Digital Logic Design

Administrative

- My office hours have moved to Friday 10-12.
- First lab assignment (MIPS assembly) starts Today and is due by midnight in one week (9-Nov).
- Quick evaluation.
- The assigned reading is fair game for the exam.

Topics

- Combinational logic and gates
- Logic → Truth tables
- Truth tables → gates with Karnaugh maps
- Multiplexors, Encoders, and Decoders
- Finite State Machines
- Memories and Latches

Combinational Logic

- Basic Operations
  - NOT \( !A \)
  - AND \( A \cdot B \)
  - OR \( A + B \)
  - NOR \( ! (A + B) = NOT \ (A \ OR \ B) \)
  - NAND \( ! (A \cdot B) = NOT \ (A \ AND \ B) \)
  - XOR \( A ^ B \)
  - XNOR \( !(A ^ B) = NOT \ (A \ XOR \ B) \)

Basic Gate Truth Tables

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Out</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAND</th>
<th>NOR</th>
<th>XNOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Out</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Logic Equations

- \( A + 1 = 1 \)
- \( A \cdot 1 = A \)
- \( A+0 = A \)
- \( A \cdot 0 = 0 \)
De Morgan’s Law

- !(A•B) = !A+!B

\[
\begin{array}{c|c|c|c|c}
A & B & A•B & !(A•B) & !A+!B \\
0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

Another One For You To Try

- !(A+B) = !A•!B

\[
\begin{array}{c|c|c|c|c}
A & B & A+B & !(A+B) & !A•!B \\
0 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

A More Complex Example

- Show that: A•(B+C) = A•B + A•C

\[
\begin{array}{c|c|c|c|c|c}
A & B & C & B+C & A•(B+C) & A•B & A•C \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

Going the Other Way…

- We just did equations \(\rightarrow\) truth tables, but how do do truth tables \(\rightarrow\) equations?
- Why do we want to do this?
  - Does this look familiar? (Think back to the first lecture)

\[
\begin{array}{c|c|c|c}
A & B & C \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

How Do We Build a Half-Adder? (S)

- What is a half-adder?
  - An adder with no carry in. \((A+B = \text{Sum} + \text{Carry})\)
  - Why? Simpler to start with than a full adder
- Start with S
  - What logic function is this? \((\text{Inputs are } A \text{ and } B)\)
  - Find all the places where \(S = 1\), then look at the inputs.
  - \(S = A \oplus B\)

\[
\begin{array}{c|c|c|c|c|c}
A & B & S & & & \text{XOR} \\
0 & 0 & 0 & & & 0 \\
0 & 1 & 1 & & & 0 \\
1 & 0 & 1 & & & 0 \\
1 & 1 & 0 & & & 0 \\
\end{array}
\]

This pattern is XOR.

How Do We Build a Half-Adder (C)

- Now Look at C
  - What logic function is this? \((\text{Inputs are } A \text{ and } B)\)
  - Find all the places where \(C = 1\), then look at the inputs.
  - \(C = A \land B\)

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
A & B & C & & & & \text{AND} \\
0 & 0 & 0 & & & & 0 \\
0 & 1 & 1 & & & & 0 \\
1 & 0 & 0 & & & & 0 \\
1 & 1 & 1 & & & & 0 \\
\end{array}
\]

This pattern is AND.
How Do We Build a Half-Adder?

• \( S = A \text{ AND } B \)
• \( C = A \text{ XOR } B \)

\[
\begin{array}{c|c|c|c}
A & B & S & C \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 \\
\end{array}
\]

Karnaugh Maps

Sum

\[
\begin{array}{c|c|c|c}
A \backslash B & 0 & 1 \\
0 & 0 & 0 \\
1 & 1 & 0 \\
\end{array}
\]

Carry

\[
\begin{array}{c|c|c|c}
A \backslash B & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

• Write the inputs across the top/bottom
• Circle the largest power-of-2 block of 1s
• And you get the equation!

More Karnaugh Maps

More Than Two Variables

• Order variables such that only 1 changes in each row/column (Grey coding)
• Groups may overlap

\[
\begin{array}{c|c|c|c|c}
A \backslash B & 00 & 01 & 11 & 10 \\
00 & 0 & 0 & 0 & 0 \\
01 & 0 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[B \cdot D \quad !A \cdot B \quad !C \cdot B \]
\[B \cdot (D + !A + !C)\]

Look more carefully:

• Recognize the pattern?
• \( C_{in} \) is true if an odd number of \( A, B, \) and \( C_{in} \) are true.
• This is XOR

Sum = \( C_{in} \text{ XOR (A XOR B)} \)

Full Adder

• \( A + B + \text{ Carry In} = \text{ Sum} + \text{ Carry Out} \)

Karnaugh Map for Full Adder Sum

\[
\begin{array}{c|c|c|c|c|c}
\text{Cin} \backslash AB & 00 & 01 & 11 & 10 \\
00 & 0 & 0 & 0 & 0 \\
01 & 0 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[C_{in} \cdot A \cdot B + \!C_{in} \cdot A \cdot B + C_{in} \cdot A \cdot B + \!C_{in} \cdot A \cdot B\]

Look more carefully:

• Recognize the pattern?
• \( C_{in} \) is true if an odd number of \( A, B, \) and \( C_{in} \) are true.
• This is XOR

Sum = \( C_{in} \text{ XOR (A XOR B)} \)
**You Try: Full Adder Carry Out**

<table>
<thead>
<tr>
<th>Cin</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Karnaugh Map Summary**

- Allows you to truth table → gates
  - Important because we describe functions in terms of their truth tables
  - Remember to circle the largest groups (power-of-two)
  - Groups may overlap
  - Inputs should only change one bit at a time (e.g., 00, 01, 11, 10, not 00, 01, 10, 11)
  - If you have outputs you don’t care about (x’s) you can circle them too
- However, computer do this better than humans
  - All modern tools (including LogicSim) will convert truth tables to logic gates for you

**Important Logic Blocks**

- **Multiplexers (MUXes)**
  - Choose from multiple inputs
  - Essential for routing signals around
- **Demultiplexers (DEMUXes)**
  - Opposite of MUXes
  - Need to define what you get in the non-selected case
- **Same symbol for busses (multiple values)**
  - E.g., in can be an 8-bit value (8 wires) and each out will also be 8-bit values (8 wires)

**Important Logic Blocks**

- **Decoders**
  - Convert from binary encoding to 1-hot
  - Binary 10 → 0100 in 1-hot
- **Encoders**
  - Convert form 1-hot to binary
  - 1-hot 00000010 → binary 010

**Example: Building a Memory**

- 1-hot row enables
- Array of SRAM Cells (Each box stores 1 bit)

**Example: Building a Memory**

- 1-hot row enables
- Array of SRAM Cells (Each box stores 1 bit)
Example: Building a Memory

Memories

- Memories store **state**
  Information that can change over time.

- Combinational Logic is **state-less**
  The output changes “instantly” with the inputs.

- You need both to make anything useful.

Example: Building a Counter

- **Next_value (3bits) = current_value (3bits) + 1**
- On **clock** the output gets the previous input

- **Next_value = current_value + 1**
- On **clock** the output gets the previous input
Q: What limits the speed of the clock?

- When the clock edge rises (0→1)
- The \( \text{next} \) value is stored as the current value
- The new \( \text{current} \) value then goes through the adder to make a new \( \text{next} \) value
- Everything takes some time

Finite State Machines

- Choose what to do next (next state and outputs) based on what are are doing now (current state and inputs)
- Allows things to happen over time based on previous inputs
- Pure logic "instantly" changes its outputs based on the input

Example: Building a Counter

- Current value
- Next value
- Clock input

Finite State Machines

Example FSM: Missile Launcher

- 3 States: IDLE, WAIT_CODE, CODE_OK
- Change between states based on Input buttons A and B
- Set the LAUNCH output to true when
  - the A button is pressed followed by
  - the B button
- States
  - IDLE
    - \( \text{next} \) state is IDLE if \( \text{LAUNCH} \) false
    - \( \text{next} \) state is WAIT_CODE if \( \text{LAUNCH} \) true
  - WAIT_CODE
    - \( \text{next} \) state is CODE_OK if \( \text{LAUNCH} \) false
    - \( \text{next} \) state is IDLE if \( \text{LAUNCH} \) true
  - CODE_OK
    - \( \text{next} \) state is IDLE if \( \text{LAUNCH} \) false
    - \( \text{next} \) state is CODE_OK if \( \text{LAUNCH} \) true

FSM Diagram

Example FSM: Missile Launcher

- IDLE
  - LAUNCH false
  - \( \text{next} \) state is IDLE
- WAIT_CODE
  - \( \text{next} \) state is CODE_OK
- CODE_OK
  - \( \text{next} \) state is CODE_OK

FSM State Table

(State encoded as 2 bit binary number)

<table>
<thead>
<tr>
<th>Input</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAIT_CODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE_OK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAUNCH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FSM State Table

<table>
<thead>
<tr>
<th>Input</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAIT CODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE OK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAUNCH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
You Try: FSM Next State Logic

<table>
<thead>
<tr>
<th>Next State NS₂</th>
<th>00</th>
<th>01</th>
<th>12</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>00</td>
<td>01</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>NS₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next State NS₂ =

FSM Launch Logic

<table>
<thead>
<tr>
<th>Launch</th>
<th>00</th>
<th>01</th>
<th>12</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>00</td>
<td>01</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Launch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Launch = S₂

FSM Combinational Logic

- Next State
  - NS₃ = S₂⁺A⁺B
  - NS₄ = S₂⁺A⁺B + S₁⁺S₂⁺A⁺B
- Launch = S₁

Building Memories: SRAM

- Static Random Access Memory
  - What is “static”?
  - It doesn’t change on its own (Unlike “dynamic” memory which does.)
- How do we write the memory?
- Question: What is the difference between the input and output?

Using Clocks to Make Latches

- Transparent Latch
  - When clock is high (1) the input goes straight through to the output
  - When clock is low (0) the last value is “latched” or held and the input is ignored

Problem: The output changes with the input when the clock is high. Glitches in combinational logic can cause bad things.

Edge-Triggered FlipFlops

- Input is only sampled on the rising edge (0→1) of the clock
Edge-Triggered FlipFlop

- Clock = 0
  - Value latched into first latch
- Clock = 1
  - First latch transferred to second latch and output
  - But input ignored

![FlipFlop Diagram]

DRAM

- Dynamic Random Access Memory
- Uses a capacitor to store charge
  - Charge dissipates over time
  - Must constantly refresh the data (hence the dynamic part)
  - Reading data uses up charge [must re-write after reading]
  - But, 6-10x smaller than SRAM = infinitely cheaper

![Capacitor Diagram]

Summary

- Combinational Logic
  - Inputs “immediately” produce output
  - Use truth tables to determine logic equations
  - Common functions such as MUX, DEMUX, decode, encode
- Sequential Logic
  - Current state is updated to next state by the clock
  - Store current state in a latch/memory/flipflop
  - Combinational logic used to determine the next state, and update the current state on the clock

- What do you need for the labs?
  - Combinational logic for the ALU and adder
  - Combinational logic to decode instructions and connect up the ALU
- What do you need for life?
  - Understand that state is stored in memories
  - ...that state is updated by combinational logic
  - ...that clock speed depends on how long it takes to calculate the next state

You Try: Full Adder Carry Out

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

Next State $S_0$

<table>
<thead>
<tr>
<th>Next State $S_0$</th>
<th>Next State $S_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Next State $S_0 = S_0 \cdot A + B$

Next State $S_{1} = S_0 \cdot A + B + \overline{S_0} \cdot S_1 \cdot A + B$

You Try: FSM Next State Logic

<table>
<thead>
<tr>
<th>Input</th>
<th>$S_0$</th>
<th>$S_1$</th>
<th>Next State $S_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Next State $S_{0} = S_0 \cdot A + B$

Next State $S_1 = S_0 \cdot A + B + \overline{S_0} \cdot S_{1} \cdot A + B$