Basics of Logic Design
Arithmetic Logic Unit (ALU)

CPS 104
Lecture 9

Today’s Lecture

• Homework #3 Assigned Due March 2
• Project Groups form groups of 3 by Wed, I will assign after that
• Project Specification(s) will be on Web Due April 20
• Building the building blocks...

Outline
• Review
• Digital building blocks
• An Arithmetic Logic Unit (ALU)

Reading
• Appendix B, Chapter 3
Review: Digital Design

- Logic Design, Switching Circuits, Digital Logic

Recall: Everything is built from transistors
- A transistor is a switch
- It is either on or off
- On or off can represent True or False

Given a bunch of bits (0 or 1)...
- Is this instruction a lw or a beq?
- What register do I read?
- How do I add two numbers?
- Need a method to reason about complex expressions

Review: Boolean Functions

- Boolean functions have arguments that take two values (\{T,F\} or {0,1}) and they return a single or a set of (\{T,F\} or {0,1}) value(s).
- Boolean functions can always be represented by a table called a “Truth Table”

Example: \(F: \{0,1\}^3 \rightarrow \{0,1\}^2\)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
<th>f_2</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</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>
</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>
Review: Boolean Functions and Expressions

\[ F(A, B, C) = (A \times B) + (\neg A \times C) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Review: Boolean Gates

- **Gates** are electronics devices that implement simple Boolean functions

Examples:

- `a` \(\rightarrow\) \(\times\) `b` \(\rightarrow\) \(\rightarrow\) \(\neg\) `a` \(\rightarrow\) \(\rightarrow\)
- `a` \(\rightarrow\) \(\rightarrow\) `b` \(\rightarrow\) \(\rightarrow\) `a` \(\rightarrow\) \(\rightarrow\)`a` \(\rightarrow\) \(\rightarrow\) `b` \(\rightarrow\) \(\rightarrow\)`a` \(\rightarrow\) \(\rightarrow\)`a` \(\rightarrow\) \(\rightarrow\)`a` \(\rightarrow\) \(\rightarrow\)`a` \(\rightarrow\) \(\rightarrow\)
Boolean Functions, Gates and Circuits

- Circuits are made from a network of gates. (function compositions).

\[ F = \neg a \cdot b + \neg b \cdot a \]

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( \text{XOR}(a,b) )</th>
</tr>
</thead>
<tbody>
<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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Digital Design Examples

Input: 2 bits representing an unsigned number (n)
Output: \( n^2 \) as unsigned binary number

Input: 2 bits representing an unsigned number (n)
Output: \( 3-n \) as unsigned binary number
More Design Examples

- **X** is a 3-bit quantity
  1. Write a logic function that is true if and only if **X** contains at least two 1s.
  2. Implement the logic function from problem 1. using only AND, OR and NOT gates. (Note there are no constraints on the number of gate inputs.) By implement, I mean draw the circuit diagram.
  3. Write a logic function that is true if and only if **X**, when interpreted as an unsigned binary number, is greater than the number 5.
  4. Implement the logic function from problem 3. using only AND, OR and NOT gates. (Note there are no constraints on the number of gate inputs.)

Parity Example

- The parity code of a binary word counts the number of ones in a word. If there are an even number of ones the parity code is 0, if there are an odd number of ones the parity code is 1. For example, the parity of 0101 is 0, and the parity of 1101 is 1.
- Construct the truth table for a function that computes the parity of a four-bit word. Implement this function using AND, OR and NOT gates. (Note there are no constraints on the number of gate inputs.)
Design Example

- Consider machine with 4 registers
- Given 2-bit input (register specifier, \(I_1, I_0\))
- Want one of 4 output bits (\(O_3-O_0\)) to be 1
  - E.g., allows a single register to be accessed
- What is the circuit for this?

Circuit Example: Decoder

\[
\begin{array}{c|c|c|c|c}
\text{I}_1 & \text{I}_0 & Q_0 & Q_1 & Q_2 \\
\hline
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
\end{array}
\]
Circuit Example: 2x1 MUX

Multiplexor (MUX) selects from one of many inputs

\[ \text{MUX}(A, B, S) = (A \times S) + (B \times \neg S) \]

Example 4x1 MUX
Arithmetic and Logical Operations in ISA

- What operations are there?
- How do we implement them?
  - Consider a 1-bit Adder

### Truth Table for 1-bit Addition

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>C_in</th>
<th>Sum</th>
<th>C_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

What is the circuit for Sum and for Cout?
A 1-bit Full Adder

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Cin</th>
<th>Sum</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>1</td>
<td>0</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>

Example: 4-bit adder
Subtraction

- How do we perform integer subtraction?
- What is the HW?
### Example: Adder/Subtracter

![Diagram of an ALU slice](image)

<table>
<thead>
<tr>
<th>$B_{inv}$</th>
<th>$F$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$a + b$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$a - b$</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>NOT $b$</td>
</tr>
<tr>
<td>-2</td>
<td></td>
<td>$a \lor b$</td>
</tr>
<tr>
<td>-3</td>
<td></td>
<td>$a \land b$</td>
</tr>
</tbody>
</table>

- Add/Sub = 0 => Addition
- Add/Sub = 1 => Subtraction
Overflow

Example 1:
\[
\begin{array}{c}
0100000 \\
0110101_2 (= 53_{10}) \\
+0101010_2 (= 42_{10}) \\
\hline
1011111_2 (=-33_{10})
\end{array}
\]

Example 2:
\[
\begin{array}{c}
1000000 \\
1010101_2 (=-43_{10}) \\
+1001010_2 (=-54_{10}) \\
\hline
0011111_2 (= 31_{10})
\end{array}
\]

Example 3:
\[
\begin{array}{c}
1100000 \\
0110101_2 (= 53_{10}) \\
+1101010_2 (=-22_{10}) \\
\hline
0011111_2 (= 31_{10})
\end{array}
\]

Example 4:
\[
\begin{array}{c}
0000000 \\
0010101_2 (= 21_{10}) \\
+0101010_2 (= 42_{10}) \\
\hline
0111111_2 (= 63_{10})
\end{array}
\]

Add/Subtract With Overflow detection

Add/Sub

\[
\begin{array}{c}
a_{n-1} \quad b_{n-1} \\
a_{n-2} \quad b_{n-2} \\
a_1 \quad b_1 \\
a_0 \quad b_0
\end{array}
\]

S_{n-1} \quad S_{n-2} \quad S_1 \quad S_0

OVERFLOW
### The new ALU Slice

**Truth Table:**

<table>
<thead>
<tr>
<th>A</th>
<th>F</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a + b</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>a - b</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>NOT b</td>
</tr>
<tr>
<td>-2</td>
<td></td>
<td>a OR b</td>
</tr>
<tr>
<td>-3</td>
<td></td>
<td>a AND b</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Input:** a, b, Cin
- **Output:** Q, Cout, F
- **Logic Operations:**
  - Add/sub
  - Overflow
  - Zero

### The ALU

**Diagram:**

- **Input:** a0, b0, a1, b1, a2, b2, ... (up to an-2, bn-2)
- **Output:** Q0, Q1, Qn-2, Qn-1
- **Logic Operations:**
  - Overflow
  - Zero
  - ALU control
Abstraction: The ALU

• General structure
• Two operand inputs
• Control inputs

The Shift Operation

• Consider an 8-bit machine
• How do I implement the shift operation?
Summary thus far

- Given Boolean function, generate a circuit that "realizes" the function.
- Constructed circuits that can add and subtract.
- The ALU: a circuit that can add, subtract, detect overflow, compare, and do bit-wise operations (AND, OR, NOT)
- Shifter

Next up: Storage Elements: Registers, Latches, Buses
Memory Elements

- All the circuit we looked at so far are **combinational circuits**: the output is a Boolean function of the inputs.
- We need circuits that can remember values. (registers)
- The output of the circuit is a function of the input AND a stored value (state).
- Circuits with memory are called **sequential circuits**.

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q</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>-</td>
</tr>
</tbody>
</table>
Set-Reset Latch (Continued)

\[
\begin{array}{ccc}
\text{R} & \text{S} & \text{Q} \\
0 & 0 & \text{Q} \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 1 & - \\
\end{array}
\]

Set-Reset Latch (Continued)

Time

\[
\begin{array}{c}
\text{S} \\
1 \\
\text{R} \\
1 \\
\text{Q} \\
1 \\
\end{array}
\]
Data Latch (D Latch)

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>Q</td>
</tr>
</tbody>
</table>

**Does not affect Output**

D Flip-Flop

- On C ↑ D is transferred to the first D latch and the second is stable.
- On C ↓ the output of the first stage is transferred to the second (output), and the first stage is stable.
The Tri-State driver is like a (one directional) switch:
- When the Enable is on (E=1) it transfers the input to the output.
- When the Enable is off (E=0) it disconnects the output.

\[
\begin{array}{c|c|c}
D & E & Q \\
0 & 1 & 0 \\
1 & 1 & 1 \\
- & 0 & Z \\
\end{array}
\]

Z :- High Impedance
Bus Connections

- The Bus: Many to many connections.
- Mutual exclusion: At most one Enable is on!

Register Cells on a bus

One can “source” and “sink” from any cell on the bus by activating the right controls (WE and RE).
3-Port Register Cell

- Stores one bit of a register
- Can Read onto Bus-A & Bus-B and Write from Bus-C Simultaneously

3-Port Register File

- Bit-0
- Bit-1
Address Decode Circuit

Register address: 01

Register File (Four 4-bit Registers)
Summary

• Given Boolean function, generate a circuit that “realize” the function.
• Constructed circuits that can add and subtract.
• The ALU: a circuit that can add, subtract, detect overflow, compare, and do bit-wise operations (AND, OR, NOT)
• Shifter
• Memory Elements: SR-Latch, D Latch, D Flip-Flop
• Tri-state drivers & Bus Communication
• Register Files
• Control Signals modify what circuit does with inputs
  ➢ ALU, Shift, Register Read/Write