Storage Elements, Busses, Integer Arithmetic

CPS 104
Lecture 10

Admin

• Homework #4 Due March 6
• Project specification on the web, Due April 14
• Partners determined by Thursday or we assign
  ➢ Email kevin (yike@cs.duke.edu)

Outline
• Review (work problems)
• Logic for Storage
• Register File
• Busses
• Integer Multiply and Divide

Reading
  Appendix B.5, 4.6-4.8
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: \( F: \{0,1\}^3 \rightarrow \{0,1\}^2 \)

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{f, f}_2 \\
0 & 0 & 0 & 0 \ 1 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

Review: Boolean Functions and Expressions

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

\[
\begin{array}{ccc|c}
A & B & C & F \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 \\
\end{array}
\]
Review: Boolean Gates

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

**Examples**

- AND \((a, b)\)
- OR \((a, b)\)
- NOT \((a)\)
- XOR \((a, b)\)
- NAND \((a, b)\)
- NOR \((a, b)\)
- XNOR \((a, b)\)

\[ F = \overline{a} \cdot b + \overline{b} \cdot a \]

Review: Boolean Functions, Gates, and Circuits

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

\[ F = \overline{a} \cdot b + \overline{b} \cdot a \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>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>
Practice Problems

1. (5 pts) Write a logic function that is true if and only if X contains at least two 1s.

2. (5 pts) 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. (5 pts) 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. (5 pts) 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.)

Logic Functions

1. (5 pts) Write a logic function that is true if and only if X, when interpreted as a signed two’s complement number is less than the number -2.

2. (5 pts) Implement the logic function from problem 5. using only AND, OR and NOT gates. (Note there are no constraints on the number of gate inputs.)
Parity Example

1. (15 pts) 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.)

Circuit to Function

- What is Truth Table and Function for this Circuit?
**Review: 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>

```
01101100
+00101100
10011001
```

**Review: The new ALU Slice**

<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>
ALU Modifications

1. (30 pts) Modify the 32-bit ALU of Figure 4.18 to support a new instruction (addpar) that performs four 8-bit additions in parallel. The 8-bits are taken from each of the 32-bit operands. For example, 32-bit input operand A is divided into four 8-bit operands \(A_3\) (bits 31-24), \(A_2\) (bits 23-16), \(A_1\) (bits 15-8), \(A_0\) (bits 0-7). For this new instruction, the addition is performed such that \(A_0+B_0\), \(A_1+B_1\), \(A_2+B_2\), and \(A_3+B_3\) occur simultaneously and independently (i.e., the operation of \(A_0+B_0\) should not affect \(A_1+B_1\), etc.) Draw a diagram similar to Figure 4.18 clearly showing your modifications to support this new instruction.

Review: Abstraction--The ALU

- General structure
  - Input
    - Two operands
    - Control
  - Output
    - Result
    - Overflow
    - Zero

![ALU Diagram](image-url)

© Alvin R. Lebeck  
CPS 104  
13-14
Review: Circuit Example: Decoder

<table>
<thead>
<tr>
<th>I₁</th>
<th>I₀</th>
<th>Q₀</th>
<th>Q₁</th>
<th>Q₂</th>
<th>Q₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>1</td>
<td>0</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>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Review: Example 4x1 MUX
Memory Elements

- All the circuits 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 function of a stored values (state).
- Circuits with memory are called sequential circuits.

Set-Reset Latch

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>Q</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
Set-Reset Latch (Continued)

<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)

Time

© Alvin R. Lebeck
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>

- 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.
- Output changes only on the edge of a clock

D Flip-Flop
Register File

- How do I build a Register File using D Flip-Flops?
- What other components do I need?
**Tri-State Driver**

- 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.

<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>Z</td>
</tr>
</tbody>
</table>

Z :- High Impedance

---

**Bus Connections**

- The Bus: Many to many connections.
- Mutual exclusion: At most one Enable is on!
- Control must ensure this!
Register Cells on a bus

One can “source” and “sink” from any cell on the bus by activating the right controls, IE--input enable, and OE--output enable.

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

Address Decode Circuit
Digital Logic 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
Arithmetic

- Integer Addition---Done
- Integer Multiplication (Ch 4.6)
- Integer Division (Ch 4.7)
- Floating Point Addition (Ch 4.8)
- Floating Point Multiplication (Ch 4.8)

Integer Multiplication

- Product = Multiplicand x Multiplier

- Example: \(0011_{\text{ten}} \times 0101_{\text{ten}}\)

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>0 0 1 1_{\text{ten}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>0 1 0 1_{\text{ten}}</td>
</tr>
<tr>
<td>Product</td>
<td>0 0 0 1 1 1_{\text{ten}}</td>
</tr>
</tbody>
</table>

© Alvin R. Lebeck  CPS 104 33
Multiplication Algorithm #1

- **From Right-Left:**
  - If multiplier digit = 1: add (shifted) copy of multiplicand to result.
  - If multiplier digit = 0: add 0 to result.
- **32 steps when multiplier is 32-bit number.**
- **Example:** $3_{10} \times 5_{10}$ or $0011_2 \times 0101_2$
  
  Product = $00001111_2$

---

**Flowchart: Multiplication Algorithm #1**

1. **Start**
2. **Test Multiplier**
   - **Multiplier0 = 1**
     - 1a. Add multiplicand to product and place the result in Product register
   - **Multiplier0 = 0**
3. **Shift the Multiplier register right 1 bit**
4. **Shift the Multiplicand register left 1 bit**
5. **2nd repetition?**
   - No: < 32 repetitions
   - Yes: 32 repetitions
6. **Done**
Multiplication Hardware #1

- Multiplicand starts in right half of register
- MIPS: 64-bit product in Hi & Lo Regs
  - Move from Lo (mflo) to get 32-bit product
  - Move from hi (mfhi) to get upper 32-bits & test for overflow

Multiplication Hardware #2

- Shift Multiplicand Left ~ Shift Product Right
- Only need 32 bits for multiplicand
- Possible to combine multiplier and product registers
Multiplication Algorithm #2

1. Test Multiplier0
   - Add multiplicand to the left half of the product and place the result in the left half of the Product register

2. Shift the Product register right 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?
   - No: < 32 repetitions
   - Yes: 32 repetitions

Done

Booth Encoding

- Observation:
  - Can write number as difference of two numbers.
  - In particular: Can replace a string of 1s with initial subtract when we see a 1, and then an add when we see the bit AFTER the last 1

- Example 1: $7_{10}$
  - $7_{10} = -1_{10} + 8_{10}$
  - $0111_2 = -0001_2 + 1000_2$

- Example 2: $110_{10} = 01101110_2$
  - $110_{10} = (-2_{10} + 16_{10}) + (-32_{10} + 128_{10})$
  - $01101110_2 = (-00000001_2 + 00010000_2) + (-00100000_2 + 10000000_2)$

- Works for signed numbers as well!
Booth’s Algorithm

• Similar to previous multiply algorithm.

• (Current, Previous) bits of Multiplier:
  - 0,0: middle of string of 0s; do nothing
  - 0,1: end of a string of 1s; add multiplicand
  - 1,0: start of string of 1s; subtract multiplicand
  - 1,1: middle of string of 1s; do nothing

• Shift Product/Multiplier right 1 bit (as before)

Signed Multiplication

• Convert negative numbers to positive and remember the original signs.

• In 2s-complement, can multiply directly using Booth’s Algorithm.
  - Sign extend when shifting.
**Integer Division**

- **Dividend** = **Quotient** x **Divisor** + **Remainder**
- **Example**: $1,001,010_{\text{ten}} / 1000_{\text{ten}}$

\[
\begin{array}{c}
1 & 0 & 0 & 1_{\text{ten}} \\
\hline
\text{Divisor} & 1000_{\text{ten}} & \text{Dividend} & 1001010_{\text{ten}} \\
-1 & 0 & 0 & 0 \\
\hline
1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
\hline
1_{\text{ten}} & \text{Remainder}
\end{array}
\]

**Division Hardware #1**

- **Divisor starts in left half of divisor register**

![Division Hardware Diagram]

Figure Copyright Morgan Kaufmann
Division (contd.)

• Similar to multiplication
  ➢ Shift remainder left instead of shifting divisor right
  ➢ Combine quotient register with right half of remainder register
  ➢ MIPS: Hi contains remainder, Lo contains quotient

• Signed Division
  ➢ Remember the signs and negate quotient if different.
  ➢ Make sign of remainder match the dividend

• Same hardware can be used for both multiply and divide.
  ➢ Need 64-bit register that can shift left and right
  ➢ ALU that adds or subtracts
  ➢ Optimizations possible

Summary

• Storage elements
  ➢ S-R latch, D-Latch, D Flip-Flop
• Register File
• Integer Multiplication & Division

• Homework #4 Due March 6
• Project: form groups and read description.