Logic and Computer Design Fundamentals

Chapter 6 – Registers and Register Transfers

Part 3 – Control of Register Transfers

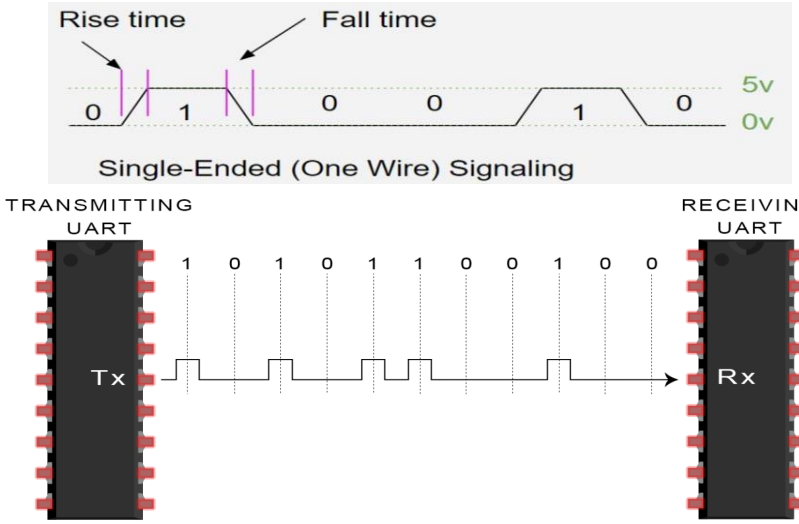
Ming Cai
cm@zju.edu.cn
College of Computer Science and Technology,
Zhejiang University

Overview

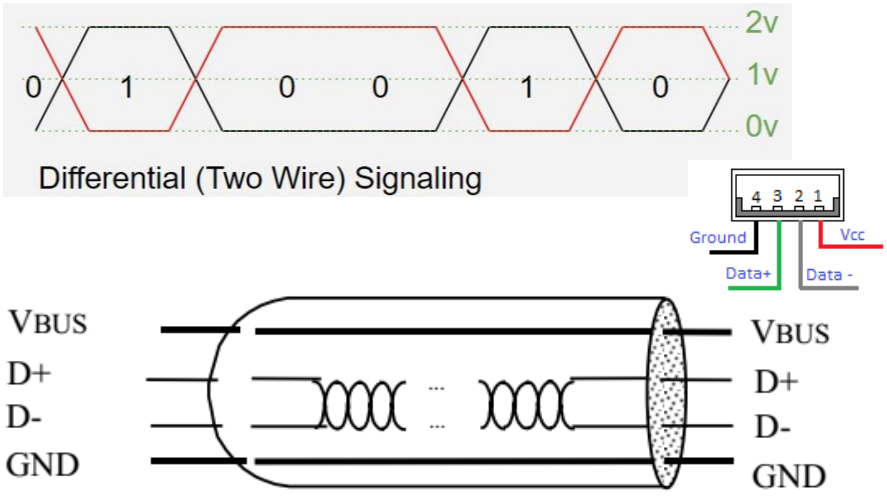
- **Part 1 – Registers, Microoperations and Implementations**
- **Part 2 – Counters, register cells, buses, & serial operations**
  - **Microoperations on single register (continued)**
    - **Counters**
  - **Serial transfers and microoperations**
  - **Register cell design**
- **Part 3 – Control of Register Transfers**

Serial Transfers and Microoperations

- A digital system is said to operate in a serial mode when information in the system is transferred or manipulated one bit at a time.
- Serial transfer method is in contrast to parallel transfer, in which all the bits of the register are transferred at the same time.



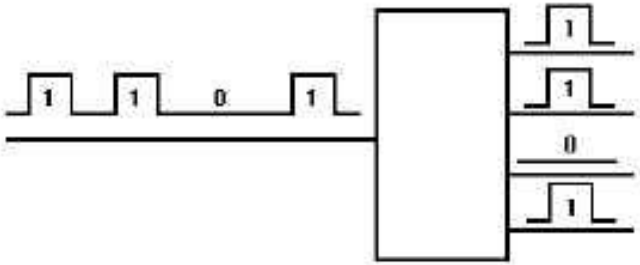
Serial data transfer by UART



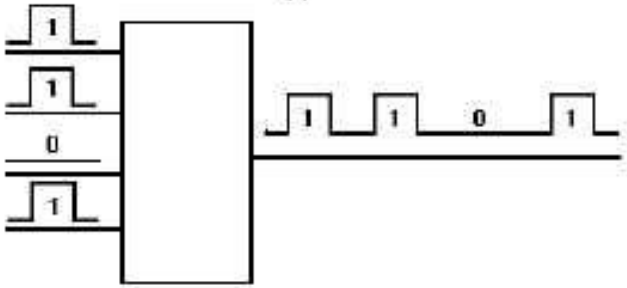
Serial data transfer by USB 2.0

Serial Transfers and Microoperations (continued)

- **Serial Transfers**
  - **Used for "narrow" transfer paths**
- **Example 1: Telephone or cable line**
  - **Serial-to-Parallel conversion at destination**
  - **Parallel-to-Serial conversion at source**



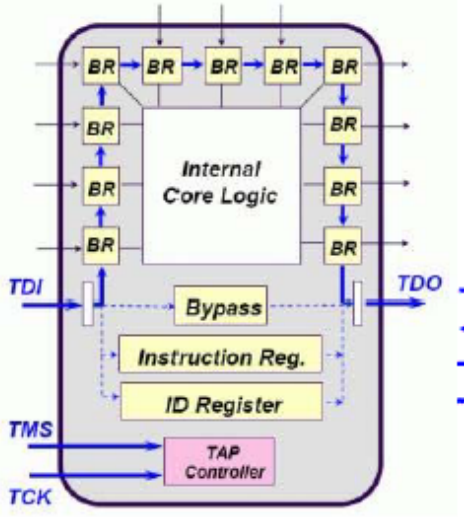
SERIAL-TO-PARALLEL CONVERSION
A



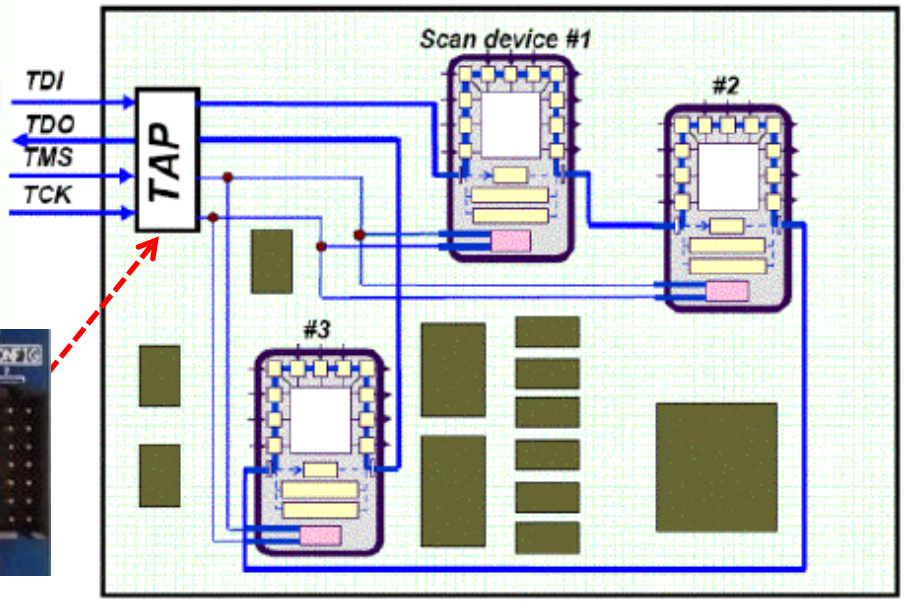
PARALLEL-TO-SERIAL CONVERSION
B

Serial Transfers and Microoperations (continued)

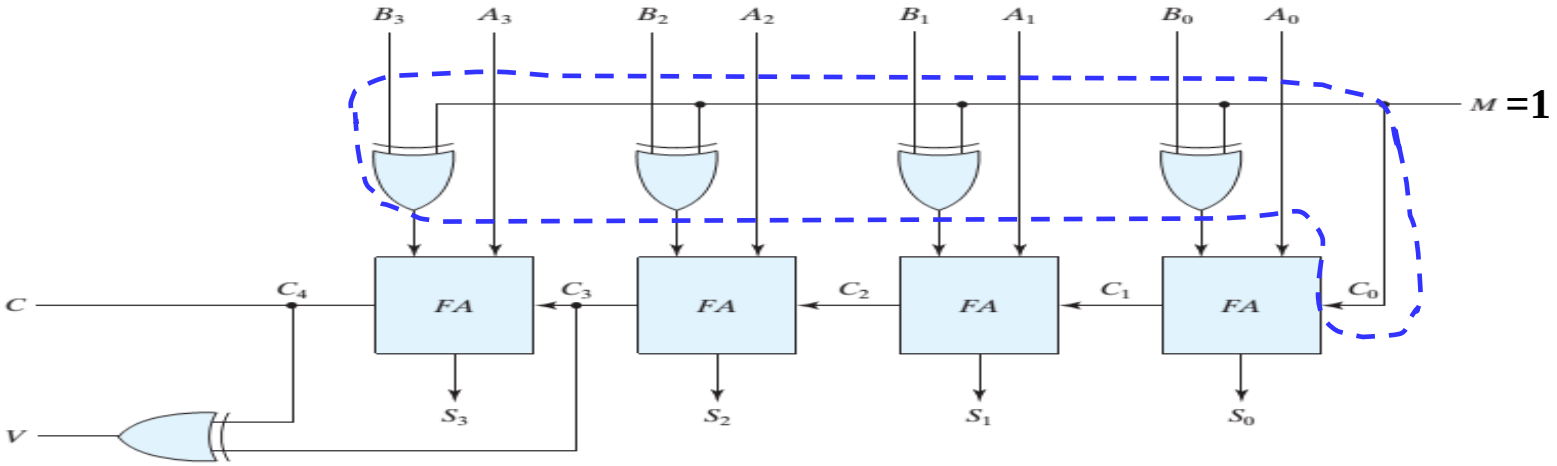
- **Example 2: Initialization and capture of the contents of many flip-flops for test purposes**
  - **Add shift function to all flip-flops and form large shift register**
  - **Use shifting for simultaneous initialization and capture operations**



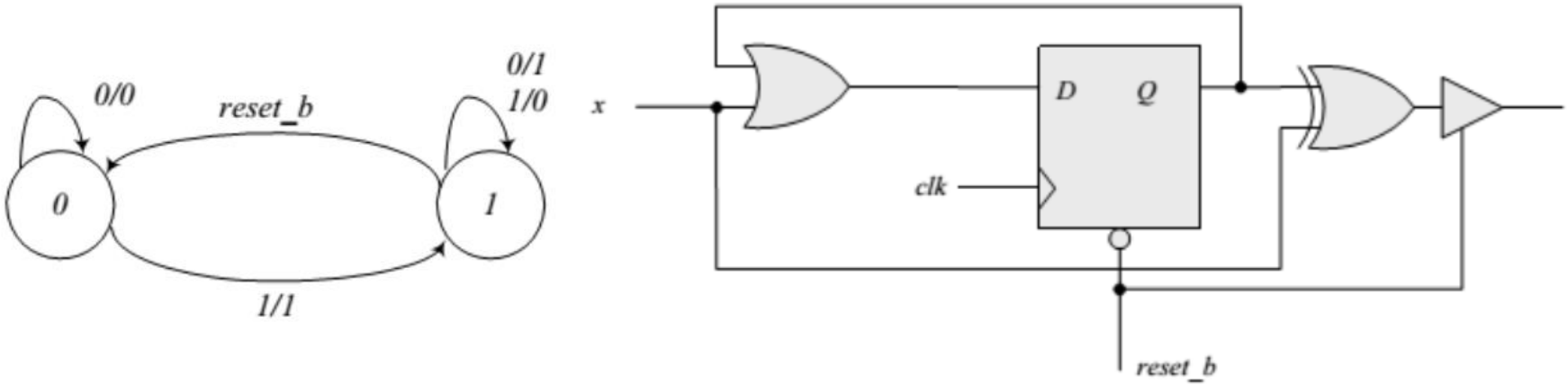
USB-JTAG



Parallel and Serial Operation Example



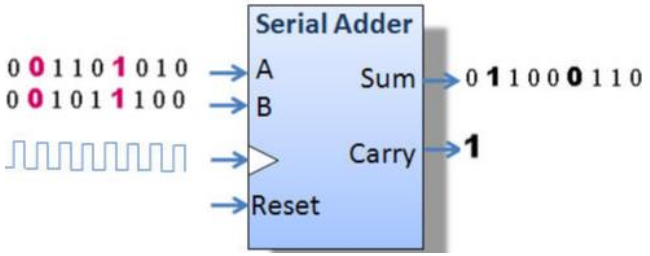
parallel 2's complementer



serial 2's complementer

Serial Microoperations

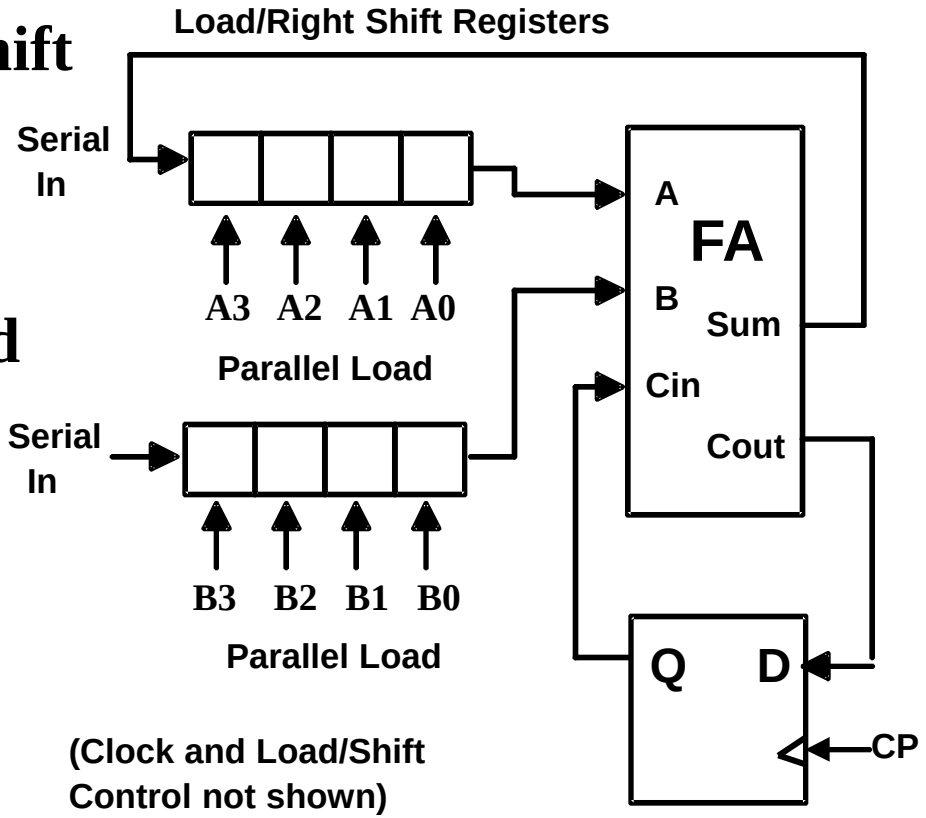
- Serial microoperations
  - Example: Addition



- By using two shift registers for operands, a full adder, and a flip flop (for the carry), we can add two numbers serially, starting at the least significant bit.
- Serial addition is a low cost way to add large numbers of operands, since a "tree" of full adder cells can be made to any depth, and each new level doubles the number of operands.
- Other operations can be performed serially as well, such as parity generation/checking or more complex error-check codes.

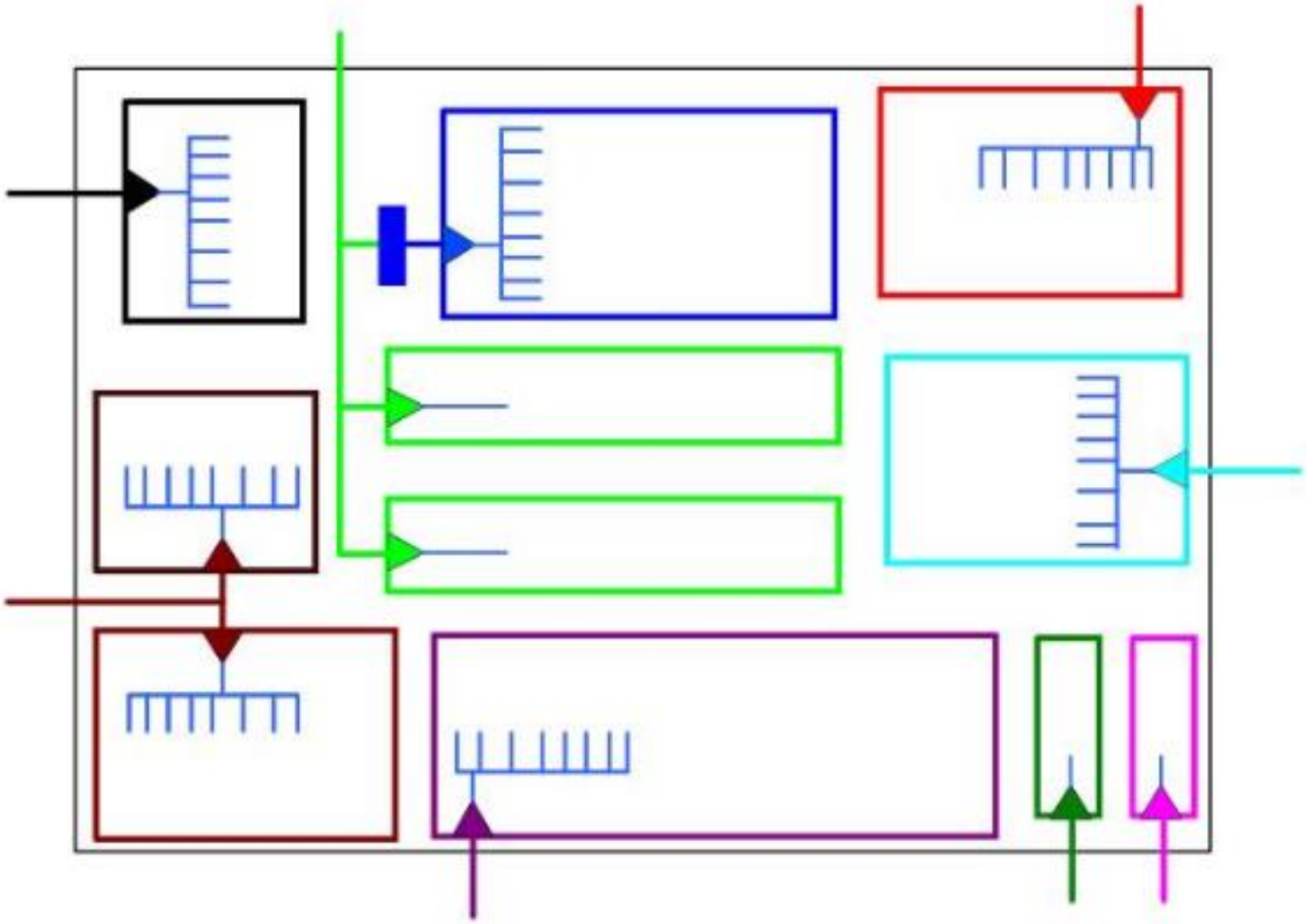
Serial Adder

- The circuit shown uses two shift registers for operands A(3:0) and B(3:0).
- A full adder, and one more flip flop (for the carry) is used to compute the sum.
- The result is stored in the A register and the final carry in the flip-flop



- With the operands and the result in shift registers, a tree of full adders can be used to add a large number of operands. Used as a common digital signal processing technique.

Digital System with Multiple Clock Domains

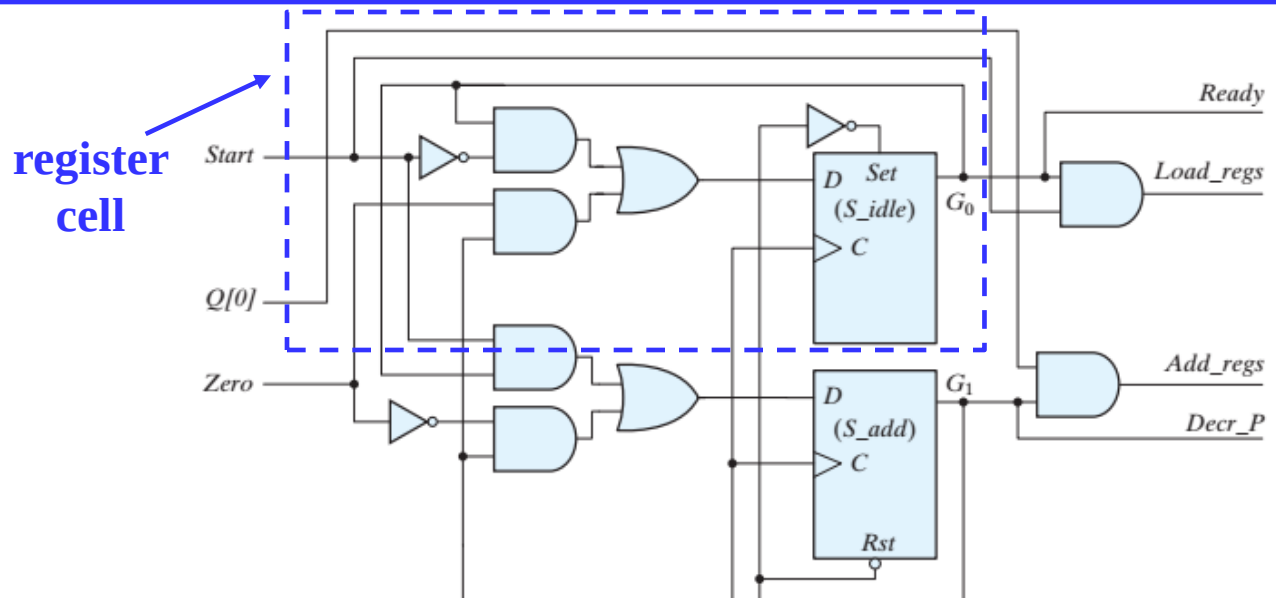


Appendix A: Signal Transfer across the Clock Boundary

Overview

- **Part 1 – Registers, Microoperations and Implementations**
- **Part 2 – Counters, register cells, buses, & serial operations**
  - **Microoperations on single register (continued)**
    - **Counters**
  - **Serial transfers and microoperations**
  - **Register cell design**
- **Part 3 – Control of Register Transfers**

Register Cell



- **We can connect iterative combinational circuits to flip-flops to form sequential circuits.**
- **A single-bit cell of an iterative combinational circuit, connected to a flip-flop that provides the output, forms a two-state sequential circuit called a register cell.**
- **We can design an n-bit register with one or more associated microoperations by designing a register cell and making n copies of it.**

Register Cell Design

- Assume that a register consists of identical cells
- Then register design can be approached as follows:
  - Design representative cell for the register
  - Connect copies of the cell 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
- Two design approaches
  - Sequential Circuit Design Approach
  - Multiplexer Approach

Register Cell Specifications

- A register
- Data inputs to the register
- Control input combinations to the register
  - Example 1: Not encoded
    - Control inputs: Load, Shift, Add
    - At most, one of Load, Shift, Add is 1 for any clock cycle
      (0,0,0), (1,0,0), (0,1,0), (0,0,1)
  - Example 2: Encoded
    - Control inputs: S1, S0
    - All possible binary combinations on S1, S0
      (0,0), (0,1), (1,0), (1,1)
- Register function

Register Cell Specifications (continued)

- **A set of register functions (typically specified as register transfers)**
  - **Example:**

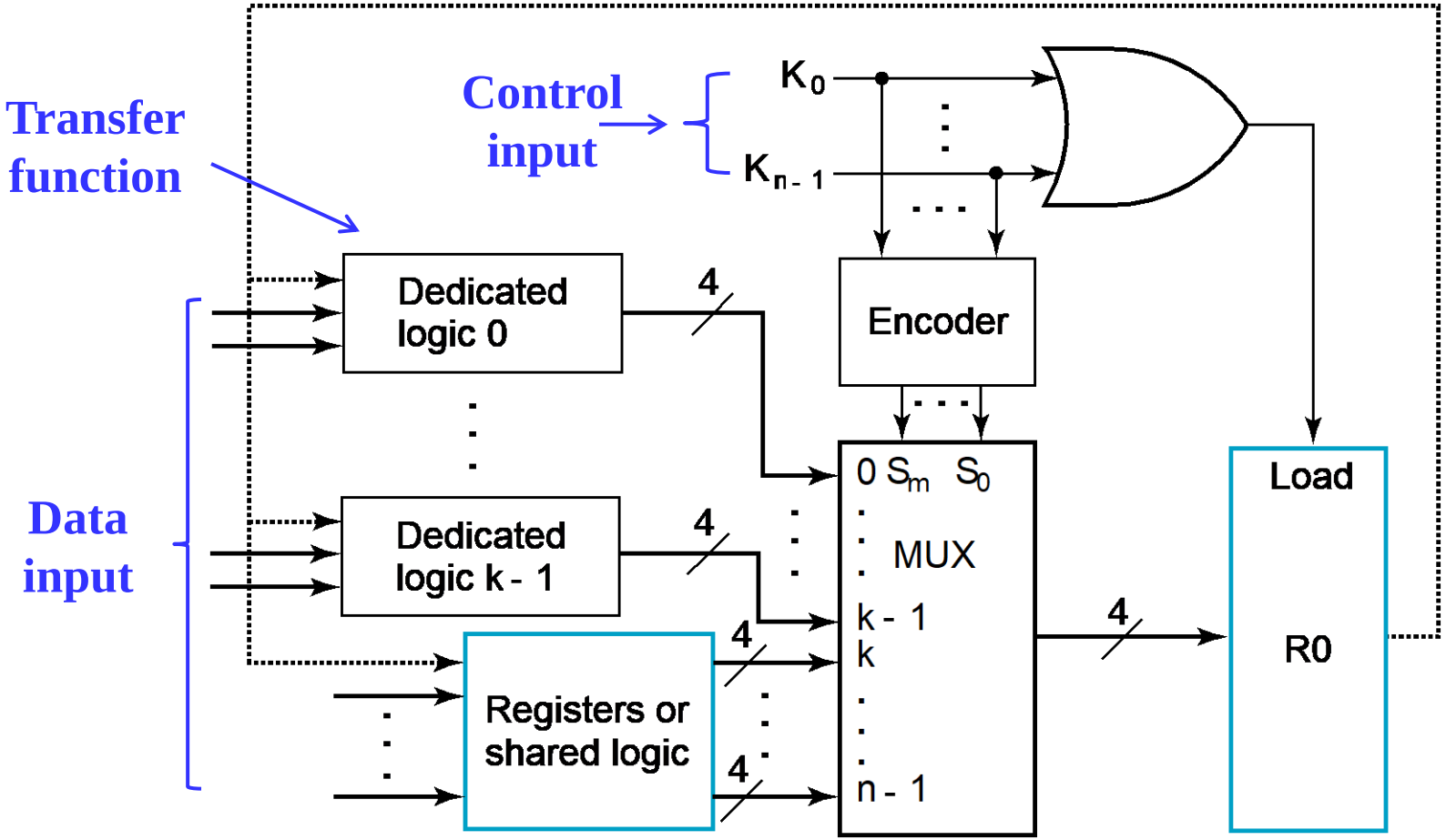
    Load: A ← B

    Shift: A ← sr B

    Add: A ← A + B
  - **Register cell specification example:**
    - **Control inputs: Load, Shift, Add**
    - **A hold state: if all control inputs are 0, hold the current register state**
    - **Data input: B, sr B, A + B, A**

Multiplexer Approach

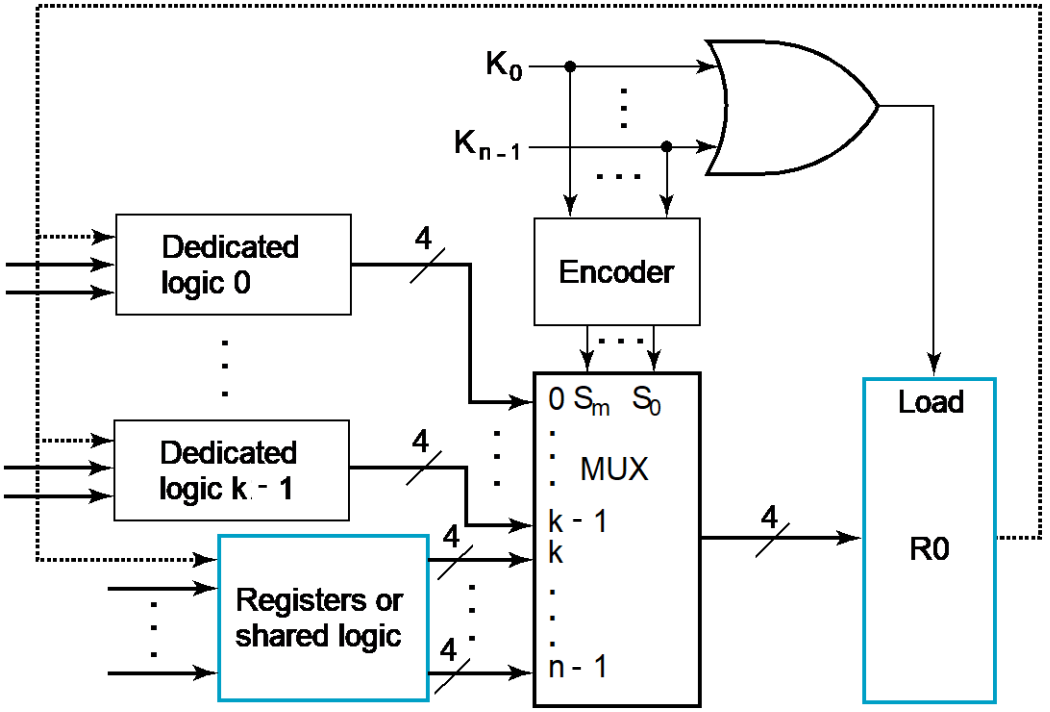
- Uses an n-input multiplexer with a variety of transfer sources and functions



Multiplexer Approach

- **Load enable by OR of control signals K_0, K_1, … K_{n-1} - assumes no load for 00…0**
- **Use:**
  - **Encoder + Multiplexer (shown) or**
  - **n x 2 AND-OR**

to select sources and/or transfer functions



Example 1: Register Cell Design

- **Register A (m-bits) Specification:**
  - **Data input: B**
  - **Control inputs (CX, CY)**
  - **Control input combinations (0,0), (0,1) (1,0)**
  - **Register transfers:**
  - **CX: A ← B ∨ A**
  - **CY :A ← B⊕A**
  - **Hold state: (0,0)**
- **Two design approaches**
  - **Multiplexer Approach**
  - **Sequential Circuit Design Approach**

Example 1: Register Cell Design (continued)

- **Multiplexer Approach**
- **Load Control**

  Load = CX + CY

- **Since all control combinations appear as if encoded (0,0), (0,1), (1,0) can use multiplexer without encoder:**

control signal

S1 = CX

S0 = CY

data input

D0 = A_i — **Hold A**

D1 = $A_i \leftarrow B_i \vee A_i$ — **CX = 1**

D2 = $A_i \leftarrow B_i \oplus A_i$ — **CY = 1**

- **Note that the decoder part of the 3-input multiplexer can be shared between bits if desired**

Sequential Circuit Design Approach

- **Find a state diagram or state table**
  - **Note that there are only two states with the state assignment equal to the register cell output value**
- **Use the design procedure in Chapter 4 to complete the cell design**
- **For optimization:**
  - **Use K-maps for up to 4 to 6 variables**
  - **Otherwise, use computer-aided or manual optimization**

Example 1: Register Cell Design (continued)

- **State Table:**

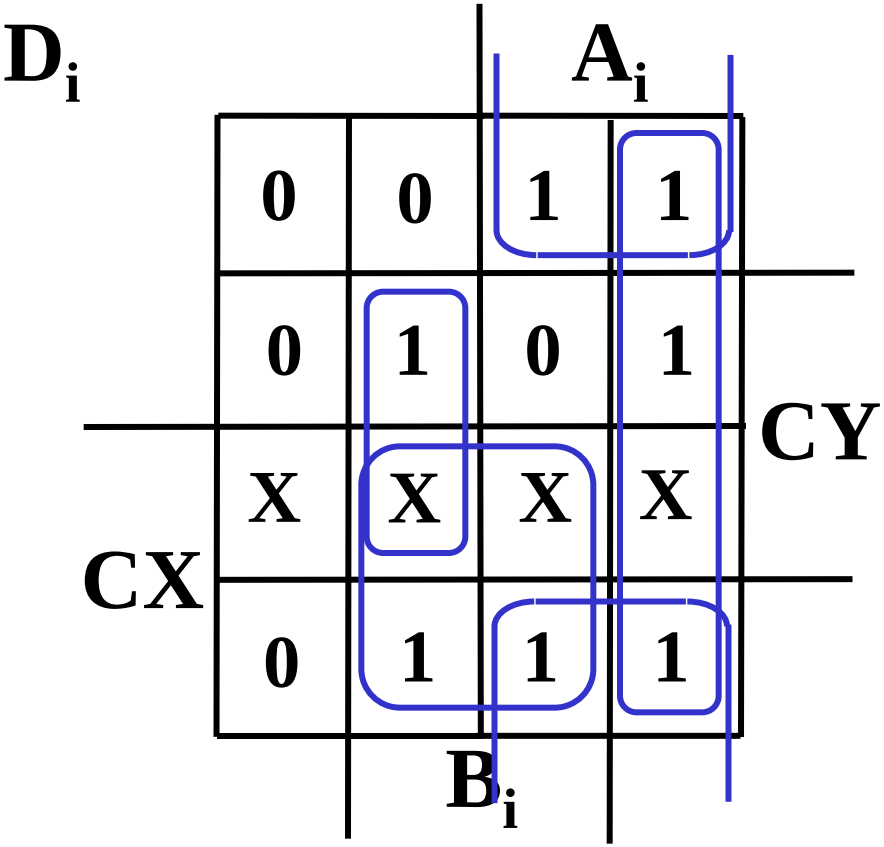
	Hold	Ai ∨ Bi		Ai ⊕ Bi	
state → A_i	CX = 0 CY = 0 B_i = x	CX = 1 CY = 0 B_i = 0	CX = 1 CY = 0 B_i = 1	CX = 0 CY = 1 B_i = 0	CX = 0 CY = 1 B_i = 1 ← input
0	0	0	1	0	1
1	1	1	1	1	0

- Four variables 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: Register Cell Design (continued)

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



Example 1: Register Cell Design (continued)

- **The resulting SOP equation:**

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

- **Using factoring and DeMorgan's law:**

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

  $$D_i = CX\,B_i + A_i \oplus (CY\,B_i)$$

  The gate input cost per cell = 2 + 8 + 2 + 2 = 14

- **The gate input cost per cell for the multiplexer approach is:**

  Per cell: 19

  Shared decoder logic: 8

- **Cost gain by sequential design > 5 per cell**

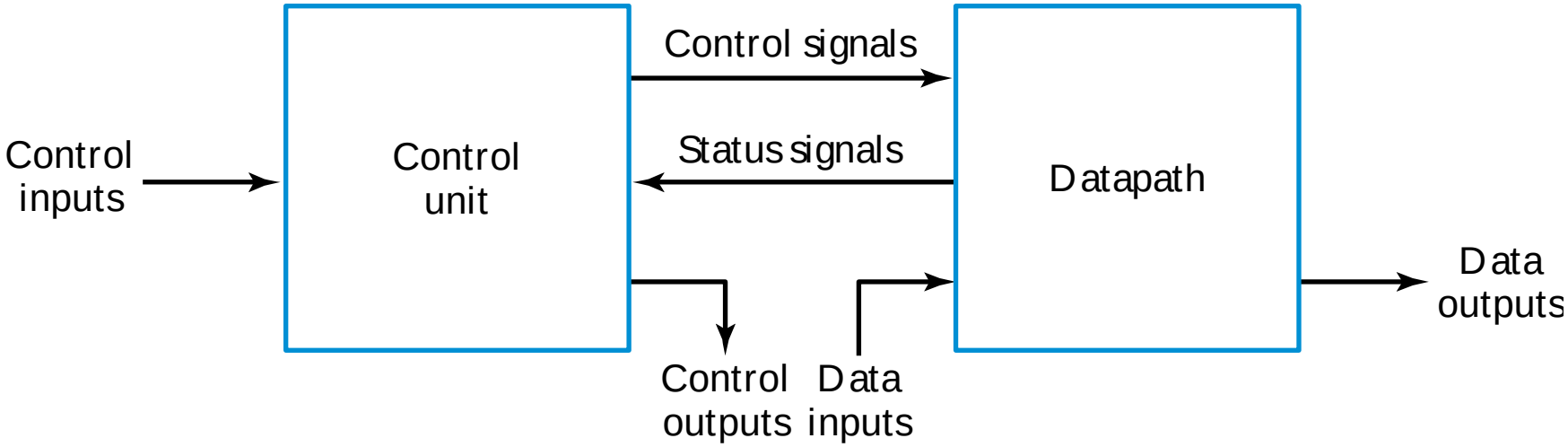
- **Also, no Enable on the flip-flop makes it cost less**

Overview

- **Part 1 – Registers, Microoperations and Implementations**
- **Part 2 – Counters, Register Cells, Buses, & Serial Operations**
- **Part 3 – Control of Register Transfers**
  - **Introduction to register transfer systems**
  - **Register transfer system design procedure**
  - **A design example**
  - **Microprogrammed control**

Introduction: Digital System Design Method

- **Three essential elements**
  - **Set of registers: mostly in Datapath with some in Control Unit**
  - **Basic operation (microoperation): Register transfers performed on registers**
  - **Control: that supervises the sequencing of the register transfers**



Register Transfer System Design Procedure

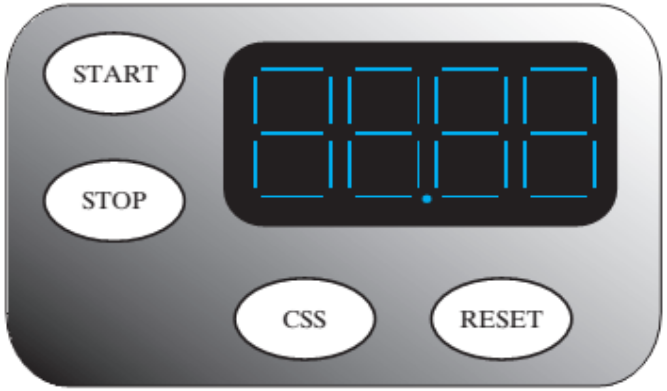
- **Write a detailed system specification.**
- **Determine all data, control and status input signals, all data, control and status output signals, and registers of the datapath and control unit.**
- **Find a state machine diagram for the system including register transfers for the datapath and control unit as outputs.**
- **Determine all internal control and status signals. Use these signals to separate output conditions and actions, including register transfers, from the state machine diagram flow and represent them in tabular form.**

Register Transfer System Design Procedure (continued)

- **Draw a block diagram of the datapath including all control and status inputs and outputs. Draw a block diagram of the control if it includes register transfer hardware.**
- **Design any specialized register transfer logic as needed for the datapath and the control.**
- **Design the control unit logic.**
- **Verify the correct operation of the combined datapath and control unit. If verification fails, debug the system and verify the changed system.**

Design Example – DASHWATCH - Specs

- Very Inexpensive Stop Watch for "dash" runners
- Times intervals to at most 99.99 seconds (RESET)
- Inputs: START, STOP, CSS (compare and store shortest), RESET
- The START button resets a timer to 0 and then starts the timer, and the STOP button stops the timer and the latest dash time is displayed on the 4 digit BCD LCD
- Output: 4 digit BCD LCD with decimal point
- Registers: 4-digit BCD Counter and 16-bit Parallel Load Register



(a)

TM (Timer)

4-Digit BCD Counter

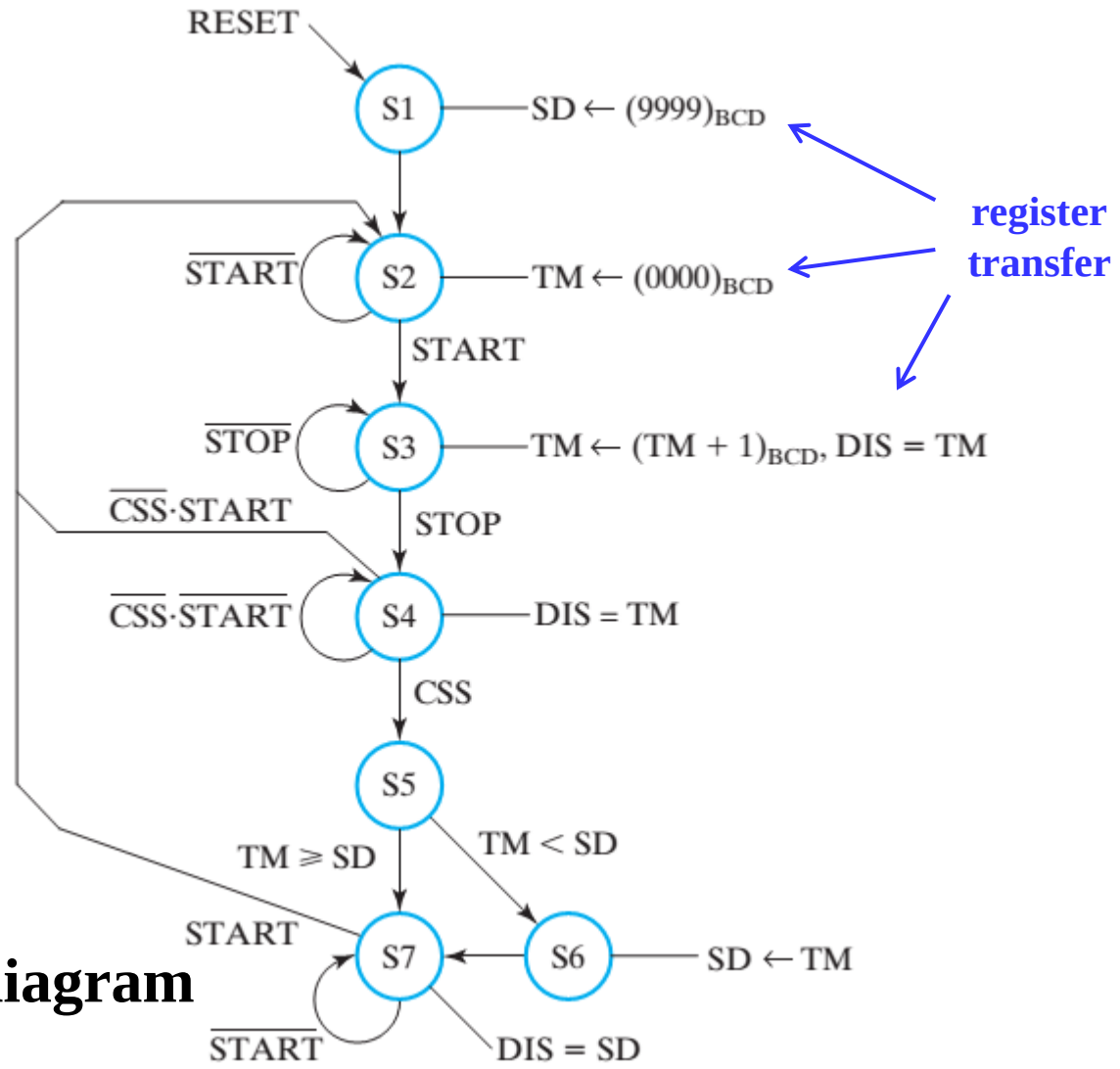
SD (Shortest Dash)

16-Bit Parallel Load Register

DASHWATCH Inputs, Outputs, and Registers

	Symbol	Function	Type
Button	START	Initialize timer to 0 and start timer	Control input
	STOP	Stop timer and display timer	Control input
	CSS	Compare, store, and display shortest dash time	Control input
	RESET	Set shortest value to 10011001	Control input
LCD	B_1	Digit 1 data vector a, b, c, d, e, f, g to display	Data output vector
	B_0	Digit 0 data vector a, b, c, d, e, f, g to display	Data output vector
	DP	Decimal point to display (= 1)	Data output
	B_{-1}	Digit −1 data vector a, b, c, d, e, f, g to display	Data output vector
	B_{-2}	Digit −2 data vector a, b, c, d, e, f, g to display	Data output vector
	B	The 29-bit display input vector (B_1, B_0, DP, B_{-1}, B_{-2})	Data output vector
Register	TM	4-Digit BCD counter	16-Bit register
	SD	Parallel load register	16-Bit register

DASHWATCH State Machine Diagram with Register Transfer Outputs



Which state machine diagram is it? Mealy or Moore?

State Machine Diagram Design

- **Specify only Moore outputs (no particular reason)**
- **S1: Reset state - in this state, initialize SD to 100110011001001 (99.99), the maximum possible dash time.**
- **S2: Because of Moore output spec, S1 cannot be used for this state since SD is not to be initialized again to 99.99 after having passed through states S4 or S7. TM is initialized to $(0000)_{BCD}$ for next dash.**
- **S3: State during dash. Entered with START and exited with STOP. While in state, 1 (0.01 seconds) is added to TM for each clock pulse. (Clock frequency is 100 Hz), and DIS shows TM value.**
- **S4: Decision state whether to Compare, Store, and display Shortest dash time, or to continue to display TM. Also START begins new dash.**
- **S5: State for comparison of TM to SD.**
- **S6: State for loading TM into SD if TM is smaller.**
- **S7: State for START to begin new dash and display of SD as shortest dash time.**

DASHWATCH Output Control/Status Table

Action or Status	Control or Status Signals	Meaning for Values 1 and 0
$TM \leftarrow (0000)_{BCD}$	RSTM	1: Reset TM to 0 (synchronous reset) 0: No reset of TM
$TM \leftarrow (TM + 1)_{BCD}$	ENTM	1: BCD count up TM by 1, 0: hold TM value
$SD \leftarrow (9999)_{BCD}$	UPDATE	0: Select 1001100110011001 for loading SD 1: Select TM for loading SD
$SD \leftarrow TM$	LSR	1: Enable load SD, 0: disable load SD
DIS = TM DIS = SD	DS	0: Select TM for DIS 1: Select SD for DIS
TM < SD TM ≥ SD	ALTB	1: TM less than SD 0: TM greater than or equal to SD

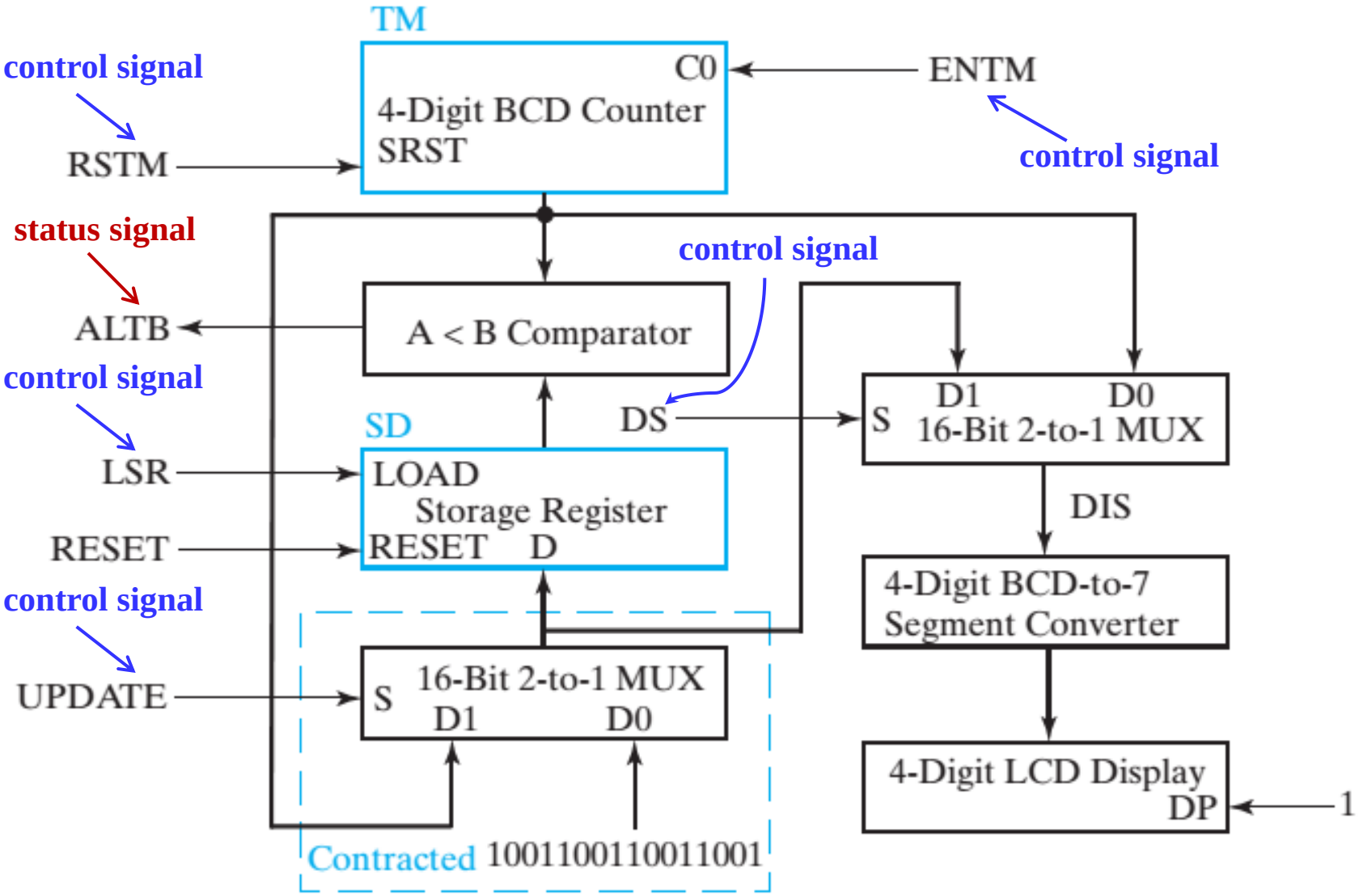
control signal

status signal

Determination of Internal Control/Status Signals

- **TM – Timer**
  - **Reset to 0000: RSTM**
  - **Enable to Count Up: ENTM**
- **SD – Shortest Dash**
  - **Load SD: LSR = 1;**
  - **Select input 9999: UPDATE = 0**
  - **Select input TM: UPDATE = 1**
- **DIS – Display (B_1, B_0, DP, B_{-1}, B_{-2})**
  - **Select input TM: DS = 0**
  - **Select input SD: DS = 1**
- **Compare TM and SD (Status)**
  - **TM < SD: ALTB = 1**
  - **TM ≥ SD: ALTB = 0**

DASHWATCH Datapath



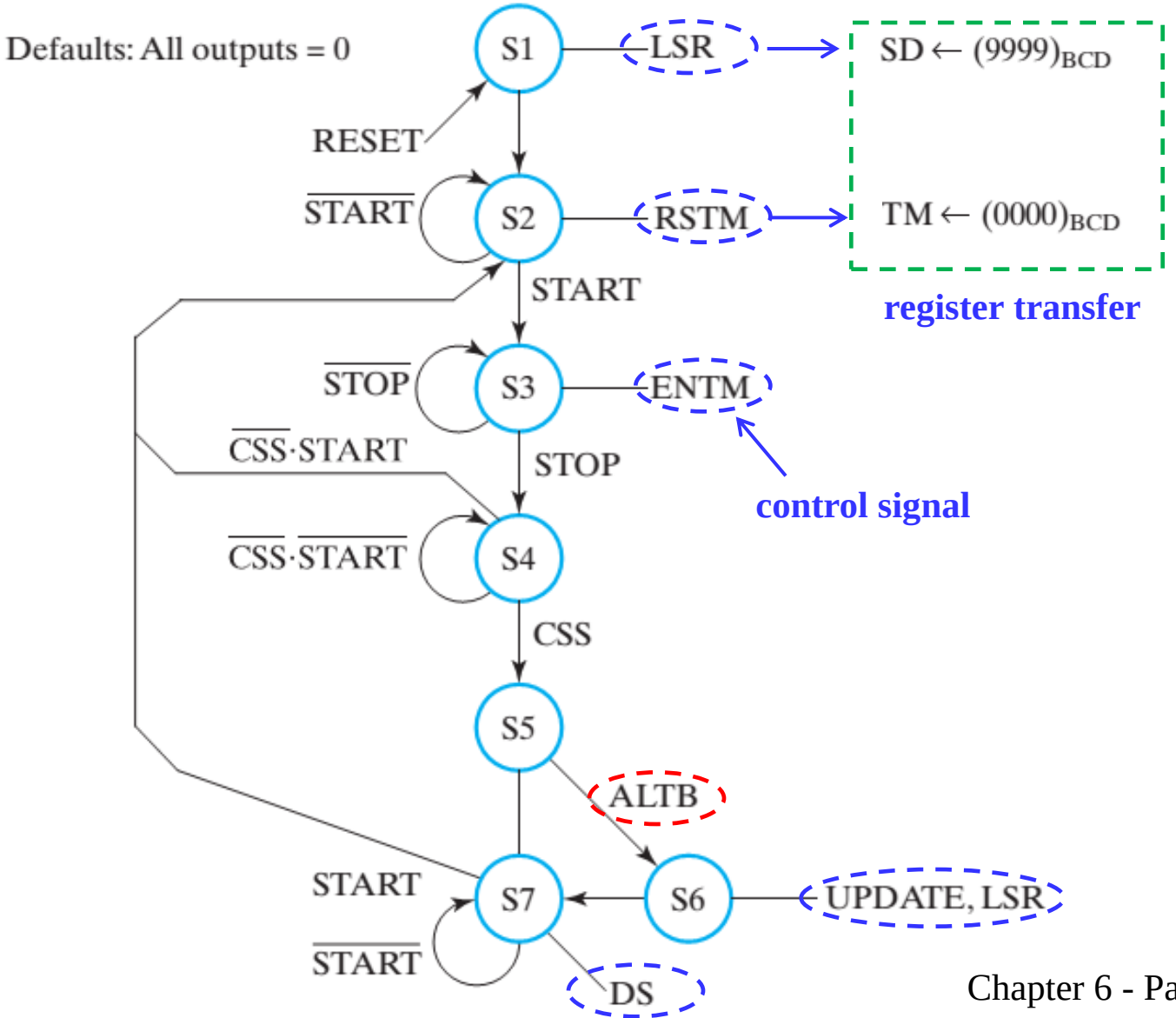
DASHWATCH – Datapath Development

- **TM: 4-digit BCD Counter with Synchronous Reset**
  - **Based on previous BCD adder digit design**
  - **synchronous reset SRST added**
  - **SRST = RSTM**
  - **C0 (Incoming carry) = ENTM**
- **A < B Comparator**
  - **Compares TM to SD**
  - **Designed as left-to-right iterative cell array with output C0**
- **SD: Standard 16-bit parallel load register**
  - **LOAD = LSR**
  - **Contracted standard 2-way, 16-bit multiplexer used to select between 9999_{BCD} and TM as parallel load input D**
  - **S = UPDATE**

DASHWATCH – Datapath Development – Display Logic

- **2-way 16-bit multiplexer**
  - **Selects between TM and SD**
  - **S = DS**
- **4-digit BCD-to-7 Segment Converter**
  - **Uses previous design**
- **4-digit 7-Segment Display with Decimal Point**
  - **2-digit fractional part**
  - **Decimal Point control = DP**
  - **DP = 1**

DASHWATCH – SMD with Control Signal Outputs Replacing Register Transfers



DASHWATCH – FF Input Equations

- **One-Hot State Assignment – 7 bits**
- **State S1 entered only by using asynchronous RESET**

$$D_{S1} = S1(t+1) = 0$$

$$D_{S2} = S2(t+1) = S1 + S2 \cdot \overline{START} + S4 \cdot \overline{CSS} \cdot START + S7 \cdot START$$

$$D_{S3} = S3(t+1) = S2 \cdot START + S3 \cdot \overline{STOP}$$

$$D_{S4} = S4(t+1) = S3 \cdot STOP + S4 \cdot \overline{CSS} \cdot \overline{START}$$

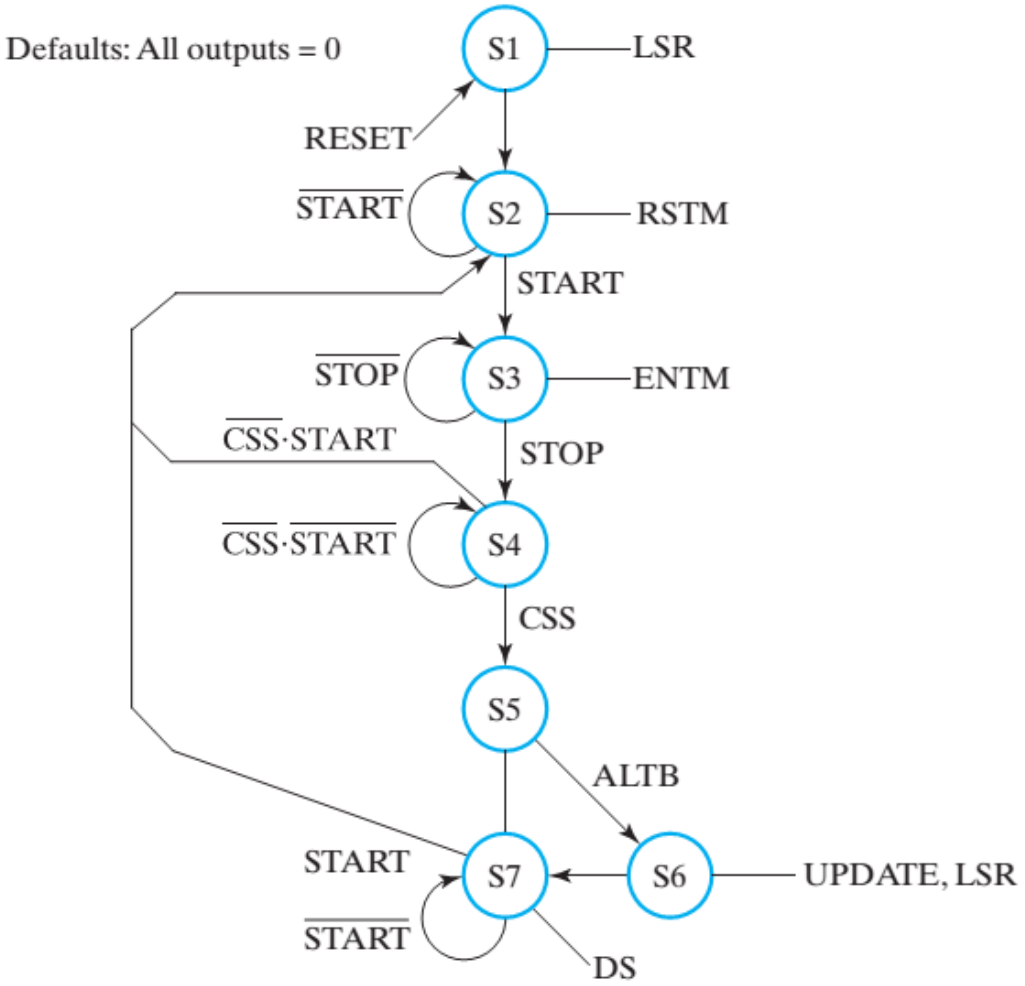
$$D_{S5} = S5(t+1) = S4 \cdot CSS$$

$$D_{S6} = S5 \cdot ALTB$$

$$D_{S7} = S7(t+1) = S5 \cdot \overline{ALTB} + S6 + S7 \cdot \overline{START}$$

DASHWATCH – Output Equations

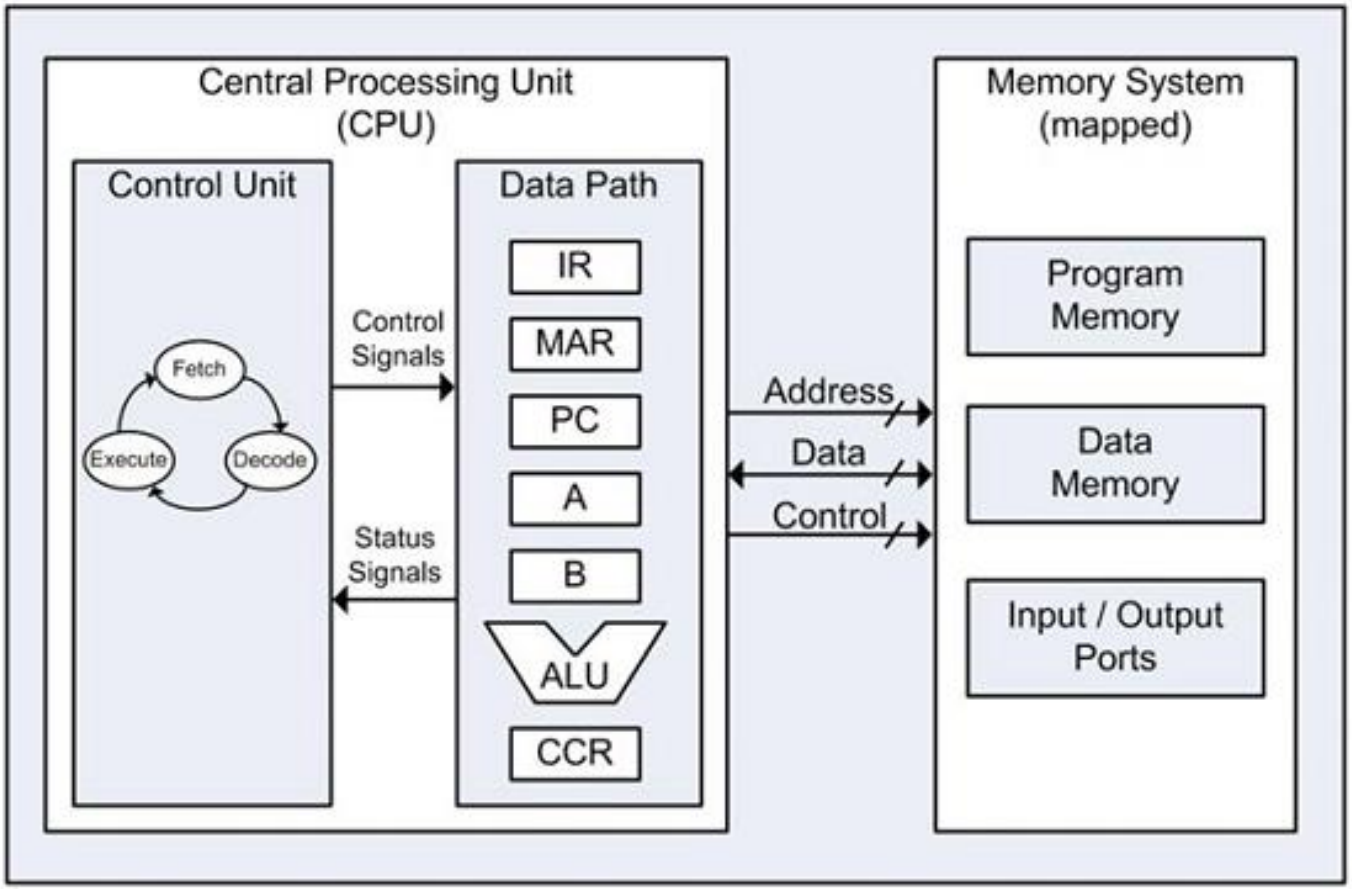
$$LSR = S1 + S6$$
$$RSTM = S2$$
$$ENTM = S3$$
$$UPDATE = S6$$
$$DS = S7$$



Programmable and Non-Programmable Systems

- **Non-programmable System: Specific system**
  - **the control unit does not deal with fetching and executing instructions.**
  - **There is no PC or similar register in such a system.**
  - **Instead, the control unit determines the operations to be performed and the sequence of those operations, based on its inputs and the status bits from the datapath.**
- **Programmable System: General system**
  - **A portion of the input consists of a sequence of instructions called a program.**
  - **Typically stored in a memory and addressed by a program counter.**
  - **The Control Unit is responsible for fetching and executing these instructions.**

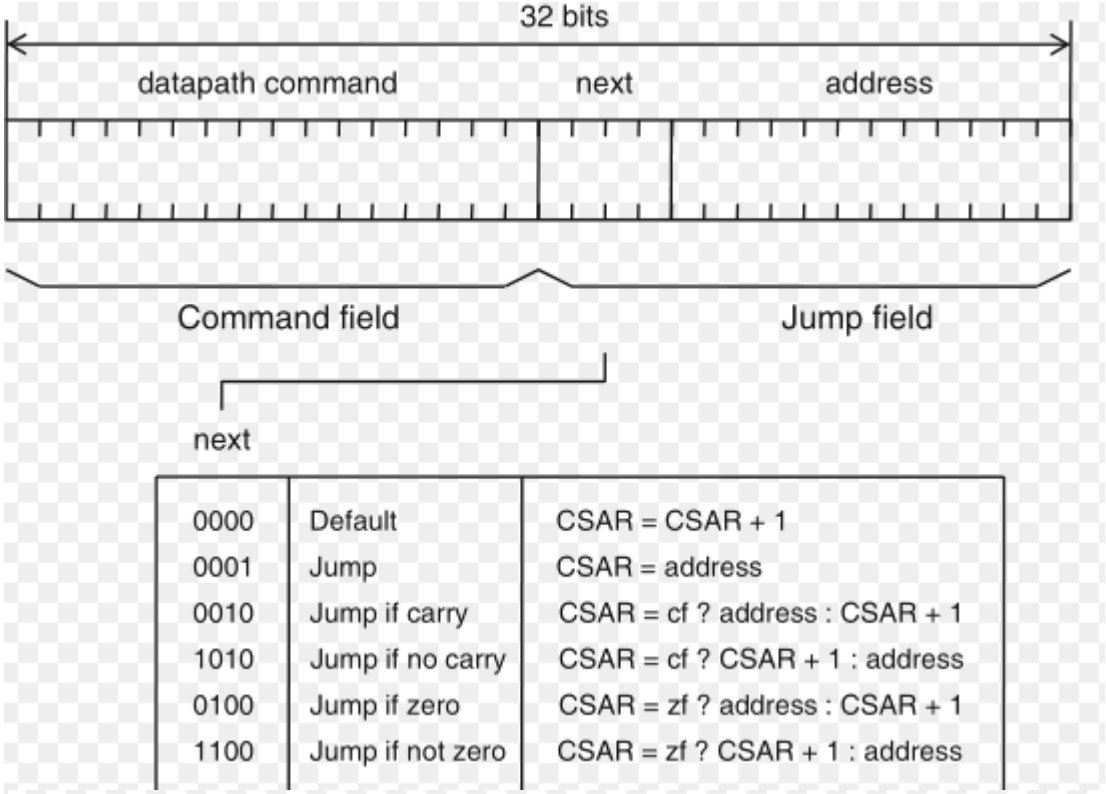
Microprogrammed Control



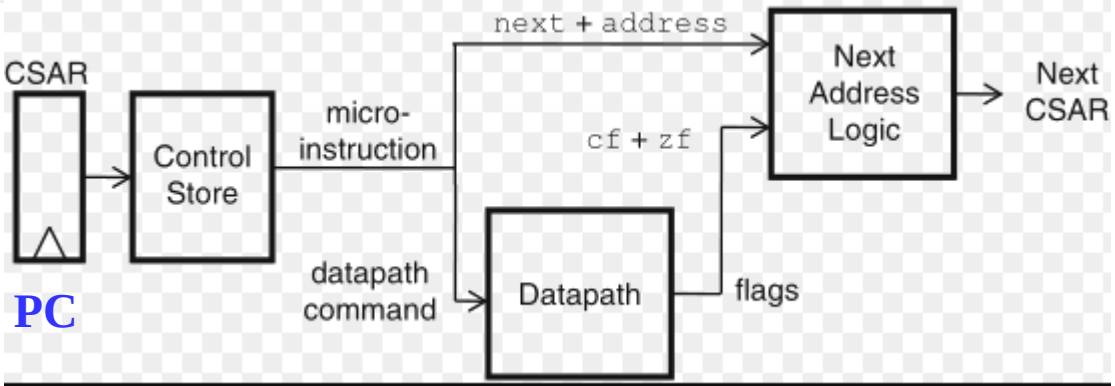
Microprogrammed Control (continued)

- ***Microprogrammed Control*** **— a control unit with binary control values stored as words in memory.**
- ***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**

Microprogrammed Control (continued)

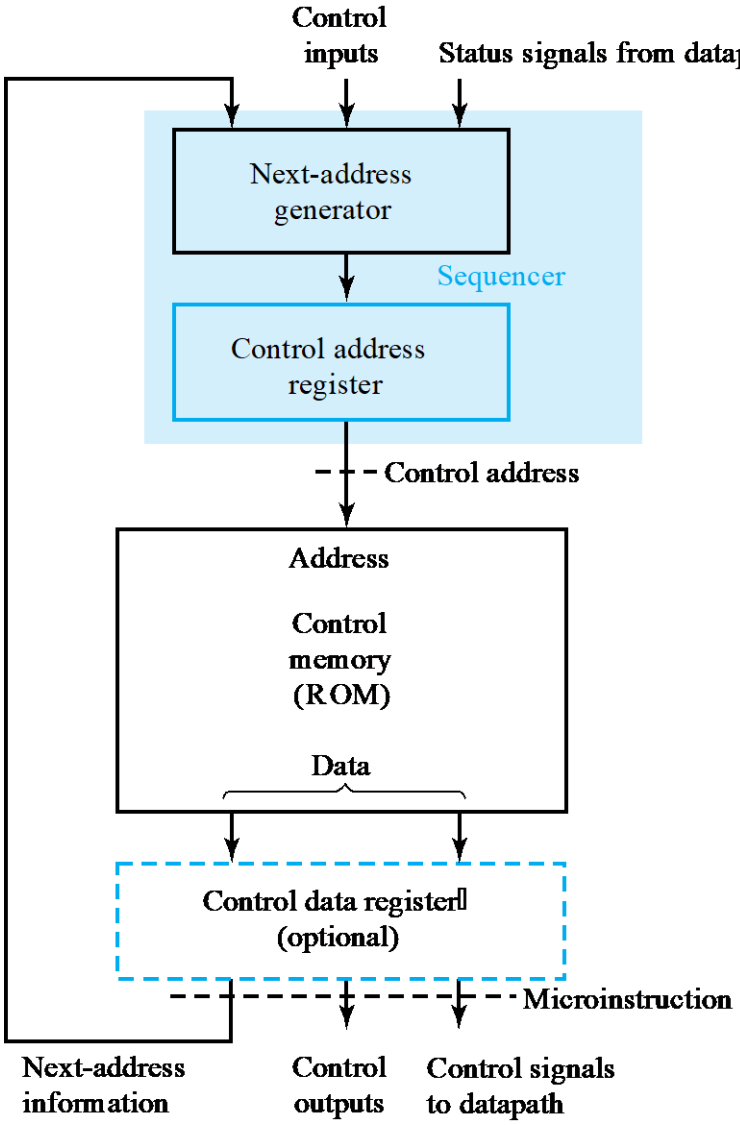


Typical microinstruction format



Microinstruction execution

Microprogrammed Control (continued)



Reset Vector

x86 -> 0xFFF0

i386 -> 0xFFFFFFF0

Assignments

Reading:

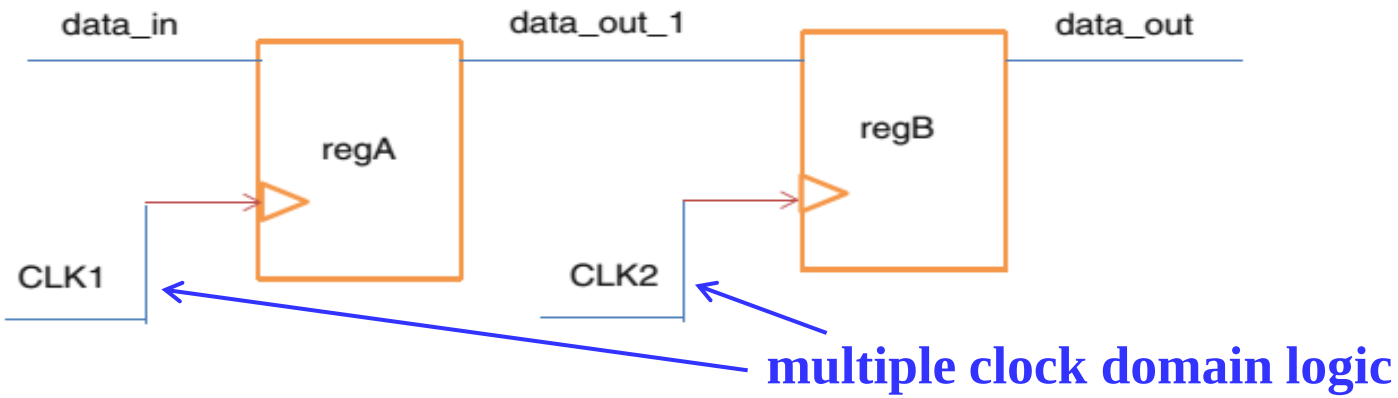
- **6-7, 6-9, 6-10**

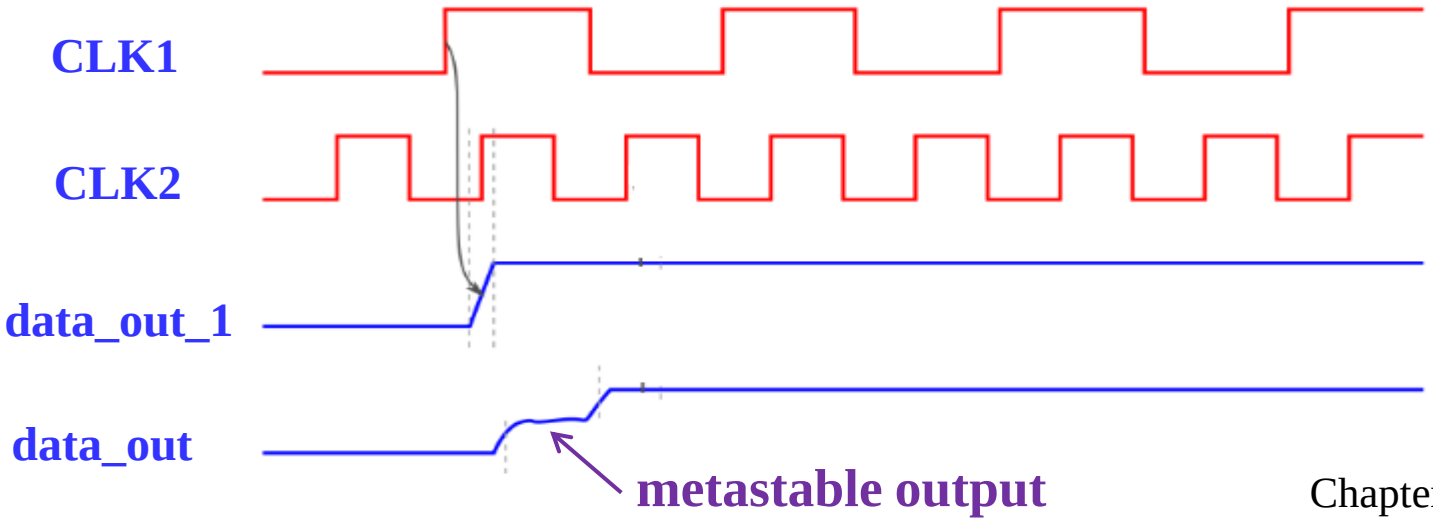
Problem assignment:

- **6-23**

Appendix A: Signal Transfer across the Clock Boundary

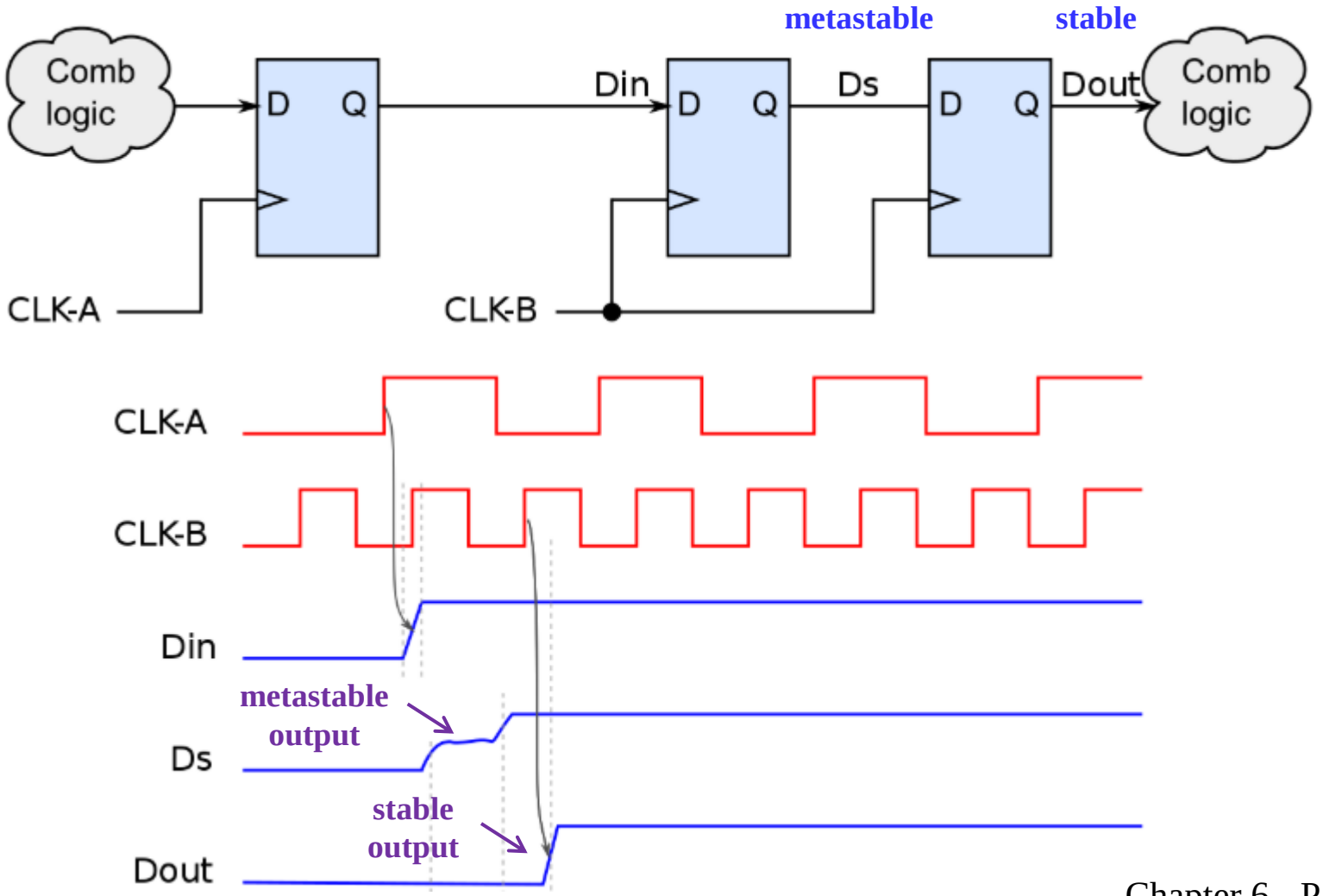
- **Passing data from one clock domain to another clock domain is difficult and error-prone task.**





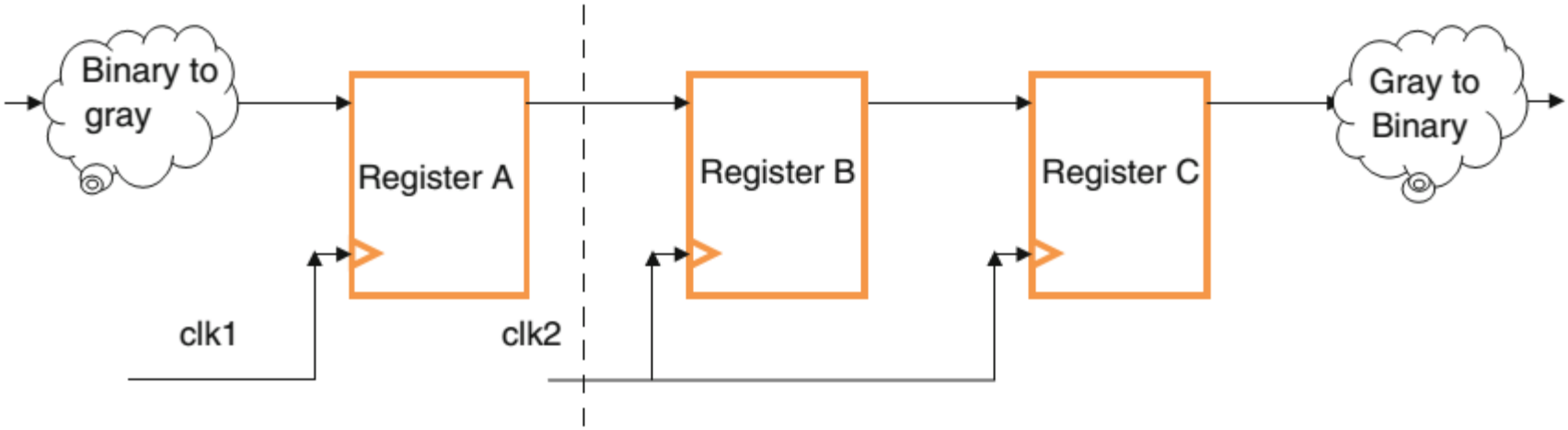
Signal Transfer across the Clock Boundary (continued)

- **Two Flip-Flop Synchronizer**



Signal Transfer across the Clock Boundary (continued)

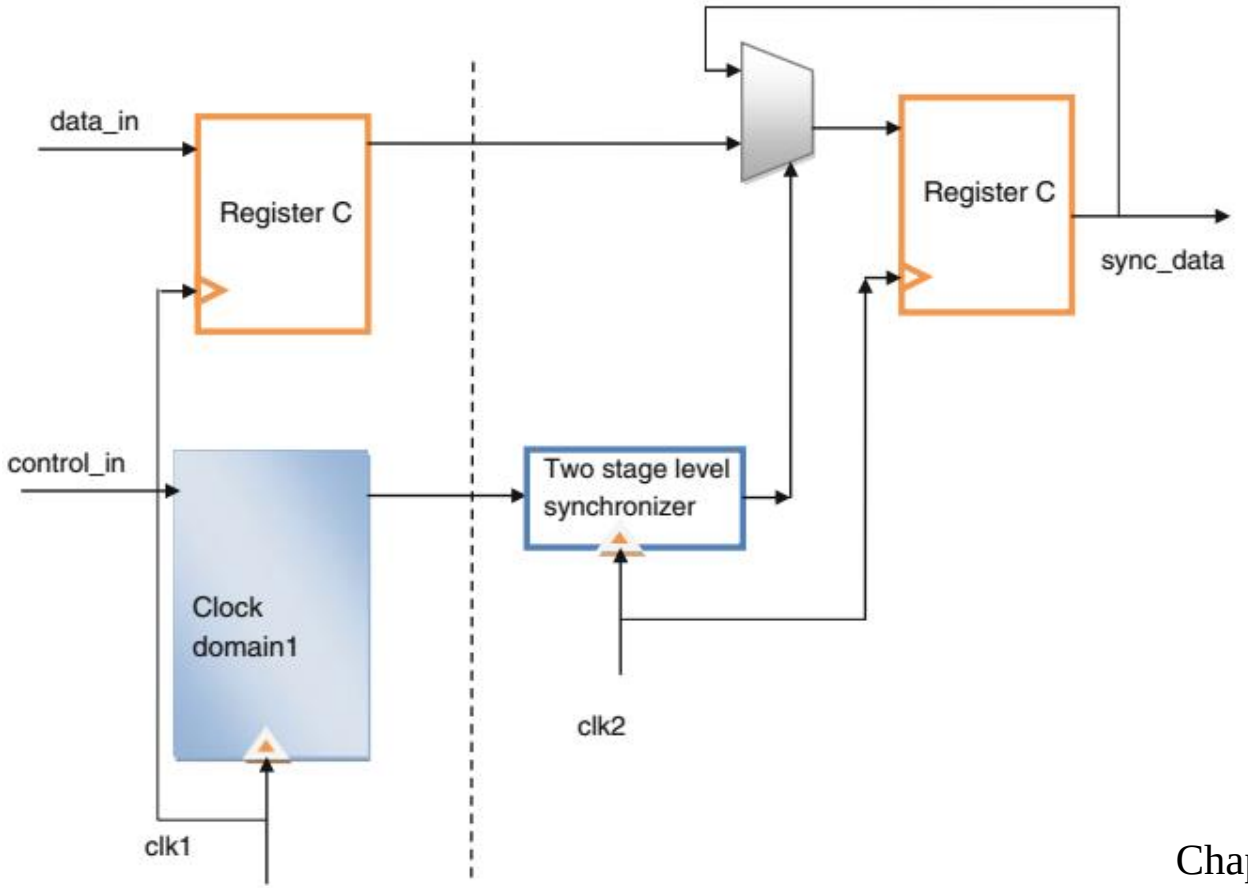
- **Gray Encoding**
  - **While passing multiple bits of the data or control signals it is essential to use the gray encoding technique.**
  - **Gray encoding is guaranteed to sample the one-bit change across the clocking boundary.**



Binary: 0111 ⟶ 1000(XXXX) ⟶ XXXX
Gray: 0100 ⟶ 1100(X100) ⟶ 0100/1100

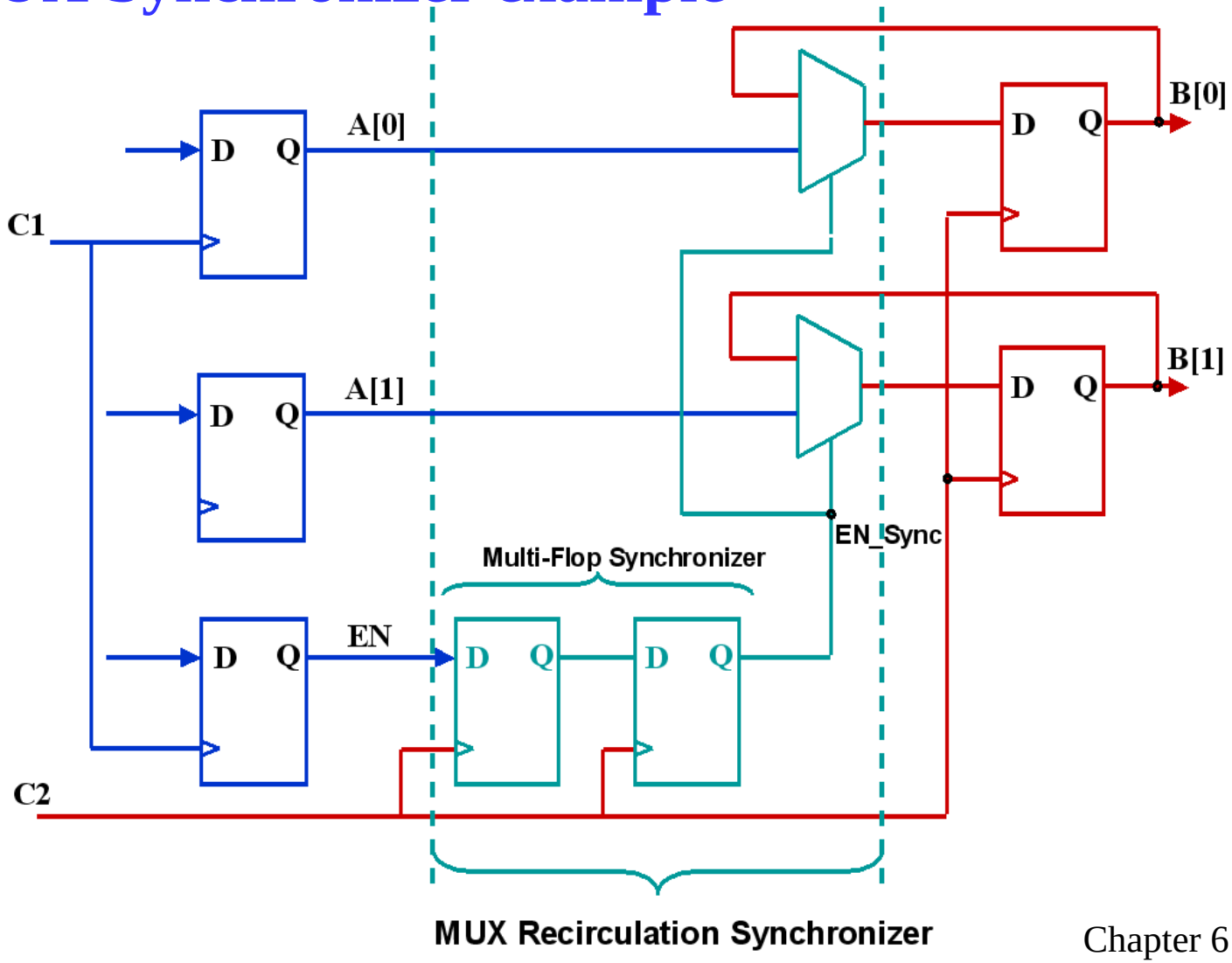
Signal Transfer across the Clock Boundary (continued)

- **MUX Synchronizer**
  - **Use the pair of the data and control signals while sending the information across clock domain.**



Signal Transfer across the Clock Boundary (continued)

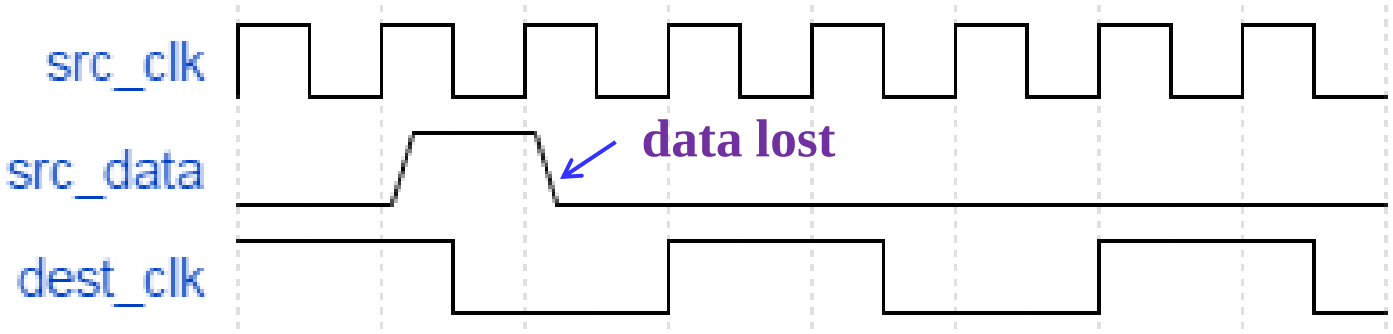
- MUX Synchronizer example



Signal Transfer across the Clock Boundary (continued)

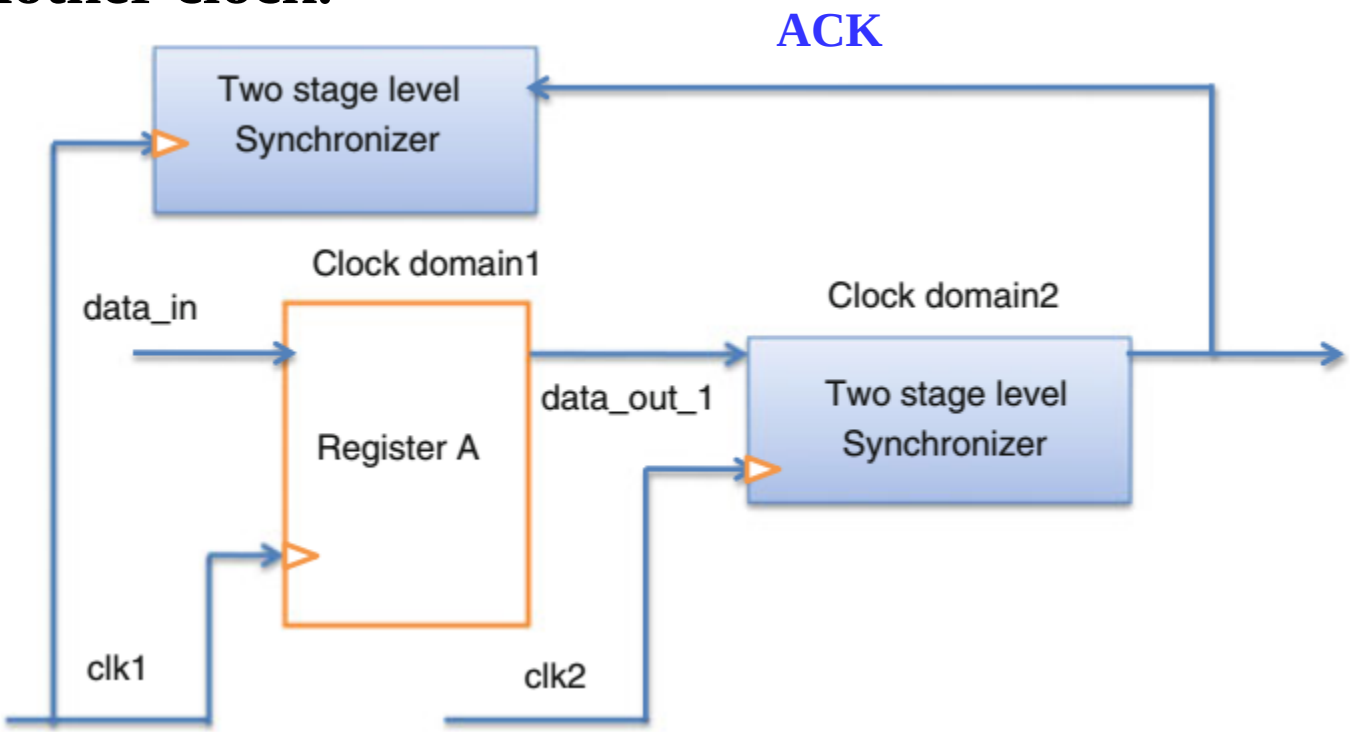
- **Two scenarios of clock domain crossing**
  - **Synchronizing slow signals into fast clock domain**

    Using "open-loop" synchronizers without acknowledgement.
  - **Synchronizing fast signals into slow clock domain**
    - **Using pulse stretcher ("open-loop" synchronizers)**
    - **Using "closed-loop" synchronizers with acknowledgement.**



Signal Transfer across the Clock Boundary (continued)

- **Handshaking Mechanism - synchronizer with feedback acknowledge**
  - **One or more than one handshake signals are required while passing the data from one of the clock domains to another clock.**



Signal Transfer across the Clock Boundary (continued)

- **FIFO (First in First Out) Synchronizer**
  - **FIFO memory buffers can be used as a synchronizer to pass the data between multiple clock domains.**