Lecture 15

◆ Logistics
- HW4 is due today
- HW5 posted today
- Exam questions: to me
- Class feedback

◆ Last lecture
- Adders

◆ Today
- More on Adder timing issues (hard!)
- Summary of Combinational Logic
- Introduction to Sequential Logic
  - The basic concepts
  - An example

---

Binary full adder

◆ 1-bit full adder
- Computes sum, carry-out
  - Carry-in allows cascaded adders
- Sum = Cin xor A xor B
- Cout = ACin + BCin + AB

![Binary full adder diagram](image)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![Binary full adder truth table](image)
Problem: Ripple-carry delay

- Carry propagation limits adder speed

\[ \begin{array}{c}
\text{A} \quad \text{B} \\
\text{Cin} \\
\text{Cout} \end{array} \]

\[ \begin{array}{c}
\text{XOR} \\
\text{AND} \\
\text{OR} \\
\text{XOR} \end{array} \]

Cout takes two gate delays
Cin arrives late

\[ \begin{aligned}
0111 & \quad A \\
+ & \quad 0001 \quad B \\
\hline
1000 & \quad \text{Sum} \\
\end{aligned} \]

Speeding up the adder

- Need to find a way to “predict” Cout for all bits
- Without knowing what Cin is
  
  \[ \begin{aligned}
  \text{Cout is always 0} & \quad + \quad 0 \\
  \text{Predict Cout} & \\
  \text{Cout is 0 if Cin is 0} & \quad 0 \\
  \text{Cout is 1 if Cin is 1} & \quad + \quad 1 \\
  \text{Call this PROPAGATE} & \\
  \end{aligned} \]

- Let’s try all cases:
  
  - A = 0, B = 0 but not sure of Cin
  - A = 0, B = 1 but not sure of Cin
  - A = 1, B = 0 but not sure of Cin
  - A = 1, B = 1 but not sure of Cin

\[ \begin{aligned}
\text{Call this GENERATE} & \quad + \quad 1 \\
\text{Predict Cout} & \\
\end{aligned} \]
Solution: Create a carry lookahead logic

Getting Pi and Gi

- **Carry generate**: \( G_i = A_iB_i \) for i-th bit
  - Generate Cout when \( A = B = 1 \)

- **Carry propagate**: \( P_i = A_i \oplus B_i \) for i-th bit
  - Propagate Cin to Cout when \( (A \oplus B) = 1 \)

- So, Cout = G + PCin

\[ C_{i+1} = G_i + P_iC_i \]

One Solution: Carry lookahead logic

- Get Pi (propagate) and Gi (generate)

\[ C_i = G_i + P_iC_i \]

\[ C_2 = G_2 + P_1C_1 = G_1 + P_1G_0 + P_1P_2C_0 \]

\[ C_3 = G_3 + P_2C_2 \]

\[ C_4 = G_4 + P_3C_3 \]

\[ C_5 = G_5 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \]

\[ C_6 = G_6 + P_3P_2G_2 + P_3P_2P_1G_1 + P_3P_2P_1P_0C_0 \]
We've finished combinational logic...

- Negative numbers in binary
- Truth tables
- Basic logic gates
- Schematic diagrams
- Minterm and maxterm expansions (canonical, minimized)
- de Morgan's theorem
- AND/OR to NAND/NOR logic conversion
- K-maps, logic minimization, don't cares
- Multiplexers/demultiplexers
- PLAs/PALs
- ROMs
- Multi-level logics
- Timing diagrams
- Hazards
- Adders

We had no way to store memory: When the input changed, the output changed

Next: Sequential logic can store memory…

Sequential Logic (next 5 weeks!)

- We learn the details
  - Latches, flip-flops, registers (storage)
  - Shift registers, counters (we can count now!)
  - State machines (when we can store, we have states)
  - Moore and Mealy machines (types of state machines)
  - Timing and timing diagrams
    - timing more important than combinational logic
  - Synchronous and asynchronous inputs
    - Metastability (problem!)
The “WHY” slide

- Learning sequential logic
  - Having the ability to hold memory is important. If you couldn’t use your prior knowledge stored in the memory, you wouldn’t be very smart (and same goes for a computer).

Sequential versus combinational

Apply fixed inputs A, B
When the clock ticks, the output becomes available
Observe C
Wait for another clock tick
Observe C again

Combinational: C will stay the same
Sequential: C may be different
Sequential versus combinational

- **Combinational systems are memoryless**
  - Outputs depend only on the present inputs

- **Sequential systems have memory**
  - Outputs depend on the present and the previous inputs

Synchronous sequential systems

- **Memory holds a system’s state**
  - Changes in state occur at specific times
  - A periodic signal times or clocks the state changes
  - The clock period is the time between state changes

- State changes occur at rising edge of clock

- Duty cycle = pulsewidth / period (here it is 50%)
Steady-state abstraction

- Outputs retain their *settled values*
  - The clock period must be long enough for all voltages to settle to a *steady state* before the next state change.

![Diagram of a steady-state abstraction with inputs A and B, output C, and clock input.]

What did I just say about sequential logic?

- Has clock
  - *Synchronous* = clocked
  - Exception: *Asynchronous*

- Has state
  - *State* = memory

- Employs feedback

- Assumes *steady-state signals*
  - Signals are valid after they have settled
  - State elements hold their settled output values
Example: A sequential system

- Door combination lock
  - Enter three numbers in sequence and the door opens
  - When one number is entered, press ‘enter’
  - If there is an error the lock must be reset
  - After the door opens the lock must be reset
  - Inputs: Sequence of numbers, reset, enter
  - Outputs: Door open/close
  - Memory: Must remember the combination

We will go through the motion of designing a real system

We will teach details of “how” to do these steps in the next few weeks

Understand the problem

- Consider I/O and unknowns
  - How many bits per input?
  - How many inputs in sequence?
  - How do we know a new input is entered?
  - How do we represent the system states?
Implement using sequential logic

- **Behavior**
  - Clock tells us when to look at inputs
    - After inputs have settled
  - Sequential: Enter sequence of numbers
  - Sequential: Remember if error occurred

- **A diagram may be helpful**
  - Assume synchronous inputs
  - State sequence
    - Enter 3 numbers serially
    - Remember if error occurred
  - All states have outputs
    - Lock open or closed

A diagram (called finite-state diagram)

- **States:** 5
  - Each state has outputs
- **Outputs:** open/closed
- **Inputs:** reset, new, results of comparisons
  - Assume synchronous inputs

We use state diagrams to represent sequential logic

System transitions between finite numbers of states
Separate data path and control

- **Data path**
  - Stores combination
  - Compares inputs with combination

- **Control**
  - Finite state-machine controller
  - Control for data path
  - State changes clocked

---

Refine diagram; generate state table

- Refine state diagram to include internal structure

- Generate state table

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>S1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>S3</td>
<td>closed</td>
<td>closed</td>
</tr>
<tr>
<td>OPEN</td>
<td>-</td>
<td>open</td>
</tr>
</tbody>
</table>
Encode state table

- **State can be**: S1, S2, S3, OPEN, or ERR
  - Need at least 3 bits to encode: 000, 001, 010, 011, 100
  - Can use 5 bits: 00001, 00010, 00100, 01000, 10000
  - Choose 4 bits: 0001, 0010, 0100, 1000

- **Output to mux can be**: C1, C2, or C3
  - Need 2 or 3 bits to encode
  - Choose 3 bits: 001, 010, 100

- **Output open/closed can be**: Open or closed
  - Need 1 or 2 bits to encode
  - Choose 1 bit: 1, 0

Encode state table (con’t)

- **Good encoding choice!**
  - Mux control is identical to last 3 state bits
  - Open/closed is identical to first state bit
  - Output encoding ⇒ the outputs and state bits are the same

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0001</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>0000</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>0010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0100</td>
<td>1000</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing the controller

- We will learn how to design the controller given the encoded state-transition table.

![Diagram of controller](image)

Designing the datapath

- Four multiplexers
  - 2-input ANDs and 3-input OR
- Four single-bit comparators
  - 2-input XNORs
- 4-input AND

![Diagram of datapath](image)
Where did we use **memory**?

- **Memory**: Stored combination, state (errors or successes in past inputs)

Where did we use **feedback**?

- **Feedback**: Comparator output ("equal" signal)
Where did we use clock?

- **Clock** synchronizes the inputs
  - Accept inputs when clock goes high

- **Controller is clocked**
  - Mux-control and open/closed signals change on the clock edge