University Of Diyala
College Of Engineering
Computer Engineering Department

Digital System Design II

Dr. Yasir Al-Zubaidi

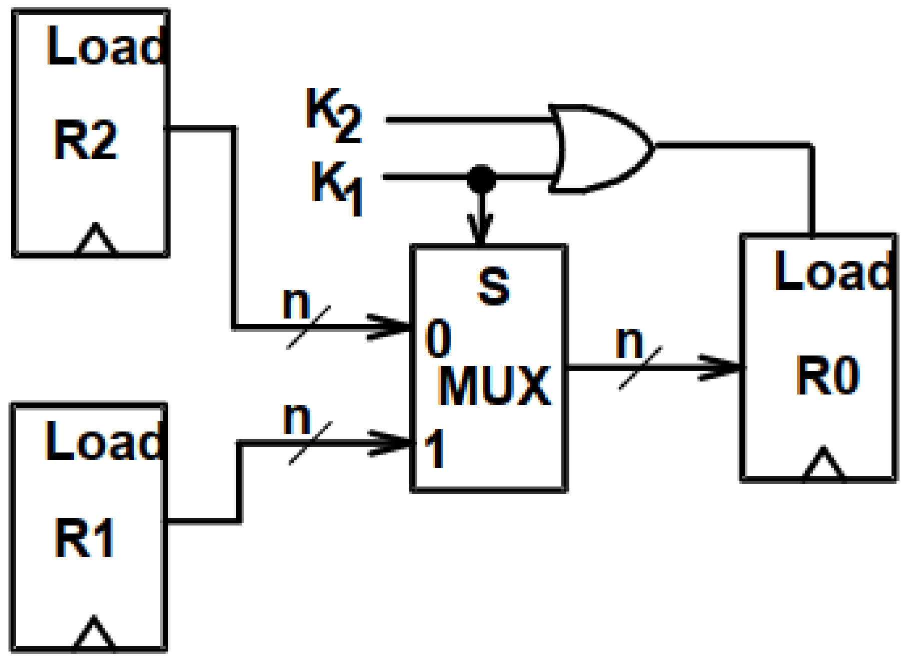
Third stage

2019

Multiplexer-Based Single Register Transfers

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

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

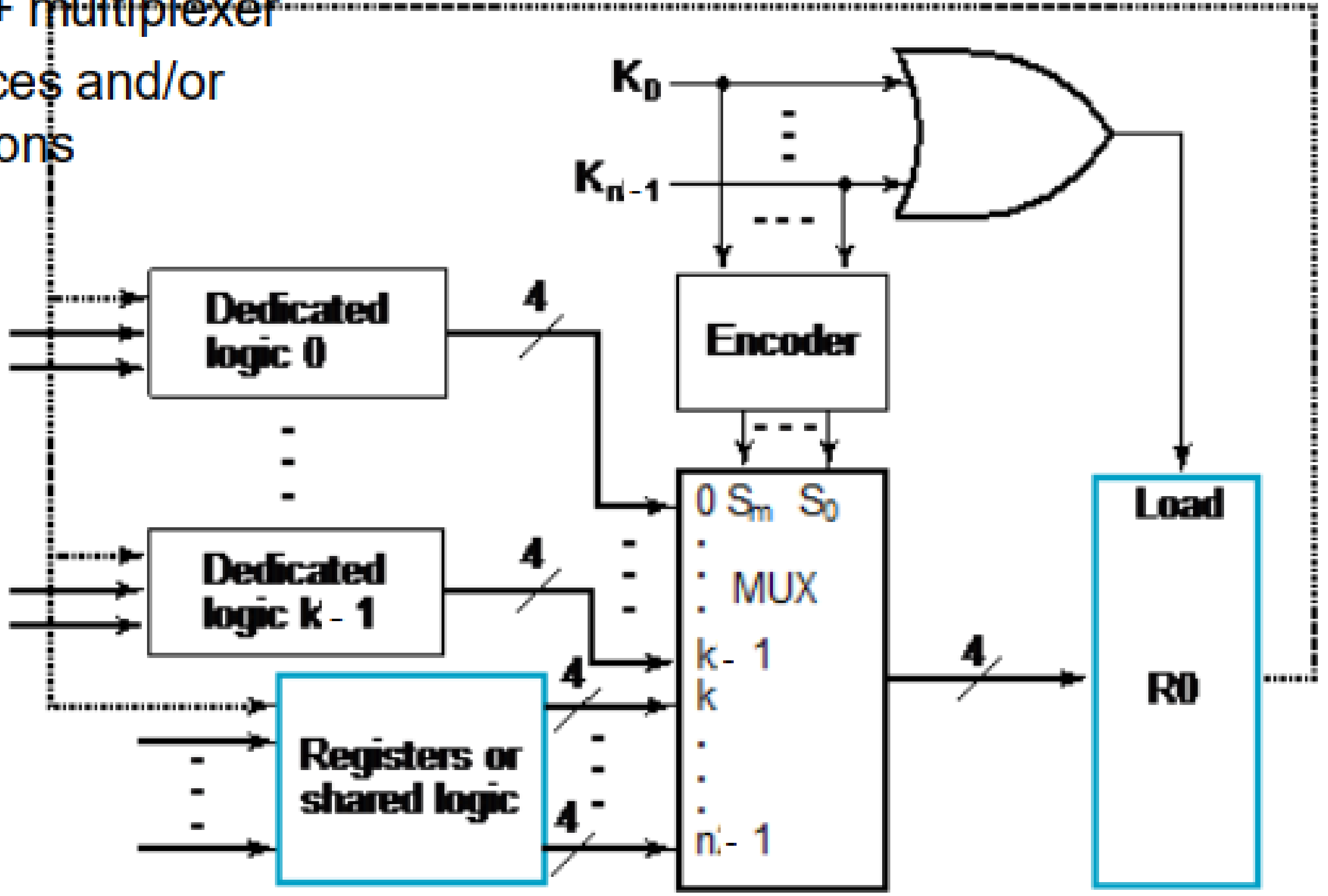


Register Design

- 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
- <u>Register cell design</u> is the first step of the above process

Approach I: Multiplexer-based

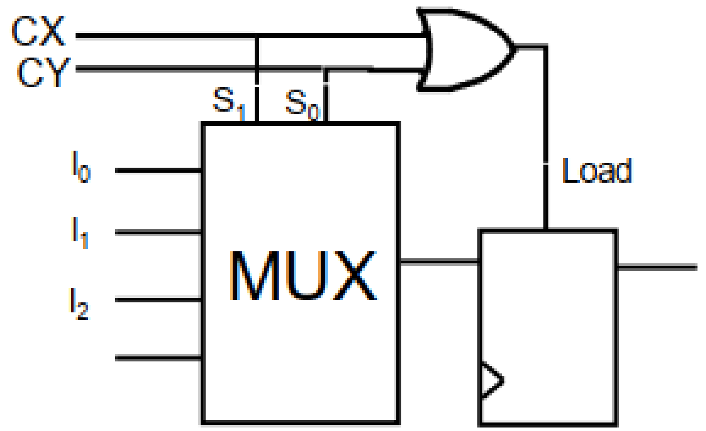
- An n-input multiplexer with a variety of sources and functions
- Load enable by OR of control signals K_0, K_1, … K_{n-1} (for 00…0, no load)
- Use encoder + multiplexer to select sources and/or transfer functions



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:

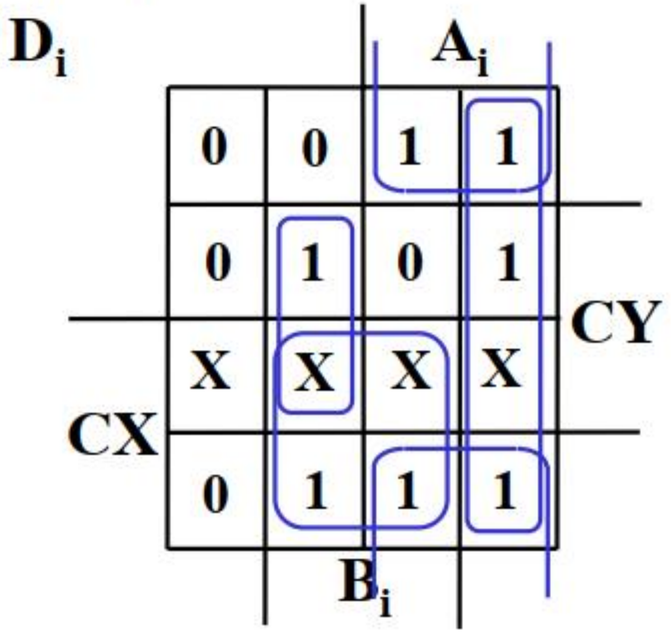
	Hold	Ai **v** Bi		Ai$\oplus$Bi	
A_i	CX = 0 CY = 0	CX = 1 CY = 0 B_i = 0	CX = 1 CY = 0 B_i = 1	CX = 0 CY = 1 B_i = 0	CX = 0 CY = 1 B_i = 1
0	0	0	1	0	1
1	1	1	1	1	0

- 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



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:

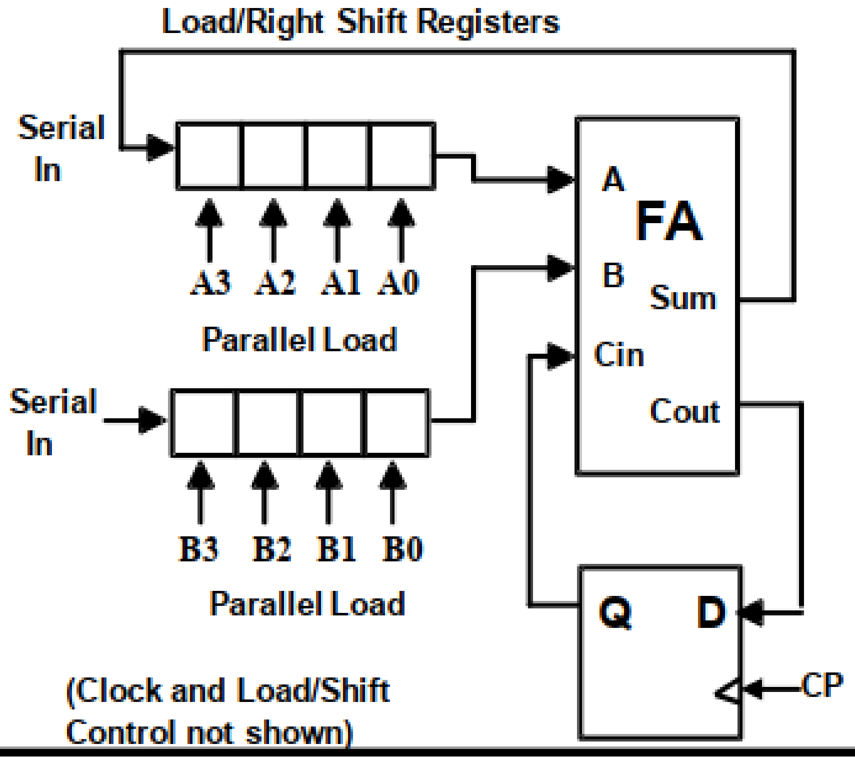
  Per cell: 19

  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

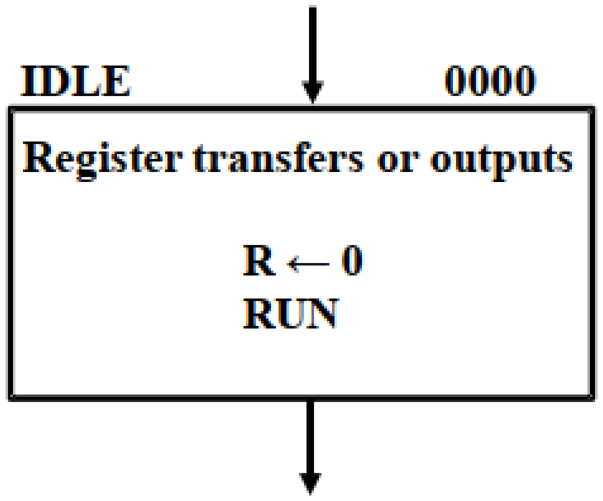
Algorithmic State Machines

- 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

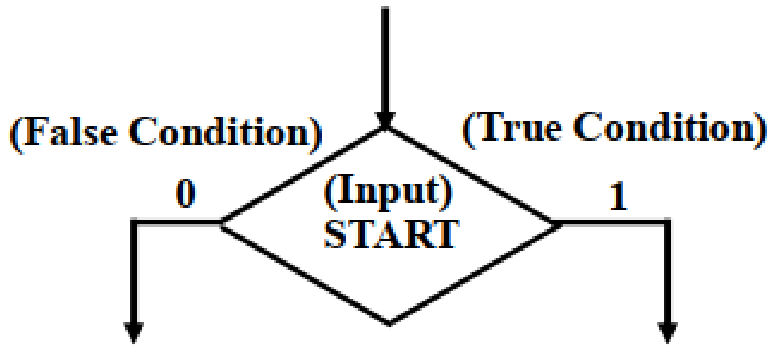
- A rectangle with:
  - The symbolic name for the state
  - An optional state code
  - Containing register transfer operations, and outputs activated within or while leaving the state

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

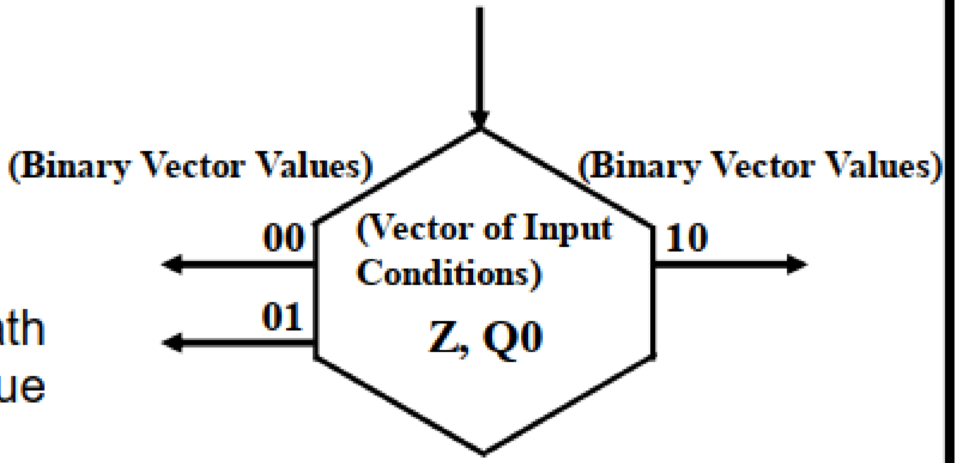


Decision Box

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

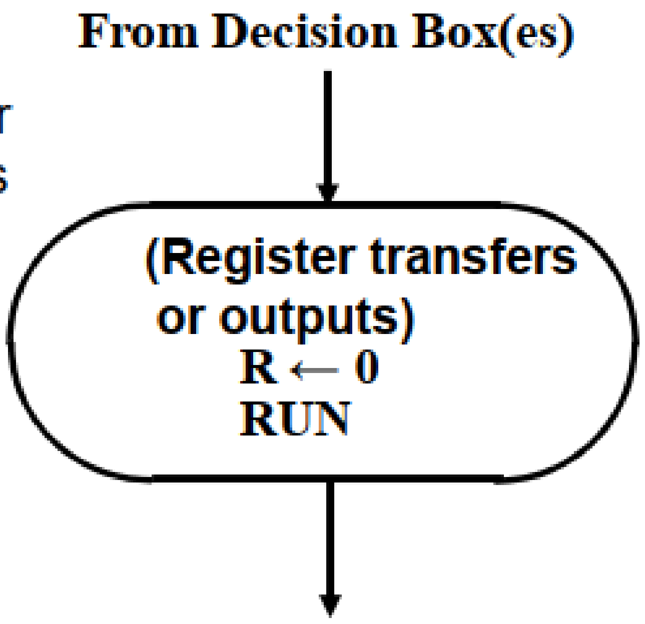


- Vector: A hexagon with:
  - One input path (entry point).
  - A vector of input conditions tested.
  - Up to 2^n output paths. The path taken has a binary vector value that matches the vector input condition



Conditional Output Box

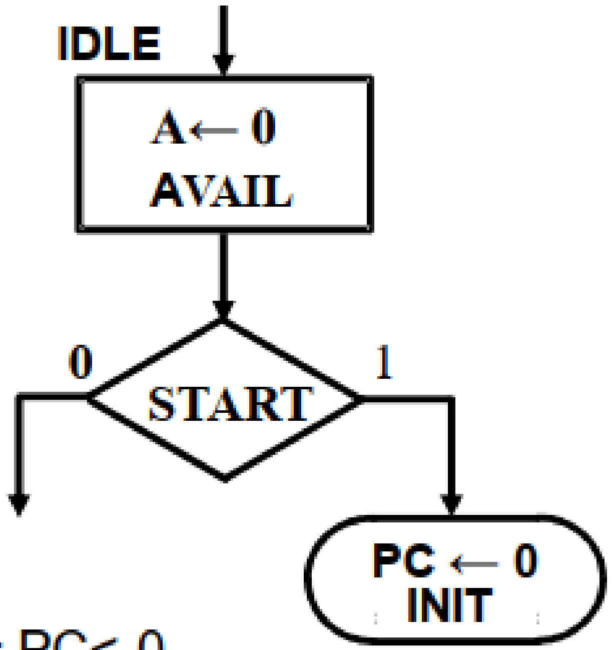
- An oval with:
  - One input path from a decision box(es)
  - One output path
  - Register transfers or outputs that occur only if the conditional path to the box is taken.
- Transfers and outputs
  - in a state box are Moore type - dependent only on state
  - in a conditional output box are Mealy type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



Connecting Boxes Together

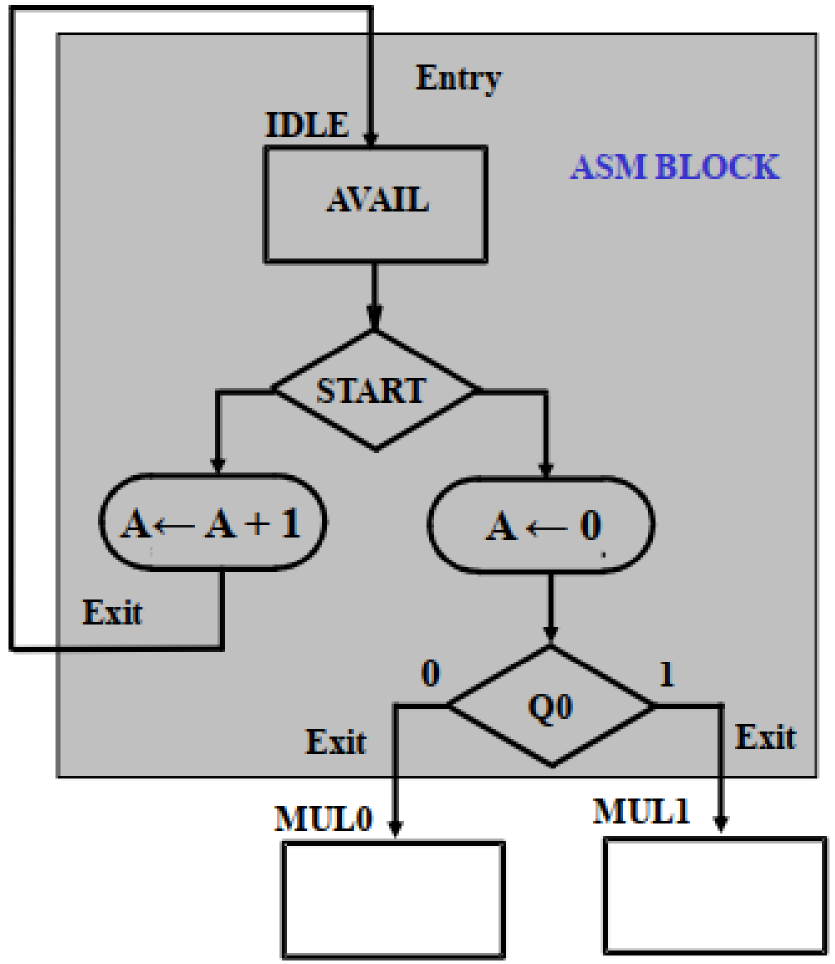
- By connecting boxes together, we see the power of expression.

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



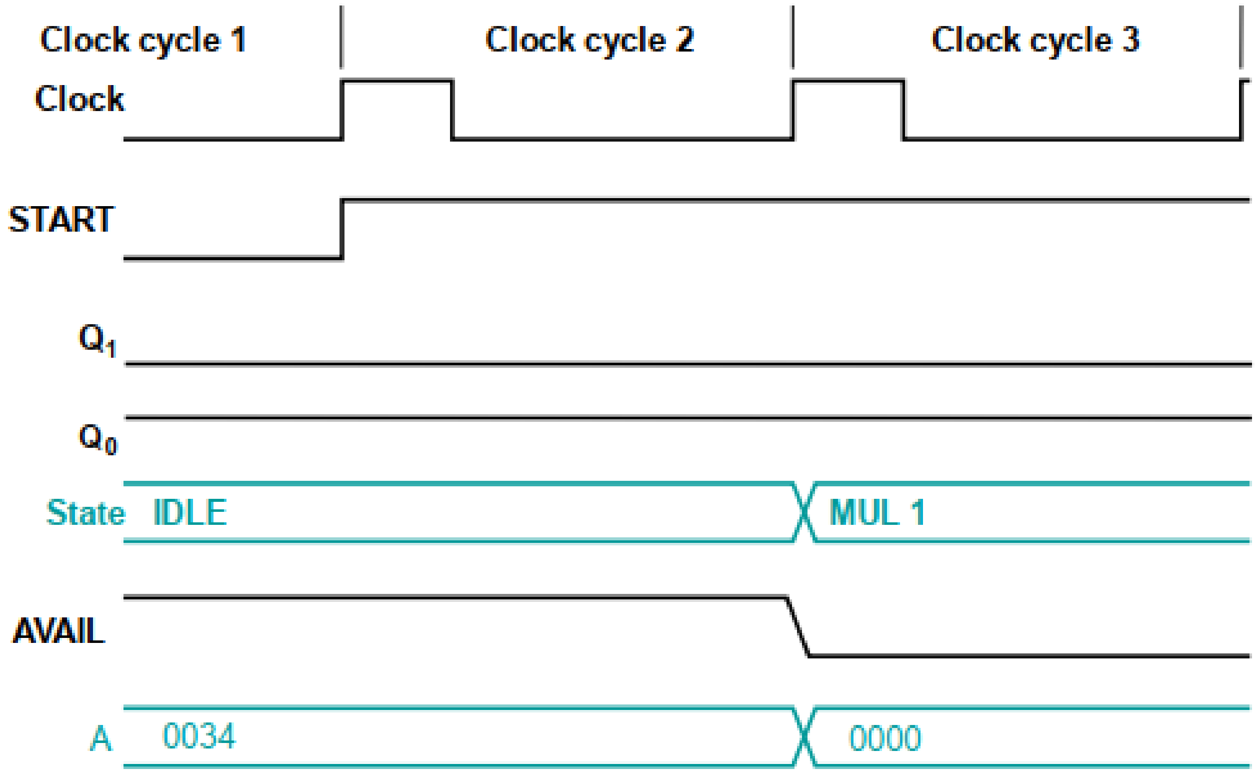
ASM Blocks

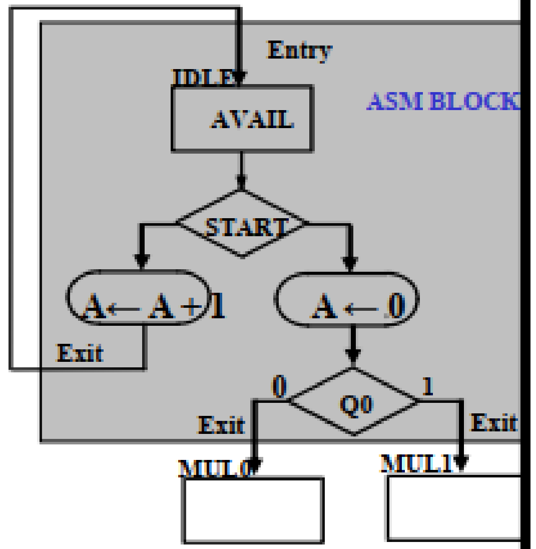
- One state box along with all decision and conditional output boxes connected to it, called an ASM Block. i.e., the ASM Block includes all items on the path from the current state to the same or other states.



ASM Timing

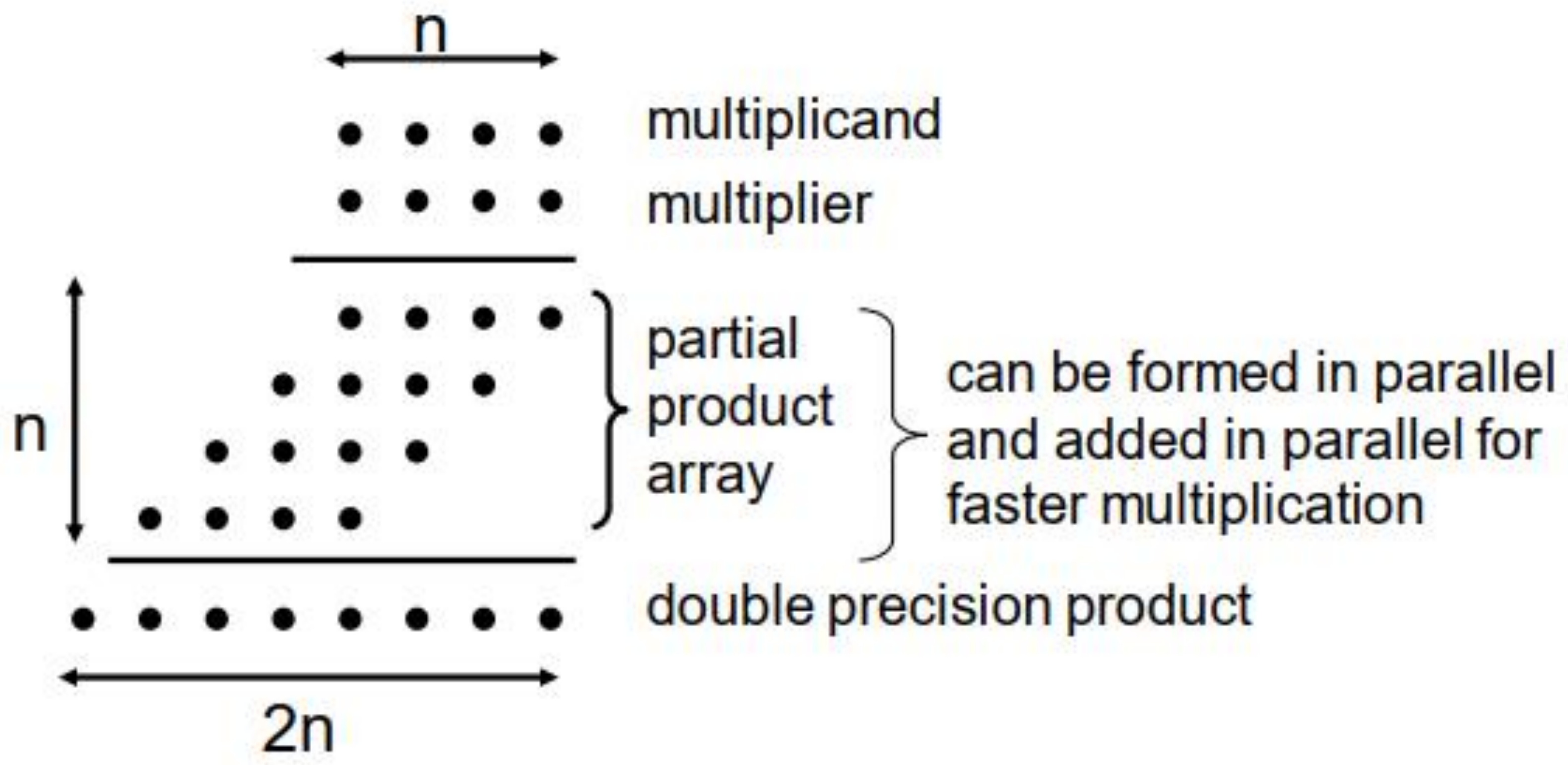
- Outputs appear while in the state
- Register transfers and conditional outputs occur at the clock while exiting the state - New value occur in the next state!





Multiply Overview

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



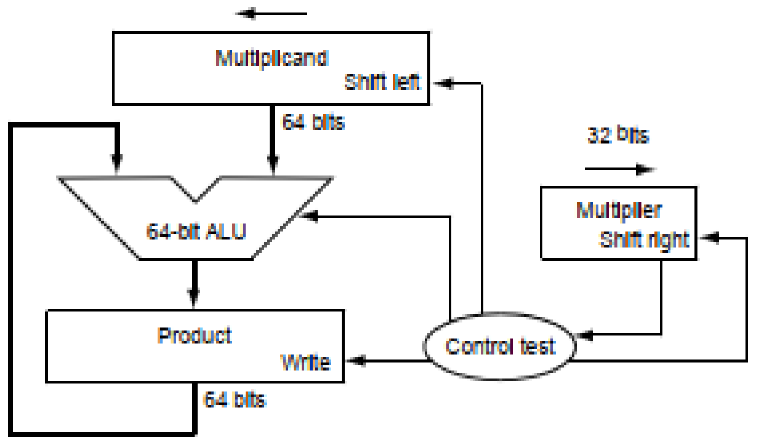
n
multiplicand
multiplier
n
partial product array
can be formed in parallel and added in parallel for faster multiplication
double precision product
2n

Multiplier Example

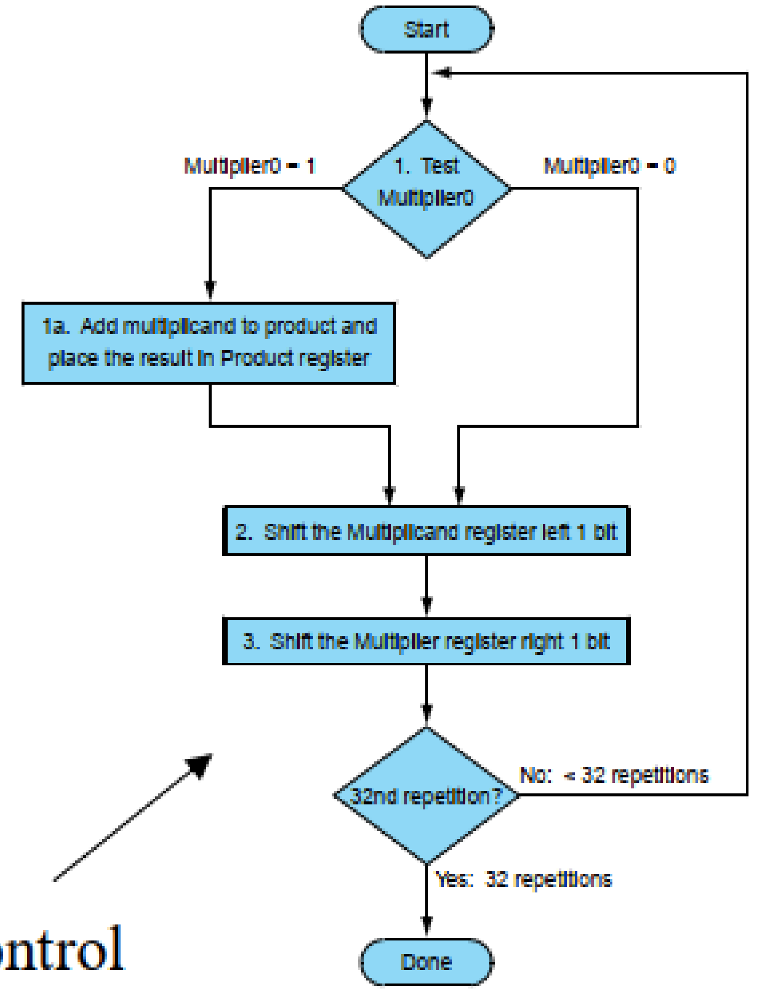
- Example: (101 **x** 011) Base 2
- Partial products are:
  101 x 0, 101 x 1, and 101 x 1

```
            1   0   1   multiplicanc
        x   0   1   1   multiplier
        -------------
            1   0   1
        1   0   1
    0   0   0
-----------------------
0   0   1   1   1   1
```

Multiplication: Implementation (version 1)



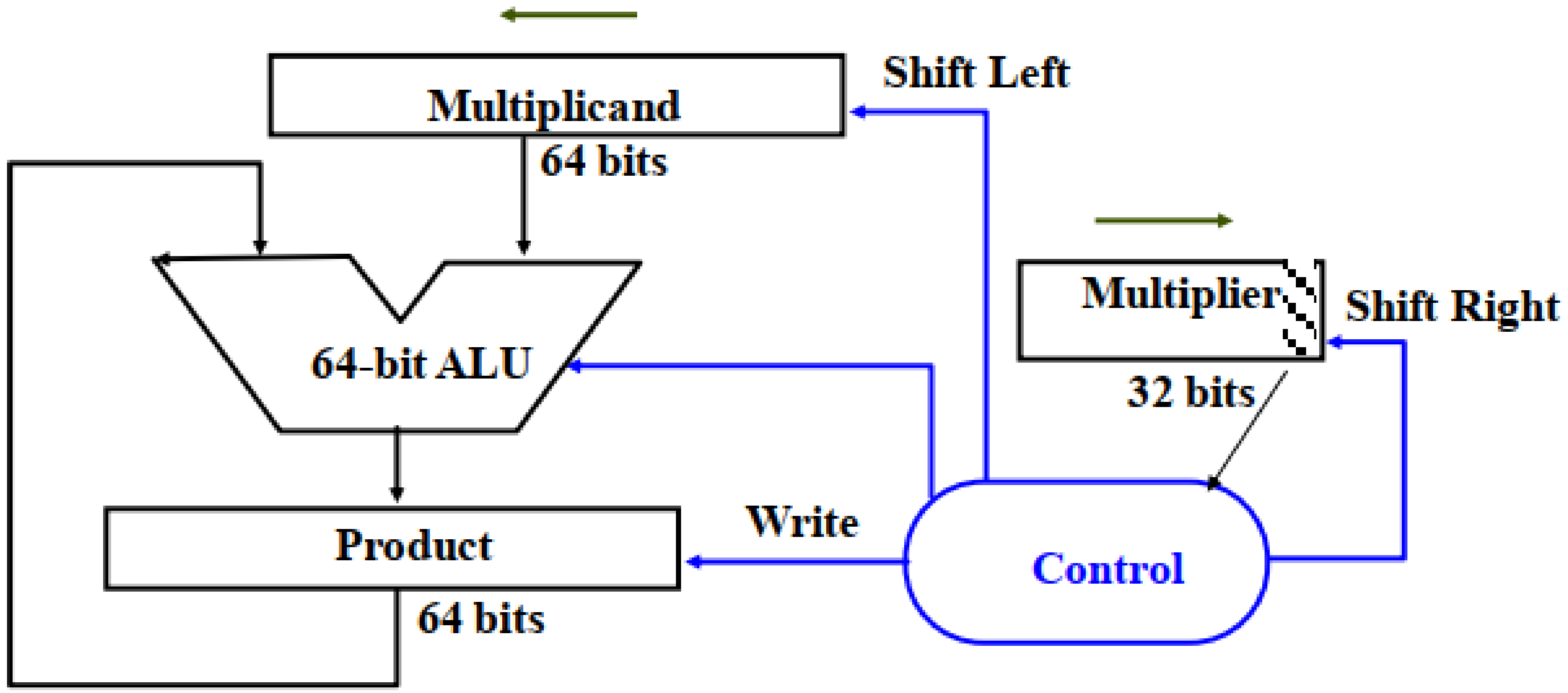
Datapath


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Control

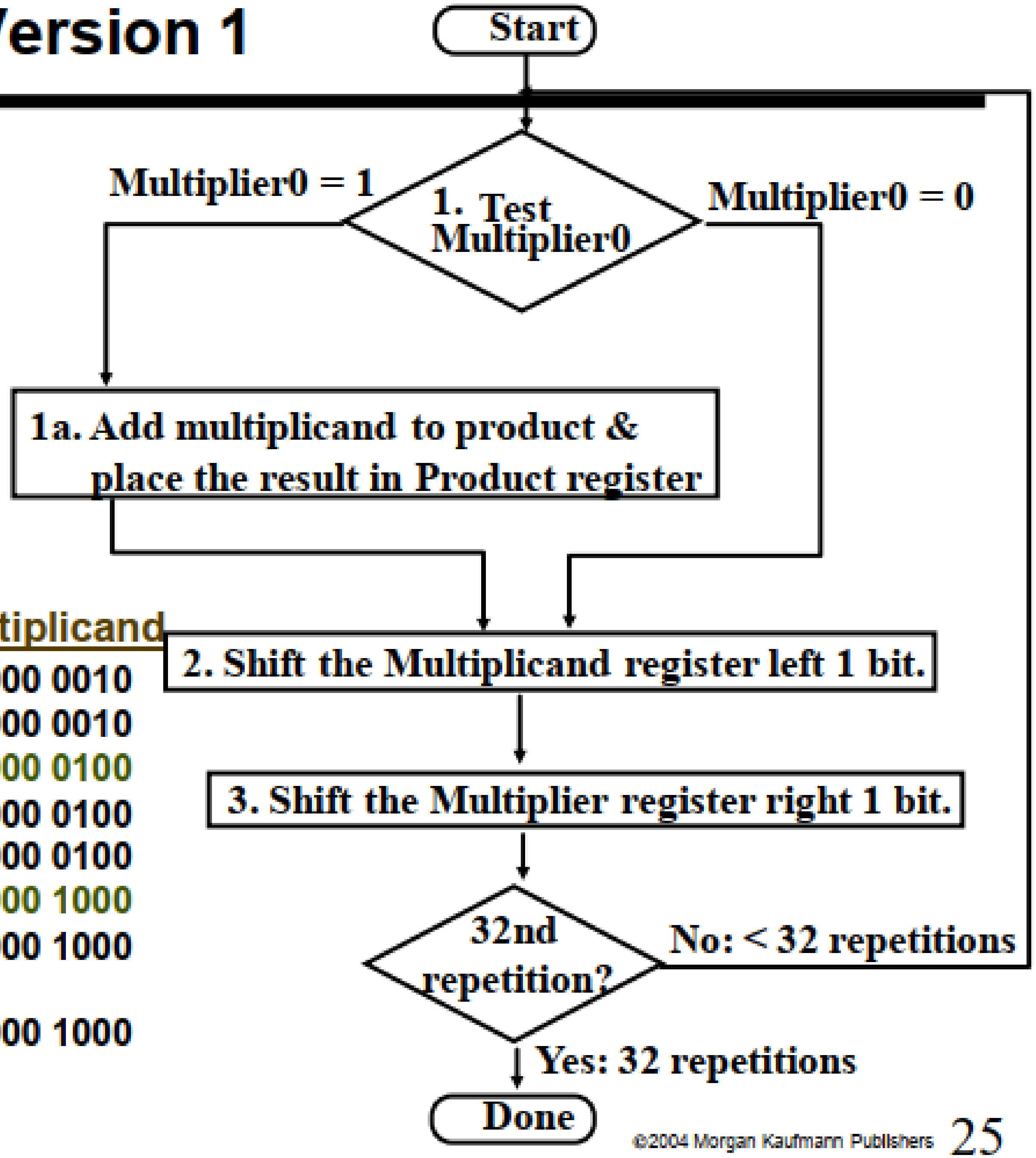
Unisigned shift-add multiplier (version 1)

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



Multiplier = datapath + control

ultiply Algorithm Version 1


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product & place the result in Product register
2. Shift the Multiplicand register left 1 bit.
3. Shift the Multiplier register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

	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

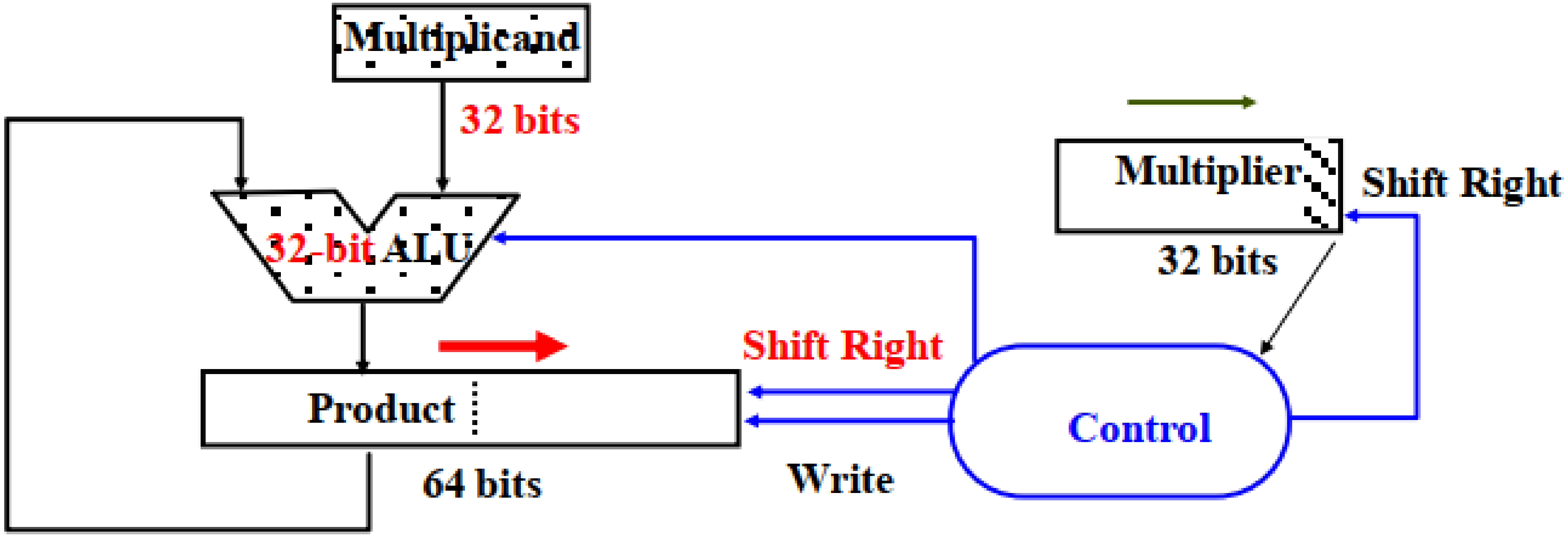
- 1 clock per cycle => ≈ 100 clocks per multiply because of 32 repetitions, 3 steps in one repetition
  - Ratio of add/sub to multiply is from 5:1 to 100:1
  - Slow
- 0's inserted in the rightmost bit of multiplicand as shifting left
  => least significant bits of product never changed once formed
-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

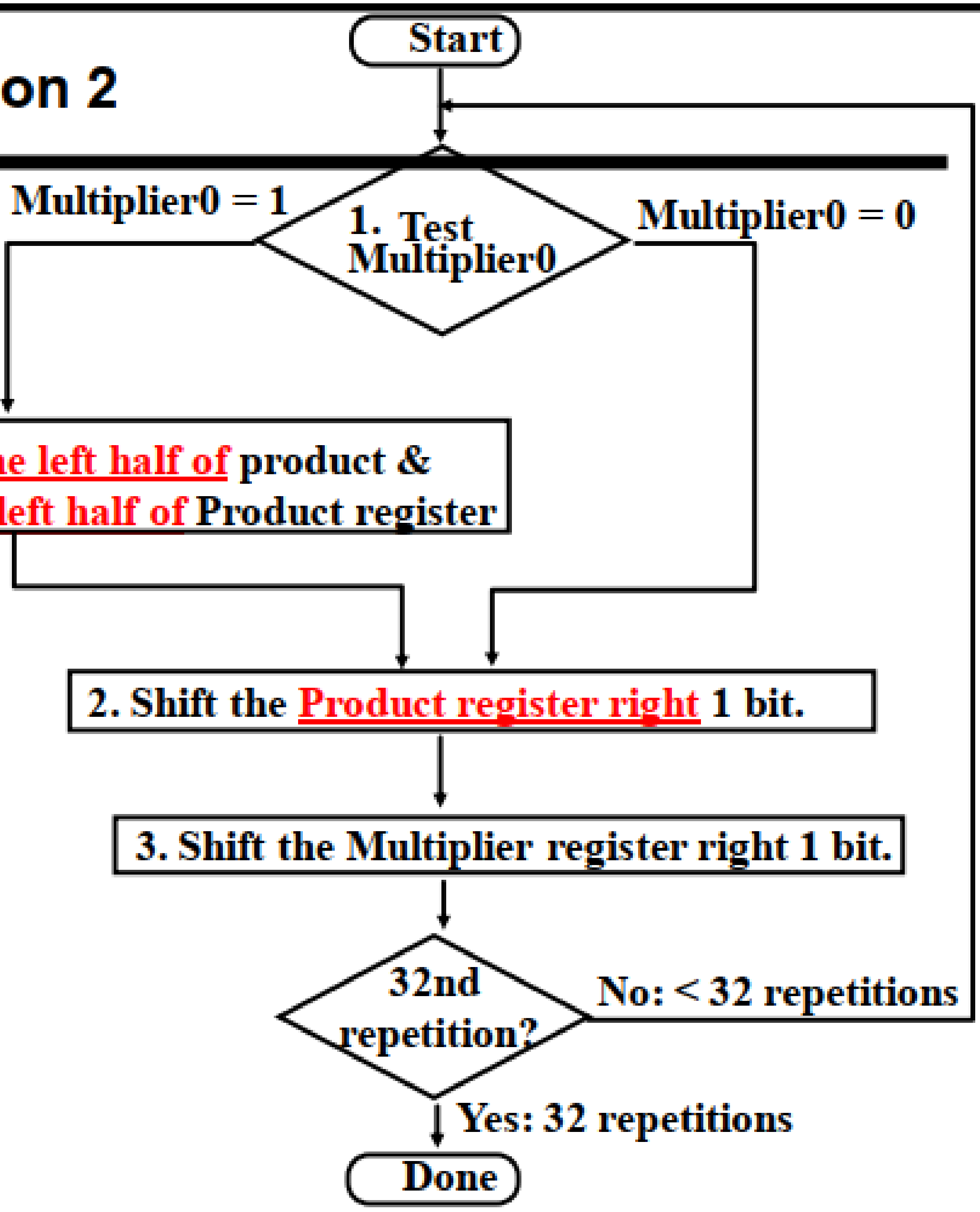
  => 64-bit adder is wasted
- Instead of shifting multiplicand to left, let's shift *product* to right

MULTIPLY HARDWARE Version 2

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

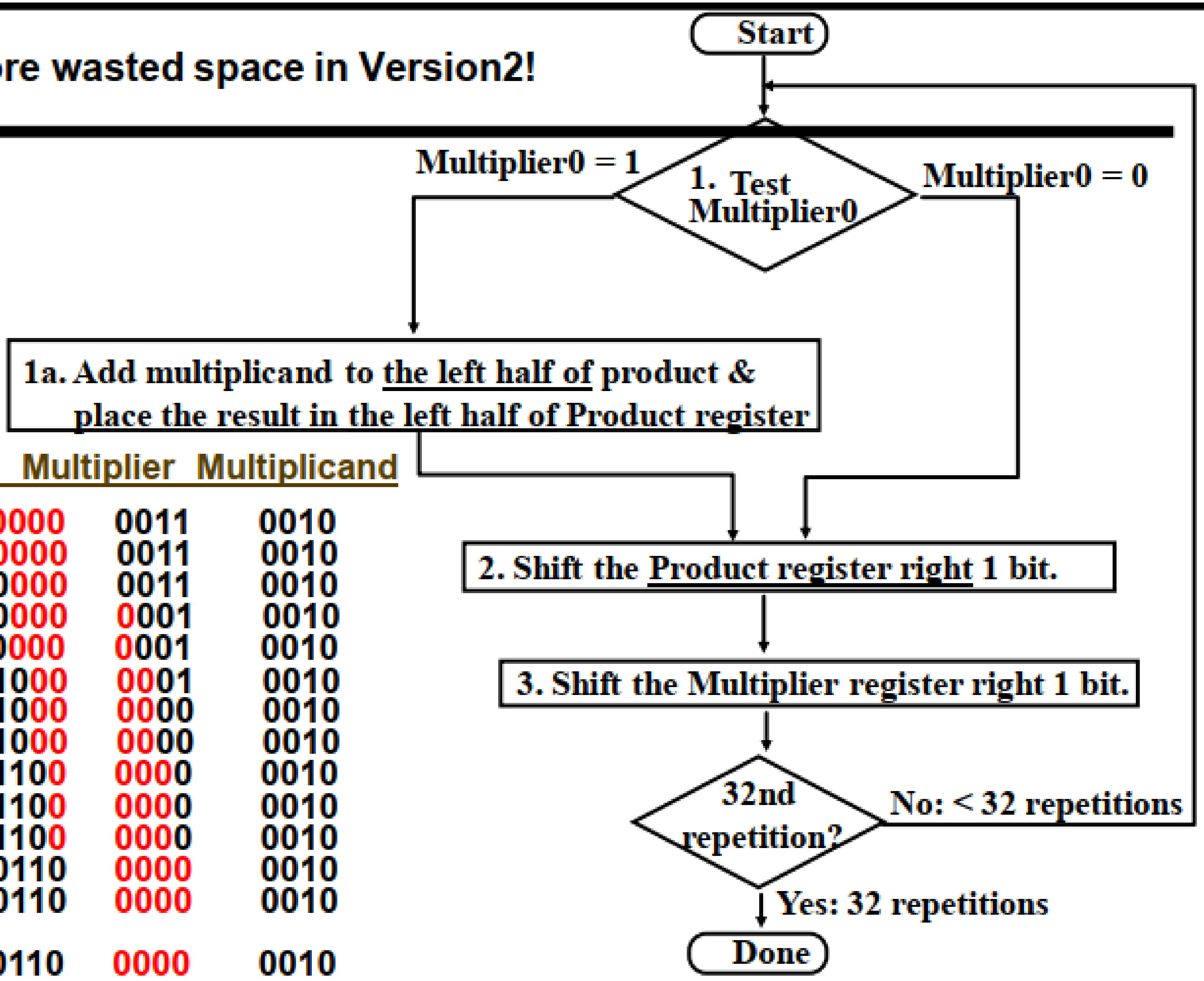


Multiply Algorithm Version 2


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to the left half of product & place the result in the left half of Product register
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

	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

Still more wasted space in Version2!


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to the left half of product & place the result in the left half of Product register
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

	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

Observations on Multiply Version 2

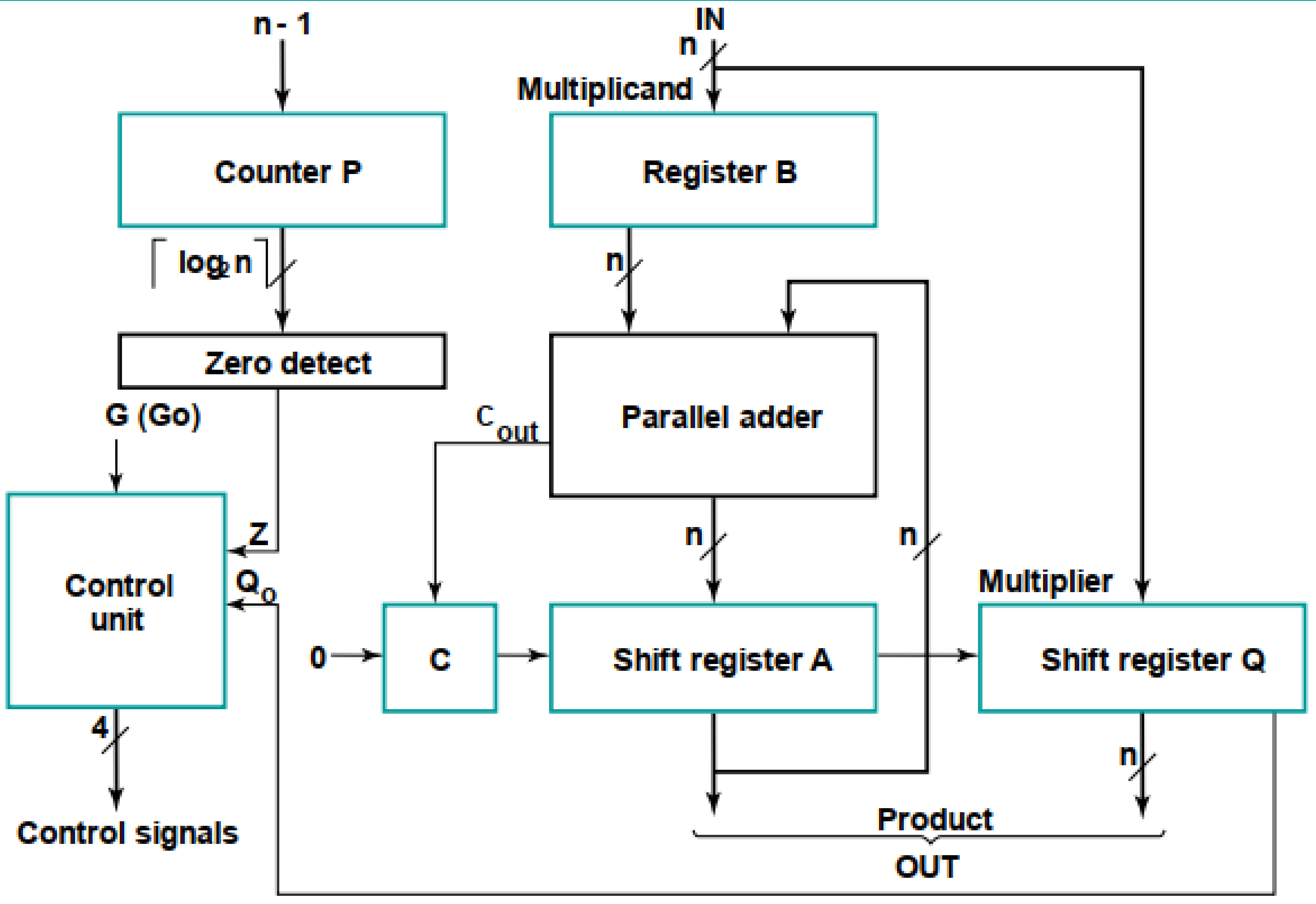
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

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				**Multipler$_0$ = 1 => Add B**
0	1	0	1				**Addition**
0	0	1	0	1			**Shift Right (Zero-fill C)**
+	1	0	1				**Multipler$_1$ = 1 => Add B**
0	1	1	1	1			**Addition**
0	0	1	1	1	1		**Shift Right**
0	0	0	1	1	1	1	**Multipler$_2$ = 0 => 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.