The Design Process

“To Design Is To Represent”
Design activity yields description/representation of an object

-- Traditional craftsman does not distinguish between the conceptualization and the artifact
-- Separation comes about because of complexity
-- The concept is captured in one or more representation languages
-- This process IS design

Design Begins With Requirements
-- Functional Capabilities: what it will do
-- Performance Characteristics: Speed, Power, Area, Cost, . . .
Design Process (cont.)

*Design Finishes As Assembly*

-- Design understood in terms of components and how they have been assembled

-- Top Down *decomposition* of complex functions (behaviors) into more primitive functions

-- bottom-up *composition* of primitive building blocks into more complex assemblies

*Design is a "creative process," not a simple method*

---

**Design Refinement**

- Informal System Requirement
- Initial Specification
- Intermediate Specification
- Final Architectural Description
- Intermediate Specification of Implementation
- Final Internal Specification
- Physical Implementation

refinement increasing level of detail
Design as Search

Design involves educated guesses and verification
-- Given the goals, how should these be prioritized?
-- Given alternative design pieces, which should be selected?
-- Given design space of components & assemblies, which part will yield the best solution?

Feasible (good) choices vs. Optimal choices

Design as Representation (example)

(1) Functional Specification "VHDL Behavior"
Inputs: 2 x 16 bit operands- A, B; 1 bit carry input- Cin.
Outputs: 1 x 16 bit result- S; 1 bit carry output- Co.
Operations: PASS, ADD (A plus B plus Cin), SUB (A minus B minus Cin), AND, XOR, OR, COMPARE (equality)
Performance: left unspecified for now!

(2) Block Diagram "VHDL Entity"
Understand the data and control flows

---

ECE468 ALU design
Adapted from ©VC and © UCB
Elements of the Design Process

° Divide and Conquer
  • Formulate a solution in terms of simpler components.
  • Design each of the components (subproblems)

° Generate and Test
  • Given a collection of building blocks, look for ways of putting them together that meets requirement

° Successive Refinement
  • Solve “most” of the problem (i.e., ignore some constraints or special cases), examine and correct shortcomings.

° Formulate High-Level Alternatives
  • Articulate many strategies to “keep in mind” while pursuing any one approach.

° Work on the Things you Know How to Do
  • The unknown will become “obvious” as you make progress.

---

Summary of the Design Process

Hierarchical Design to manage complexity

Top Down vs. Bottom Up vs. Successive Refinement

Importance of Design Representations:

- Block Diagrams
- Decomposition into Bit Slices
- Truth Tables, K-Maps
- Circuit Diagrams

Other Descriptions: state diagrams, timing diagrams, reg xfer, . . .

Optimization Criteria:

- Gate Count
- Logic Levels
- Fan-in/Fan-out
- Area
- Delay
- Power
- Pin Out
- Cost
- Design time
- [Package Count]
Introduction to Binary Numbers

° Consider a 4-bit binary number

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

° Examples:
  • $3 + 2 = 5$
  • $3 + 3 = 6$

Two’s Complement Representation

° 2’s complement representation of negative numbers
  • Bitwise inverse and add 1
  • The MSB is always “1” for negative number => sign bit

° Biggest 4-bit Binary Number: 7
° Smallest 4-bit Binary Number: -8
Two’s Complement Arithmetic

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>

Examples: \(7 - 6 = 7 + (-6) = 1\) \(3 - 5 = 3 + (-5) = -2\)

Functional Specification of the ALU

ALU Control Lines (ALUop) Function
- 000  And
- 001  Or
- 010  Add
- 110  Subtract
- 111  Set-on-less-than
A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD

```
\[ \begin{array}{cccc}
    A & B & \text{CarryIn} & \text{CarryOut} \\
    0 & 0 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 1 & 1 & 1 \\
    1 & 0 & 0 & 1 \\
    1 & 0 & 1 & 1 \\
    1 & 1 & 0 & 0 \\
    1 & 1 & 1 & 1 \\
\end{array} \]
```

A One-bit Full Adder

- This is also called a (3, 2) adder
- Half Adder: No CarryIn nor CarryOut
- Truth Table:
# Logic Equation for CarryOut

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\* CarryOut = (!A & B & CarryIn) | (A & !B & CarryIn) | (A & B & !CarryIn) | (A & B & CarryIn)

# Logic Equation for Sum

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\* Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)
Logic Equation for Sum (continue)

° Sum = (\!A \& \!B \& \!CarryIn) \mid (\!A \& B \& \!CarryIn) \mid (A \& \!B \& \!CarryIn) \\
\mid (A \& B \& \!CarryIn)

° Sum = A \ XOR \ B \ XOR \ CarryIn

° Truth Table for XOR:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X XOR Y</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>

Logic Diagrams for CarryOut and Sum

° CarryOut = B \& \!CarryIn \mid A \& \!CarryIn \mid A \& B

° Sum = A \ XOR \ B \ XOR \ CarryIn
How About Subtraction?

- Keep in mind the followings:
  - \((A - B)\) is the that as: \(A + (-B)\)
  - 2’s Complement: Take the inverse of every bit and add 1

- Bit-wise inverse of \(B\) is \(!B\):
  - \(A + !B + 1 = A + (!B + 1) = A + (-B) = A - B\)
Overflow

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th></th>
<th>Decimal</th>
<th>2's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>2</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>3</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>6</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>7</td>
<td>1001</td>
</tr>
</tbody>
</table>

Examples: 7 + 3 = 10 but ...

-4 - 5 = -9 but ...

Overflow Detection

° Overflow: the result is too large (or too small) to represent properly
  - Example: -8 <= 4-bit binary number <= 7

° When adding operands with different signs, overflow cannot occur!

° Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive

° Homework exercise: Prove you can detect overflow by:
  - Carry into MSB ! = Carry out of MSB
Overflow Detection Logic

- Carry into MSB ≠ Carry out of MSB
- For a N-bit ALU: Overflow = CarryIn[N-1] XOR CarryOut[N-1]

Zero Detection Logic

- Zero Detection Logic is just a one BIG NOR gate
- Any non-zero input to the NOR gate will cause its output to be zero
The Disadvantage of Ripple Carry

- The adder we just built is called a "Ripple Carry Adder"
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for a N-bit adder: 2N-gate delay

```
ECE468 ALU design
```

```
The Disadvantage of Ripple Carry
ECE468 ALU design
```

```
A0 → 1-bit ALU → Result0
B0 →
CarryIn0

A1 → 1-bit ALU → Result1
B1 →
CarryIn1

A2 → 1-bit ALU → Result2
B2 →
CarryIn2

A3 → 1-bit ALU → Result3
B3 →
CarryIn3

```

```
Carry Select Header
ECE468 ALU design
```

```
Carry Select Header
ECE468 ALU design
```

```
Carry Select Header
Consider building a 8-bit ALU
  - Simple: connects two 4-bit ALUs in series
```

```
**Carry Select Header (Continue)**

- Consider building a 8-bit ALU
  - Expensive but faster: uses three 4-bit ALUs

![ALU Diagram](image)

**The Theory Behind Carry Lookahead**

- Recalled: \( \text{CarryOut} = (B \& \text{CarryIn}) \mid (A \& \text{CarryIn}) \mid (A \& B) \)
  - \( \text{Cin2} = \text{Cout1} = (\text{B1} \& \text{Cin1}) \mid (\text{A1} \& \text{Cin1}) \mid (\text{A1} \& \text{B1}) \)
  - \( \text{Cin1} = \text{Cout0} = (\text{B0} \& \text{Cin0}) \mid (\text{A0} \& \text{Cin0}) \mid (\text{A0} \& \text{B0}) \)

- Substituting \( \text{Cin1} \) into \( \text{Cin2} \):
  - \( \text{Cin2} = (\text{A1} \& \text{A0} \& \text{B0}) \mid (\text{A1} \& \text{A0} \& \text{Cin0}) \mid (\text{A1} \& \text{B0} \& \text{Cin0}) \mid (\text{B1} \& \text{A0} \& \text{B0}) \mid (\text{B1} \& \text{A0} \& \text{Cin0}) \mid (\text{B1} \& \text{A0} \& \text{Cin0}) \mid (\text{A1} \& \text{B1}) \)

- Now define two new terms:
  - Generate Carry at Bit \( i \) \( \text{gi} = \text{Ai} \& \text{Bi} \)
  - Propagate Carry via Bit \( i \) \( \text{pi} = \text{Ai} \) or \( \text{Bi} \)
The Theory Behind Carry Lookahead (Continue)

- Using the two new terms we just defined:
  - Generate Carry at Bit i \( g_i = A_i \& B_i \)
  - Propagate Carry via Bit i \( p_i = A_i \lor B_i \)

- We can rewrite:
  - \( C_{in1} = g_0 \lor (p_0 \& C_{in0}) \)
  - \( C_{in2} = g_1 \lor (p_1 \& g_0) \lor (p_1 \& p_0 \& C_{in0}) \)
  - \( C_{in3} = g_2 \lor (p_2 \& g_1) \lor (p_2 \& p_1 \& g_0) \lor (p_2 \& p_1 \& p_0 \& C_{in0}) \)

- Carry going into bit 3 is 1 if
  - We generate a carry at bit 2 (g2)
  - Or we generate a carry at bit 1 (g1) and bit 2 allows it to propagate (p2 & g1)
  - Or we generate a carry at bit 0 (g0) and bit 1 as well as bit 2 allows it to propagate (p2 & p1 & g0)
  - Or we have a carry input at bit 0 (Cin0) and bit 0, 1, and 2 all allow it to propagate (p2 & p1 & p0 & Cin0)

A Partial Carry Lookahead Adder

- It is very expensive to build a “full” carry lookahead adder
  - Just imagine the length of the equation for Cin31

- Common practices:
  - Connects several N-bit Lookahead Adders to form a big adder
  - Example: connects four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder
Summary

° An Overview of the Design Process
  • Design is an iterative process-- successive refinement
  • Do NOT wait until you know everything before you start

° An Introduction to Binary Arithmetics
  • If you use 2’s complement representation, subtract is easy.

° ALU Design
  • Designing a Simple 4-bit ALU
  • Other ALU Construction Techniques

° More information from Chapter 4 of the textbook