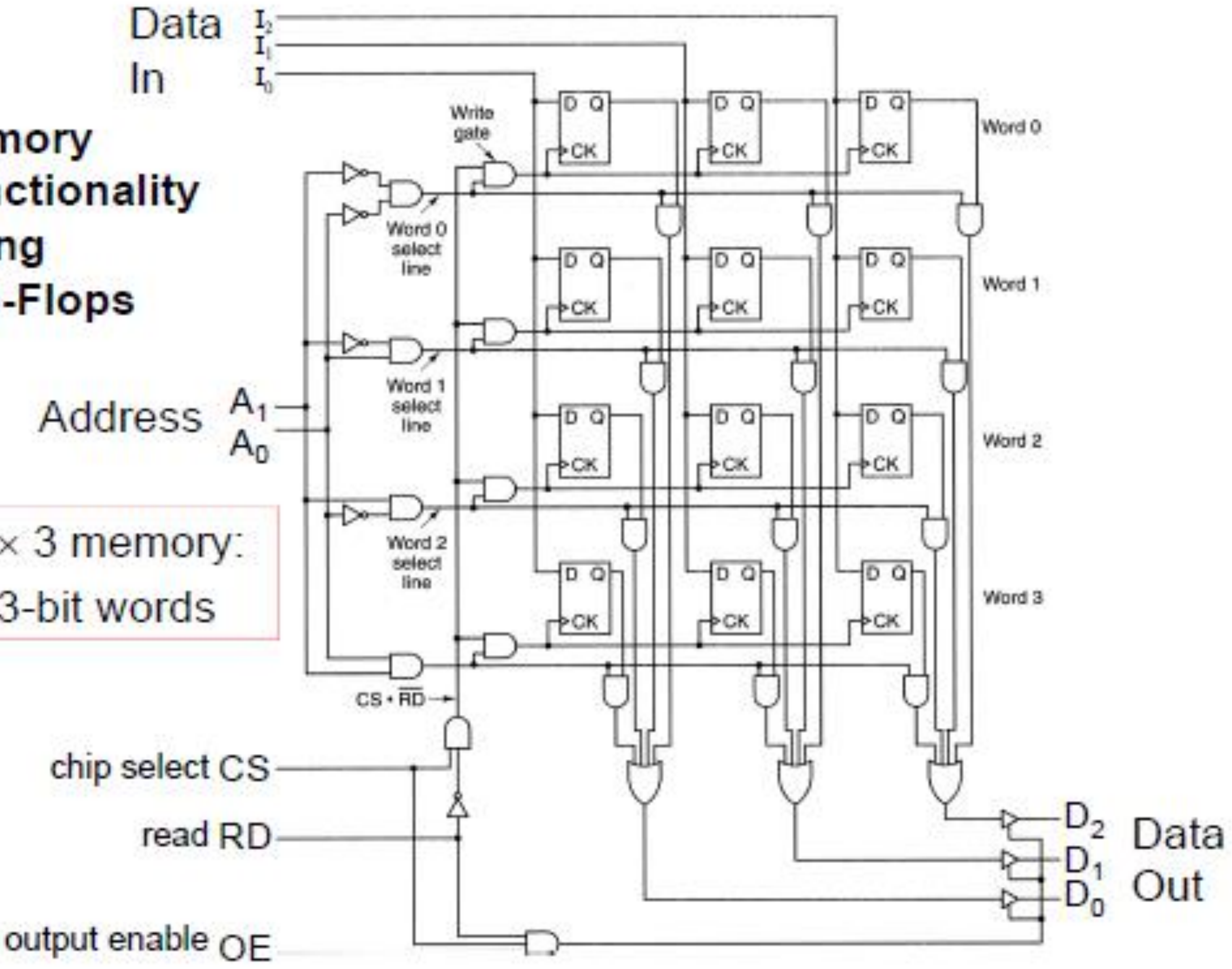
Figure 11.10: BIT-SERIAL ADDER.

Basic Storage Technologies

- Static RAM
  - Fast: 1-20 ns access time
  - Expensive, small
  - Typical Usage: register file, caches
  - Typical Size: KB to few MB
- Dynamic RAM
  - Medium speed: 30-50 ns access time
  - Medium cost/size
  - Typical Usage: Main Memory
  - Typical Size: Few GB
- Magnetic Disk
  - Slowest but Cheapest cost and Largest size

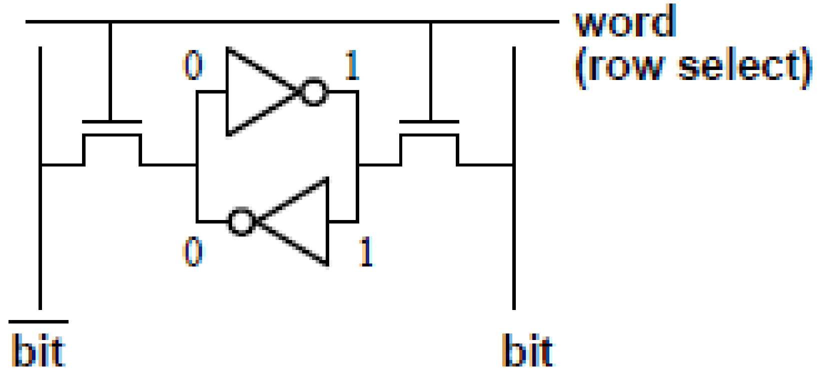
Memory Functionality Using Flip-Flops

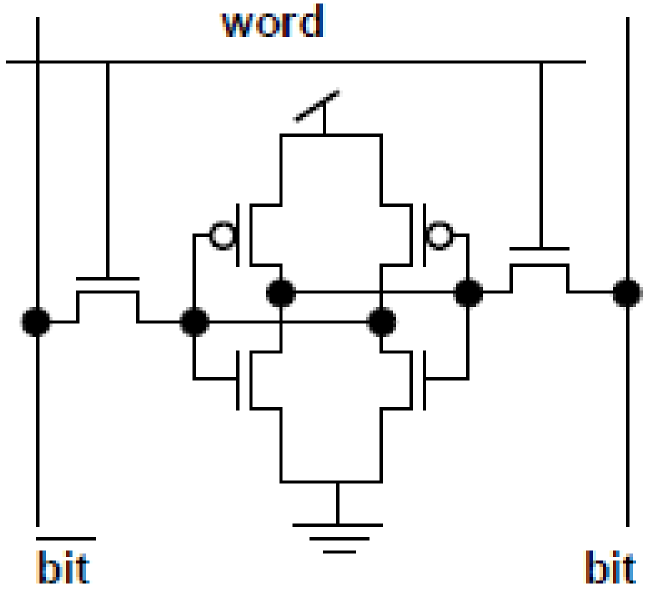
a 4 × 3 memory:
4 3-bit words



Static RAM Cell

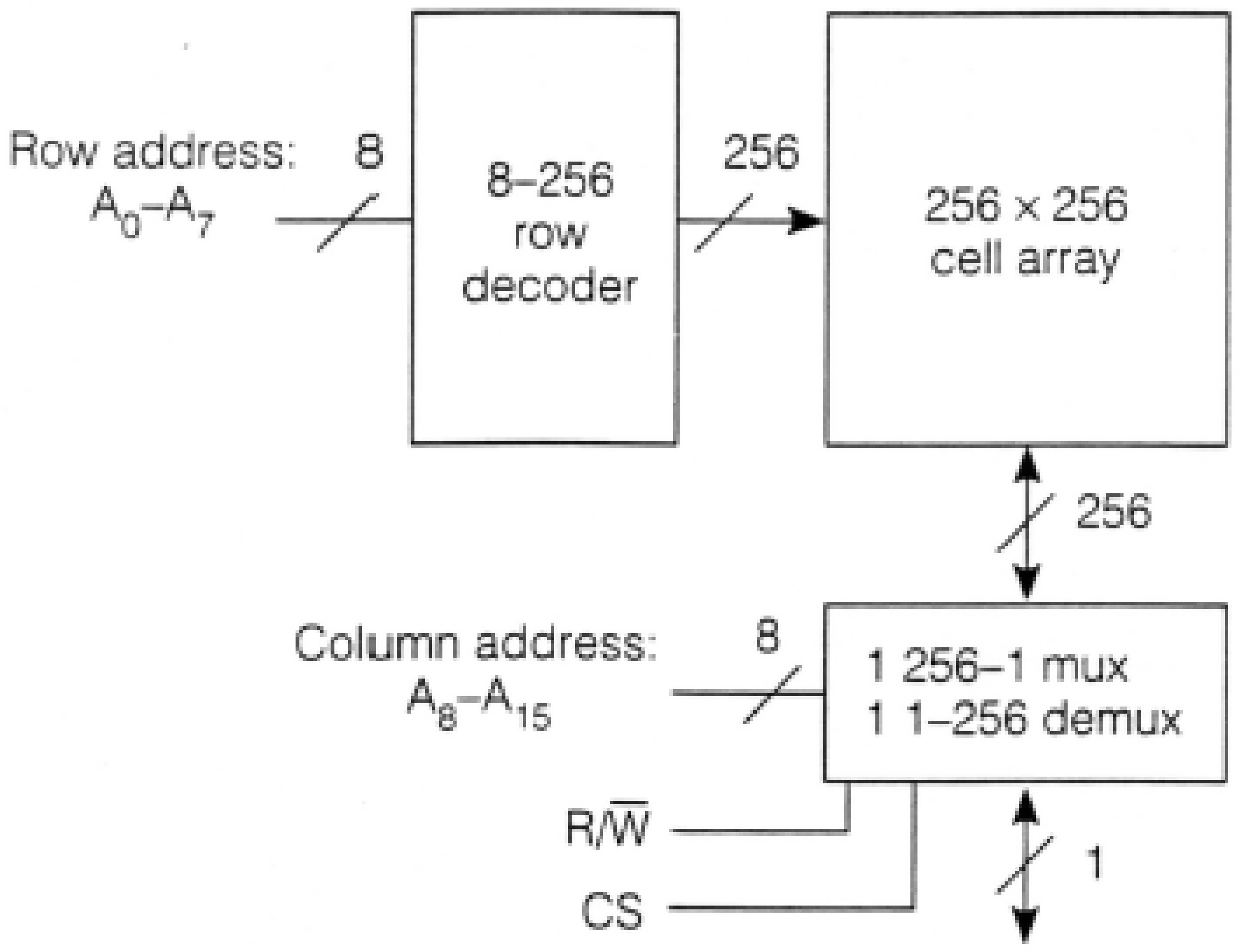
6-Transistor SRAM Cell



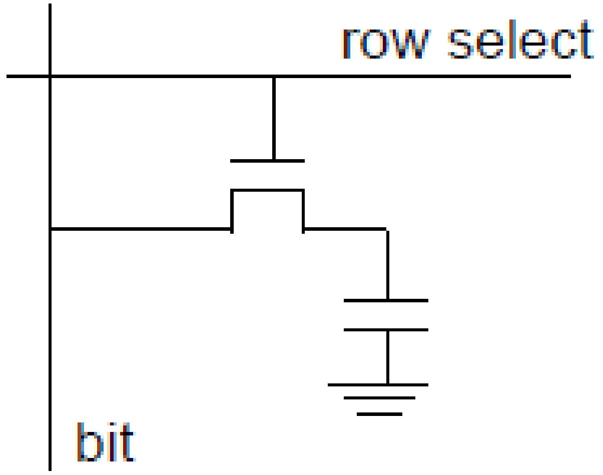


- **write:**
  1. drive bit lines (bit=1, $\overline{\text{bit}}$=0)
  2. select row
- **read:**
  1. precharge bit and $\overline{\text{bit}}$ to Vdd
  2. select row
  3. cell pulls one line low
  4. sense amp on column detects difference between bit and $\overline{\text{bit}}$

A 64K×1 Static RAM Chip



Dynamic RAM (DRAM): 1-Transistor Memory Cell



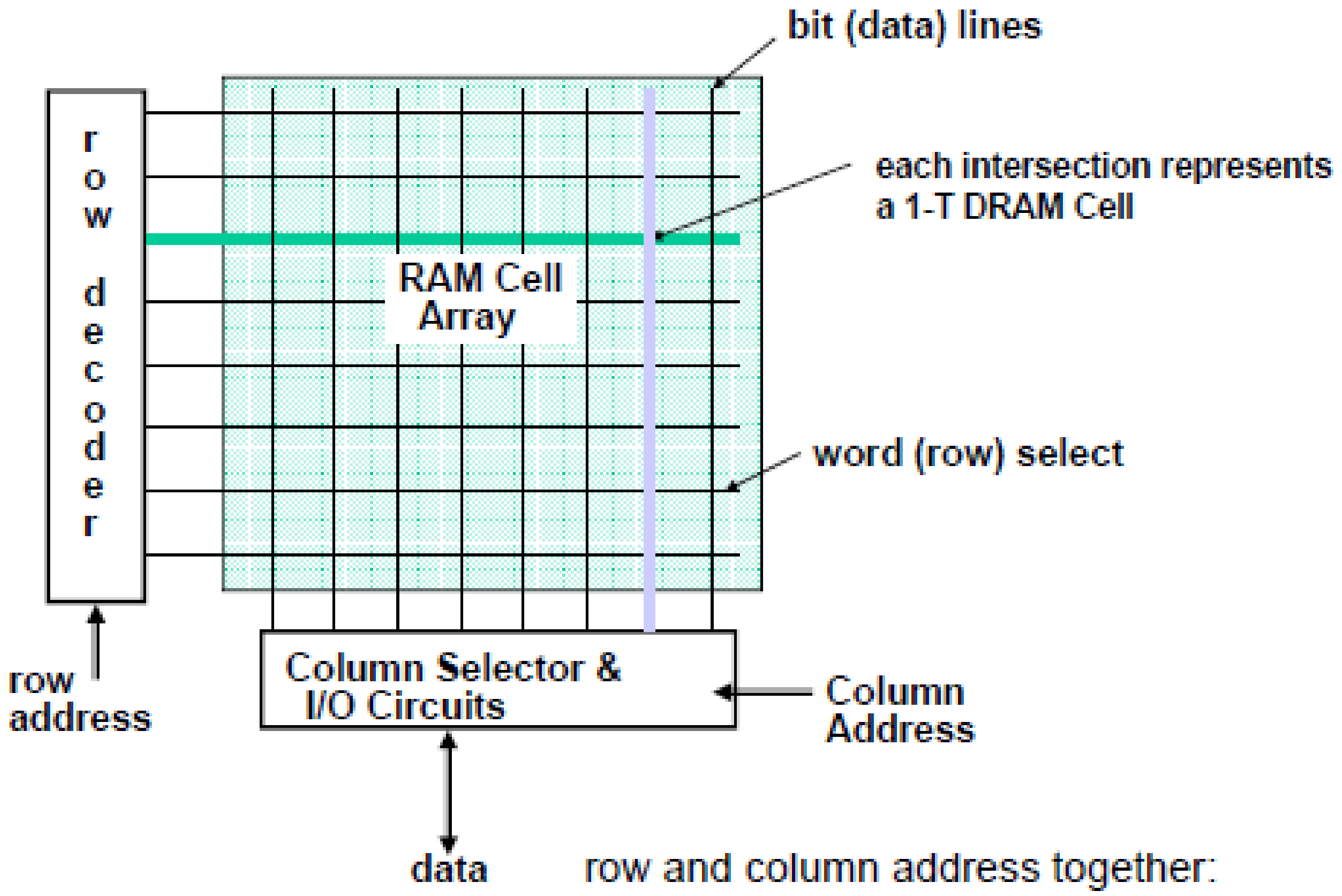
- **write:**
  1. drive bit line
  2. select row
- **read:**
  1. precharge bit line to Vdd
  2. select row
  3. cell and bit line share charges

     → very small voltage changes on the bit line
  4. sense (sense amplifier)

     can detect changes of ~1 million electrons
  5. write: restore the value
- **refresh**

  a dummy read to every cell

Classical DRAM Organization (square)



row and column address together:
select 1 bit a time

Central Processing Unit (CPU)

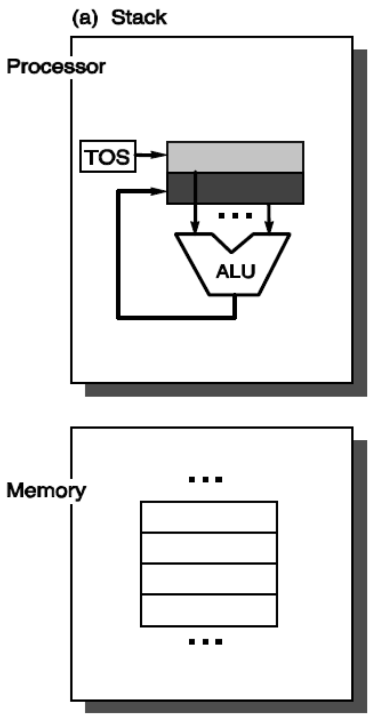
- CPU executes a program
  - A program consists of machine instructions and dat a.
  - Basic steps of executing instructions
    - Fetch
      - Fetch instruction at PC address from main memory
    - Decode
      - Decide what to (CTRL unit generates proper control signals)
    - Execute
      - Perform ALU operation or Address calculation
    - Memory
      - Access Memory to Load or Store data
    - Write Back
      - Store the ALU result or Load data

Instruction Set Architecture

- ISA is the structure of a computer that a machine language programmer (or a co mpiler) must understand to write a corre ct program for that machine
- Classic ISA types
  - Stack: No registers
  - Accumulator: 1 Registers for ALU op
  - Register-Memory: Register file (GPR)s
  - Load-Store machines: Register file

Stack Machine

- No Registers, just a stack of value operations
- Push / Pop stack, Arithmetic operation
- Perform "OP" with the first two stack entries, store the Result in t he 2nd entry of stack, then pop the stack top

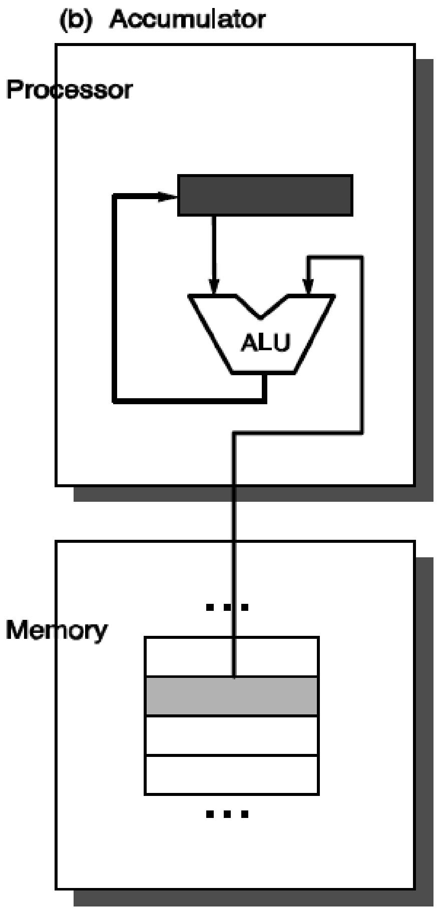


- Implicit operands on stack
- Ex. C = A + B

```
Push A
Push B
Add
Pop C
```

- Good code density; used in 60's-70's; now in Java VM

Accumulator Machine



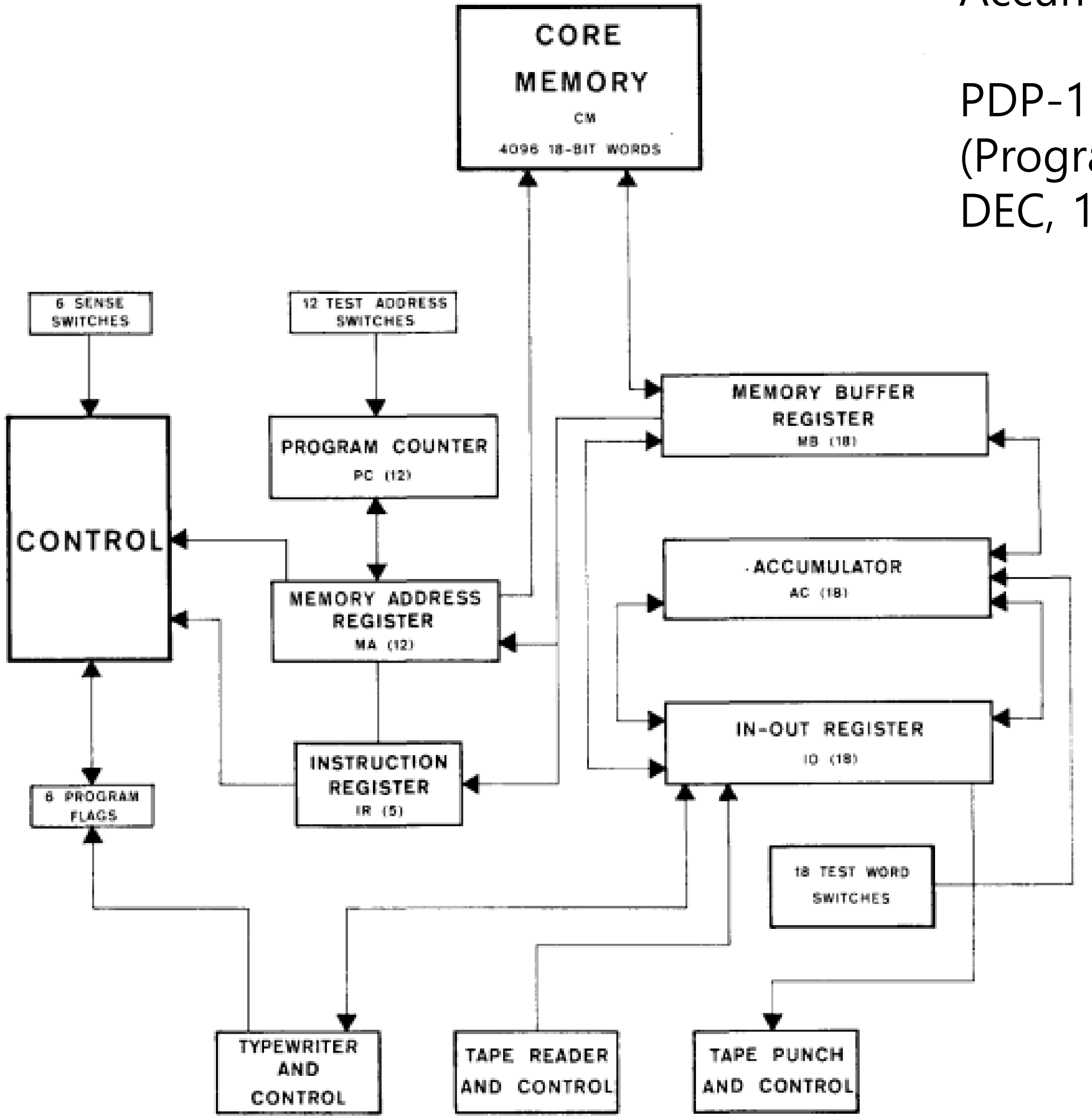
The accumulator provides an implicit input, and is the implicit place to store the result

Ex) C=A+B	
Load A	-> Load A in the ACC
ADD B	-> ACC <- ACC+B
STORE C	-> Store ACC in C

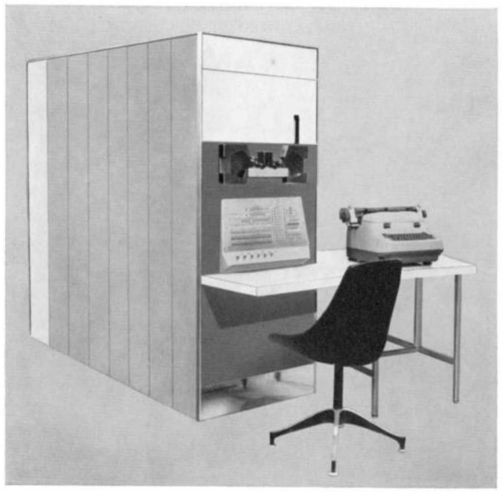
*Used before 1980

Accumulator Machine example

PDP-1
(Programmed Data Processor-1)
DEC, 1961



PDP-1 System Block Diagram



Programmed Data Processor-1

General Purpose Registers

- Benefit
  - Reduce memory traffic
  - Improve program speed
  - Improve code density
- Usage
  - Holding temporal result in expression evaluation
  - Passing parameters
  - Holding variables
- Examples
  - Register-Memory (ex., Intel x86)
    - Arith. Inst can use data in registers and/or memory
  - Load-Store (ARM, MIPS[CS151B])
    - Arith. Inst can only use data in registers

Comparing #Instructions

Code sequence for **C = A + B**

Stack	Accumulator	Register-Memory	Load-Store
Push A	**Load A**	**Add C, A, B**	**Load R1,A**
Push B	**Add B**		**Load R2,B**
Add	**Store C**		**Add R3,R1,R2**
Pop C			**Store C,R3**

Powerful instruction ⇒ higher performance?

- Fewer instructions required
- But complex instructions are hard to implement
  - May slow down all instructions, including simple ones
- Compilers are good at making fast code from simple instructions