Computer Architecture

ECE 361
Lecture 5: The Design Process & ALU Design
Quick Review of Last Lecture
MIPS ISA Design Objectives and Implications

- Support general OS and C-style language needs
- Support general and embedded applications
- Use dynamic workload characteristics from general purpose program traces and SPECint to guide design decisions
- Implement processor core with a relatively small number of gates
- Emphasize performance via fast clock

Traditional data types, common operations, typical addressing modes

RISC-style: Register-Register / Load-Store
### MIPS jump, branch, compare instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1,$2,100</td>
<td>if ($1 == $2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Equal test; PC relative branch</td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>if ($1 != $2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not equal test; PC relative</td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare less than; 2’s comp.</td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare &lt; constant; 2’s comp.</td>
</tr>
<tr>
<td>set less than uns.</td>
<td>sltu $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare less than; natural numbers</td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>sltiu $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare &lt; constant; natural numbers</td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jump to target address</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Example: MIPS Instruction Formats and Addressing Modes

- All instructions 32 bits wide
MIPS Instruction Formats

- Fixed instruction size: 4 bytes
- I-type:
  - \( rt \leftrightarrow Memory \ [rs + IMM] \)
  - \( rt \leq rs \ op \ IMM \)
  - \( if (rs == 0) \ PC \ += IMM \)
  - \( [r31 = PC+4] \ PC \ <= rs \)
- R-type
  - \( rd \leq rs \ op \ rt \)
- J-type
  - \( PC \ += Offset \)
  - \( r31 \leq PC+4; \ PC \ += Offset \)
MIPS Operation Overview

° Arithmetic logical
  ° Add, AddU, AddI, ADDIU, Sub, SubU
  ° And, AndI, Or, Orl
  ° SLT, SLTI, SLTU, SLTIU
  ° SLL, SRL

° Memory Access
  ° LW, LB, LBU
  ° SW, SB
Branch & Pipelines

By the end of Branch instruction, the CPU knows whether or not the branch will take place.

However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

Why not execute it?
The next Destination

Single/multicycle Datapaths

Begin ALU design using MIPS ISA.

Pipelining

Memory Systems

IFetch Dcd Exec Mem WB
IFetch Dcd Exec Mem WB
IFetch Dcd Exec Mem WB
IFetch Dcd Exec Mem WB
Outline of Today’s Lecture

° An Overview of the Design Process

° Illustration using ALU design

° Refinements
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

*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
Problem: Design a “fast” ALU for the MIPS ISA

- Requirements?
- Must support the Arithmetic / Logic operations
- Tradeoffs of cost and speed based on frequency of occurrence, hardware budget
MIPS ALU requirements

° Add, AddU, Sub, SubU, AddI, AddIU
  • => 2’s complement adder/sub with overflow detection
° And, Or, AndI, OrI, Xor, Xori, Nor
  • => Logical AND, logical OR, XOR, nor
° SLTI, SLTIU (set less than)
  • => 2’s complement adder with inverter, check sign bit of result
### MIPS arithmetic instruction format

#### R-type:

<table>
<thead>
<tr>
<th>Type</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>10</td>
<td>xx</td>
</tr>
<tr>
<td>ADDIU</td>
<td>11</td>
<td>xx</td>
</tr>
<tr>
<td>SLTI</td>
<td>12</td>
<td>xx</td>
</tr>
<tr>
<td>SLTIU</td>
<td>13</td>
<td>xx</td>
</tr>
<tr>
<td>ANDI</td>
<td>14</td>
<td>xx</td>
</tr>
<tr>
<td>ORI</td>
<td>15</td>
<td>xx</td>
</tr>
<tr>
<td>XORI</td>
<td>16</td>
<td>xx</td>
</tr>
<tr>
<td>LUI</td>
<td>17</td>
<td>xx</td>
</tr>
</tbody>
</table>

#### I-Type:

<table>
<thead>
<tr>
<th>Type</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>00</td>
<td>40</td>
</tr>
<tr>
<td>ADDU</td>
<td>00</td>
<td>41</td>
</tr>
<tr>
<td>SUB</td>
<td>00</td>
<td>42</td>
</tr>
<tr>
<td>SUBU</td>
<td>00</td>
<td>43</td>
</tr>
<tr>
<td>AND</td>
<td>00</td>
<td>44</td>
</tr>
<tr>
<td>OR</td>
<td>00</td>
<td>45</td>
</tr>
<tr>
<td>XOR</td>
<td>00</td>
<td>46</td>
</tr>
<tr>
<td>NOR</td>
<td>00</td>
<td>47</td>
</tr>
</tbody>
</table>

○ Signed arith generate overflow, no carry
Design Trick: divide & conquer

- Break the problem into simpler problems, solve them and glue together the solution

- Example: assume the immediates have been taken care of before the ALU
  - 10 operations (4 bits)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>add</td>
</tr>
<tr>
<td>01</td>
<td>addU</td>
</tr>
<tr>
<td>02</td>
<td>sub</td>
</tr>
<tr>
<td>03</td>
<td>subU</td>
</tr>
<tr>
<td>04</td>
<td>and</td>
</tr>
<tr>
<td>05</td>
<td>or</td>
</tr>
<tr>
<td>06</td>
<td>xor</td>
</tr>
<tr>
<td>07</td>
<td>nor</td>
</tr>
<tr>
<td>12</td>
<td>slt</td>
</tr>
<tr>
<td>13</td>
<td>sltU</td>
</tr>
</tbody>
</table>
Refined Requirements

(1) Functional Specification
inputs: 2 x 32-bit operands A, B, 4-bit mode (sort of control)
outputs: 32-bit result S, 1-bit carry, 1 bit overflow
operations: add, addu, sub, subu, and, or, xor, nor, slt, sltU

(2) Block Diagram (CAD-TOOL symbol, VHDL entity)
Behavioral Representation: VHDL

Entity ALU is
  generic (c_delay: integer := 20 ns;
            S_delay: integer := 20 ns);
  port ( signal A, B:  in vlbit_vector (0 to 31);
         signal    m:  in vlbit_vector (0 to 3);
         signal    S: out vlbit_vector (0 to 31);
         signal    c: out vlbit;
         signal ovf: out vlbit)
end ALU;

  ...

  S <= A + B;
Design Decisions

° Simple bit-slice
  • big combinational problem
  • many little combinational problems
  • partition into 2-step problem

° Bit slice with carry look-ahead

° . . .
Refined Diagram: bit-slice ALU
7-to-2 Combinational Logic

° start turning the crank . . .

<table>
<thead>
<tr>
<th>Function</th>
<th>Inputs</th>
<th>Outputs</th>
<th>K-Map</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M0 M1 M2 M3 A B Cin</td>
<td>S Cout</td>
<td></td>
</tr>
<tr>
<td>0 (add)</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD
A One-bit Full Adder

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

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 + 0 + 0 = 00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 + 0 + 1 = 01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 + 1 + 0 = 01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 + 1 + 1 = 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 + 0 + 0 = 01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 + 0 + 1 = 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 + 1 + 0 = 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 + 1 + 1 = 11</td>
</tr>
</tbody>
</table>
## Logic Equation for CarryOut

\[ \text{CarryOut} = (!A \& B \& \text{CarryIn}) \mid (A \& !B \& \text{CarryIn}) \mid (A \& B \& !\text{CarryIn}) \mid (A \& B \& \text{CarryIn}) \]

\[ \text{CarryOut} = B \& \text{CarryIn} \mid A \& \text{CarryIn} \mid A \& B \]

<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>0</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>
Logic Equation for Sum

\[
\text{Sum} = (!A \land !B \land \text{CarryIn}) \lor (!A \land B \land !\text{CarryIn}) \lor (A \land !B \land !\text{CarryIn}) \lor (A \land B \land \text{CarryIn})
\]

Inputs | Outputs | Comments
--- | --- | ---
| A | B | CarryIn | CarryOut | Sum | | 0 | 0 | 0 | 0 | 0 | 0 + 0 + 0 = 00
| 0 | 0 | 1 | 0 | 1 | 0 + 0 + 1 = 01
| 0 | 1 | 0 | 0 | 1 | 0 + 1 + 0 = 01
| 0 | 1 | 1 | 1 | 0 | 0 + 1 + 1 = 10
| 1 | 0 | 0 | 0 | 1 | 1 + 0 + 0 = 01
| 1 | 0 | 1 | 1 | 0 | 1 + 0 + 1 = 10
| 1 | 1 | 0 | 1 | 0 | 1 + 1 + 0 = 10
| 1 | 1 | 1 | 1 | 1 | 1 + 1 + 1 = 11
Logic Equation for Sum (continue)

° Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)

° Sum = A XOR B XOR CarryIn

° Truth Table for XOR:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th></th>
<th>X XOR Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Logic Diagrams for CarryOut and Sum

- **CarryOut** = B & CarryIn | A & CarryIn | A & B

![CarryOut Diagram](image)

- **Sum** = A XOR B XOR CarryIn

![Sum Diagram](image)
Seven plus a MUX?

- Design trick 2: take pieces you know (or can imagine) and try to put them together

- Design trick 3: solve part of the problem and extend
A 4-bit ALU

1-bit ALU

- 1-bit Full Adder
- CarryIn
- CarryOut
- Mux
- Result

4-bit ALU

- CarryIn0
- A0
- B0
- 1-bit ALU
- Result0
- CarryOut0
- CarryIn1
- A1
- B1
- 1-bit ALU
- Result1
- CarryOut1
- CarryIn2
- A2
- B2
- 1-bit ALU
- Result2
- CarryOut2
- CarryIn3
- A3
- B3
- 1-bit ALU
- Result3
- CarryOut3
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
Additional operations

° A - B = A + (– B)
  • form two complement by invert and add one

Set-less-than? – left as an exercise
Revised Diagram

- LSB and MSB need to do a little extra
## Overflow

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

Examples:
- $7 + 3 = 10$ but ...
- $-4 - 5 = -9$ but ...

```
  0  1  1  1
+  0  1  1  1
    1 0 1 0 0
    1 0 1 1 1
      1 0 1 0
        1 0 1
          1
```

```
  1  1  1  0  0
+  1  0  1  1
    0 1 1 1 1
    1 1 1 1 1
      1 1 1
        1
```

1 0 1 0 0 0 - 4
1 0 1 1 1 1 7
Overflow Detection

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

° 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

° On your own: Prove you can detect overflow by:
  • Carry into MSB ° Carry out of MSB

\[
\begin{array}{c}
\text{0 1 1 1 1} \\
+ \quad \text{0 0 1 1 1} \\
\hline
\quad 1 0 1 0 0
\end{array}
\]

\[
\begin{array}{c}
\text{1 0} \\
+ \quad \text{1 1 0 0 0} \\
\hline
\quad 0 1 1 0 1
\end{array}
\]

\[
\begin{array}{c}
\text{1 0 1 0 0} \\
+ \quad \text{0 1 1 1 1} \\
\hline
\quad 0 0 1 1 1
\end{array}
\]

\[
\begin{array}{c}
\text{1 0 1 0 1} \\
+ \quad \text{1 0 1 1 1} \\
\hline
\quad 0 1 1 1 1
\end{array}
\]
Overflow Detection Logic

- Carry into MSB  
- Carry out of MSB

- For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]

\[\text{Overflow} = \text{CarryIn}[N - 1] \oplus \text{CarryOut}[N - 1]\]
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
More Revised Diagram

- LSB and MSB need to do a little extra

signed-arith and cin xor co

Ovflw

C/L to produce select, comp, c-in
But What about Performance?

° Critical Path of n-bit Rippled-carry adder is n*CP

Design Trick: throw hardware at it
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
Carry Look Ahead (Design trick: peek)

\[
C1 = G0 + C0 \cdot P0
\]

\[
C2 = G1 + G0 \cdot P1 + C0 \cdot P0 \cdot P1
\]

\[
C3 = G2 + G1 \cdot P2 + G0 \cdot P1 \cdot P2 + C0 \cdot P0 \cdot P1 \cdot P2
\]

\[
C4 = \ldots
\]

A  B  C-out
0 0 0  “kill”
0 1  C-in  “propagate”
1 0  C-in  “propagate”
1 1 1  “generate”

P = A \text{ xor } B
G = A \text{ and } B
Plumbing as Carry Lookahead Analogy
The Idea 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 \( p_i = A_i \text{xor} B_i \)

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

° Carry going into bit 3 is 1 if
  • We generate a carry at bit 2 \( g_2 \)
  • Or we generate a carry at bit 1 \( g_1 \) and
    bit 2 allows it to propagate \( p_2 \& g_1 \)
  • Or we generate a carry at bit 0 \( g_0 \) and
    bit 1 as well as bit 2 allows it to propagate \( p_2 \& p_1 \& g_0 \)
  • Or we have a carry input at bit 0 \( C_{in0} \) and
    bit 0, 1, and 2 all allow it to propagate \( p_2 \& p_1 \& p_0 \& C_{in0} \)
The Idea Behind Carry Lookahead

° Recall: \( \text{CarryOut} = (B \& \text{CarryIn}) \mid (A \& \text{CarryIn}) \mid (A \& B) \)

\[ \begin{align*}
\text{Cin2} & = \text{Cout1} = (B_1 \& \text{Cin1}) \mid (A_1 \& \text{Cin1}) \mid (A_1 \& B_1) \\
\text{Cin1} & = \text{Cout0} = (B_0 \& \text{Cin0}) \mid (A_0 \& \text{Cin0}) \mid (A_0 \& B_0)
\end{align*} \]

° Substituting \( \text{Cin1} \) into \( \text{Cin2} \):

\[ \begin{align*}
\text{Cin2} & = (A_1 \& A_0 \& B_0) \mid (A_1 \& A_0 \& \text{Cin0}) \mid (A_1 \& B_0 \& \text{Cin0}) \mid (B_1 \& A_0 \& B_0) \mid (B_1 \& A_0 \& \text{Cin0}) \mid (B_1 \& A_0 \& \text{Cin0}) \mid (A_1 \& B_1)
\end{align*} \]

° Now define two new terms:

\[ \begin{align*}
\text{Generate Carry at Bit } i & = g_i = A_i \& B_i \\
\text{Propagate Carry via Bit } i & = p_i = A_i \ xor \ B_i \\
\text{READ and LEARN Details}
\end{align*} \]
Cascaded Carry Look-ahead (16-bit): Abstraction

\[ C_1 = G_0 + C_0 \cdot P_0 \]

\[ C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \]

\[ C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \]

\[ C_4 = \ldots \]
2nd level Carry, Propagate as Plumbing
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
Design Trick: Guess

\[ CP(2n) = 2 \times CP(n) \]

\[ CP(2n) = CP(n) + CP(\text{mux}) \]

Carry-select adder
Carry Select

- Consider building a 8-bit ALU
  - Simple: connects two 4-bit ALUs in series
Consider building a 8-bit ALU

- Expensive but faster: uses three 4-bit ALUs
Carry Skip Adder: reduce worst case delay

Just speed up the slowest case for each block

Exercise: optimal design uses variable block sizes
Additional MIPS ALU requirements

- Mult, MultU, Div, DivU (next lecture)
  => Need 32-bit multiply and divide, signed and unsigned

- Sll, Srl, Sra (next lecture)
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits

- Nor (leave as exercise to reader)
  => logical NOR or use 2 steps: (A OR B) XOR 1111....1111
Elements of the Design Process

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

° Generate and Test (e.g., ALU)
  • Given a collection of building blocks, look for ways of putting them together that meets requirement

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

° Formulate High-Level Alternatives (e.g., carry select)
  • 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
- [Package Count]
- Area
- Logic Levels
- Delay
- Fan-in/Fan-out
- Power
- Pin Out
- Cost
- Design time

mux design meets at TT