Logic and Computer Design Fundamentals

Chapter 7 – Registers and Register Transfers

Part 1 – Registers, Microoperations and Implementations

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.
(Hyperlinks are active in View Show mode)
Overview

- **Part 1 - Registers, Microoperations and Implementations**
  - Registers and load enable
  - Register transfer operations
  - Microoperations - arithmetic, logic, and shift
  - Microoperations on a single register
    - Multiplexer-based transfers
    - Shift registers

- **Part 2 - Counters, Register Cells, Buses, & Serial Operations**

- **Part 3 – Control of Register Transfers**
Registers

- **Register** – a collection of binary storage elements
- In theory, a register is sequential logic which can be defined by a state table
- More often, think of a register as storing a vector of binary values
- Frequently used to perform simple data storage and data movement and processing operations
Registers
Example: 2-bit Register

- How many states are there? \(2^2=4\)
- How many input combinations? \(2^2=4\)
- How many output combinations? \(2^2=4\)
- What is the output function?
  \(Y_1 = A_1\)
  \(Y_0 = A_0\)
- What is the next state function?
  \(A_1(t+1) = \text{IN}_1\)
  \(A_0(t+1) = \text{IN}_0\)
- Moore or Mealy?
  Moore
- What are the quantities above for an \(n\)-bit register?
  \(\text{States} = 2^n\)
  \(\text{Input Combinations} = 2^n\)
  \(\text{Output Combinations} = 2^n\)
Register Design Models

- Due to the large numbers of states and input combinations as \( n \) becomes large, the state diagram/state table model is not feasible!

- What are methods we can use to design registers?
  - Add predefined combinational circuits to registers
    - Example: To count up, connect the register flip-flops to an incrementer
  - Design individual cells using the state diagram/state table model and combine them into a register
    - A 1-bit cell has just two states
    - Output is usually the state variable
Add Combinational Circuits to Registers

Incrementer

Register

D3 Q3
D2 Q2
D1 Q1
D0 Q0

Combinational Circuit

Clock

Counter
Design Individual Cells
Register Storage

- **Expectations:**
  - A register can store information for multiple clock cycles
  - To “store” or “load” information should be controlled by a signal

- **Reality:**
  - A D flip-flop register loads information on every clock cycle

- **Realizing expectations:**
  1. Use a signal to block the clock to the register,
  2. Use a signal to control feedback of the output of the register back to its inputs, or
  3. Use other SR or JK flip-flops, that for (0,0) applied, store their state

- **Load** is a frequent name for the signal that controls register storage and loading
  - **Load = 1:** Loads input values (load new values)
  - **Load = 0:** Loads register contents (hold current values)
Registers with Clock Gating

- The *Load* signal enables the clock signal to pass through if 1 and prevents the clock signal from passing through if 0.
- **Example:** For Positive Edge-Triggered or Negative Pulse Master-Slave Flip-flop:

  - What logic is needed for gating? *Gated Clock = Clock + Load*
  - What is the problem? *Clock Skew of gated clocks with respect to clock or each other*
Registers with Clock Gating
A more reliable way to selectively load a register:

- Run the clock continuously, and
- Selectively use a load control to change the register contents.

Example: 2-bit register with Load Control:

- For Load = 0, loads register contents (hold current values)
- For Load = 1, loads input values (load new values)

Hardware more complex than clock gating, but free of timing problems
Registers with Load-Controlled Feedback

In $Q_0$

In $Q_1$

In $Q_2$

In $Q_3$

Load

In$_0$

In$_1$

In$_2$

In$_3$

Clock
Registers with Load-Controlled Feedback

In0
In1
In2
In3
Load
Clock

Q0
Q1
Q2
Q3

Register with Parallel Load

Clock

Load

In0

D Q

Q0

Q1

Q2

Q3

In1

In2

In3
Register Transfer Operations

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

- Three basic components:
  - Set of Registers
  - Operations
  - Control of Operations

- Elementary Operations:
  - load, count, shift, add, bitwise "OR", etc.

- Elementary operations called microoperations
Register Notation

- **Letters and numbers** – denotes a register (ex. R2, PC, IR)
- **Parentheses ( )** – denotes a range of register bits
  
  Example: R1(1), PC(7:0), PC(L)
- **Arrow (←)** – denotes data transfer
  
  Example: R1 ← R2, PC(L) ← R0
- **Comma** – separates parallel operations
- **Brackets [ ]** – Specifies a memory address
  
  Example: R0 ← M[AR], R3 ← M[PC]
Conditional Transfer

- If \((K_1 = 1)\) then \((R2 \leftarrow R1)\) is shortened to \(K_1: (R2 \leftarrow R1)\)

*where* \(K_1\) *is a control variable specifying a conditional execution of the microoperation.*

![Diagram of conditional transfer with control variable (K1) and microoperations (R1 and R2).](image)
Microoperations

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

<table>
<thead>
<tr>
<th>Arithmetic operations</th>
<th>Logical operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Addition</td>
<td>⊕ Logical Exclusive OR</td>
</tr>
<tr>
<td>– Subtraction</td>
<td>– Not</td>
</tr>
<tr>
<td>* Multiplication</td>
<td>⊗ Logical AND</td>
</tr>
<tr>
<td>/ Division</td>
<td>⊕ Logical OR</td>
</tr>
</tbody>
</table>
Example Microoperations

- **Add** the content of R1 to the content of R2 and place the result in R1.
  \[ R1 \leftarrow R1 + R2 \]

- **Multiply** the content of R1 by the content of R6 and place the result in PC.
  \[ PC \leftarrow R1 \times R6 \]

- **Exclusive OR** the content of R1 with the content of R2 and place the result in R1.
  \[ R1 \leftarrow R1 \oplus R2 \]
Example Microoperations (Continued)

- Take the **1's Complement** of the contents of R2 and place it in the PC.

  \[ \text{PC} \leftarrow \overline{R2} \]

- On condition \( K_1 \text{ OR } K_2 \), the content of R1 is Logic **bitwise Ored** with the content of R3 and the result placed in R1.

  \[ (K1 + K2): \text{R1} \leftarrow \text{R1} \lor \text{R3} \]

- **NOTES:**
  
  ""' +"" (as in \( K_1 + K_2 \)) and means “OR.”
  In R1 \( \leftarrow \text{R1} + \text{R3}, + \) means “plus.”
Control Expressions

- The control expression for an operation appears to the left of the operation and is separated from it by a colon.

- Control expressions specify the logical condition for the operation to occur.

- Control expression values of:
  - Logic "1" -- the operation occurs.
  - Logic "0" -- the operation does not occur.

- Example:
  \[
  \bar{X} K_1 : R1 \leftarrow R2 + R1 \\
  X K_1 : R1 \leftarrow R2 + \bar{R1} + 1
  \]

  - Variable \( K_1 \) enables the add or subtract operation.
  - If \( X = 0 \), then \( \bar{X} = 1 \) so \( \bar{X} K_1 = 1 \), activating the addition of \( R1 \) and \( R2 \).
  - If \( X = 1 \), then \( X K_1 = 1 \), activating the addition of \( R2 \) and the two's complement of \( R1 \) (subtract).
Arithmetic Microoperations

\( \overline{X} K_1: \quad R1 \leftarrow R2 + R1 \)

\( X K_1: \quad R1 \leftarrow R2 + \overline{R1} + 1 \)
Arithmetic Microoperations

\[ \overline{X} K_1: \quad R1 \leftarrow R2 + R1 \]

\[ X K_1: \quad R1 \leftarrow R2 + \overline{R1} + 1 \]

X = \begin{cases} 
0 & \rightarrow R1 \leftarrow R2 + R1 \\
1 & \rightarrow R1 \leftarrow R2 - R1
\end{cases}

\[ X'K_1 + XK_1 = K_1 \]
Arithmetic Microoperations

\( \overline{X} K_1: \quad R1 \leftarrow R2 + R1 \)

\( X K_1: \quad R1 \leftarrow R2 + \overline{R1} + 1 \)

\[ X = \begin{cases} 
0 & \Rightarrow R1 \leftarrow R2 + R1 \\
1 & \Rightarrow R1 \leftarrow R2 - R1 
\end{cases} \]
Two's Complement Arithmetic

$(9)_{10} = (0\ 1001)_2 \Rightarrow (-9)_{10} = (1\ 0111)_2$

$(4)_{10} = (0\ 0100)_2 \Rightarrow (-4)_{10} = (1\ 1100)_2$

$9 - 4 = 9 + (-4) = (0\ 1001)_2 + (1\ 1100)_2$

\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 \\
\hline
0 & 0 & 1 & 0 \\
\end{array}
\Rightarrow (5)_{10}
\]
Two's Complement Arithmetic

\[(9)_{10} = (0\ 1001)_{2} \rightarrow (-9)_{10} = (1\ 0111)_{2}\]
\[(4)_{10} = (0\ 0100)_{2} \rightarrow (-4)_{10} = (1\ 1100)_{2}\]

\[4 - 9 = 4 + (-9) = (0\ 0100)_{2} + (1\ 0111)_{2}\]

\[
\begin{array}{c}
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 \\
\hline
1 & 1 & 0 & 1 & 1
\end{array}
\rightarrow (-5)_{10}\]
Two's Complement Arithmetic

\[(9)_{10} = (0\ 1001)_2 \rightarrow (-9)_{10} = (1\ 0111)_2\]
\[(8)_{10} = (0\ 1000)_2 \rightarrow (-8)_{10} = (1\ 1000)_2\]

\[9 + 8 = (0\ 1001)_2 + (0\ 1000)_2\]

\[0\ 1\ 0\ 0\ 0\]

\[0\ 1\ 0\ 0\ 1\]

\[+\]

\[0\ 1\ 0\ 0\ 0\]

\[1\ 0\ 0\ 0\ 1\ \rightarrow\ OVERFLOW\]

\[\frac{-2^4}{4} \leq \text{Number} \leq \frac{2^4}{4} - 1 \rightarrow -16 \leq \text{Number} \leq +15\]
Two's Complement Arithmetic

\[(9)_{10} = (0 1001)_2 \rightarrow (-9)_{10} = (1 0111)_2\]
\[(8)_{10} = (0 1000)_2 \rightarrow (-8)_{10} = (1 1000)_2\]

\[(-9) + (-8) = (1 0111)_2 + (1 1000)_2\]

\[\begin{array}{cccc}
1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 \\
+ & & & \\
1 & 1 & 0 & 0 & 0 \\
\hline
0 & 1 & 1 & 1 & 1 \\
\end{array}\]

\[\rightarrow \text{OVERFLOW}\]

\[\begin{array}{c}
\frac{-2^3}{4} \leq \text{Number} \leq \frac{2^3-1}{4} \\
\rightarrow -16 \leq \text{Number} \leq +15
\end{array}\]
Arithmetic Microoperations

From Table 7-3:

<table>
<thead>
<tr>
<th>Symbolic Designation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 ← R1 + R2</td>
<td>Addition</td>
</tr>
<tr>
<td>R0 ← R\overline{}1</td>
<td>Ones Complement</td>
</tr>
<tr>
<td>R0 ← R\overline{}1 + 1</td>
<td>Two's Complement</td>
</tr>
<tr>
<td>R0 ← R2 + R\overline{}1 + 1</td>
<td>R2 minus R1 (2's Comp)</td>
</tr>
<tr>
<td>R1 ← R1 + 1</td>
<td>Increment (count up)</td>
</tr>
<tr>
<td>R1 ← R1 – 1</td>
<td>Decrement (count down)</td>
</tr>
</tbody>
</table>

Note that any register may be specified for source 1, source 2, or destination.

These simple microoperations operate on the whole word.
Logical Microoperations

<table>
<thead>
<tr>
<th>Symbolic Designation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_0 \leftarrow \overline{R_1} )</td>
<td>Bitwise NOT</td>
</tr>
<tr>
<td>( R_0 \leftarrow R_1 \lor R_2 )</td>
<td>Bitwise OR (sets bits)</td>
</tr>
<tr>
<td>( R_0 \leftarrow R_1 \land R_2 )</td>
<td>Bitwise AND (clears bits)</td>
</tr>
<tr>
<td>( R_0 \leftarrow R_1 \oplus R_2 )</td>
<td>Bitwise EXOR (complements bits)</td>
</tr>
</tbody>
</table>
Logical Microoperations

Example:

- Let $R_1 = 10101010$, and $R_2 = 11110000$
- Then after the operation, $R_0$ becomes:

<table>
<thead>
<tr>
<th>R0</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010101</td>
<td>$R_0 \leftarrow \overline{R_1}$</td>
</tr>
<tr>
<td>11111010</td>
<td>$R_0 \leftarrow R_1 \lor R_2$</td>
</tr>
<tr>
<td>10100000</td>
<td>$R_0 \leftarrow R_1 \land R_2$</td>
</tr>
<tr>
<td>01011010</td>
<td>$R_0 \leftarrow R_1 \oplus R_2$</td>
</tr>
</tbody>
</table>
Shift Microoperations

- From Table 7-5:
- Let R2 = 11001001
- Then after the operation, R1 becomes:

<table>
<thead>
<tr>
<th>Symbolic Designation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 ← sl R2</td>
<td>Shift Left</td>
</tr>
<tr>
<td>R1 ← sr R2</td>
<td>Shift Right</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010010</td>
<td>R1 ← sl R2</td>
</tr>
<tr>
<td>01100100</td>
<td>R1 ← sr R2</td>
</tr>
</tbody>
</table>

- 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) (see Chapter 10).
Shift Registers

4-Bit Shift Register

Data input

Serial input

Serial output

$SI \rightarrow C \rightarrow C \rightarrow C \rightarrow C \rightarrow SO$

$CLK$

Data input: 1000

1st data bit = 0 (LSB)
After CLK1

2nd data bit = 1
After CLK2

3rd data bit = 0
After CLK3

4th data bit = 1 (MSB)
After CLK4, the 4-bit number is completely stored in register.
Bidirectional Shift Register with Parallel Load

<table>
<thead>
<tr>
<th>S_1</th>
<th>S_0</th>
<th>Register Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No Change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Shift Left</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Shift Right</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Parallel Load</td>
</tr>
</tbody>
</table>

Register Operation

<table>
<thead>
<tr>
<th></th>
<th>S_1</th>
<th>S_0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN_i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SR ➔ SL

4 × 1 MUX

Register

Clock

Logic and Computer Design Fundamentals, 4e
PowerPoint® Slides
© 2008 Pearson Education, Inc.
Bidirectional Shift Register with Parallel Load

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_0$</th>
<th>Register Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No Change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Shift Left</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Shift Right</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Parallel Load</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IN0</th>
<th>SR Input</th>
<th>Q0</th>
<th>4 × 1 MUX</th>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td></td>
<td>1</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>Q3</td>
<td>2</td>
<td></td>
<td>2</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>Q4</td>
<td>3</td>
<td></td>
<td>3</td>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IN1</th>
<th>Q1</th>
<th>4 × 1 MUX</th>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN2</td>
<td>Q2</td>
<td>4 × 1 MUX</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>IN3</td>
<td>Q3</td>
<td>4 × 1 MUX</td>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

SL Input

Clock
Bidirectional Shift Register with Parallel Load

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_0$</th>
<th>Register Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No Change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Shift Left</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Shift Right</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Parallel Load</td>
</tr>
</tbody>
</table>

SL Input → Bidirectional Shift Register with Parallel Load → SR Input

$Q_3$, $Q_2$, $Q_1$, $Q_0$
Bidirectional Shift Register with Parallel Load

<table>
<thead>
<tr>
<th>S₁</th>
<th>S₀</th>
<th>Register Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No Change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Shift Left</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Shift Right</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Parallel Load</td>
</tr>
</tbody>
</table>

Case 1: Shift Right

Case 2: Shift Left

Case 3: Parallel Load
### Shift Register with Parallel Load

<table>
<thead>
<tr>
<th>Shift</th>
<th>Load</th>
<th>Register Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No Change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Load Parallel Data</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>Shift Left</td>
</tr>
</tbody>
</table>

#### Shift Load

- **In₀**: Serial Input
- **Q₀**: Output

#### Operation

- **In₁**: Serial Input
- **Q₁**: Output

- **In₂**: Serial Input
- **Q₂**: Output

- **In₃**: Serial Input
- **Q₃**: Output

- **Clock**: Clock Input
Register Transfer Structures

- **Multiplexer-Based Transfers** - Multiple inputs are selected by a multiplexer dedicated to the register

- **Bus-Based Transfers** - Multiple inputs are selected by a shared multiplexer driving a bus that feeds inputs to multiple registers

- **Three-State Bus** - Multiple inputs are selected by 3-state drivers with outputs connected to a bus that feeds multiple registers

- **Other Transfer Structures** - Use multiple multiplexers, multiple buses, and combinations of all the above
Multiplexer-Based Transfers

- Multiplexers connected to register inputs produce flexible transfer structures (Note: Clocks are omitted for clarity)

**Example:**

- The transfers are:

  \[ K_1: \text{R0} \leftarrow \text{R1} \]

  \[ K_1 \cdot K_2: \text{R0} \leftarrow \text{R2} \]

\[ K_1 + K_1 \cdot K_2 = K_1 + K_2 \]
Multiplexer-Based Transfer
Example: Two 4-bit registers

(b) Detailed logic
Bus Transfers

Register A
Register B
Register C
Register D

Bus lines

Register D
3 2 1 0
D3 D2 D1 D0

D3 C3 B3 A3

3 2 1 0
MUX3

S0 S1

Register C
3 2 1 0
C3 C2 C1 C0

D2 C2 B2 A2

3 2 1 0
MUX2

S0 S1

Register B
3 2 1 0
B3 B2 B1 B0

D1 C1 B1 A1

3 2 1 0
MUX1

S0 S1

Register A
3 2 1 0
A3 A2 A1 A0

D0 C0 B0 A0

3 2 1 0
MUX0

S0 S1

4-Line Common Bus
Bus Transfer

Example:
For register R0 to R3 in a 4 bit system

<table>
<thead>
<tr>
<th>S1</th>
<th>S0</th>
<th>Register selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

Chapter 7 - Part 1  43
Bus Transfer

- For register R0 to R63 in a 16 bit system:
  - What is the MUX size we use? $64 \times 1$ MUX
  - How many MUX we need? 16 MUXs
  - How many select bit? 6 bits
Tri-State Buffers

- **Tri-state buffer gate:**
  - When control input = 1: The output is enabled (output $Y = \text{input } A$)
  - When control input = 0: The output is disabled (output $Y = \text{high-impedance}$)

```plaintext
If  C=1, Output Y = A
If  C=0, Output = High-impedance
```
Bus system with tri-state buffer

<table>
<thead>
<tr>
<th>S1</th>
<th>S0</th>
<th>Register selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

Select input

Enable input

2*4 decoder

A0

B0

C0

D0
Multiplexer Approach

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

- Load enable by OR of control signals $K_0, K_1, \ldots, K_{n-1}$
  - Assumes no load for 00...0

- Use:
  - Encoder + Multiplexer (shown) or
  - $n \times 2$ AND-OR

  to select sources and/or transfer functions
Multiplexer and Bus-Based Transfers for Multiple Registers

- Multiplexer dedicated to each register
- Shared transfer paths for registers
  - A shared transfer object is called a bus (Plural: buses)
- Bus implementation using:
  - multiplexers
  - three-state nodes and drivers
- In most cases, the number of bits is the length of the receiving register
Dedicated MUX-Based Transfers

- Multiplexer connected to each register input produces a very flexible transfer structure =>
- Characterize the simultaneous transfers possible with this structure.
Multiplexer Bus

- A single bus driven by a multiplexer lowers cost, but limits the available transfers.
- Characterize the simultaneous transfers possible with this structure.
- Characterize the cost savings compared to dedicated multiplexers.
Three-State Bus

- The 3-input MUX can be replaced by a 3-state node (bus) and 3-state buffers.
- Cost is further reduced, but transfers are limited.
- Characterize the simultaneous transfers possible with this structure.
- Characterize the cost savings and compare.
- Other advantages?
Shift Registers

- Shift Registers move data laterally within the register toward its MSB or LSB position.
- In the simplest case, the shift register is simply a set of D flip-flops connected in a row like this:

  ![Shift Register Diagram]

  - Data input, In, is called a *serial input* or the *shift right input*.
  - Data output, Out, is often called the *serial output*.
  - The vector (A, B, C, Out) is called the *parallel output*. 
Shift Registers (continued)

- The behavior of the serial shift register is given in the listing on the lower right.

- T0 is the register state just before the first clock pulse occurs.

- T1 is after the first pulse and before the second.

- Initially unknown states are denoted by “?”. 

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
CP & In & A & B & C & Out \\
\hline
T1 & 1 & 0 & ? & ? & ? \\
T2 & 1 & 1 & 0 & ? & ? \\
T3 & 0 & 1 & 1 & 0 & ? \\
T4 & 1 & 0 & 1 & 1 & 0 \\
T5 & 1 & 1 & 0 & 1 & 1 \\
T6 & 1 & 1 & 1 & 0 & 1 \\
\hline
\end{array}
\]
Parallel Load Shift Registers

- By adding a mux between each shift register stage, data can be shifted or loaded.
- If SHIFT is low, A and B are replaced by the data on \( D_A \) and \( D_B \) lines, else data shifts right on each clock.
- By adding more bits, we can make \( n \)-bit parallel load shift registers.

Note:
- A parallel load shift register with an added “hold” operation that stores data unchanged is given in Figure 7-10 of the text.
Shift Registers with Additional Functions

- By placing a 4-input multiplexer in front of each D flip-flop in a shift register, we can implement a circuit with shifts right, shifts left, parallel load, hold.

- Shift registers can also be designed to shift more than a single bit position right or left

- Shift registers can be designed to shift a variable number of bit positions specified by a variable called a shift amount.
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
  - **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

- **Serial microoperations**
  - **Example 1**: Addition
  - **Example 2**: Error-Correction for CDs
Parallel-to-Serial / Serial-to-Parallel

Transmitter

Serial-out

Receiver

PISO = Parallel-in, serial-out

TxD = Transmit Data out

SIPO = Serial-in, parallel-out

RxD = Receive Data in

Transmit clock (TxC)

Receive clock (Rx'C = N × Tx'C)

÷ N Counter

Parallel-out

PISO

Parallel-in

msb

lsb

William Stallings “Data and Computer Communications”
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.

Shifting a binary number left is equivalent to multiplying by 2.

Shifting a binary number right is equivalent to dividing by 2.
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.
Terms of Use

- All (or portions) of this material © 2008 by Pearson Education, Inc.
- Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.
- These materials or adaptations thereof are not to be sold or otherwise offered for consideration.
- This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.