Intro to Digital Design

Encoders, Decoders, Registers, Counters

Instructor: Justin Hsia

Teaching Assistants:
Emilio Alcantara  Eujean Lee
Naoto Uemura  Pedro Amarante
Wen Li
Lab 7 – Useful Components
   - Modifying Lab 6 game to implement common circuit elements
   - Build a tunable computer opponent!

Quiz 2 is next week in lecture
   - Last 30 minutes (+ 5 min buffer), 10% of your course grade
   - On Lectures 4-5: Sequential Logic, Timing, FSMs, and Verilog
   - Past Quiz 2 (+ solutions) on website: Course Info → Quizzes
Practice Problem

❖ For an \( n \)-bit ripple-carry adder, what is the shortest and longest time that output \( S \) changes after each clock cycle?

- A, B, \( c_0 \) from registers (show up at \( t_{C2Q} \)); \( S \) goes directly to a register input.
- Assume all gates have a delay of 1 ns; use variables for all other timing values.

\[
\begin{align*}
\text{shortest (} s_0 \text{): } & t_{C2Q} + 1 \\
\text{longest (} s_n \text{): } & t_{C2Q} + 2n
\end{align*}
\]
Outline

❖ Circuit Routing Elements
❖ Register Revisited
Standard Circuit Routing Elements

- **Multiplexor (mux)**
  - Pass one of \(N\) inputs to single output

- **Simple Encoder**
  - One of \(N\) inputs is active and output tells you which one (in binary)

- **1-of-\(N\) Binary Decoder**
  - Interpret binary input to assert one of \(N\) output wires

- **Demultiplexer (demux)**
  - Pass single input onto one of \(N\) outputs
Encoder

- A device or circuit that converts information from one format or code to another
  - **Examples**: decimal to binary, keyboard press to character, rotary encoder for odometer, analog-to-digital converter

- A **simple encoder** is a one-hot to binary converter
  - One-hot means at most only one input line (out of $m \leq 2^n$) will be high
  - Output is the binary representation ($n$ bits wide) of the asserted line’s bit numbering or “address”
  - Referred to as an $m:n$ encoder (read as “$m$-to-$n$”)
Simple Encoder Implementation

❖ 4:2 Encoder

![Diagram of 4:2 Encoder]

<table>
<thead>
<tr>
<th>D_3</th>
<th>D_2</th>
<th>D_1</th>
<th>D_0</th>
<th>A_1</th>
<th>A_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

❖ Two issues:

1) What if multiple inputs are hot? 
   \[ A_1 = D_2 + D_3 \]
   \[ A_0 = D_1 + D_3 \]
   \[ D_2 \text{ and } D_1 \text{ hot acts like } D_3 \text{ was hot} \]

2) What if no inputs are hot?
   \[ D_0 \text{ acts like } D_0 \text{ was hot when it wasn't} \]
Priority Encoder

1) Use *priorities* to resolve the problem of multiple active input lines
   - **Example**: Highest ID active is given priority (“wins”)

2) Add an output to identify when at least 1 input active

<table>
<thead>
<tr>
<th>D_3 D_2 D_1 D_0</th>
<th>A_1</th>
<th>A_0</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1 X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 X X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 X X X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Implement with OR gate
Encoder Examples

- Navigation (Compass) Encoder

http://www.electronics-tutorials.ws/combination/comb_4.html
Encoder Examples

- Analog-to-Digital Converter (ADC)

Decoder

- A device or circuit that converts or interprets information from an encoded format
  - Examples: binary to decimal, CPU instruction decoder, video decoder (analog to digital)

- A binary decoder is a binary to one-hot converter
  - $n$ input bits serve as bit number or “address” specifier
  - Only corresponding output out of $m \leq 2^n$ will be asserted
  - Referred to as an $n:m$ decoder (read as “$n$-to-$m$”)
1-of-N Binary Decoder Implementation

- **2:4 Decoder**

  ![2:4 Decoder Diagram]

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_0$</th>
<th>$D_3$</th>
<th>$D_2$</th>
<th>$D_1$</th>
<th>$D_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Issue:**
  - What do we do if we want nothing to happen?
Enabled Decoder

- Only have active output when Enable signal is high

<table>
<thead>
<tr>
<th>Enable</th>
<th>$S_1$</th>
<th>$S_0$</th>
<th>$D_3$</th>
<th>$D_2$</th>
<th>$D_1$</th>
<th>$D_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Diagram of Enabled Decoder with inputs $S_1$, $S_0$, and Enable, and output bits $D_3$, $D_2$, $D_1$, $D_0$. The diagram shows the logical connections for each input combination.
Enabled Decoder in Verilog

```verilog
module enDecoder2_4 (out, in, enable);
    output logic [3:0] out;
    input logic [1:0] in;
    input logic enable;

    always_comb begin
        if (enable)
            case (in)
                2'b00: out = 4'b0001;
                2'b01: out = 4'b0010;
                2'b10: out = 4'b0100;
                2'b11: out = 4'b1000;
            endcase
        else
            out = 4'b0000;
    end

endmodule  // enDecoder2_4
```
Decoder Examples: Demultiplexer

- **1-bit 1-to-2 DEMUX:**

- **Truth Table:**

<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>D₀</th>
<th>D₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- More generally, AND dᵢ output from decoder with *every* input bit that is wired to DEMUX output Dⱼ.
Decoder Examples

- Binary to 7-seg display
  - You’ve already made this in this class!

http://www.learnabout-electronics.org/Digital/dig44.php
Decoder Examples

- MIPS instruction decoder
  - Upper 6 bits of a 32-bit MIPS instruction
  - Part of the control portion of a CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Name</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>Add Imm.</td>
<td>001000</td>
</tr>
<tr>
<td>addiu</td>
<td>Add Imm. Unsigned</td>
<td>001001</td>
</tr>
<tr>
<td>andi</td>
<td>And Imm.</td>
<td>001100</td>
</tr>
<tr>
<td>beq</td>
<td>Branch On Equal</td>
<td>000100</td>
</tr>
<tr>
<td>bne</td>
<td>Branch On Not Equal</td>
<td>000101</td>
</tr>
<tr>
<td>j</td>
<td>Jump</td>
<td>000010</td>
</tr>
<tr>
<td>jal</td>
<td>Jump and Link</td>
<td>000011</td>
</tr>
<tr>
<td>lui</td>
<td>Load Upper Imm.</td>
<td>001111</td>
</tr>
<tr>
<td>lw</td>
<td>Load Word</td>
<td>100011</td>
</tr>
<tr>
<td>ori</td>
<td>Or Imm.</td>
<td>001101</td>
</tr>
<tr>
<td>sb</td>
<td>Store Byte</td>
<td>101000</td>
</tr>
<tr>
<td>sw</td>
<td>Store Word</td>
<td>101011</td>
</tr>
</tbody>
</table>
Technology

Break
Outline

❖ Circuit Routing Elements
❖ Register Revisited
State Element Revisited: Register

- \( n \) instances of flip-flops together
  - One for every bit in input/output bus width

- Desired behaviors (synchronous)
  - Output \( Q \) resets to zero when \textbf{Reset} signal is high
  - Hold current value unless \textbf{Enable} signal is high
Controlled Register

- Here using shorthand C (clock), R (reset), E (enable)

<table>
<thead>
<tr>
<th>Reset</th>
<th>Enable</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>q =</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>q =</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>q =</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>q =</td>
</tr>
</tbody>
</table>

Diagram of D Flip-Flop with control inputs R, E, and C.
Shift Register

 fich Register that shifts the binary values in one or both directions

❖ Where do we get the input from?

- External input (e.g., delay a signal)
- Function of current bits (e.g., linear-feedback shift register)

❖ What is the output data of interest?

- Last (oldest) bit of sequence
- Entire set of current bits
Linear Feedback Shift Register (LFSR)

- Shift register input is a logical combination of the current state bits:

- **Example:** pseudo-random number generator
  - Input: no external input!
  - Output: all state bits together as a bus
Simple LFSR in Verilog

❖ How to implement this in Verilog?

```verilog
module LFSR #(
  parameter WIDTH=3
) (Q, clk);

output logic [WIDTH-1:0] Q;  // present state
input logic clk;             // clock input

always_ff @(posedge clk)
  Q <= {Q[WIDTH-2:0], ~(Q[WIDTH-1] ^ Q[WIDTH-2])};

endmodule
```
Counters

❖ A register that goes through a specific state sequence
   ▪ More general than what you typically think of as a “counter”

❖ Examples:
   ▪ \textit{n-bit Binary Counter}: counts from 0 to \(2^n-1\) in binary
   ▪ \textit{Up Counter}: Binary value increases by 1
   ▪ \textit{Down Counter}: Binary value decreases by 1

❖ 3-bit binary up counter state diagram:
LFSR Revisited

❖ A LFSR is also a counter!
  ▪ The logical combination determines the state sequence

❖ State diagram:

"pseudo-random" sequence of states determined by:
1) # of state bits
2) gate(s) used in feedback
3) which bits are connected to gate(s)
## Binary Up-Counter Implementation

### Truth Table

<table>
<thead>
<tr>
<th>$P_2$</th>
<th>$P_1$</th>
<th>$P_0$</th>
<th>$N_2$</th>
<th>$N_1$</th>
<th>$N_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Circuit Diagram

- $Dff_2$ and $Dff_1$ are triggered by $CLK$.
- $Dff_0$ is triggered by $(P_2 \oplus P_0)$.

### Calculation

- $N_0 = \overline{P_0}$
- $N_1 = P_0 \bar{P}_2 + \bar{P}_0 \bar{P}_1$  
  $= P_0 \oplus P_1$
- $N_2 = P_2 P_0 + \overline{P_2} P_1 + \overline{P_2} P_0$  
  $= P_2 (P_1 \oplus P_0) + \bar{P}_2 (P_2 P_0)$
  $= P_2 (P_1 \oplus P_0) = P_2 \oplus (P_0)$
Complex Binary Counter

<table>
<thead>
<tr>
<th>Load</th>
<th>Count</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Old Q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Up count</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Parallel load (D)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Reset</td>
</tr>
</tbody>
</table>

\[ N_0 = \overline{P_0} \]
\[ N_1 = \overline{P_1}P_0 + \overline{P_0}P_1 = P_0 \oplus P_1 \]
\[ N_2 = P_2\overline{P_0} + P_2\overline{P_1} + P_2P_1P_0 = P_2(P_0 + \overline{P_1}) + P_2P_1P_0 = P_2(P_0 + \overline{P_1} + P_1P_0) = P_2 \oplus (P_0P_1) \]
Up Counter in Verilog (no load)

module upcounter #(parameter WIDTH=8)
  (out, enable, reset, clk);

  output logic [WIDTH-1:0] out;
  input logic enable, reset, clk;

  always_ff @(posedge clk) begin
    if (reset)
      out <= 0;
    else if (enable)
      out <= out + 1;
  end
endmodule  // upcounter