University Of Diyala
College Of Engineering
Computer Engineering Department

Digital System Design II

Dr. Yasir Amer Al-Zubaidi

Third stage

2021

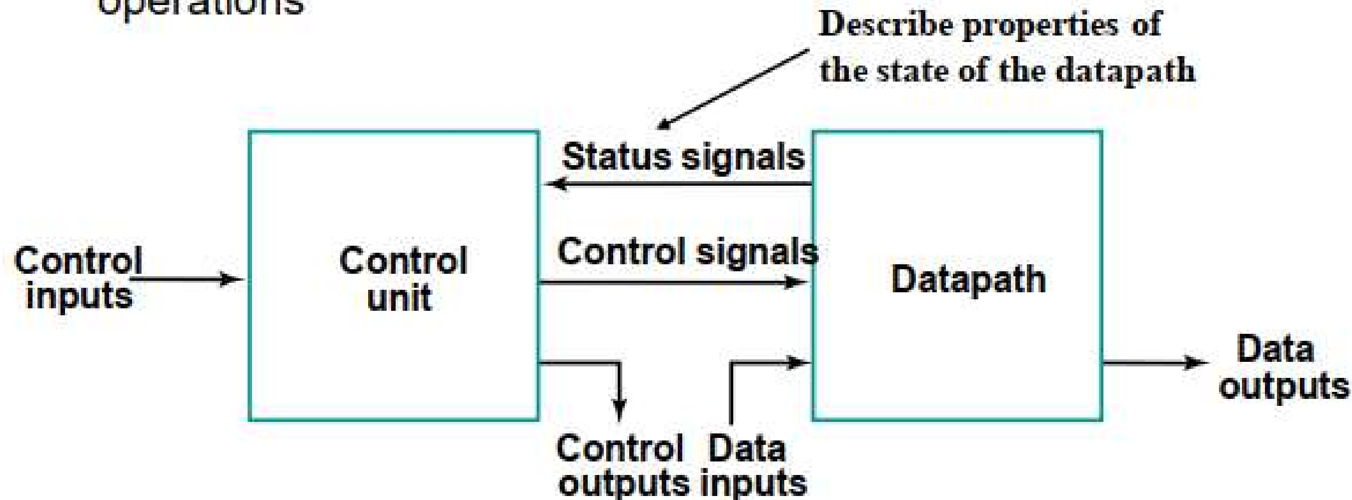
Design of Digital Sequential Circuits Using New Methods

Microprogramming Overview

- **Data path and Control**
- **Microoperations**
- **Sequencing and control**

Datapath and Control

- Datapath - performs data transfer and processing operations
- Control Unit - Determines the enabling and sequencing of the operations



- The control unit receives:
  - External control inputs
  - Status signals
- The control unit sends:
  - Control signals
  - Control outputs

Overview

- Datapath and control
- Microoperations
  - Register transfer operations
  - Microoperations - arithmetic, logic, and shift
  - Register cell design
  - Serial transfers and microoperations
- Sequencing and control

Register Transfer Operations

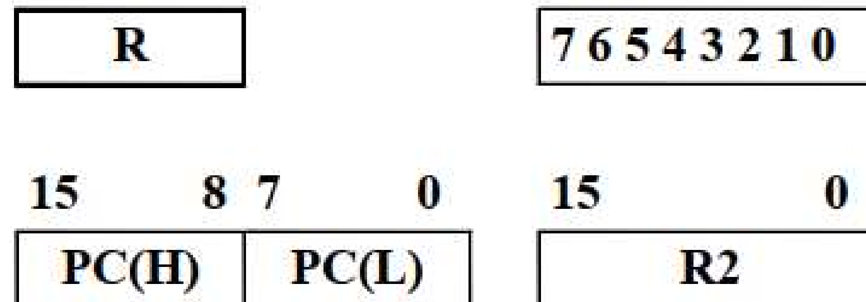
- Register Transfer Operations – the movement and processing of data stored in registers

- Three basic components:
  - A set of registers (operands)
  - Transfer operations
  - Control of operations

- Elementary operations -- called *microoperations*
  - load, count, shift, add, bitwise "OR", etc.

Register Notation

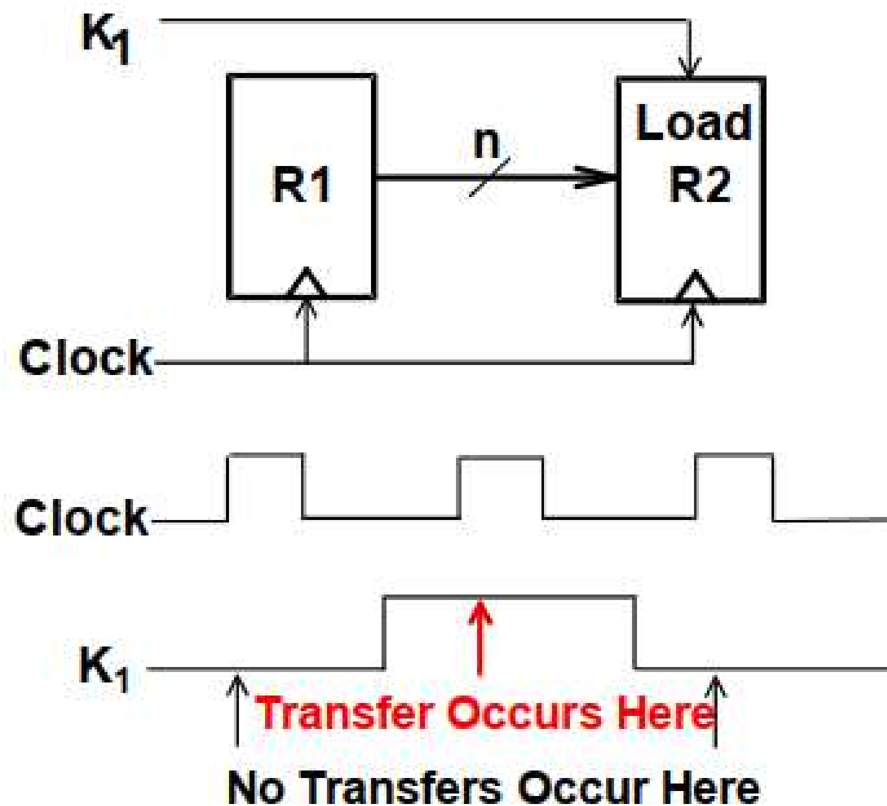
- Letters and numbers – register (e.g. R2, PC, IR)
- Parentheses ( ) – range of register bits (e.g. R1(1), PC(7:0), AR(L))



- Arrow (←) – data transfer (ex. R1 ← R2, PC(L) ← R0)
- Brackets [ ] – Specifies a memory address (ex. R0 ← M[AR], R3 ← M[PC])
- Comma – separates parallel operations

Conditional Transfer

- If (K_1 =1) then ($R2 \leftarrow R1$) $\Leftrightarrow$ K_1: ($R2 \leftarrow R1$)
  where K_1 is a control expression specifying a conditional execution of the microoperation.



Microoperations

- Logical groupings:
  - Transfer - move data from one set of registers to another
  - Arithmetic - perform arithmetic on data in registers
  - Logic - manipulate data or use bitwise logical operations
  - Shift - shift data in registers

Arithmetic operations	Logical operations
+ Addition	$\vee$ Logical OR
– Subtraction	$\wedge$ Logical AND
* Multiplication	$\oplus$ Logical Exclusive OR
/ Division	$\overline{}$ Not

Example Microoperations

- $R1 \leftarrow R1 + R2$
  - Add the content of R1 to the content of R2 and place the result in R1.
- $PC \leftarrow R1 * R6$
- $R1 \leftarrow R1 \oplus R2$
- $(K1 + K2)\text{:}\ R1 \leftarrow R1 \vee R3$
  - On condition K1 OR K2, the content of R1 is Logic bitwise Ored with the content of R3 and the result placed in R1.
  - NOTE: "+" (as in $K_1 + K_2$) means "OR." In $R1 \leftarrow R1 + R2$, + means "plus."

Arithmetic Microoperations

Symbolic Designation	Description
$R0 \leftarrow R1 + R2$	Addition
$R0 \leftarrow \overline{R1}$	Ones Complement
$R0 \leftarrow \overline{R1} + 1$	Two's Complement
$R0 \leftarrow R2 + \overline{R1} + 1$	R2 minus R1 (2's Comp)
$R1 \leftarrow R1 + 1$	Increment (count up)
$R1 \leftarrow R1 - 1$	Decrement (count down)

- Any register may be specified for source 1, source 2, or destination.
- These simple microoperations operate on the whole word

Logical Microoperations

Symbolic Designation	Description
$R0 \leftarrow \overline{R1}$	Bitwise NOT
$R0 \leftarrow R1 \vee R2$	Bitwise OR (sets bits)
$R0 \leftarrow R1 \wedge R2$	Bitwise AND (clears bits)
$R0 \leftarrow R1 \oplus R2$	Bitwise EXOR (complements bits)

Shift Microoperations

- Let R2 = 11001001

Symbolic Designation	Description	R1 content
R1 ← sl R2	Shift Left	10010010
R1 ← sr R2	Shift Right	01100100

- **Note: These shifts "zero fill". Sometimes a separate flip-flop is used to provide the data shifted in, or to "catch" the data shifted out.**
- **Other shifts are possible (rotates, arithmetic)**

University Of Diyala
College Of Engineering
Computer Engineering Department

Digital System Design II

Dr. Yasir Al-Zubaidi

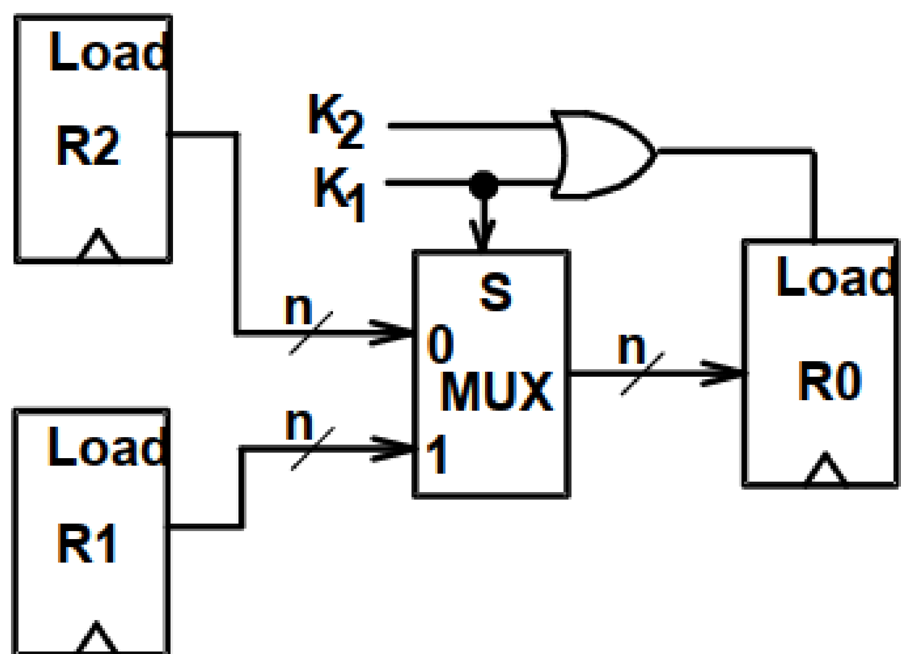
Third stage

2019

Multiplexer-Based Single Register Transfers

- MUX connected to register outputs produce flexible transfer structures
- Transfers: $K1: R0 \leftarrow R1$

  $K2\,\overline{K1}: R0 \leftarrow R2$

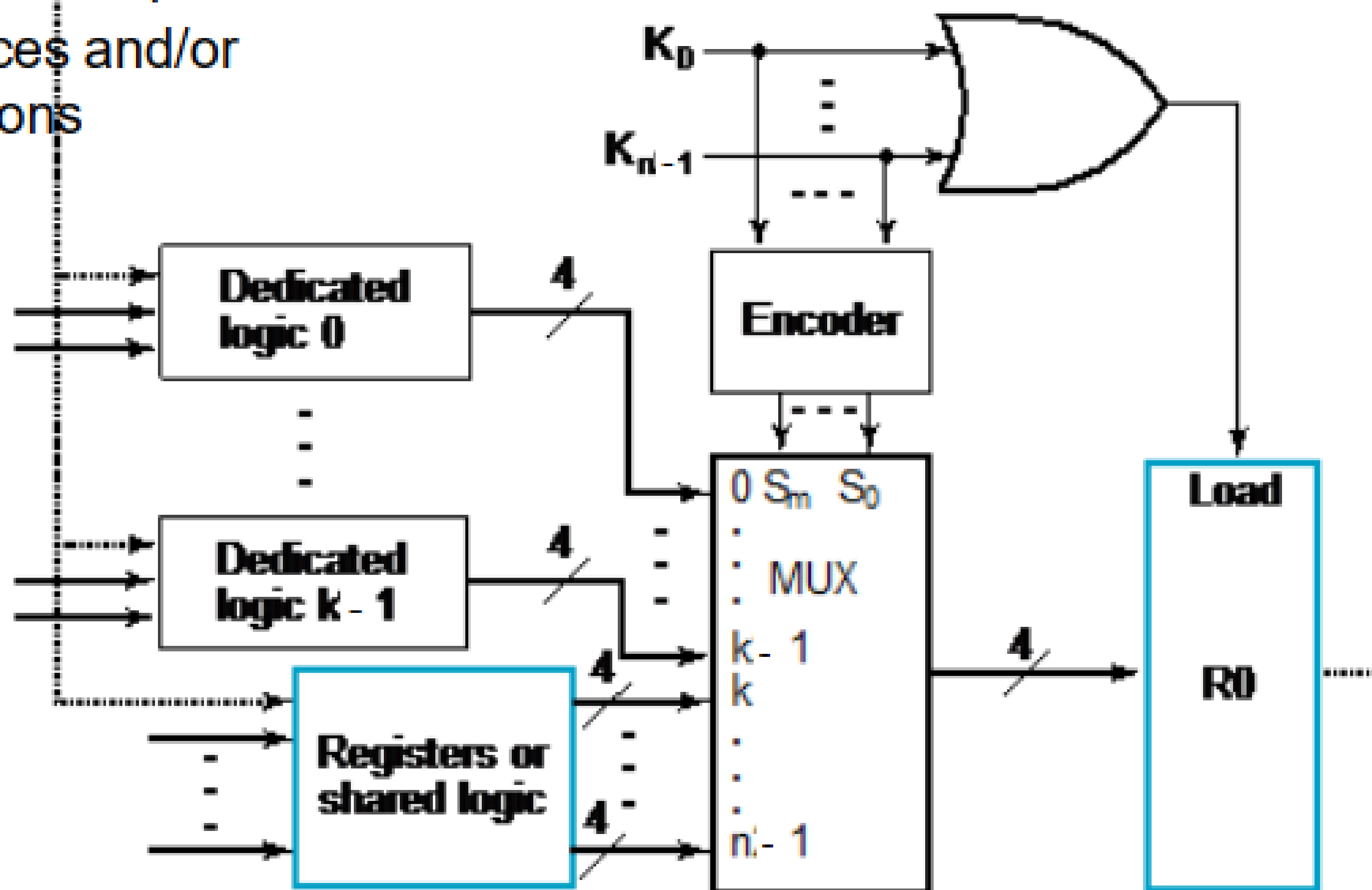


Register Design

- Assume: a register consists of identical cells
- Register design can be approached as follows:
  - Design a representative cell for the register
  - Make copies of the cell and connect together to form the register
  - Applying appropriate “boundary conditions” to cells that need to be different and contract if appropriate
- Register cell design is the first step of the above process

Approach I: Multiplexer-based

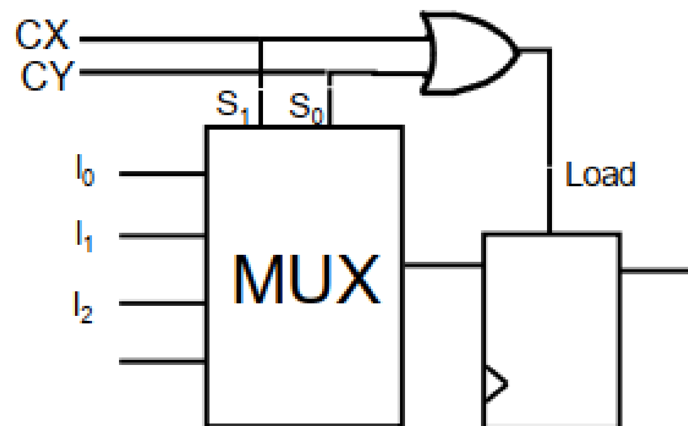
- An n-input multiplexer with a variety of sources and functions
- Load enable by OR of control signals $K_0, K_1, \ldots K_{n-1}$ (for 00…0, no load)
- Use encoder + multiplexer to select sources and/or transfer functions



Example 1: Register Cell Design

- Register A (m-bits) Specification:
  - Data input: B; Control inputs (CX, CY): (0,0), (0,1) (1,0)
  - Register transfers:
    - CX: $A \leftarrow B \vee A$; CY : $A \leftarrow B \oplus A$; Hold state: (0,0)
- Load Control: Load = CX + CY
- Since all control combinations appear as if encoded (0,0), (0,1), (1,0), can use multiplexer without encoder:

$S_1 = CX$
$S_0 = CY$
$I_0 = A_i$ Hold A
$I_1 = A_i \leftarrow B_i \oplus A_i$ CY = 1
$I_2 = A_i \leftarrow B_i \vee A_i$ CX = 1



Approach II: Sequential Circuit Design

- Find a state diagram or state table
- For optimization:
  - Use K-maps for up to 4 to 6 variables
  - Otherwise, use computer-aided or manual optimization

Example 1 Again

- State Table for D_i:

	Hold	Ai v Bi		Ai ⊕ Bi	
A_i	CX = 0 CY = 0	CX = 1 CY = 0 B_i = 0	CX = 1 CY = 0 B_i = 1	CX = 0 CY = 1 B_i = 0	CX = 0 CY = 1 B_i = 1
0	0	0	1	0	1
1	1	1	1	1	0

- Four variables (CX, CY, A, B) should give a total of 16 state table entries
- By using:
  - Combinations of variable names and values
  - Don't care conditions (for CX = CY = 1)

  only 12 entries are required to represent the 16 entries

Example 1 Again (Contd.)

- K-map - Use variable ordering CX, CY, A_i, B_i and assume a D flip-flop

D_i

A_i, B_i, CX, CY

0	0	1	1
0	1	0	1
X	X	X	X
0	1	1	1

Example 1 Again (Contd.)

- The resulting SOP equation:

$$D_i = CX\,B_i + CY\,\overline{A}_i\,B_i + A_i\,\overline{B}_i + \overline{CY}\,A_i$$
$$= CX\,B_i + \overline{A}_i\,(CY\,B_i) + A_i(\overline{CY\,B_i})$$
$$= CX\,B_i + A_i \oplus (CY\,B_i)$$

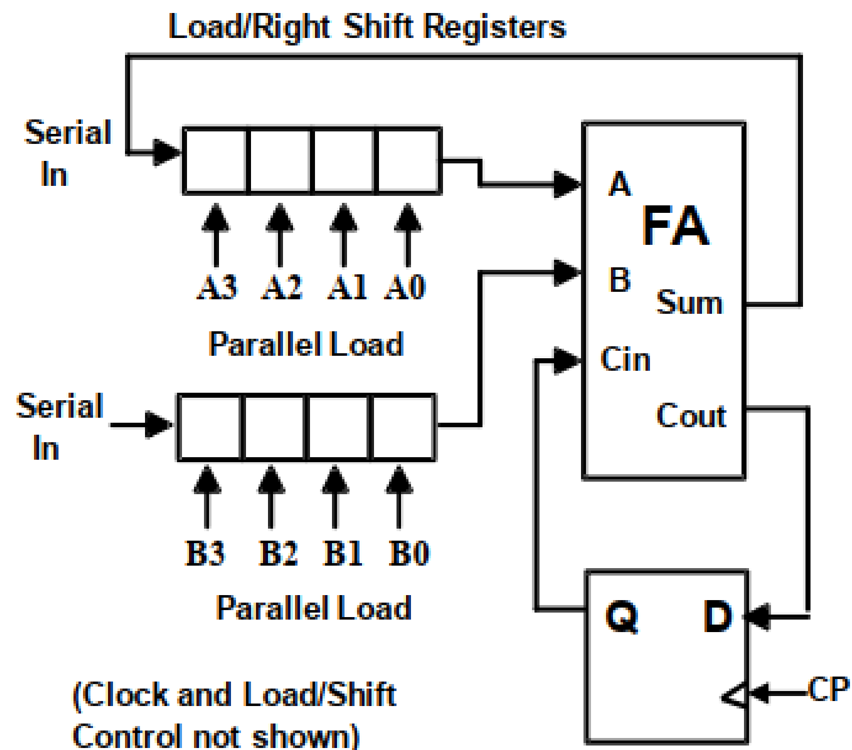
  The gate input cost per cell = 13
- The gate input cost per cell for the previous version is:

  Per cell: 19

  Shared decoder logic: 8
- Cost gain by sequential design > 6 per cell
- Also, no Enable on the flip-flop makes it cost less

Serial Transfers and Microoperations

- Serial Transfers
  - Used for “narrow” transfer paths
  - Example 1: Telephone or cable line
    - Parallel-to-Serial conversion at source
    - Serial-to-Parallel conversion at destination
- Serial microoperations
  - Example 1: Addition
    - A low cost way
    - Loss in performance



Overview

- Datapath and control
- Microoperations
- Sequencing and control
  - Algorithmic State Machines (ASM)
    - ASM chart
    - Timing considerations
    - ASM chart examples: Binary multiplier
  - Hardwired Control
    - Control design methods
    - Sequence register and decoder
    - One flip-flop per state
  - Microprogrammed control

Control Unit Types

- Two distinct classes:
  - Programmable
  - Non-programmable.
- A programmable control unit:
  - An external memory array for storing instructions and control information
  - A program counter (PC) register points to the next instruction to be executed
  - Decision logic for determining the sequence of operations and logic to interpret the instructions
- A non-programmable control unit: does not fetch instructions from a memory and is not responsible for sequencing instructions

Algorithmic State Machines

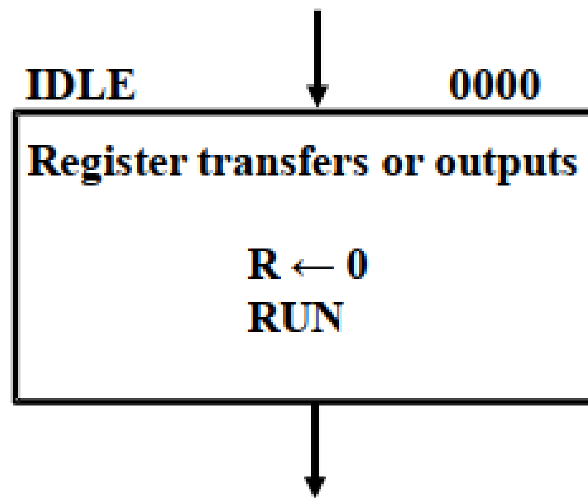
- The function of a sequential circuit can be represented by a state table or a state diagram.

- An Algorithmic State Machine (ASM) is a flowchart-like way to specify state diagrams for sequential logic and, optionally, actions performed in a datapath.
  - A flowchart is a way of showing actions and control flow in an algorithm.
  - An ASM explicitly specifies a sequence of actions and their timing relationships
  - An ASM chart directly leads to a hardware realization
- Primitives:
  1. State Box (a rectangle)
  2. Decision Box
     I. Scalar (a diamond)
     II. Vector (a hexagon)
  3. Conditional Output Box (an oval)

State Box

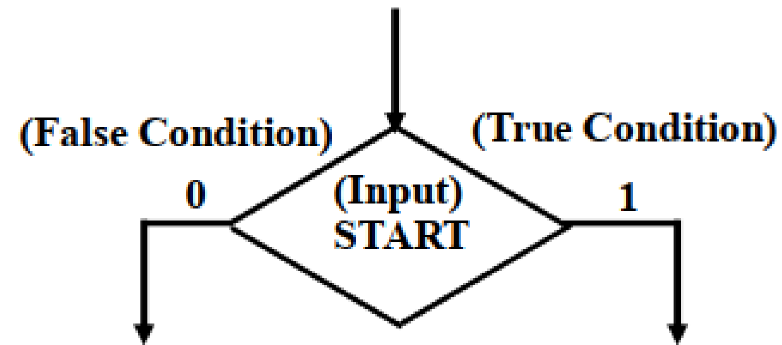
- A rectangle with:
  - The symbolic name for the state
  - An optional state code
  - Containing register transfer operations, and outputs activated within or while leaving the state

- The symbolic name for the state marked outside the upper left top
- An optional state code, if assigned, outside the upper right top

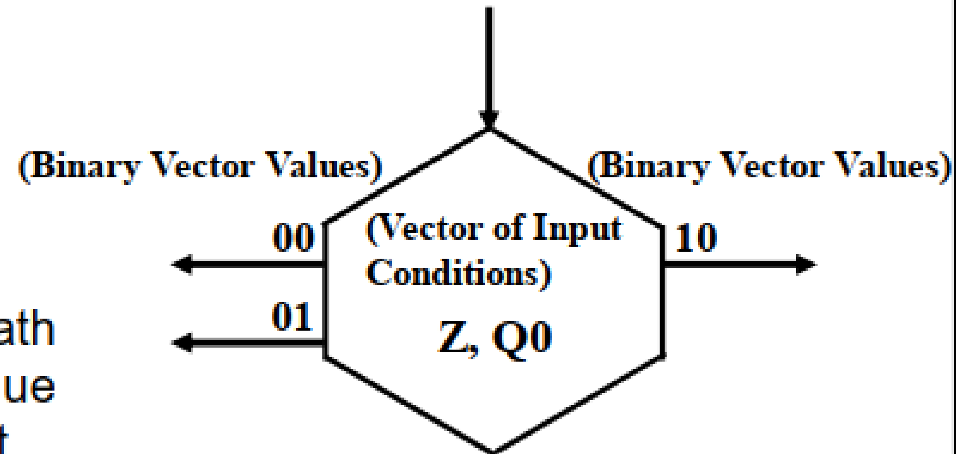


Decision Box

- Scalar : A diamond with:
  - One input path (entry point).
  - One input condition that is tested.
  - A TRUE/FALSE exit path (logic 1/0).

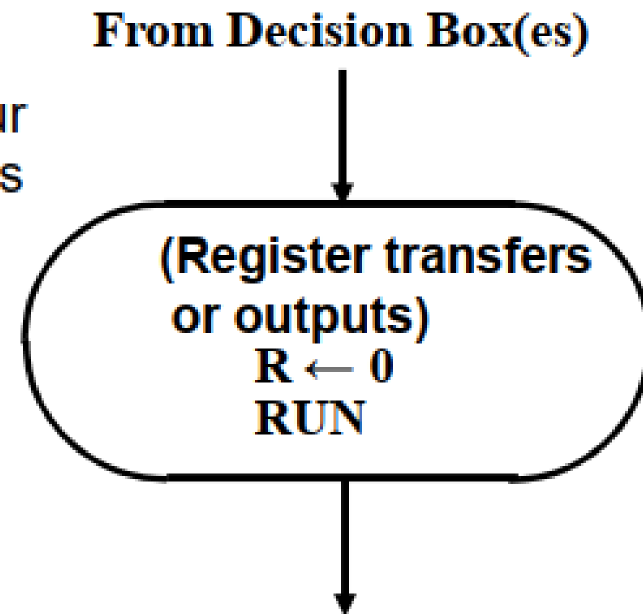


- Vector: A hexagon with:
  - One input path (entry point).
  - A vector of input conditions tested.
  - Up to 2^n output paths. The path taken has a binary vector value that matches the vector input condition



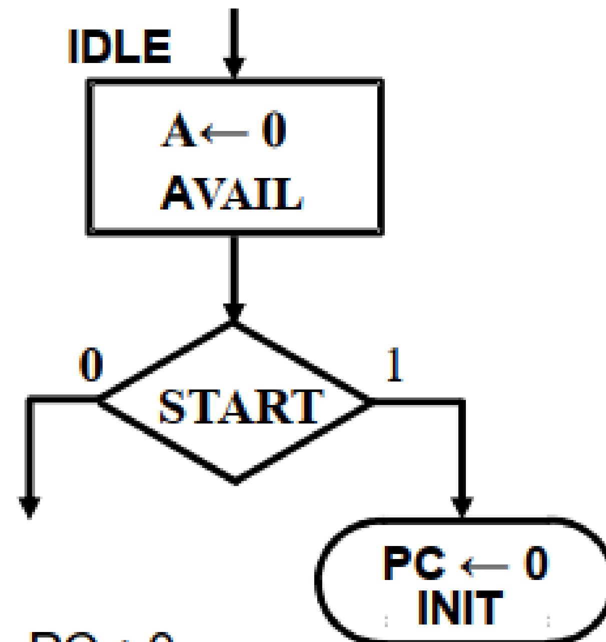
Conditional Output Box

- An oval with:
  - One input path from a decision box(es)
  - One output path
  - Register transfers or outputs that occur only if the conditional path to the box is taken.
- Transfers and outputs
  - in a state box are <u>Moore type</u> - dependent only on state
  - in a conditional output box are <u>Mealy type</u> - dependent on both state and inputs



Connecting Boxes Together

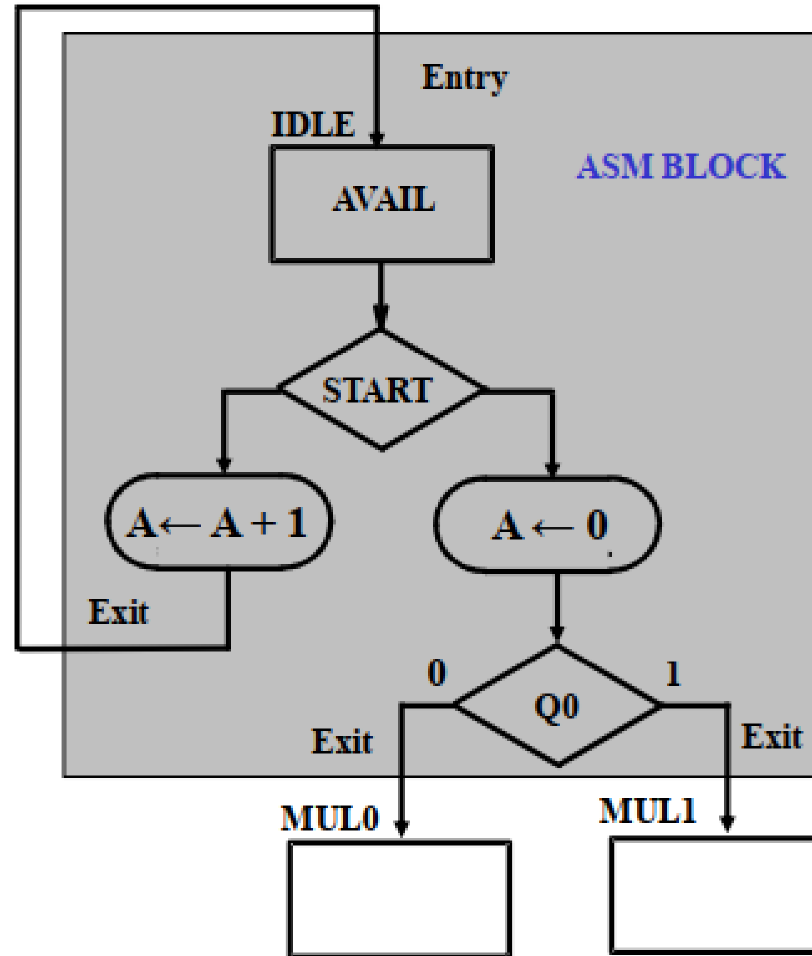
- By connecting boxes together, we see the power of expression.



- What are the:
  - Inputs? start
  - Outputs? Avail, Init
  - Conditional Outputs?
  - Transfers? A<-0, PC<-0
  - Conditional Transfers? INIT, transfer: PC<-0

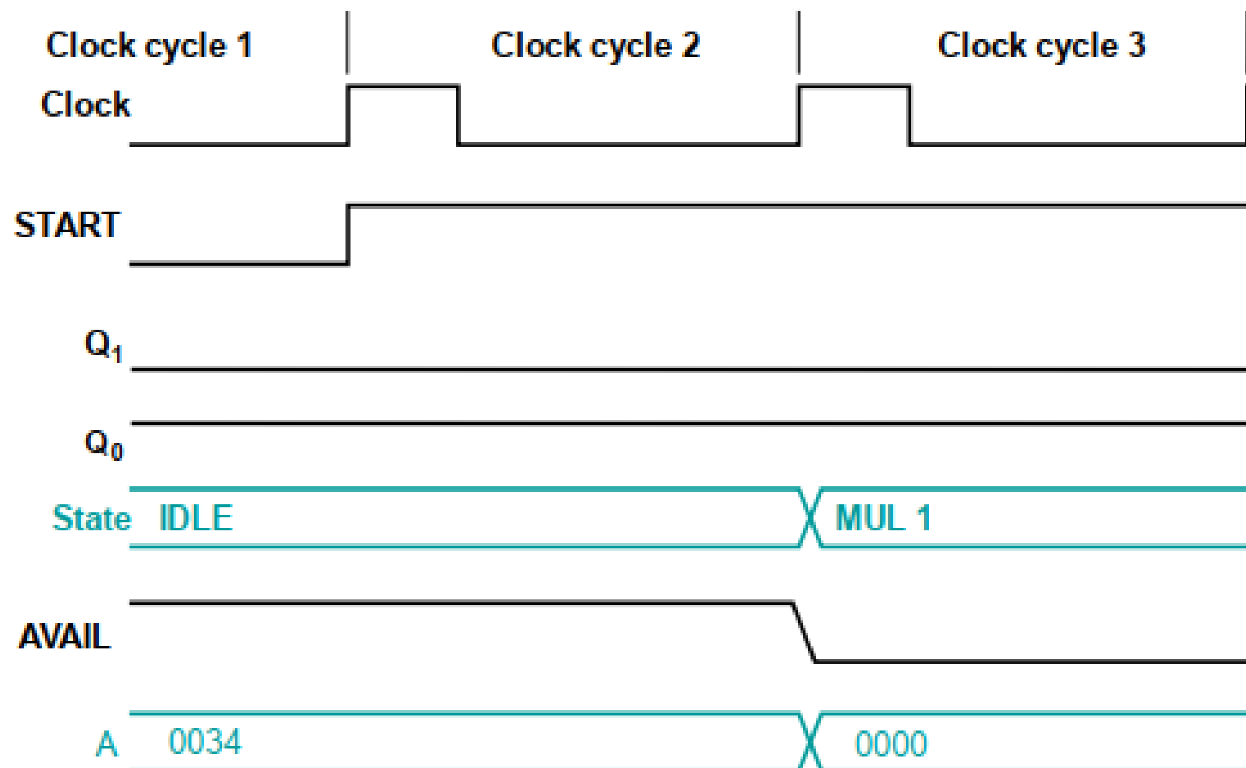
ASM Blocks

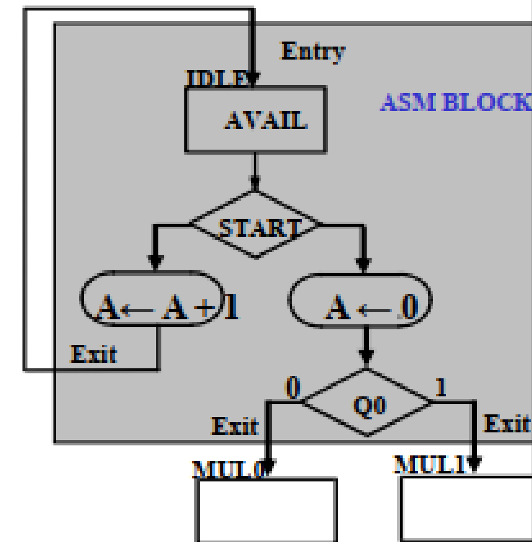
- One state box along with all decision and conditional output boxes connected to it, called an ASM Block. i.e., the ASM Block includes all items on the path from the current state to the same or other states.



ASM Timing

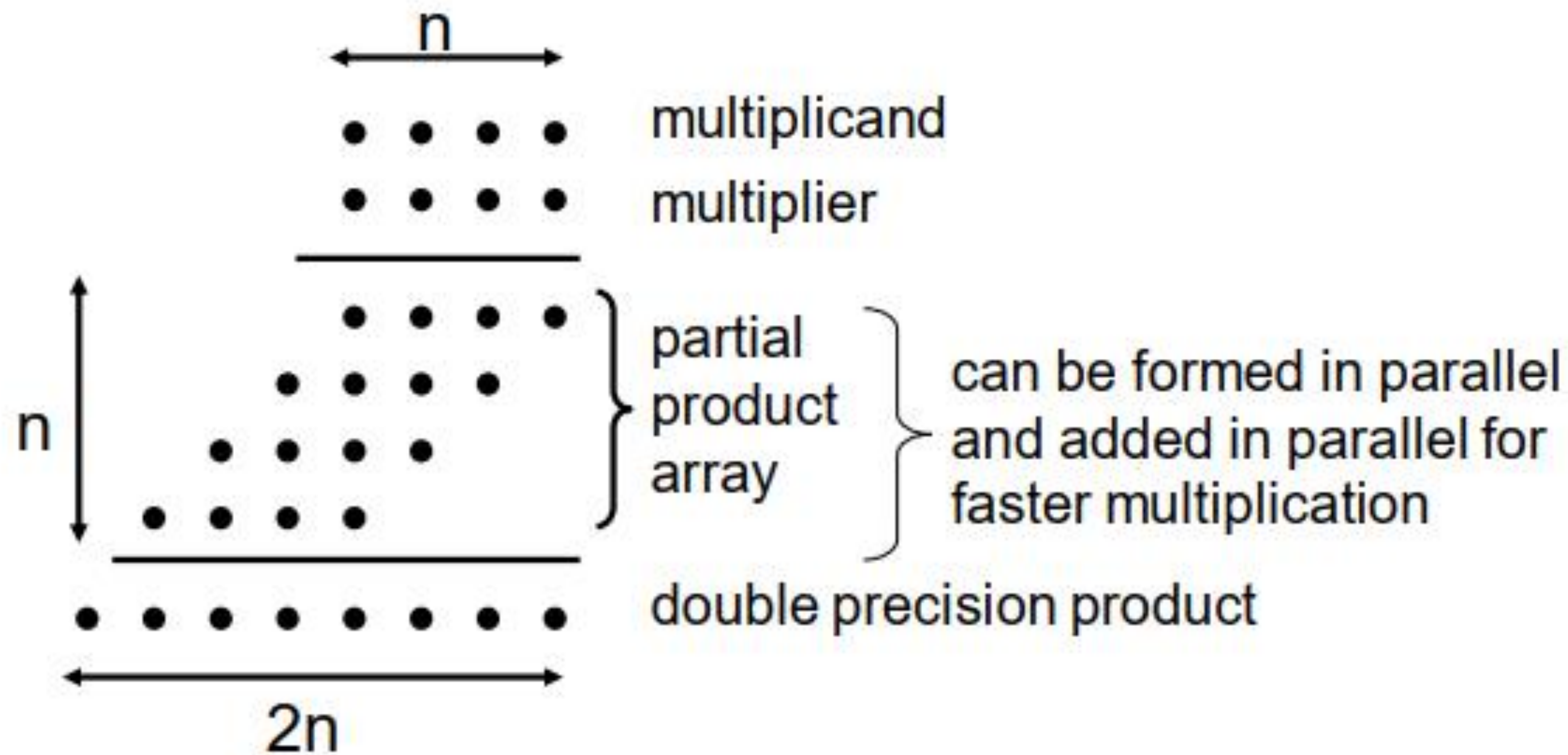
- Outputs appear while in the state
- Register transfers and conditional outputs occur at the clock while exiting the state - New value occur in the next state!





Multiply Overview

- **Binary multiplication is just a *bunch* of left shifts and adds**

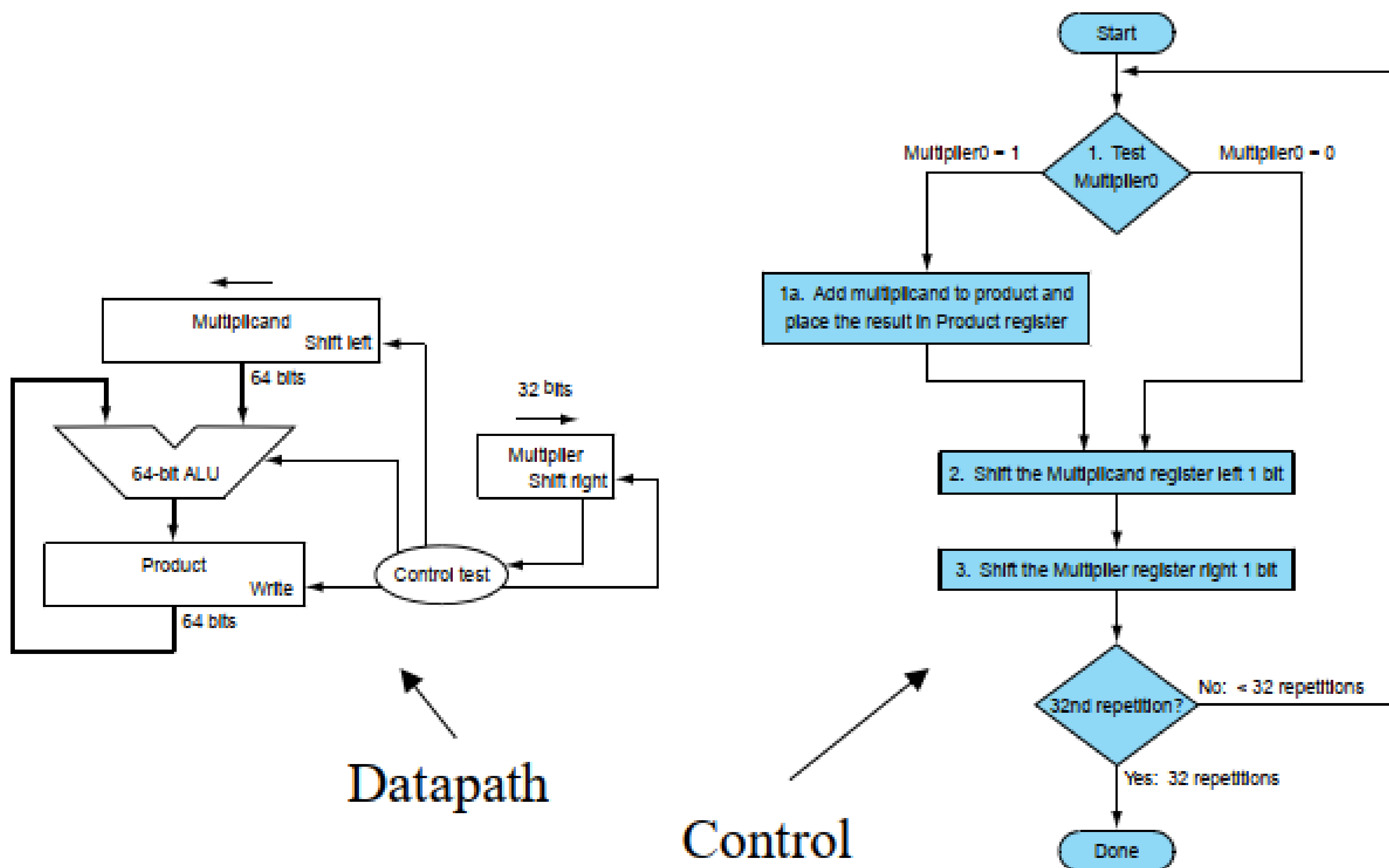


Multiplier Example

- Example: (101 **x** 011) Base 2
- Partial products are:
  101 x 0, 101 x 1, and 101 x 1

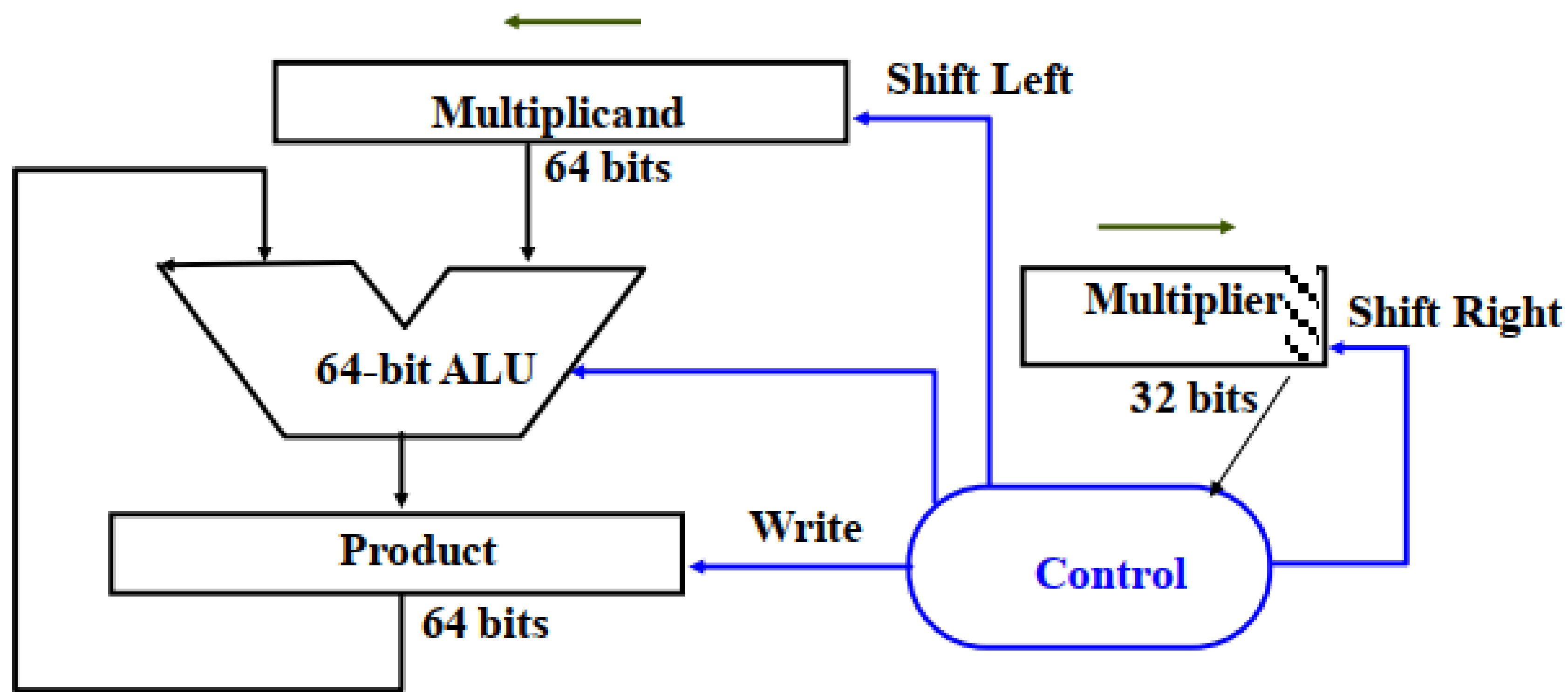
			1	0	1	multiplicand
		x	0	1	1	multiplier
			1	0	1	
		1	0	1		
	0	0	0			
0	0	1	1	1	1	

Multiplication: Implementation (version 1)

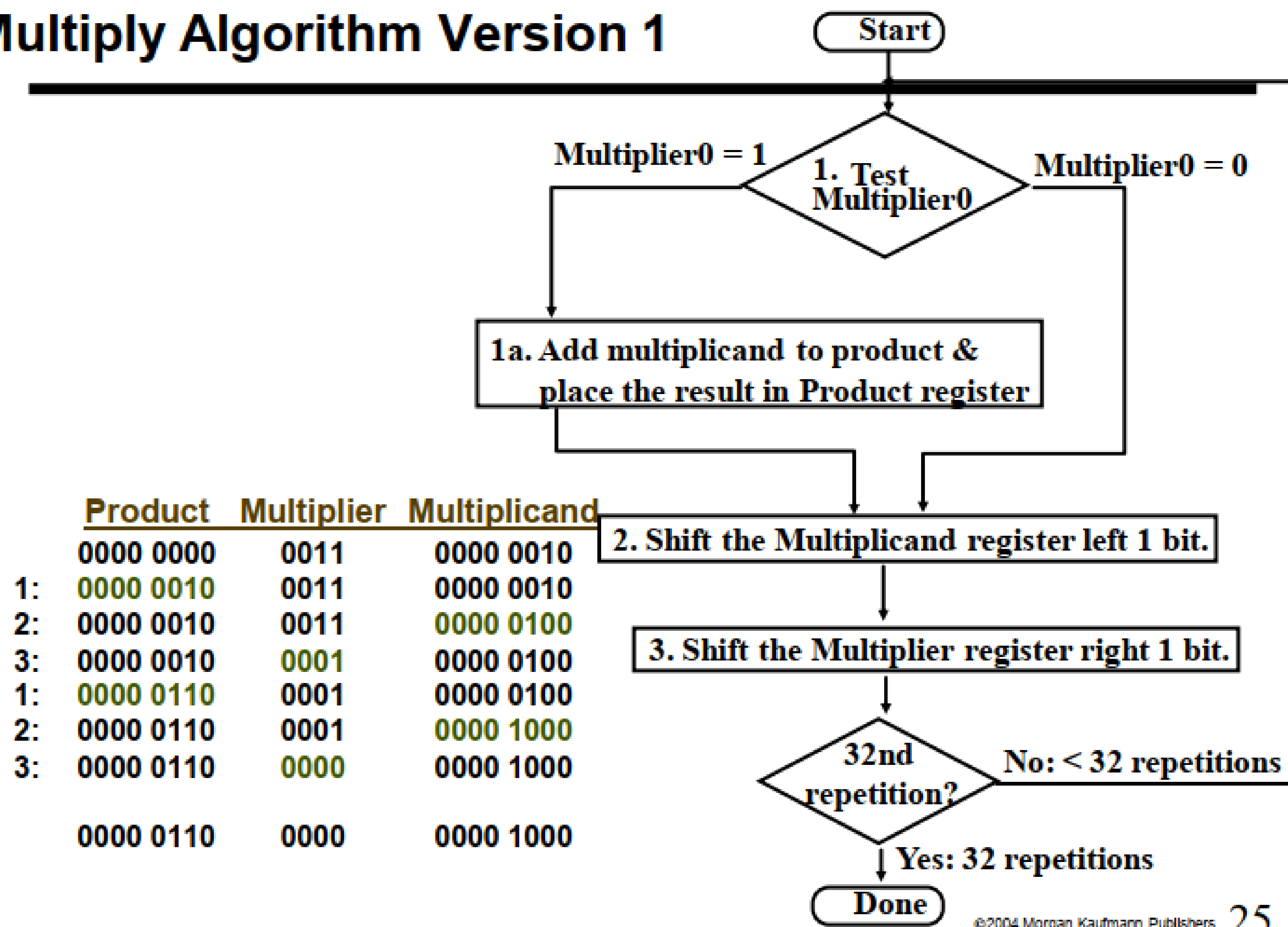


Unisigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control



	Product	Multiplier	Multiplicand
	0000 0000	0011	0000 0010
1:	0000 0010	0011	0000 0010
2:	0000 0010	0011	0000 0100
3:	0000 0010	0001	0000 0100
1:	0000 0110	0001	0000 0100
2:	0000 0110	0001	0000 1000
3:	0000 0110	0000	0000 1000
	0000 0110	0000	0000 1000

Observations on Multiply Version 1

- 1 clock per cycle => ≈ 100 clocks per multiply because of 32 repetitions, 3 steps in one repetition
  - Ratio of add/sub to multiply is from 5:1 to 100:1
  - Slow
- 0's inserted in the rightmost bit of multiplicand as shifting left
  => least significant bits of product never changed once formed
- 1/2 bits in multiplicand always 0
  - MSB are 0s at the beginning
  - 0 is inserted in LSB as multiplicand shifting left

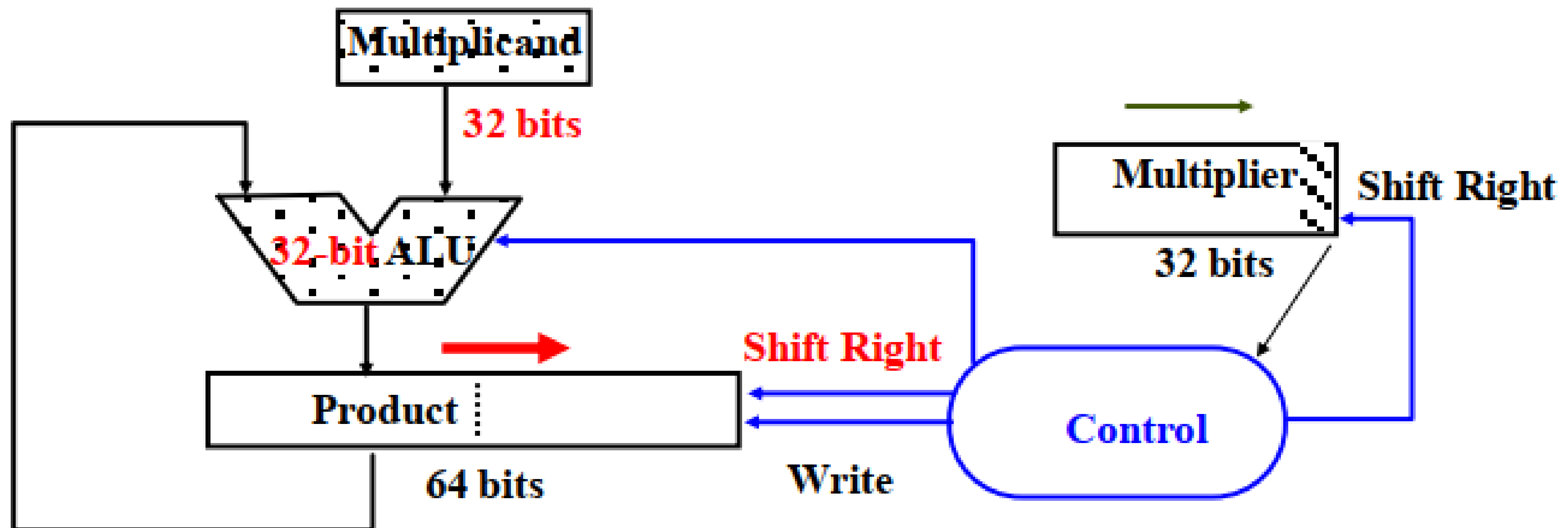
  => 64-bit multiplicand register is wasted

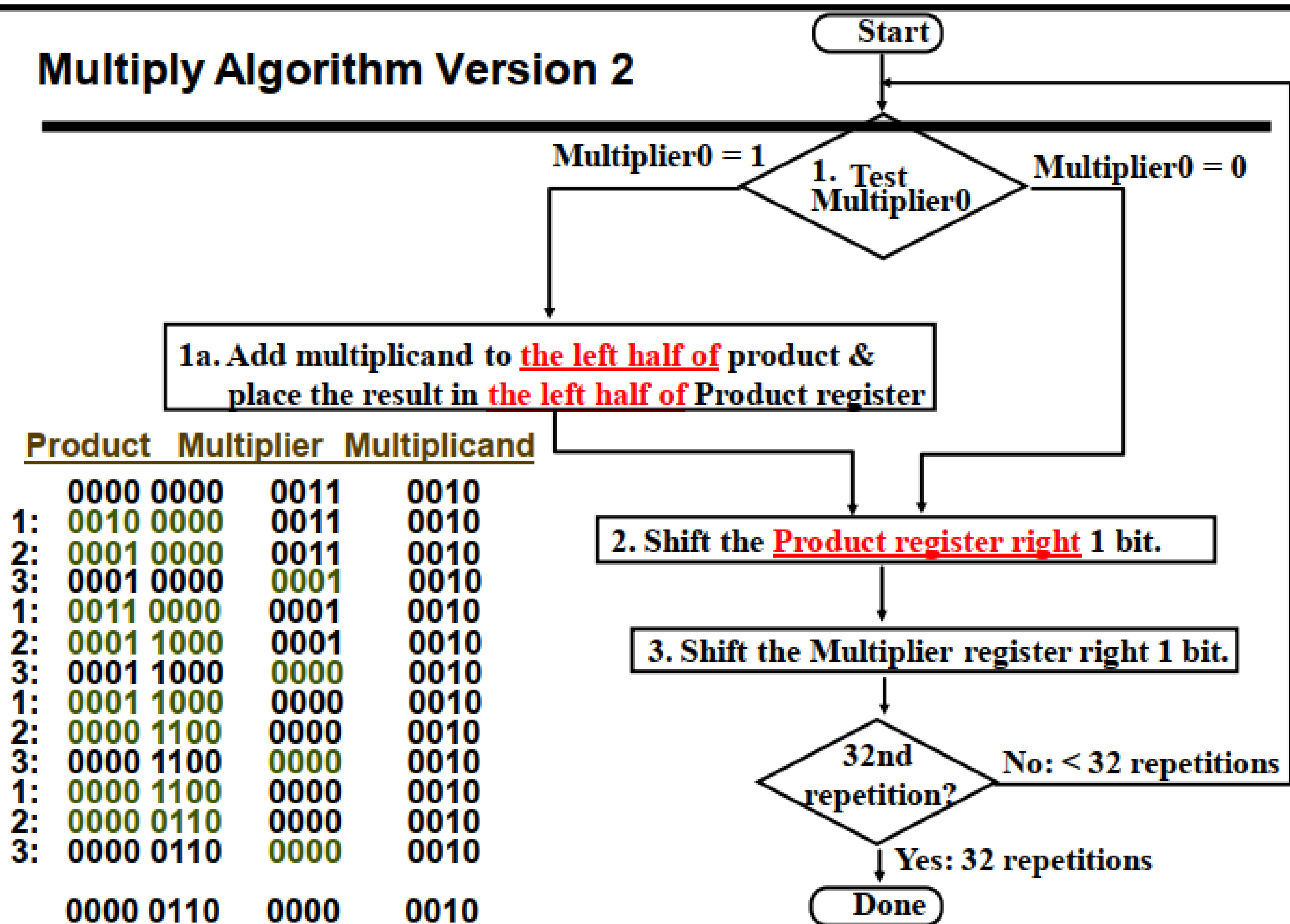
  => 64-bit adder is wasted
- Instead of shifting multiplicand to left, let's <u>shift *product* to right</u>

©2004 Morgan Kaufmann Publishers

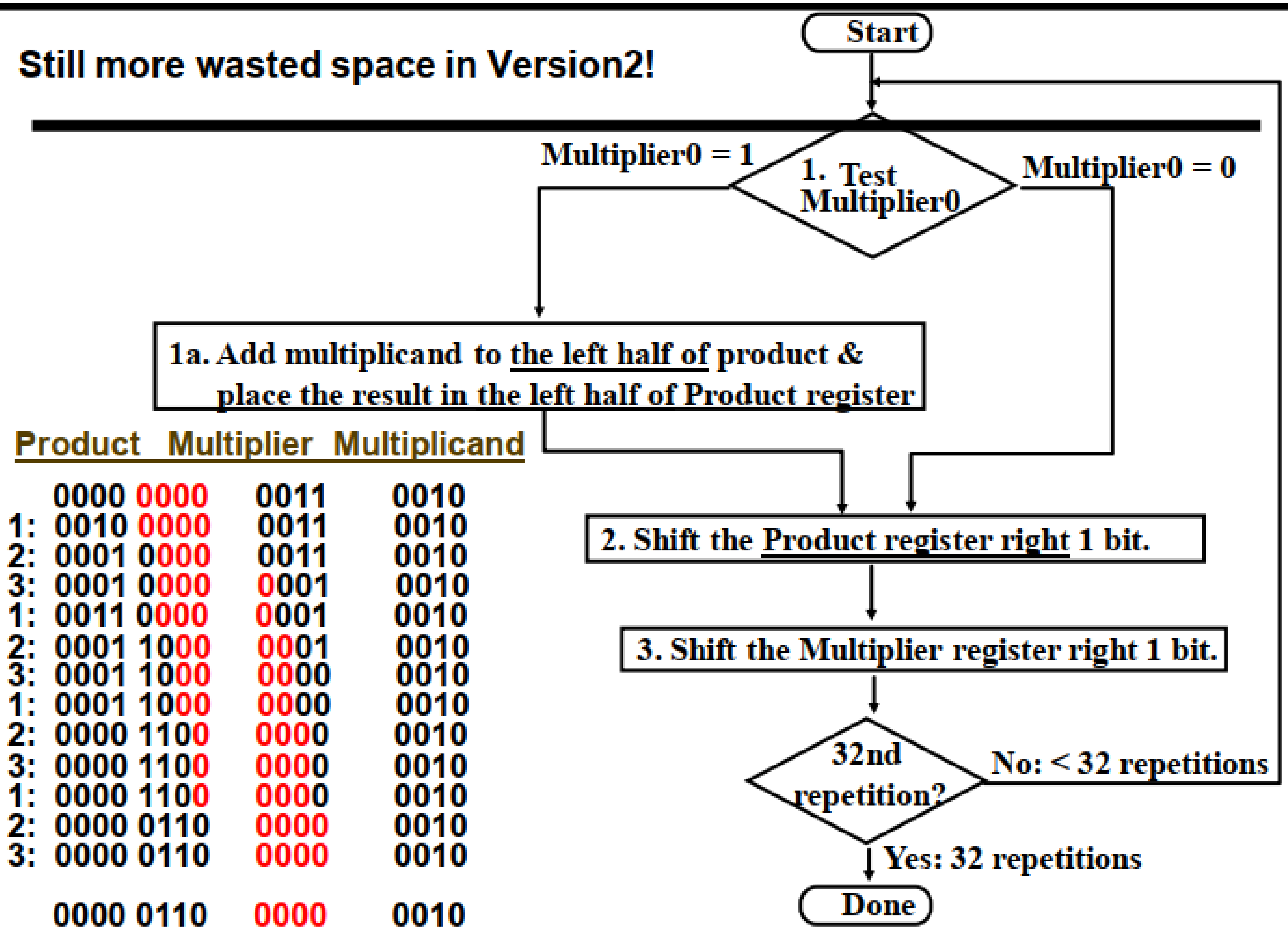
MULTIPLY HARDWARE Version 2

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg





©2004 Morgan Kaufmann Publishers


Still more wasted space in Version2!
Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to the left half of product & place the result in the left half of Product register
Product Multiplier Multiplicand
0000 0000 0011 0010
1: 0010 0000 0011 0010
2: 0001 0000 0011 0010
3: 0001 0000 0001 0010
1: 0011 0000 0001 0010
2: 0001 1000 0001 0010
3: 0001 1000 0000 0010
1: 0001 1000 0000 0010
2: 0000 1100 0000 0010
3: 0000 1100 0000 0010
1: 0000 1100 0000 0010
2: 0000 0110 0000 0010
3: 0000 0110 0000 0010
0000 0110 0000 0010
2. Shift the Product register right 1 bit.
3. Shift the Multiplier register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Observations on Multiply Version 2

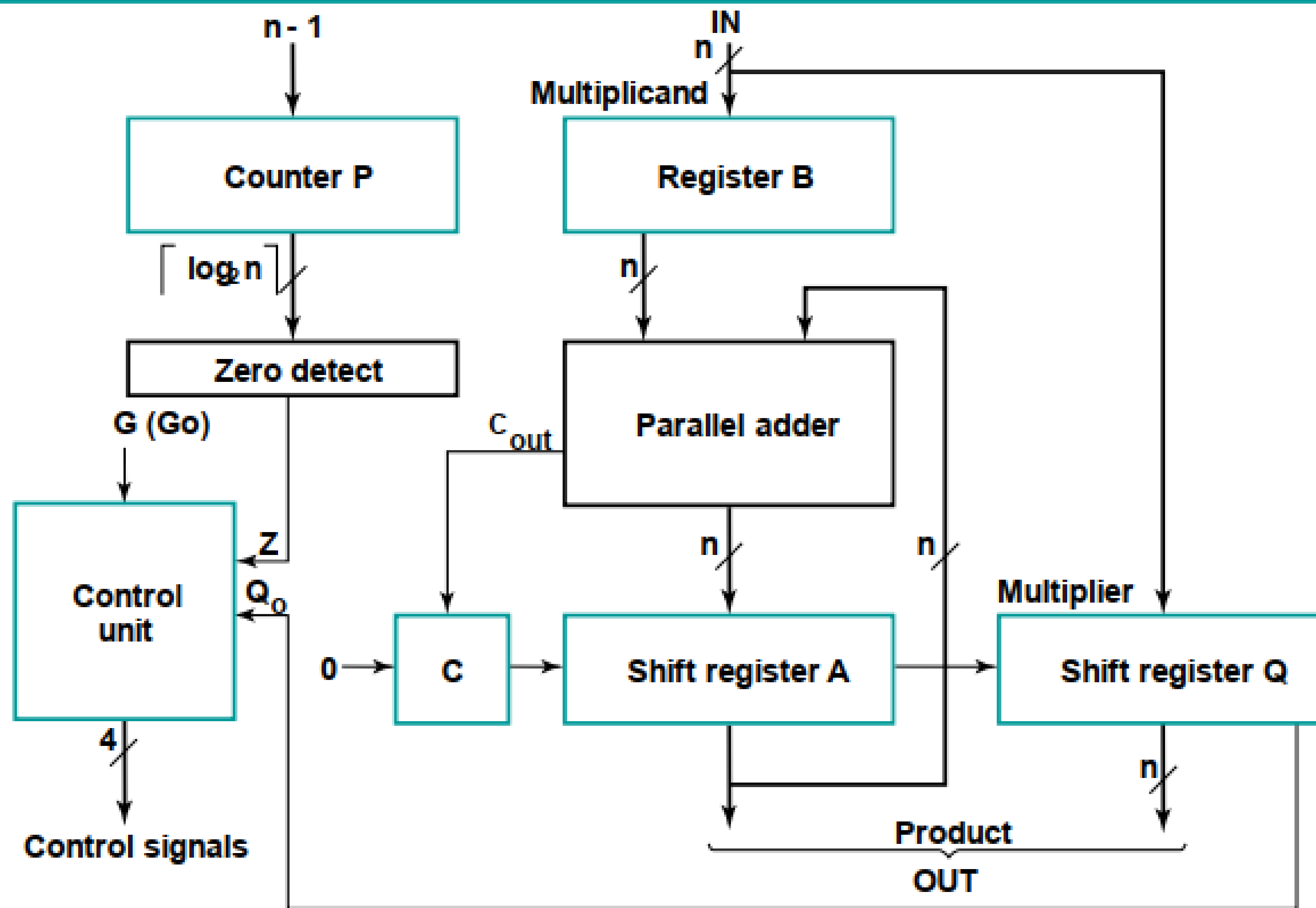
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

Example (1 0 1) x (0 1 1) Again

- Reorganizing example to follow hardware algorithm:

	1	0	1				← Multiplicand (B)
x	0	1	1				← Multiplier (Q)
0	0	0	0				Clear C || A (Carry and register A)
+	1	0	1				$\text{Multipler}_0 = 1 \Rightarrow$ Add B
0	1	0	1				Addition
0	0	1	0	1			Shift Right (Zero-fill C)
+	1	0	1				$\text{Multipler}_1 = 1 \Rightarrow$ Add B
0	1	1	1	1			Addition
0	0	1	1	1	1		Shift Right
0	0	0	1	1	1	1	$\text{Multipler}_2 = 0 \Rightarrow$ No Add, Shift Right

Multiplier Example: Block Diagram



Multiplexer Example: Operation

1. The multiplicand is loaded into register B.
2. The multiplier is loaded into register Q.
3. When G becomes 1, register C|| A is initialized to 0.
4. Down Counter P is initialized to n – 1 (n = number of bits in multiplier)
5. The partial products are formed in register C||A||Q.
6. Each multiplier (Q) bit, beginning with the LSB, is processed (if bit is 1, B is added to partial product of A; if bit is 0, do nothing)
7. C||A||Q is shifted right using the shift register
   - Partial product bits fill vacant locations in Q as multiplier is shifted out
   - If overflow during addition, the outgoing carry is recovered from C during the right shift
8. Steps 6 and 7 are repeated until P = 0 as detected by Zero detect.

University Of Diyala
College Of Engineering
Department of Computer Engineering

Digital System Design II

ASM Based Datapath and Control Design

Dr. Yasir Al-Zubaidi

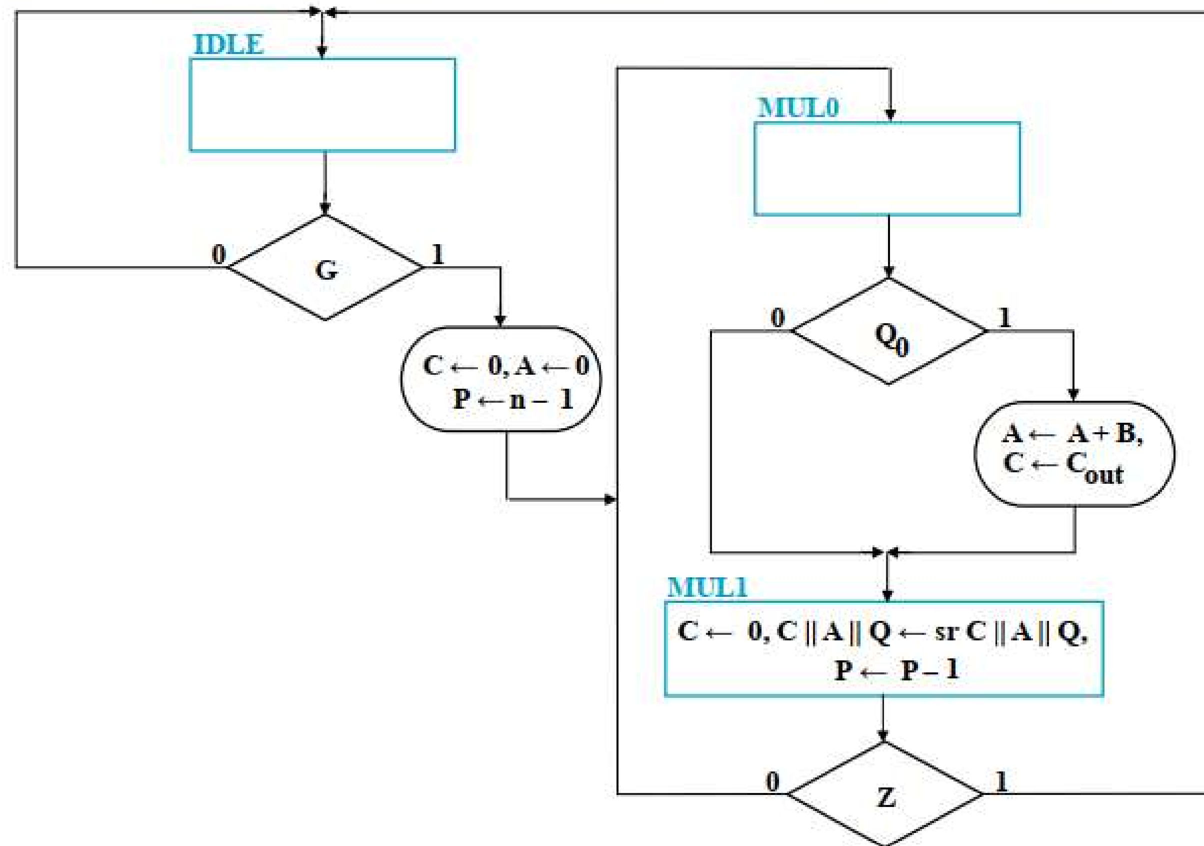
Third stage

2021

Overview

- Datapath and control
- Microoperations
- Sequencing and control
  - Algorithmic State Machines (ASM)
    - ASM chart
    - Timing considerations
    - ASM chart examples: Binary multiplier
  - Hardwired Control
    - Control design methods
    - Sequence register and decoder
    - One flip-flop per state
  - Microprogrammed control

Multiplier Example: ASM Chart



Multiplier Example: ASM Chart (Contd.)

- Three states employed here:
  - IDLE state:
    - input G is used as the condition for starting the multiplication
    - C, A, and P are initialized
  - MUL0 state: conditional addition is performed based on the value of Q_0.
  - MUL1 state:
    - right shift is performed to capture the partial product and position the next bit of the multiplier in Q_0
    - Down counter P = P - 1
    - P=0 is used to sense completion or continuation of the multiplication.

Multiplier Example: Control Signal Table

Control Signals for Binary Multiplier

Block Diagram Module	Microoperation	Control Signal Name	Control Expression
Register A:	$A \leftarrow 0$	Initialize	$\text{IDLE} \cdot G$
	$A \leftarrow A + B$	Load	$\text{MUL0} \cdot Q_0$
	$C \parallel A \parallel Q \leftarrow \text{sr } C \parallel A \parallel Q$	Shift_dec	MUL1
Register B:	$B \leftarrow IN$	*Load_B*	LOADB
Flip-Flop C:	$C \leftarrow 0$	Clear_C	$\text{IDLE} \cdot G + \text{MUL1}$
	$C \leftarrow C_{out}$	Load	—
Register Q:	$Q \leftarrow IN$	*Load_Q*	LOADQ
	$C \parallel A \parallel Q \leftarrow \text{sr } C \parallel A \parallel Q$	Shift_dec	—
Counter P:	$P \leftarrow n - 1$	Initialize	—
	$P \leftarrow P - 1$	Shift_dec	—

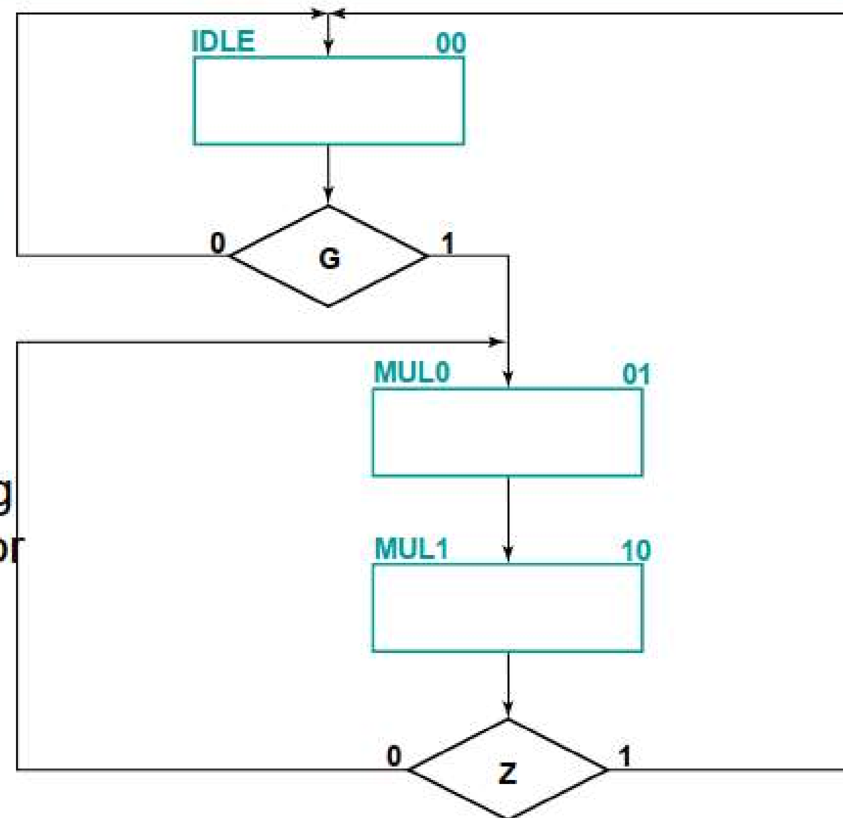
Multiplier Example: Control Signal Table (Contd.)

- Signals are defined on a register basis
- LOADQ and LOADB: external signals controlled from the system using the multiplier and will not be considered a part of this design
- Many control signals are "reused" for different registers.
  - These 4 control signals are the "outputs" of the control unit: initialize, load, shift_dec, clear_c

Multiplier Example - Sequencing Part of ASM

- With the outputs represented by the table, they can be removed from the ASM making the ASM to represent only the sequencing (next state) behavior

Similar to FSM

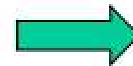


Hardwired Control

- Control Design Methods
  - Procedure specializations that use a single signal to represent each state
    - Sequence Register and Decoder
      - Sequence register with encoded states, e.g., 00, 01, 10, 11.
      - Decoder outputs produce "state" signals, e.g., 0001, 0010, 0100, 1000.
    - One Flip-flop per State
      - Flip-flop outputs as "state" signals, e. g., 0001, 0010, 0100, 1000.

Multiplier Example: Sequencer and Decoder Design - Specification

- Initially, use sequential circuit design techniques
- First, define:
  - States: IDLE, MUL0, MUL1
  - Input Signals: G, Z, Q_0 (Q_0 affects outputs, not next state)
  - Output Signals: Initialize, LOAD, Shift_Dec, Clear_C
  - State Transition Diagram (Use Sequencing ASM)
  - Output Function: Use Control Signal Table
- Second, find
  - State Assignments
  - Use two state bits to encode the three states IDLE, MUL0, and MUL1.



State	M1	M0
IDLE	0	0
MUL0	0	1
MUL1	1	0
Unused	1	1

Multiplier Example: Sequencer and Decoder Design - Formulation

- Assuming that state variables M1 and M0 are decoded into states, the next state part of the state table is:

Current State	Input G Z	Next State M1 M0
IDLE	0 0	0 0
IDLE	0 1	0 0
IDLE	1 0	0 1
IDLE	1 1	0 1
MUL0	0 0	1 0
MUL0	0 1	1 0
MUL0	1 0	1 0
MUL0	1 1	1 0

Current State M1 M0	Input G Z	Next State M1 M0
MUL1	0 0	0 1
MUL1	0 1	0 0
MUL1	1 0	0 1
MUL1	1 1	0 0
Unused	0 0	d d
Unused	0 1	d d
Unused	1 0	d d
Unused	1 1	d d

Multiplier Example: Sequencer and Decoder Design –Equations Derivation/Optimization

- Finding the equations for M1 and M0 using decoded states:

  $M1 = MUL0$

  $M0 = IDLE \cdot G + MUL1 \cdot \overline{Z}$

- The output equations using the decoded states:

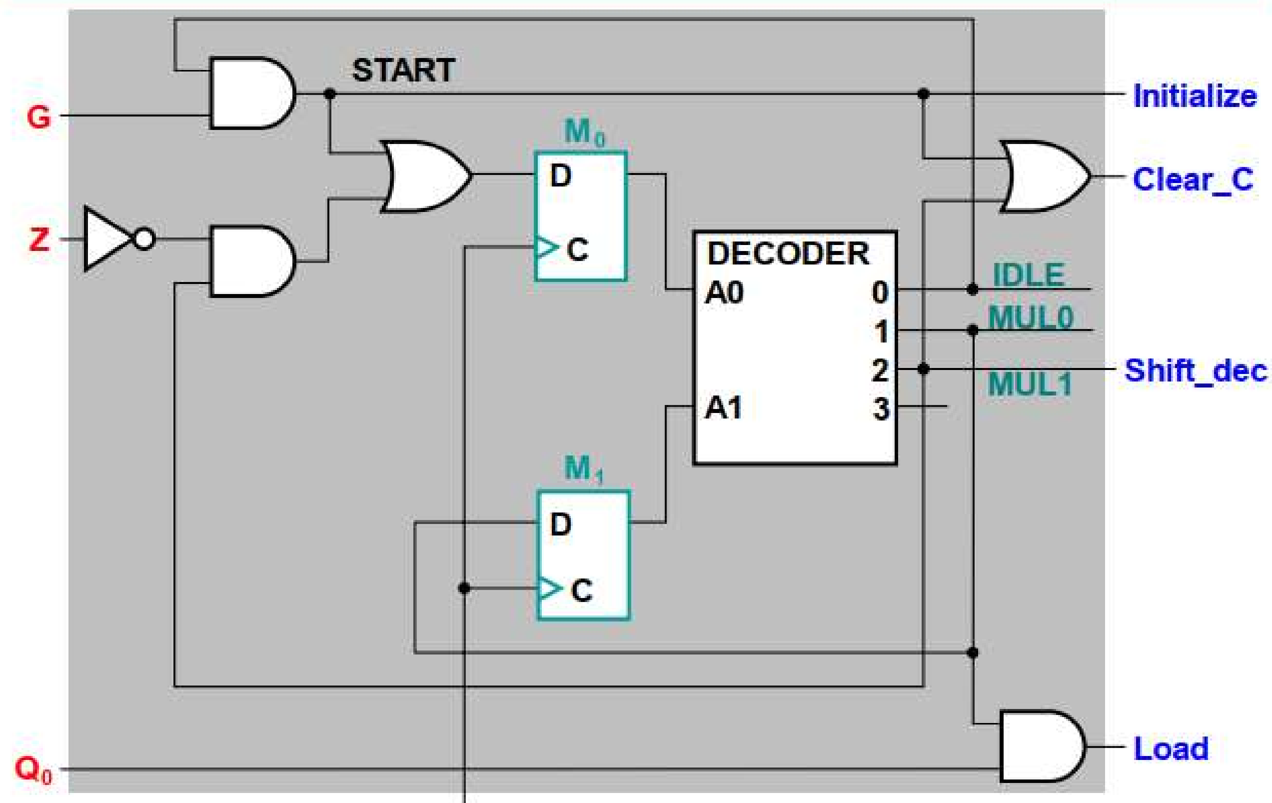
  $Initialize = IDLE \cdot G$
  $Load = MUL0 \cdot Q_0$
  $Clear_C = IDLE \cdot G + MUL1$
  $Shift_dec = MUL1$

- Doing multiple level optimization, extract IDLE · G:

  $START = IDLE \cdot G$
  $M1 = MUL0$
  $M0 = START + MUL1 \cdot \overline{Z}$
  $Initialize = START$
  $Load = MUL0 \cdot Q_0$
  $Clear_C = START + MUL1$
  $Shift_dec = MUL1$

- The resulting circuit using flip-flops, a decoder, and the above equations is given on the next slide.

Multiplier Example: Sequencer and Decoder Design - Implementation



```
--Binary multiplier with n=4
library ieee;
use ieee.std_logic_unsigned.all;
entity binary_multiplier is
      port(CLK, RESET, G, LOADB, LOADQ: in std_logic;
         MULT_IN : in std_logic_vector (3 downto 0);
         MULT_OUT : out std_logic_vector (7 downto 0));
end bianry_multiplier;

architecture behavior_4 of binary_multiplier is
      type state_type is (IDLE, MUL0, MUL1);
      variable P:=3;
      signal state, next_state : state_type;
      signal A, B, Q:std_logic_vector(3 downto 0);
      signal C, Z:std_logic;
begin
      Z<= P(1) NOR P(0);
      MULT_OUT <= A & Q;

      state_register : process (CLK, RESET)
      begin
        if (RESET = '1') then
          state <= IDLE;
        elsif (CLK'event and CLK='1') then
          state <= next_state;
        endif;
      end process;

      next_state_func : process (G, Z, state)
      begin
        case state is
          when IDLE =>
                 if G='1' then    next_state <= MUL0;
                 else             next_state <= IDLE;
                 end if;
          when MUL0 =>
                 next_state <= MUL1;
          when MUL1 =>
                 if Z='1' then    next_state <= IDLE;
                 else             next_state <= MUL0;
                 end if;
        end case;
      end process;

datapath_func : process (CLK)
variable CA: std_logic_vector (4 downto 0);
begin
  if (CLK'event and CLK='1') then
     if LOADB='1' then
       B <= MULT_IN;
     end if;
     if LOADQ = '1' then
       Q <= MULT_IN;
     end if;
     case state is
       when IDLE =>
          if G = '1' then
               C <= '0';
               A <= "0000";
               P <= "11";
          end if;
       when MUL0 =>
          if Q(0) ='1' then
               CA := ('0' & A) + ('0' & B);
          else
               CA := C & A;
          end if;
          C <= CA(4);
          A <= CA(3 downto 0);
       when MUL1 =>
          C <= '0';
          A <= C & A(3 downto 1);
          Q <= A(0) & Q(3 downto 1);
          P <= P - "01";
     end case;
  end if;
end process;

end behavior_4;
```

Speeding Up the Multiplier

- In processing each bit of the multiplier, the circuit visits states MUL0 and MUL1 in sequence.
- By redesigning the multiplier, is it possible to visit only a single state per bit processed?

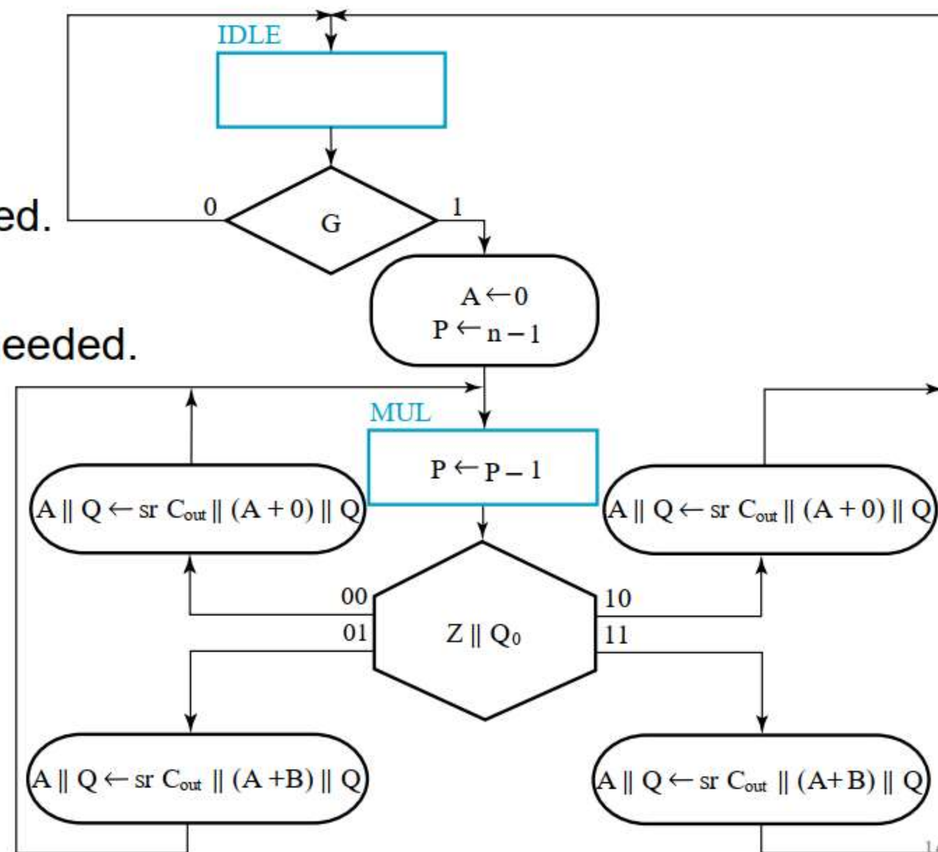
Speeding Up Multiply (Contd.)

- The operations in MUL0 and MUL1:
  - In MUL0, a conditional add of B
  - In MUL1, a right shift of C || A || Q in a shift register, the decrementing of P, and a test for P = 0 (on the old value of P)

- Any solution that uses one state must combine all of the operations listed into one state
  - The operations involving P are already done in a single state, so not a problem.
  - The right shift, however, depends on the result of the conditional addition. So these two operations must be combined!

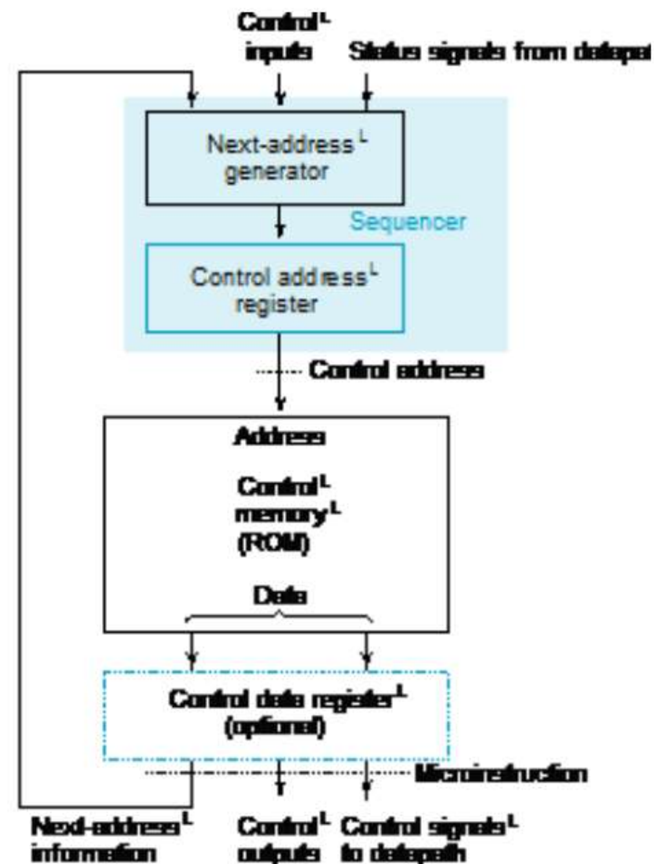
Speeding Up Multiply (Contd.)

- By replacing the shift register with a combinational shifter and combining the adder and shifter, the states can be merged.
- The C-bit is no longer needed.
- In this case, Z and Q_0 have been made into a vector.



Microprogrammed Control

- *Microprogrammed Control* — a control unit with binary control values stored as <u>words</u> in <u>memory.</u>
- *Microinstructions* — words in the control memory.
- *Microprogram* — a sequence of microinstructions.
- *Control Memory* — RAM or ROM memory holding the microinstructions.
    - *Writeable Control Memory* — RAM Memory into which microinstructions may be written



University Of Diyala
College Of Engineering
Department of Computer Engineering

Digital System Design II
Memories Overview

Dr. Yasir Al-Zubaidi

Third stage

2021

Memories

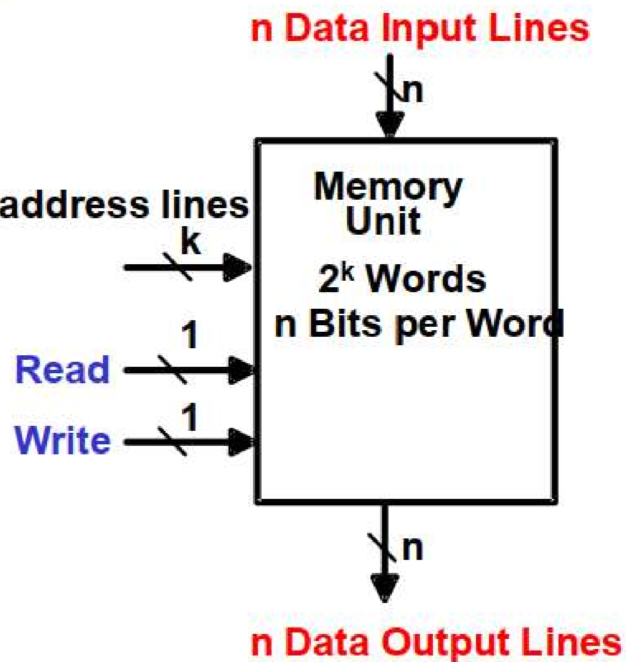
<table>
<tr><th colspan="3">Read-Write Memory</th><th>Read-Only Memory</th></tr>
<tr><td colspan="2">Volatile Memory</td><td>Non-volatile Memory</td><td rowspan="3">Mask-Programmed ROM (PROM) (nonvolatile)</td></tr>
<tr><td>Random Access</td><td>Sequential Access</td><td rowspan="2">EPROM
EEPROM
FLASH</td></tr>
<tr><td>DRAM
SRAM</td><td>FIFO
LIFO
Shift Register
CAM</td></tr>
</table>

- **Volatile: need electrical power**
- **Nonvolatile: magnetic disk, retains its stored information after the removal of power**
- **Random access: memory locations can be read or written in a random order**
- **EPROM: erasable programmable read-only memory**
- **EEPROM: electrically erasable programmable read-only memory**
- **FLASH: memory stick, USB disk**
- **Access pattern: sequential access: (video memory streaming) first-in-first-out (buffer), last-in-first-out (stack), shift register, content-addressable memory**

Memories

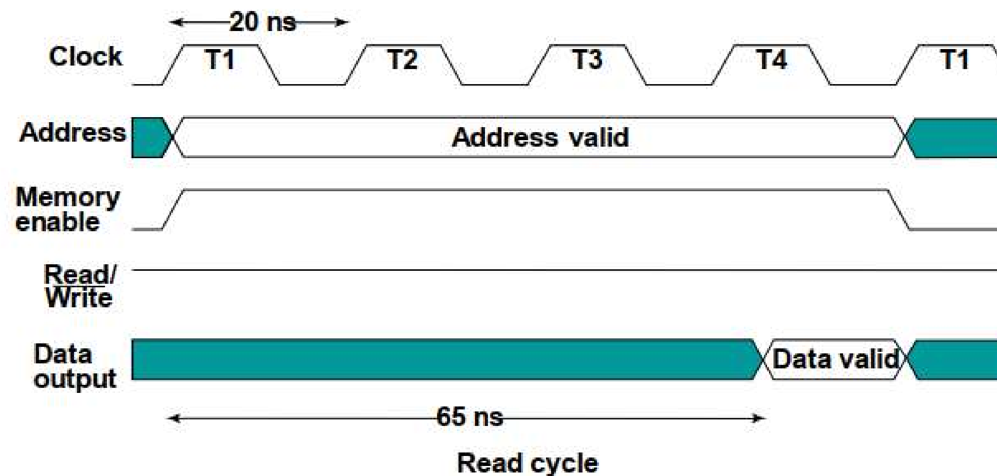
- k address lines are decoded to address 2^k words of memory.
- Each word is n bits.
- Read and Write are single control lines defining the simplest memory operations.

A basic memory system:



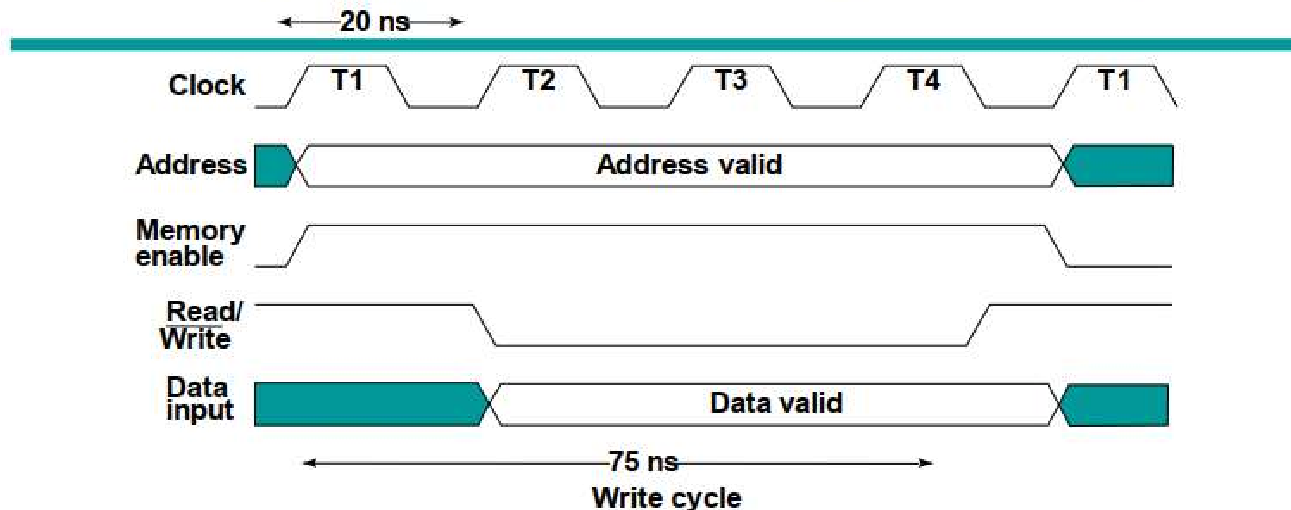
Memory Operation Timing - Reading

- Most basic memories are asynchronous
  - Storage in latches or storage of electrical charge
  - No clock
  - Controlled by control inputs and address, which are controlled by CPU and synchronized by its own clock
- Timing of signal changes/data observation is critical to the operation



- Read cycle: the access time, the maximum time from the application of the address to the appearance of the data at the Data output.

Memory Operation Timing - Writing



- Write cycle: the maximum time from the application of the address to the completion of all internal operations required to store a word
- Critical times measured with respect to edges of write pulse (1-0-1):
  - Address must be established at least a specified time before 1-0 and held for at least a specified time after 0-1 to avoid disturbing stored contents of other addresses
  - Data must be established at least a specified time before 0-1 and held for at least a specified time after 0-1 to write correctly

VHDL code for ROM

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY rom8x4 IS
 PORT (
 addr: in std_logic_vector(2 downto 0);
 q: out std_logic_vector(3 downto 0));
END rom8x4;
```

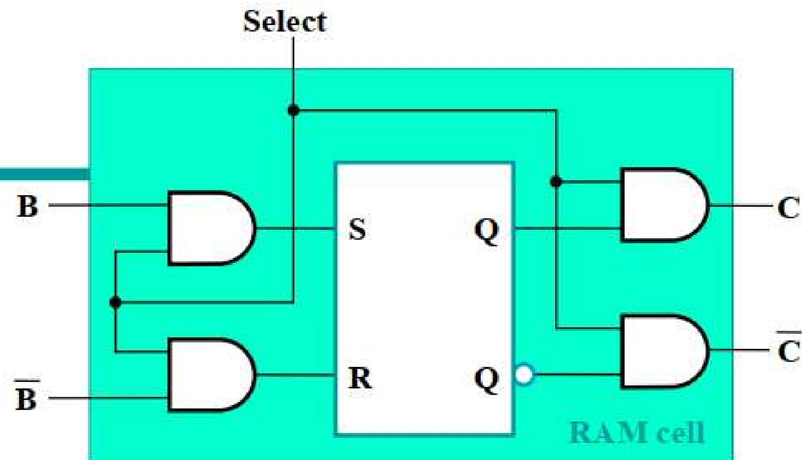
```
ARCHITECTURE behav OF rom8x4 IS
BEGIN

PROCESS(addr)
BEGIN
  CASE addr IS
    when "000" => q <= "0001";
    when "001" => q <= "0000";
    when "010" => q <= "0111";
    when "011" => q <= "1101";
    when "100" => q <= "1000";
    when "101" => q <= "1100";
    when "110" => q <= "0110";
    when "111" => q <= "1011";
    when others  =>  NULL;
  END case;
 END process;
END behav;
```

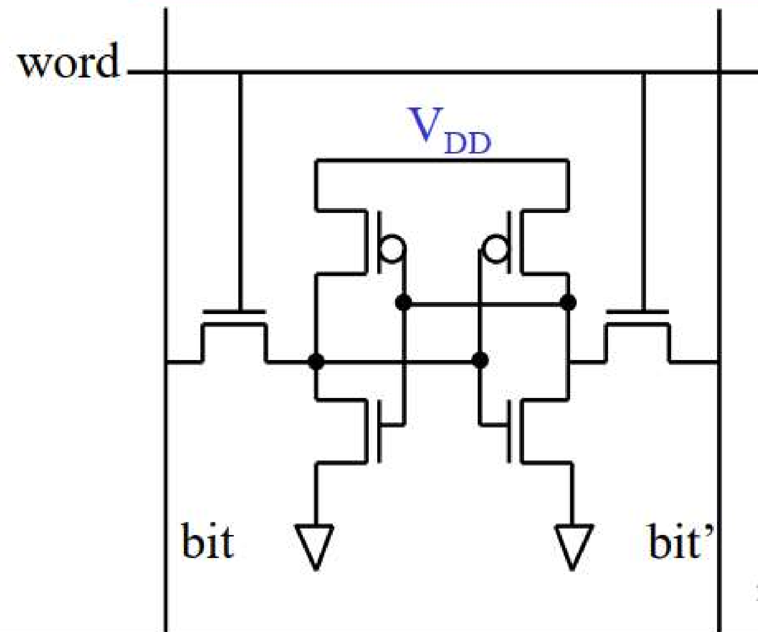
Random Access Memories (RAMs)

- Read/Write memory
- Types:
  - Static RAM (SRAM):
    - Once a word is written at a location, it remains stored as long as power is applied to the chip, unless the same location is written again.
    - Fast speed, but their cost per bit higher.
    - Application: Caches memories in Microprocessor
  - Dynamic RAM (DRAM):
    - The data stored at each location must be periodically refreshed by reading it and then writing it back again, otherwise it disappears.
    - Their density is greater and their cost per bit lower, but the speed is slower.

SRAM Cell

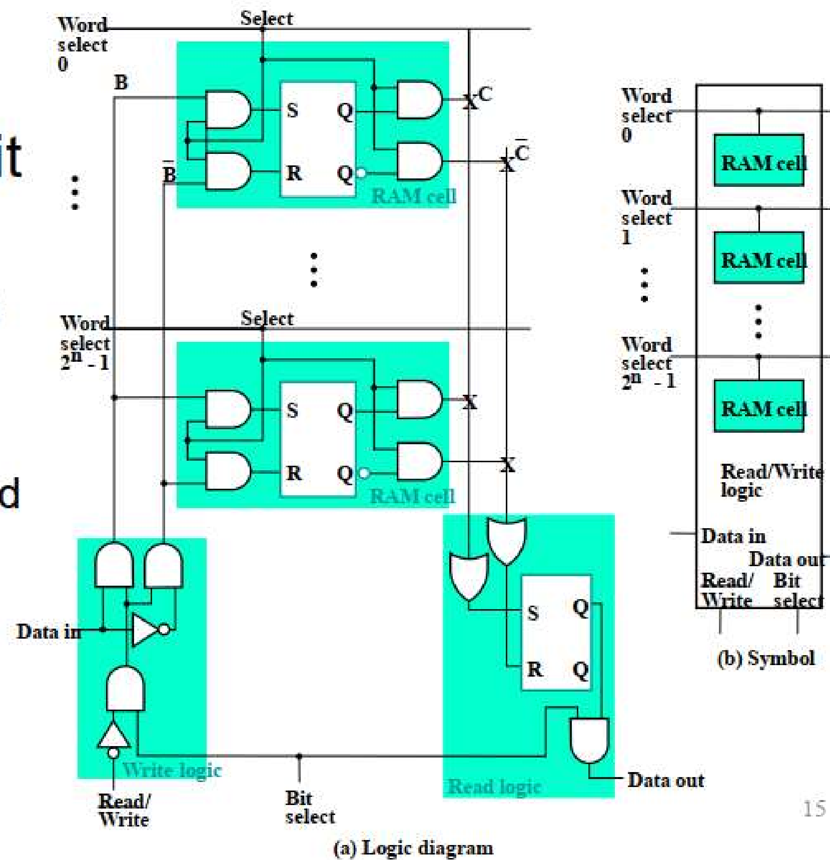
- Array of storage cells used to implement static RAM
- Storage Cell
  - SR Latch
  - Input "Select" for control
  - Dual Rail Data
    - Inputs B and $\overline{B}$
    - Outputs C and $\overline{C}$





SRAM Bit Slice

- Represents all circuitry that is required for 2^n 1-bit words
  - Multiple RAM cells
  - Control Lines:
    - Word select i – one for each word
    - Read/$\overline{\text{Write}}$
    - Bit Select
  - Data Lines:
    - Data in
    - Data out

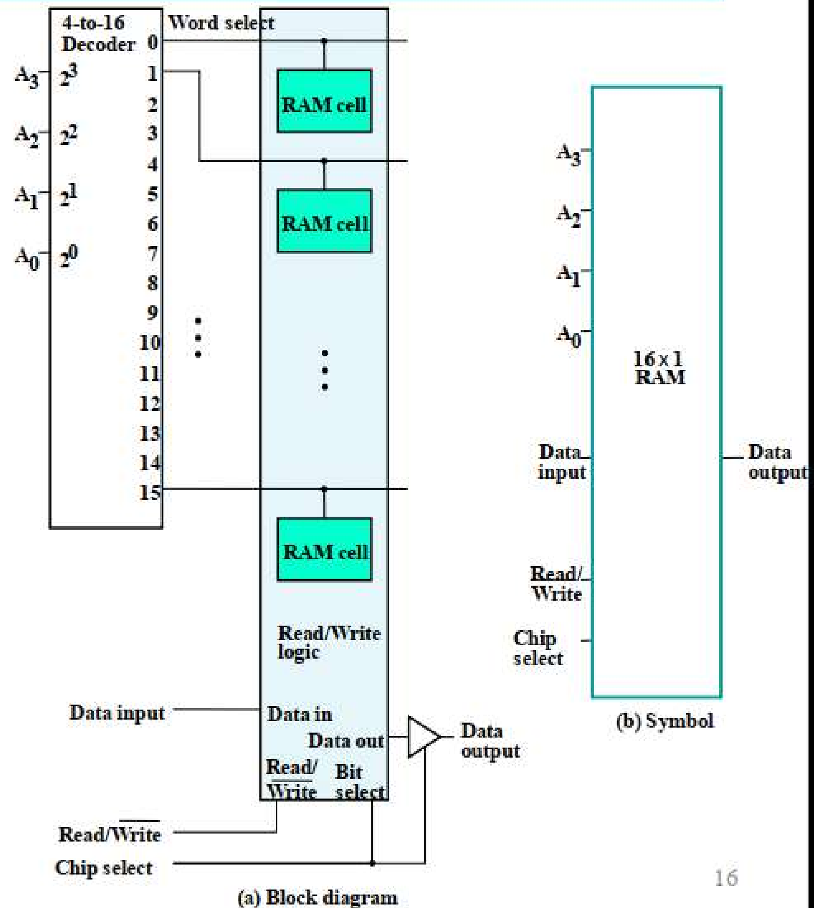


(a) Logic diagram

(b) Symbol

2^n-Word by 1-Bit RAM IC

- To build a RAM IC from a RAM slice:
  - Decoder decodes the n address lines to 2^n word select lines
  - A 3-state buffer on the data output permits RAM ICs to be combined into a RAM with $c \times 2^n$ words



Cell Arrays and Coincident Selection

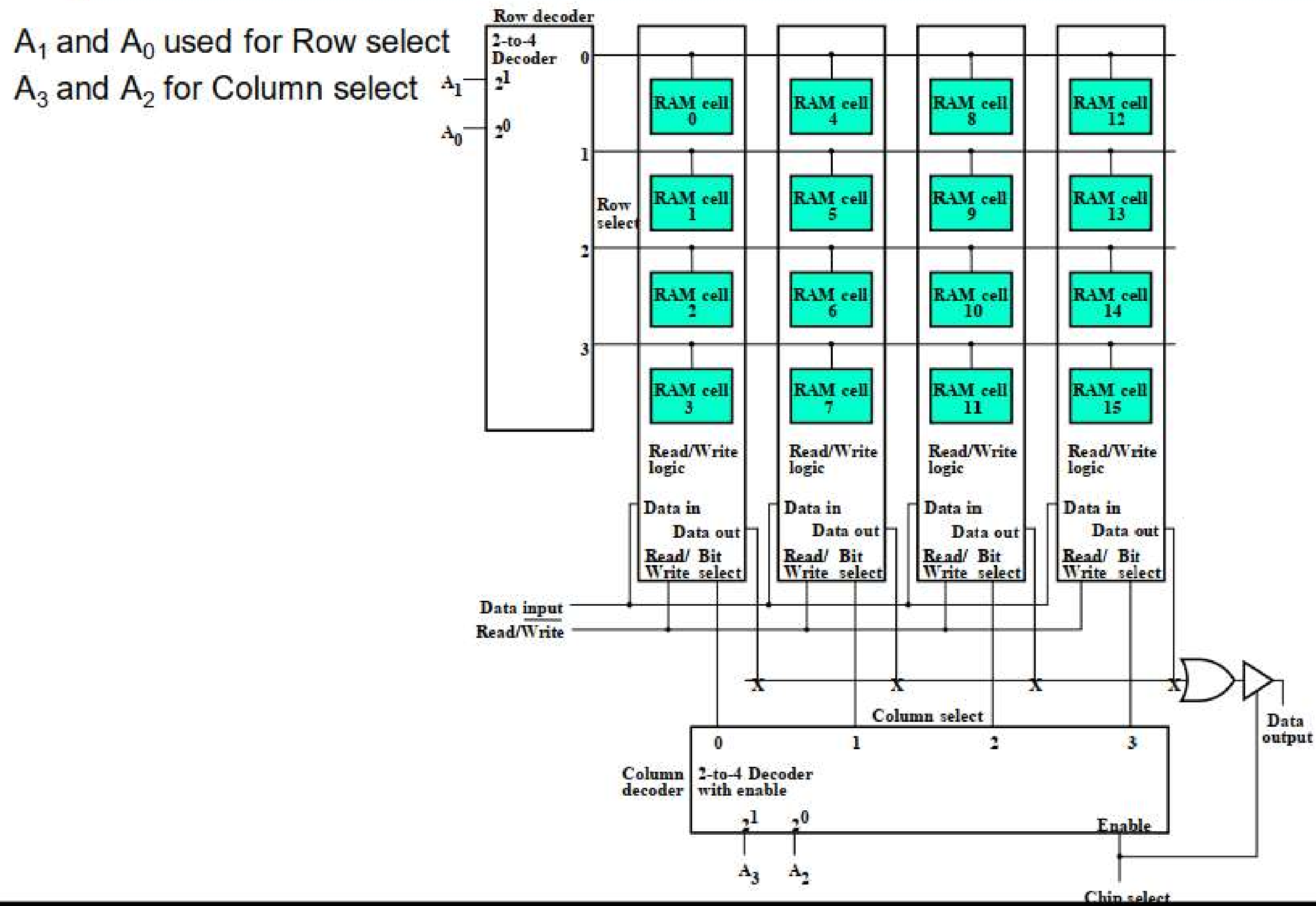
- Memory arrays can be very large =>
  - Large decoders
  - Large fanouts for the input bit lines
  - The decoder size and fanouts can be reduced by approximately $\sqrt{n}$ using a coincident selection in a 2-D array: uses two decoders, one for words and one for bits:
    - Word select becomes Row select
    - Bit select becomes Column select

- See next slide for example

Cell Arrays and Coincident Selection (Contd.)

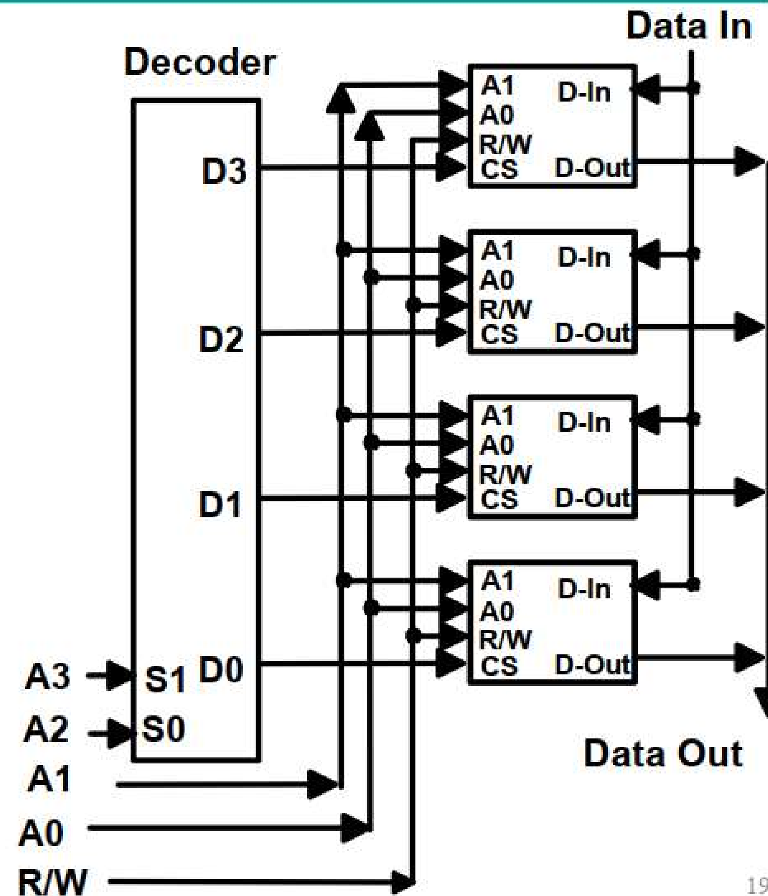
A_1 and A_0 used for Row select

A_3 and A_2 for Column select



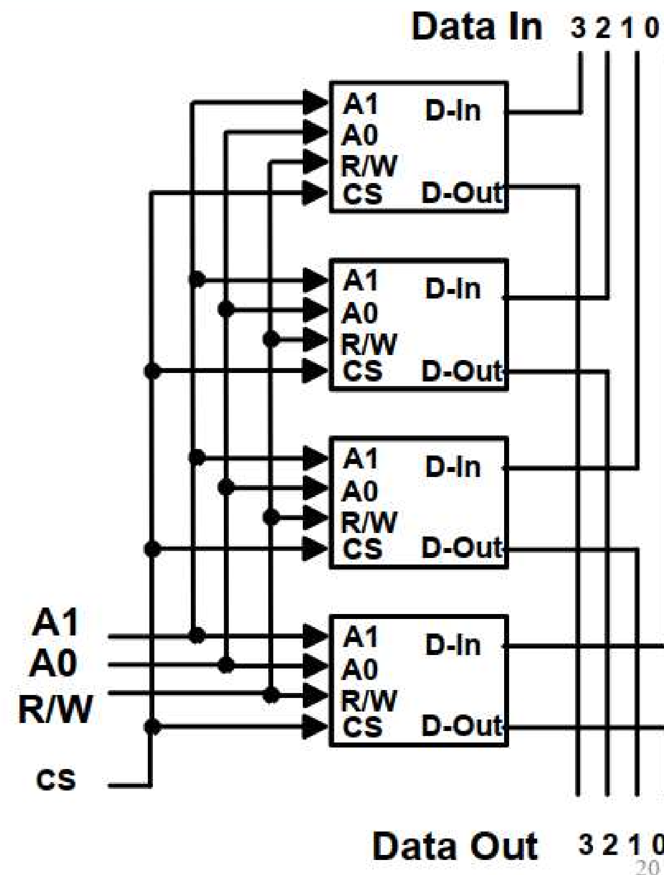
Making Larger Memories

- We can make larger memories from smaller ones by using the decoded higher order address bits to control CS (chip select) lines, tying all address, data, and R/W lines in parallel.
- A 16-Word by1-Bit memory constructed using 4-Word by 1-Bit memory.



Making Wider Memories

- Tie the address and control lines in parallel and keep the data lines separate.
- Example: make a 4-word by 4-bit memory from 4, 4-word by 1-bit memories
- Note: Both 16x1 and 4x4 memories take 4-chips and hold 16 bits of data.

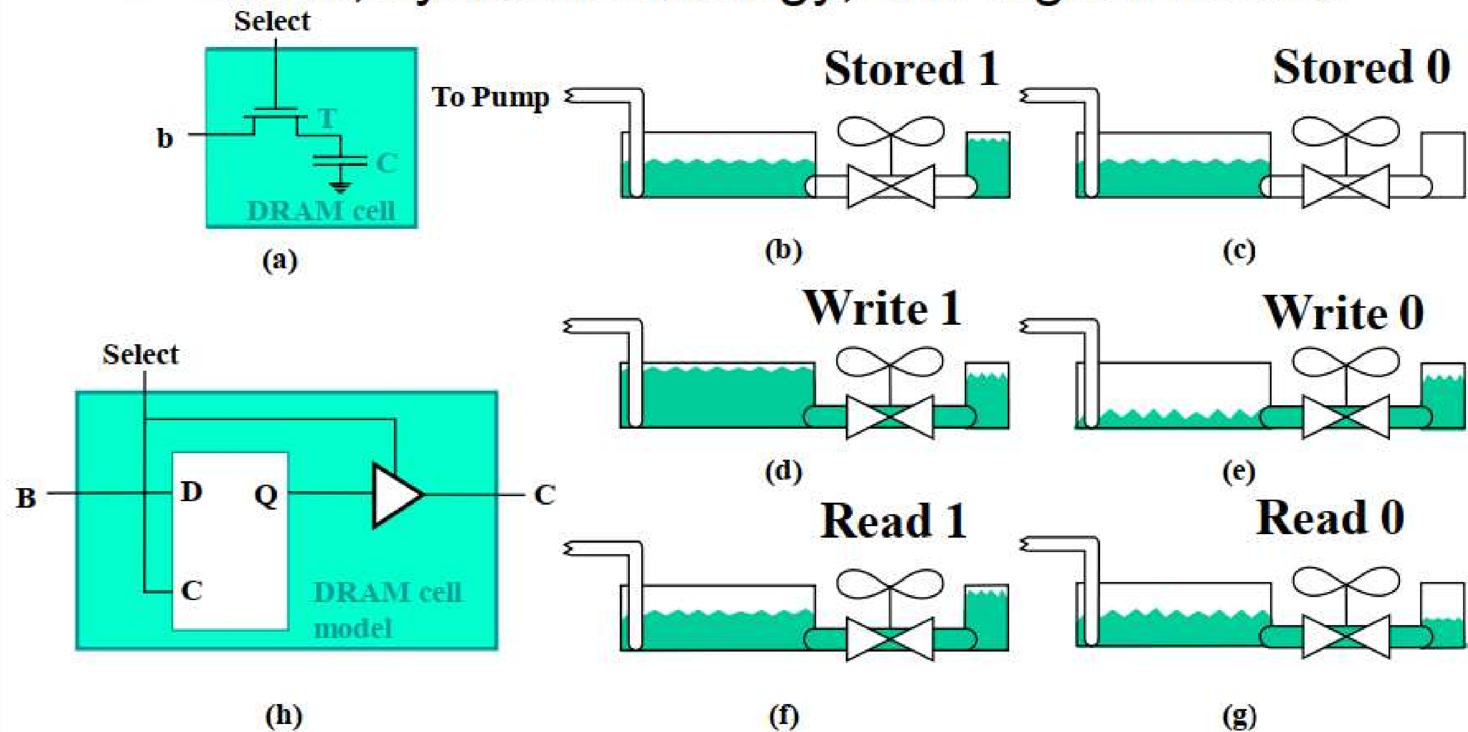


DRAM

- Basic Principle: Storage of information on capacitors.
- Charge and discharge of capacitor to change stored value
- Use of transistor as “switch” to:
  - Store charges
  - Charge or discharge

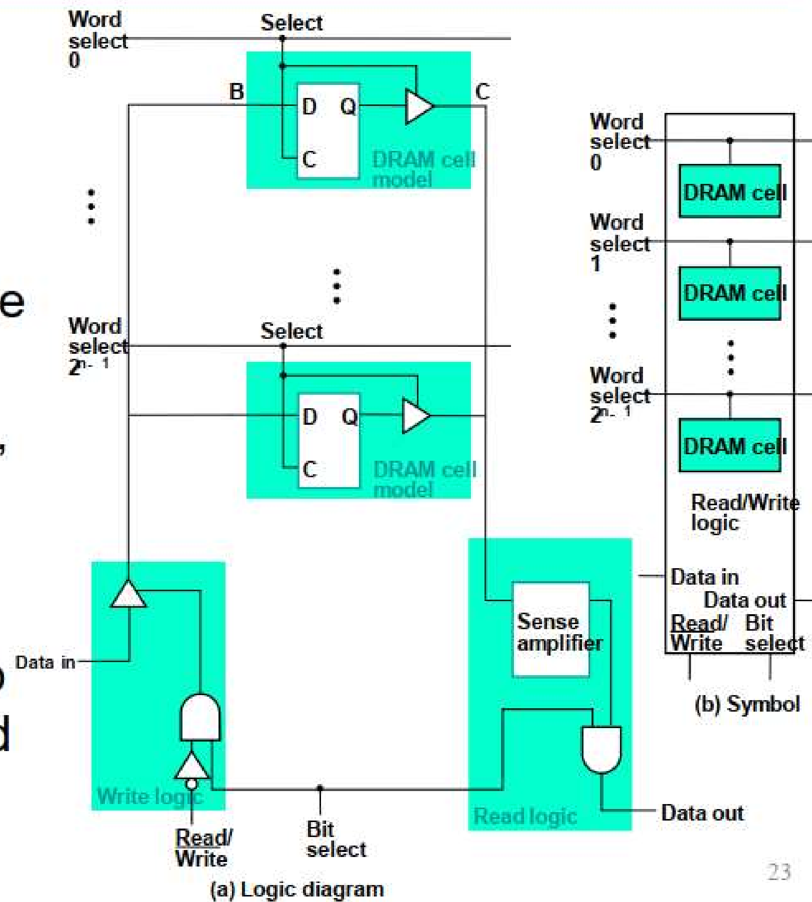
Dynamic RAM (Contd.)

- Circuit, hydraulic analogy, and logical model.



Dynamic RAM - Bit Slice

- C driven by 3-state drivers
- Sense amplifier is used to change the small voltage change on C into H or L
- In the electronics, B, C, and the sense amplifier output are connected to make destructive read into non-destructive read



University Of Diyala
College Of Engineering
Department of Computer Engineering

Digital System Design II
Microprogramming

Dr. Yasir Al-Zubaidi

Third stage

2021

Microprogramming Overview

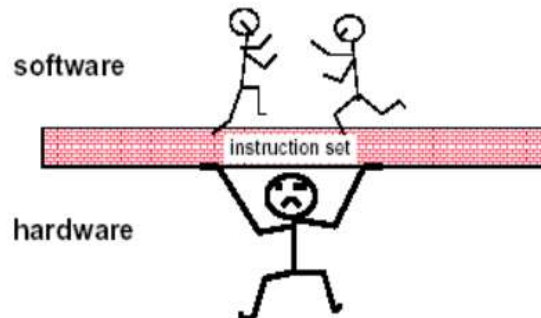
- Part 1 – Datapaths
  - Introduction
  - Datapath Example
    - Arithmetic Logic Unit (ALU)
    - Shifter
  - Datapath Representation and Control Word

- Part 2 – A Simple Computer
  - Instruction Set Architecture (ISA)
  - Single-Cycle Hardwired Control

- Part 3 – Multiple Cycle Hardwired Control
  - Single Cycle Computer Issues
  - Sequential Control Design

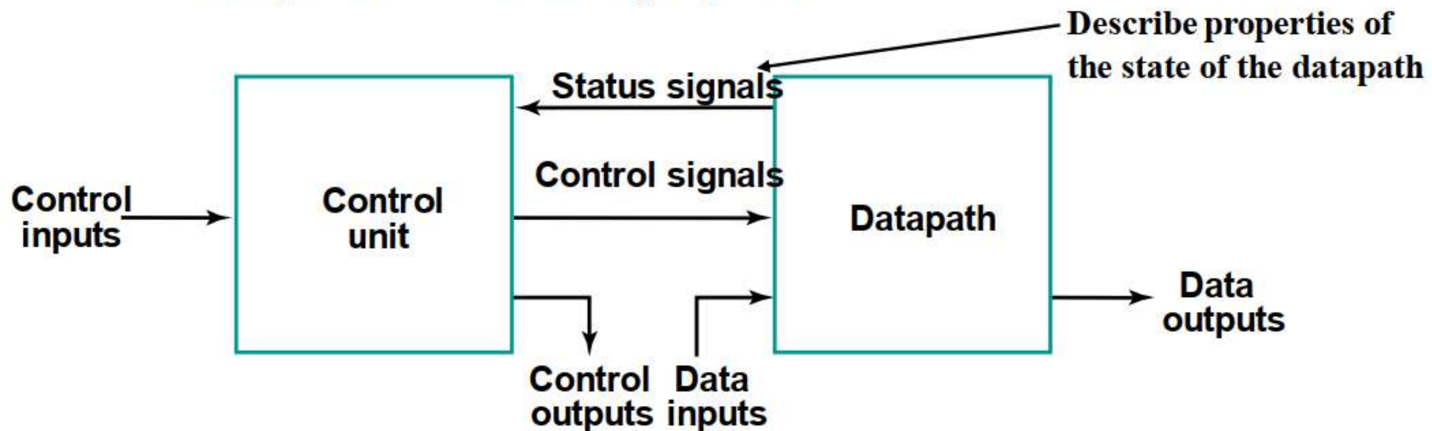
Introduction

- Computer Specification
  - *Instruction Set Architecture (ISA)* - the specification of a computer's appearance to a programmer at its lowest level
  - *Computer Architecture* - a high-level description of the hardware implementing the computer derived from the ISA
  - The architecture usually includes additional specifications such as speed, cost, and reliability.



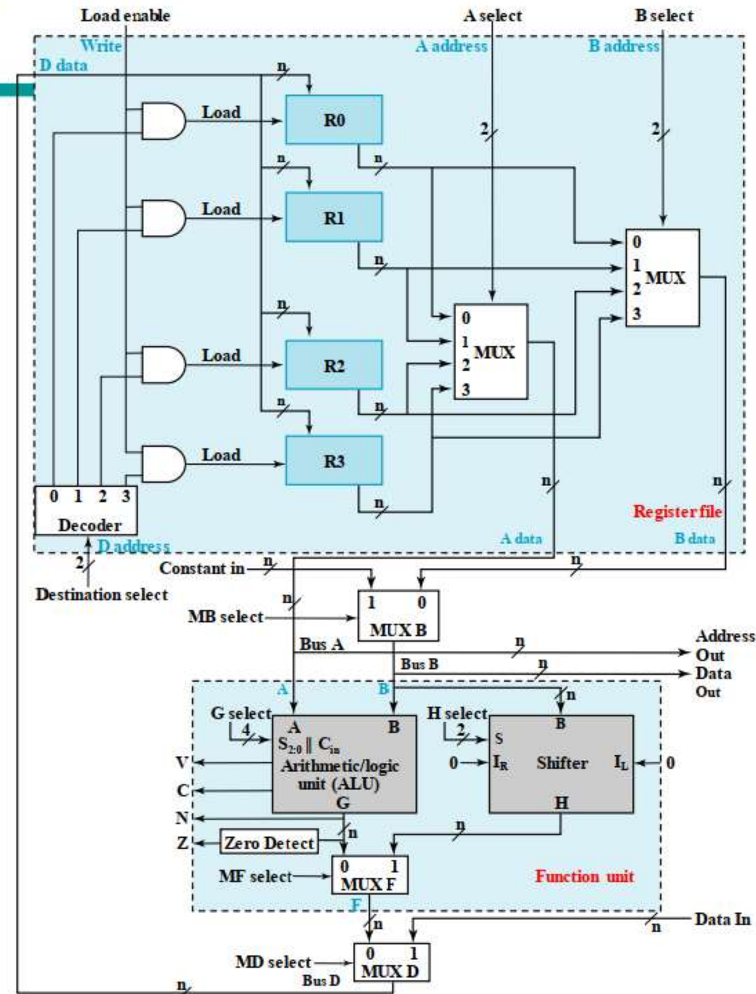
Introduction (Contd.)

- Simple computer architecture decomposed into:
  - Datapath: performing operations
    - A set of registers
    - Microoperations performed on the data stored in the registers
    - A control interface
  - Control unit: controlling datapath operations
    - Programmable & Non-programmable



Datapath Example

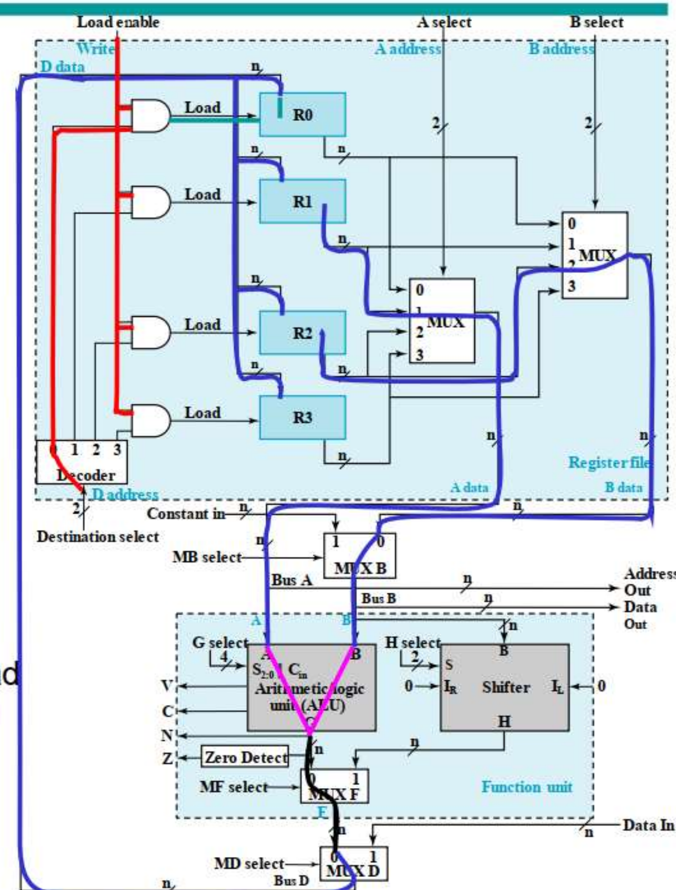
- Register file:
  - Four parallel-load regs
  - Two mux-based register selectors
  - Register destination decoder
- Microoperation Implementation
  - Mux B for external constant input
  - Buses A and B with external address and data outputs
  - Function Unit:
    - ALU and Shifter with Mux F for output select
    - Mux D for external data input
    - Logic for generating status bits V, C, N, Z



Datapath Example: Performing a Microoperation

Microoperation: R0 ← R1 + R2

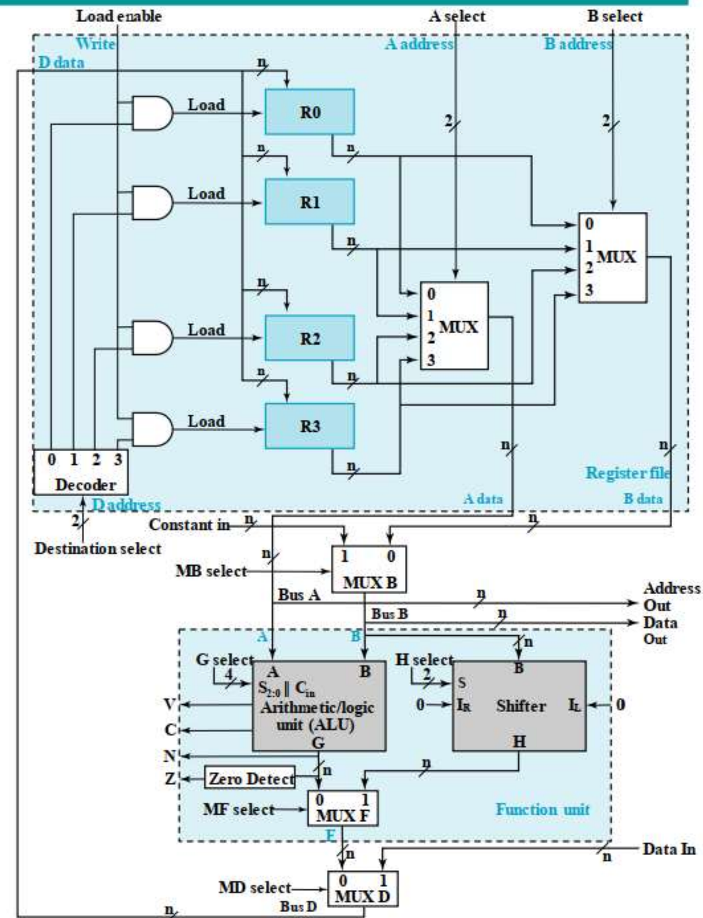
- Apply 01 to A select to place contents of R1 onto Bus A
- Apply 10 to B select to place contents of R2 onto B data and apply 0 to MB select to place B data on Bus B
- Apply 0010 to G select to perform addition G = Bus A + Bus B
- Apply 0 to MF select and 0 to MD select to place the value of G onto BUS D
- Apply 00 to Destination select to enable the Load input to R0
- Apply 1 to Load Enable to force the Load input to R0 to 1 so that R0 is loaded on the clock pulse (not shown)
- The overall microoperation requires 1 clock cycle



Datapath Example: Key Control Actions for Microoperation Alternatives

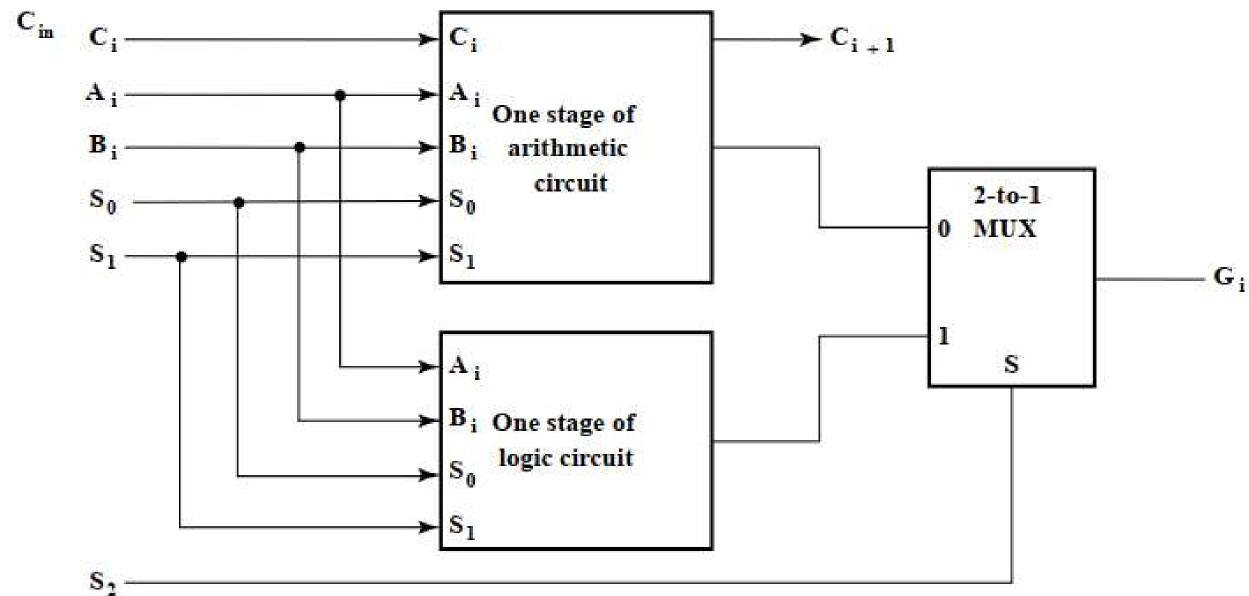
Various microoperations:

- Perform a shift microoperation: apply 1 to MF select
- Use a constant in a micro-operation using Bus B: apply 1 to MB select
- Provide an address and data for a memory or output write microoperation – apply 0 to Load enable to prevent register loading
- Provide an address and obtain data for a memory or output read microoperation – apply 1 to MD select
- For some of the above, other control signals become don't cares



Arithmetic Logic Unit (ALU)

- Decompose the ALU into:
  - An arithmetic circuit & A logic circuit
  - A selector to pick between the two circuits

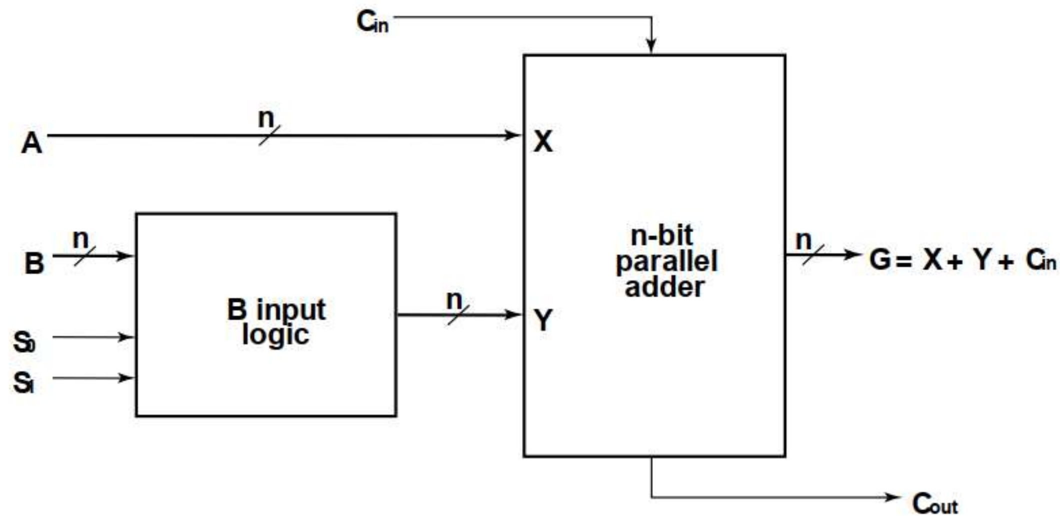


Arithmetic Circuit Design

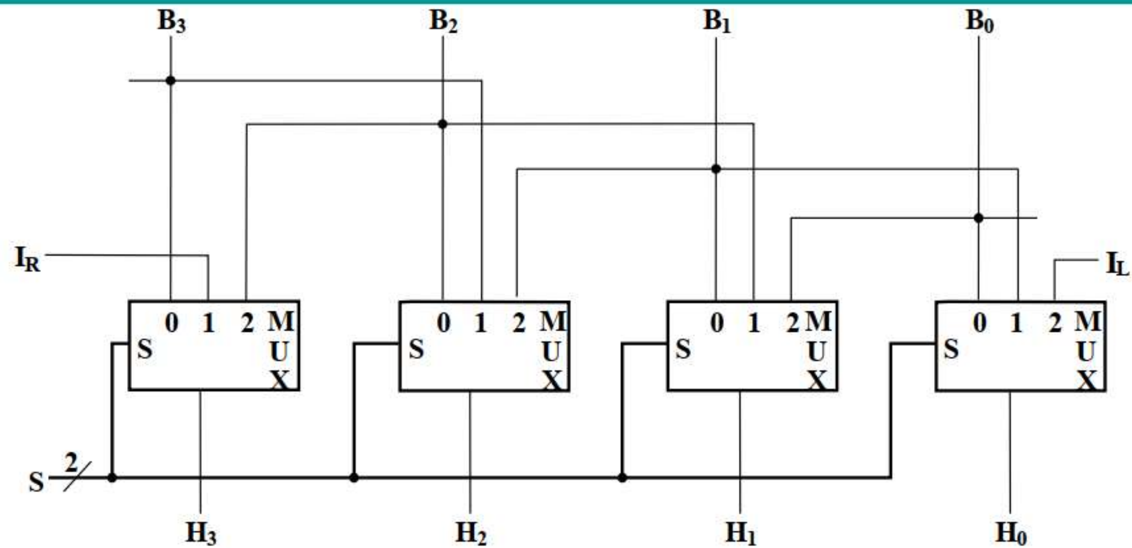
- Arithmetic circuit design
  - Decompose the arithmetic circuit into:
    - An n-bit parallel adder
    - A block of logic that selects four choices for the B input to the adder

- There are only four functions of B to select as Y in $G = A + Y + C_{in}$:

	$C_{in} = 0$	$C_{in} = 1$
0	$G = A$	$G = A + 1$
B	$G = A + B$	$G = A + B + 1$
$\overline{B}$	$G = A + \overline{B}$	$G = A + \overline{B} + 1$
1	$G = A - 1$	$G = A$



4-Bit Basic Left/Right Shifter



- Serial Inputs:
  - I_R for right shift
  - I_L for left shift

- Shift Functions:

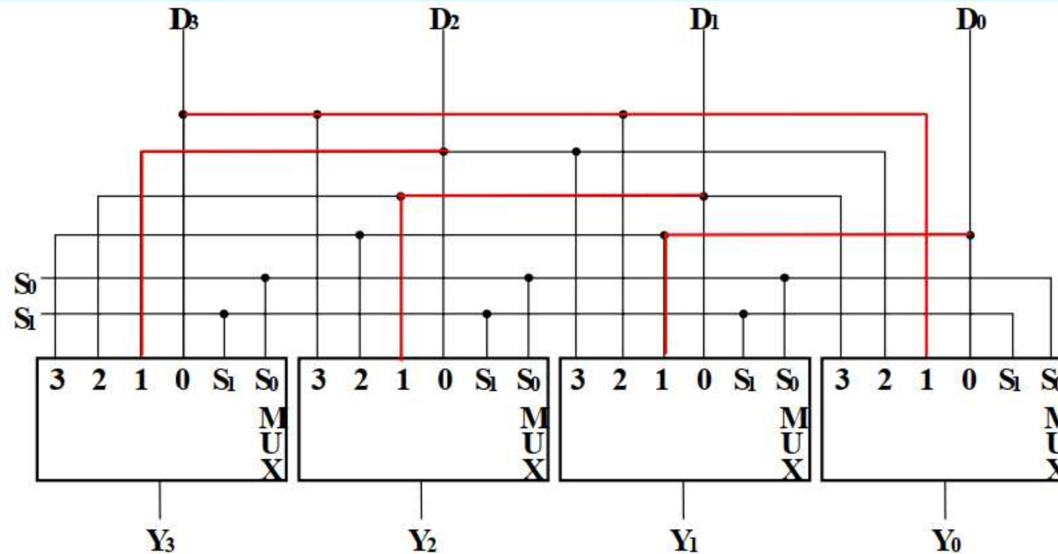
  (S_1, S_0) = 00 Pass B unchanged

  01 Right shift

  10 Left shift

  11 Unused

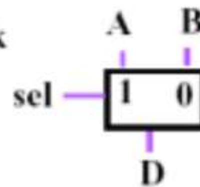
Barrel Shifter



- A rotate is a shift in which the bits shifted out are inserted into the positions vacated
- The circuit rotates its contents left from 0 to 3 positions depending on S:
  S = 00 position unchanged
  S = 01 rotate left by 1 positions
  S = 10 rotate left by 2 positions
  S = 11 rotate left by 3 positions

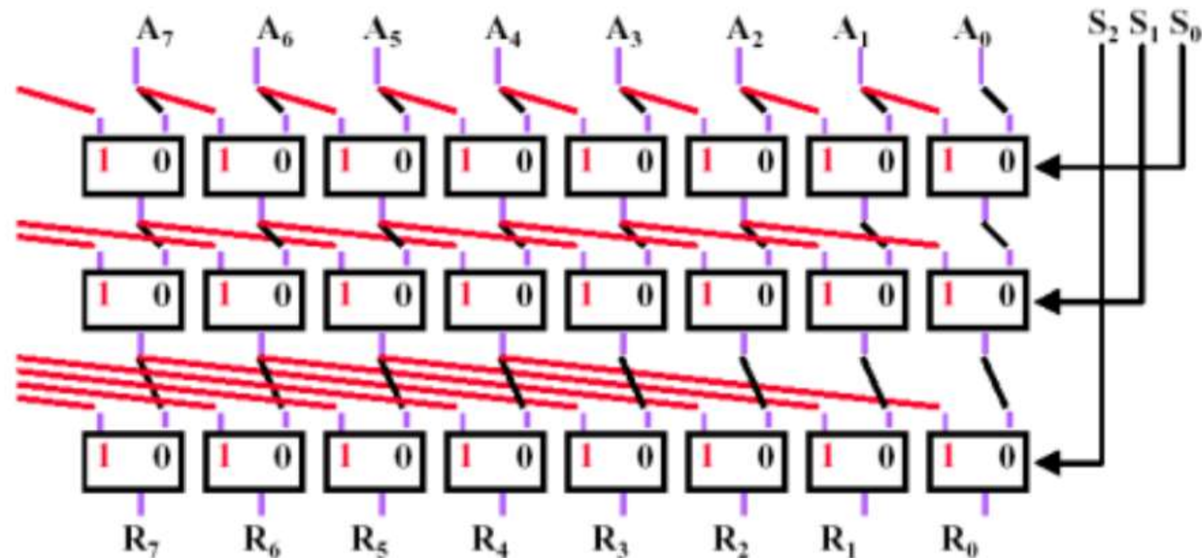
Combinational Shifter from MUXes

Basic Building Block



8-bit right shifter

- - Example 8-bit:
  - Layer 1 shifts by 0, 4
  - Layer 2 shifts by 0, 2
  - Layer 3 shifts by 0, 1



- Large barrel shifters can be constructed using:
  - Layers of multiplexers
  - 2 - dimensional array circuits designed at the electronic level