These slides have been compiled using the material provided by the publisher and the lecture notes of the text book’s author Prof. Dave Patterson (CS, UC Berkeley) and Prof. Randy Brynt (ECE, CMU).
Design Process

- **Design Finishes As Assembly**
  - Design involves components and how they are put together

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

- **Bottom-up composition** of primitive

- Use building blocks to create more complex assemblies
Design – A process of refinement

- Collect basic system requirements
- Produce initial specification
- Generate an architectural description
- Specification of implementation
- Analyze and evaluate WRT goals
- Physical Implementation
- Final analysis and specification

increasing level of detail

[Diagram showing the process with arrows indicating progression from basic requirements to final specification.]
Design – A process of refinement

- Collect basic system requirements
  - color display
    - Flexible
    - Low power
    - wireless

- Produce initial specification
  - 2K x 2K pixels
  - 120 frames/sec
  - 24 bits/pixel
  - 20 mW operating, 0.1 mW standby
  - 2 mm thick

- Generate an architectural description
  - PixelSpam interface
  - PRAM storage
  - Hierarchical block diagrams

increasing level of detail
Design – A process of refinement

- Specification of implementation
  - Technology
    » CMOS, BICMOS, Organics
  - Voltages and thresholds
  - Wiring and placement

- Analyze and evaluate WRT goals
  - Functional
  - Timing
  - Power
  - Cost

- Physical Implementation
  - Silicon, organics, PCB, thin film

- Final analysis and specification
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?

- **Good choices vs. optimum choices**
Problem: Design an ALU

◆ Specifications and constraints
  - Speed
  - Footprint
  - Power
  - Controls
  - Datapath
    » Operand width and delivery method

◆ Operations
  - Arithmetic
  - Logical

◆ Tradeoffs
  - Speed vs. footprint
  - Speed vs. power
ALU Operations

- Add, AddU, Sub, SubU, AddI, AddIU
  - 2’s complement adder/sub with overflow detection

- And, Or, AndI, OrI
  - bitwise operations

Total number of operations = 10
Design: divide & conquer method

◆ Break the problem into simpler parts
  - Work on the parts
  - Put pieces together
  - Verify solution works as a whole

◆ Example: Separate immediate instructions from the rest.
  - Process immediates before ALU
    » ALU inputs now uniform
  - 6 non-immediate operations remain
    » Need 3 bits to specify the ALU mode
Design – First Steps

- Complete functional specification first
  inputs: 2 x 32-bit operands A, B, 3-bit operation code
  outputs: 32-bit result R, 1-bit carry, 1 bit overflow
  operations: add, addu, sub, subu, and, or

- High-level block diagram completed next
Design – Reducing the problem to something simpler

- For our ALU, reduce 32-bit problem into simpler 1-bit slices.
  - Changes big combinational problem to a small combinational problem
  - Put the pieces together to solve the big problem.
Designing with lower-level block diagrams

- Consider a 1 BIT ALU block (bit slice)

Replicate 32 times for a 32-bit ALU
The 1-Bit ALU Block

- Partition into separate/independent blocks.
  - Logic
  - Arithmetic

- Complete each block at this level or further refine.
  - Complete logic block now
  - Complete function select block now
  - Decompose arithmetic block into simpler parts
Partitioning the Add/Subtract Block

◆ Computing A + B
  
  - Sum = (a * /b * /Ci) + (/a * b * /Ci) + (/a * /b * Ci) + (a * b * Ci)
  - Co = (b * Ci) + (a * Ci) + (a * b)

  - Sum requires
    » 1 4-input OR
    » 4 3-input ANDs
    » 3 inverters

  - CarryOut requires
    » 1 3-input OR
    » 3 2-input ANDs
Partitioning the Add/Subtract Block

◆ Computing A - B

- Consider simplest way to realize operation given that we already have an adder function.

- Using Adder
  » Difference = A + (-B)

- The problem then reduces to converting operand B to its negative before the Adder.
  » This is easy to do since 2’s complement is used.
  ◆ Negate all bits and add 1
Subtraction

- A – B implemented with bit complementer and full adder
  - XOR complements the input B
  - Setting CarryIn adds 1
Completing the ALU

- Overflow detection & opcode decoder

Control logic to produce select, comp, c-in
# Overflow – illustrating with 4-bit numbers

<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>
</tbody>
</table>

-8 = 1000

<table>
<thead>
<tr>
<th>Example:</th>
<th>7 + 3 = 10 but ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ 0 \quad 1 \quad 1 \quad 1 \]
\[ 0 \quad 1 \quad 1 \quad 1 \]
\[ + \]
\[ 1 \quad 0 \quad 1 \quad 0 \]
\[ -6 \]

<table>
<thead>
<tr>
<th>Example:</th>
<th>-4 - 5 = -9 but ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ 1 \]
\[ 1 \quad 1 \quad 0 \quad 0 \]
\[ + \]
\[ 1 \quad 0 \quad 1 \quad 1 \]
\[ -5 \]

\[ 0 \quad 1 \quad 1 \quad 1 \]
\[ 7 \]
Overflow

- Overflow: the result is too large (or too small) to represent properly
- Overflow cannot occur when adding operands with different signs
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive
- Overflow can be detected decoding the
  - Carry into MSB and the Carry out of MSB

\[
\begin{array}{cccccc}
0 & 1 & 1 & 1 & 1 & 7 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
0 & 1 & 1 & 1 & 1 & 3 \\
\hline
1 & 0 & 1 & 0 & 0 & -6
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 0 & 1 & 1 & 0 & 0 & -4 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
1 & 1 & 1 & 0 & 0 & -5 \\
\hline
0 & 1 & 1 & 1 & 1 & 7
\end{array}
\]
Overflow Detection Logic

- Carry into MSB \( \oplus \) Carry out of MSB
  
  \[
  \text{Overflow} = \text{CarryIn}[N - 1] \oplus \text{CarryOut}[N - 1]
  \]

\[
\begin{array}{c|c|c}
X & Y & X \oplus Y \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
Evaluating Performance – Logic part first

- Logic path has three gate delays
  - XOR + AND/ OR + MUX
  - Could be reduced by one gate delay by routing around XOR
Evaluating Performance – Add/Subtract Block

◆ 1 gate delay for XOR
◆ 3 gate delays for SUM and 2 for CarryOut
  - Sum = (a * b * Ci) + (a * b * Ci) + (a * Ci) + (a * b * Ci)
  - Co = (b * Ci) + (a * Ci) + (a * b)
◆ Each bit slice depends on Ci: the output of the previous slice.
◆ For an N-bit Adder the worst case delay is then 2 * N gate delays
◆ This worst case delay describes a ripple adder
Evaluating Performance – Add/Subtract

◆ Rippled-carry adder
Evaluating Performance – ALU Block

◆ The ALU speed is limited by its slowest block.
  - The logic block has 2 gate delays
  - The add/subtract has $2N + 1$ gate delays, where $N \gg 1$

◆ The arithmetic block is significantly limiting performance

◆ Consider ways to reduce gate delays in Adder
Speeding up the ripple carry adder

◆ Eliminating the ripple

\[
\begin{align*}
    c_1 &= b_0 c_0 + a_0 c_0 + a_0 b_0 \\
    c_2 &= b_1 c_1 + a_1 c_1 + a_1 b_1 \\
    c_3 &= b_2 c_2 + a_2 c_2 + a_2 b_2 \\
    c_4 &= b_3 c_3 + a_3 c_3 + a_3 b_3
\end{align*}
\]
Carry Look Ahead

<table>
<thead>
<tr>
<th>Carry Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

- When both inputs 0, no carry
- When one is 0, the other is 1, propagate carry input
- When both are 1, then generate a carry

<table>
<thead>
<tr>
<th>Alternate Formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Carry-lookahead adder

Generate
\[ g_i = a_i b_i \]

Propagate
\[ p_i = a_i + b_i \]

- Write carry out as function of preceding \( g, p, \) & co

\[
\begin{align*}
  c_1 &= g_0 + p_0 c_0 \\
  c_2 &= g_1 + p_1 c_1 \\
  c_3 &= g_2 + p_2 c_2 \\
  c_4 &= g_3 + p_3 c_3
\end{align*}
\]