Computer Organization and Structure

Bing-Yu Chen
National Taiwan University
Arithmetic for Computers

- Addition and Subtraction
- Gate Logic and K-Map Method
- Constructing a Basic ALU
  - Arithmetic Logic Unit
- Multiplication and Division
- Floating Point
Arithmetic

\[
\begin{align*}
\text{ALU} & \quad | \quad \text{operation} \\
\text{a} & \quad 32 \\
\text{b} & \quad 32 \\
\text{result} & \quad 32
\end{align*}
\]
Integer Addition & Subtraction

- Example: 7 + 6 (just like in grade school)

```
+7: 0000 0000 ... 0000 0111
-6: 1111 1111 ... 1111 1010
+1: 0000 0000 ... 0000 0001
```

- Subtraction = Add negation of second operand

```
Example: 7 - 6 = 7 + (-6)
```

... \( \begin{array}{cccccc}
(0) & 0 & 0 & 0 & 0 & 1 \\hline
(0) & 0 & 0 & 1 & 1 & 1 \\hline
\end{array} \)

(carries)

```plaintext
... 0 0 0 1 1 1 0
```

... (0) (0) (0) 1 (1) 0 (0) 1
Overflow

- Overflow if result out of range
  - Adding
    - +ve and −ve operands → no overflow
    - two +ve operands → overflow if result sign is 1
    - two −ve operands → overflow if result sign is 0
  - Subtracting
    - two +ve or two −ve operands → no overflow
    - +ve from −ve operand → overflow if result sign is 0
    - −ve from +ve operand → overflow if result sign is 1

Consider the operations A + B, and A − B
- Can overflow occur if B is 0?
- Can overflow occur if A is 0?
## Detecting Overflow

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand A</th>
<th>Operand B</th>
<th>Result indicating overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+B</td>
<td>≥0</td>
<td>≥0</td>
<td>&lt;0</td>
</tr>
<tr>
<td>A+B</td>
<td>&lt;0</td>
<td>&lt;0</td>
<td>≥0</td>
</tr>
<tr>
<td>A-B</td>
<td>≥0</td>
<td>&lt;0</td>
<td>&lt;0</td>
</tr>
<tr>
<td>A-B</td>
<td>&lt;0</td>
<td>≥0</td>
<td>≥0</td>
</tr>
</tbody>
</table>
Dealing with Overflow

- An exception (interrupt) occurs
  - On overflow, invoke exception handler
  - Save PC in exception program counter (EPC) register
  - Jump to predefined handler address
  - \texttt{mfc0} (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

- Some languages (e.g., C) ignore overflow
  - new MIPS instructions: \texttt{addu, addiu, sub}
  - \textit{note:} \texttt{addiu still sign-extends!}
  - \textit{note:} \texttt{sltu, sltiu for unsigned comparisons}
An ALU (Arithmetic Logic Unit)

- build an ALU to support \texttt{andi} & \texttt{ori} instructions
  - just build a 1 bit ALU, and use 32 of them

Possible Implementation (sum-of-products):
Review:
The NOT, AND, OR Gates

**Description**

**NOT**
- If $X = 0$ then $X' = 1$
- If $X = 1$ then $X' = 0$

**AND**
- $Z = 1$ if $X$ and $Y$ are both 1

**OR**
- $Z = 1$ if $X$ or $Y$ (or both) are 1

**Gates**

<table>
<thead>
<tr>
<th>X</th>
<th>$\bar{X}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</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>1</td>
</tr>
</tbody>
</table>
Review: NAND, NOR, XOR, XNOR

- 16 functions of two variables:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F9</th>
<th>F10</th>
<th>F11</th>
<th>F12</th>
<th>F13</th>
<th>F14</th>
<th>F15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- X, X', Y, Y', X•Y, X+Y, 0, 1 only half of the possible functions
Review: NAND, NOR

**Description**

NAND

Z = 1 if X is 0 or Y is 0

NOR

Z = 1 if both X and Y are 0

**Gates**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>

**Truth Table**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Review: XOR, XNOR

- **XOR:** X or Y but not both ("inequality", "difference")
  
  \[ X \oplus Y = \overline{X}Y + XY \]

- **XNOR:** X and Y are the same ("equality", "coincidence")
  
  \[ X \oplus Y = \overline{XY} + X \overline{Y} \]

### Description | Gates | Truth Table

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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Review: Truth Tables

- Tabulate all possible input combinations and their associated output values

*Example:* half adder adds two binary digits to form Sum and Carry

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

*Example:* full adder adds two binary digits and Carry in to form Sum and Carry Out

<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>

NOTE: 1 plus 1 is 0 with a carry of 1 in binary
Deriving Boolean Equations from Truth Tables *for Half Adder*

- OR'd together *product* terms for each truth table row where the function is 1
- if input variable is 0, it appears in complemented form;
- if 1, it appears uncomplemented

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

Sum = $A \overline{B} + A \overline{B}$

Carry = $A B$
Example: Full Adder

\[\text{Sum} = \overline{A} \overline{B} \overline{Cin} + \overline{A} B \overline{Cin} + A \overline{B} \overline{Cin} + A B \overline{Cin}\]

<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>

\[\text{Cout} = \overline{A} B \overline{Cin} + A \overline{B} \overline{Cin} + A \overline{B} \overline{Cin} + A B \overline{Cin}\]
Reducing the Complexity of Boolean Equations

- each product term in the above equation covers exactly two rows in the truth table; several rows are "covered" by more than one term

\[
C_{out} = A \cdot C_{in} + B \cdot C_{in} + A \cdot B
\]
Review: Gates & Net

- most widely used primitive building block in digital system design
- Standard Logic Gate Representation

- Net: electrically connected collection of wires
Two-Level Simplification

Key Tool: The Uniting Theorem —

\[ A (B' + B) = A \]

Essence of Simplification:

find two element subsets of the ON-set where only one variable changes its value. This single varying variable can be eliminated!
Karnaugh Map Method

- K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions.
- Beyond that, computer-based methods are needed.
Karnaugh Map Method

- **Numbering Scheme:** 00, 01, 11, 10
  - Gray Code — only a single bit changes from code word to next code word
K-Map Method Examples

F = A

A asserted, unchanged
B varies

B complemented, unchanged
A varies

G = B'

\[ F(A,B,C) = A \]

\[ \text{Cout} = A \overline{B} + B \text{ Cin} + A \text{ Cin} \]
K-Map Method Examples, 3 Variables

F(A,B,C) = \Sigma m(0,4,5,7)

F = B' C' + A C

In the K-map, adjacency wraps from left to right and from top to bottom

F'(A,B,C) = \Sigma m(1,2,3,6)

F' = B C' + A' C

compare with the method of using DeMorgan's Theorem and Boolean Algebra to reduce the complement!
K-map Method Examples: 4 variables

F(A,B,C,D) = Σm(0,2,3,5,6,7,8,10,11,14,15)
F = C' + A' B D + B' D'

find the smallest number of the largest possible subcubes that cover the ON-set
K-map Example: Don't Cares

Don't Cares can be treated as 1's or 0's if it is advantageous to do so.

\[ F = A'D + B' C' D \] \quad \text{w/o don't cares}

\[ F = C' D + A' D \] \quad \text{w/ don't cares}

by treating this DC as a "1", a 2-cube can be formed rather than one 0-cube.

In PoS form: \( F = D (A' + C') \)

same answer as above, but fewer literals.
Design Example: Two Bit Comparator

Block Diagram and Truth Table

A 4-Variable K-map for each of the 3 output functions
Design Example: Two Bit Comparator

F1 = A' B' C' D' + A' B C' D + A B C D + A B' C D'

F2 = A' B' D + A' C + B' C D

F3 = B C' D' + A C' + A B D'

K-map for F1

K-map for F2

K-map for F3

(A xnor C)(B xnor D)
Design Example: Two Bit Adder

Block Diagram and Truth Table

A 4-variable K-map for each of the 3 output functions

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design Example: Two Bit Adder

\[ X = A C + B C D + A B D \]
\[ Z = B D' + B' D = B \text{xor} D \]
\[ Y = A' B' C + A B' C' + A' B C' D + A B C' D' + A B C D' + A B C D \]
\[ = B' (A \text{xor} C) + A' B (C \text{xor} D) + A B (C \text{xnor} D) \]
\[ = B' (A \text{xor} C) + B (A \text{xor} B \text{xor} C) \]

1's on diagonal suggest XOR!
Y K-Map not minimal as drawn

Gate count reduced if XOR available
Two Level Simplification

Definition of Terms

*implicant*: single element of the ON-set or any group of elements that can be combined together in a K-map

*prime implicant*: implicant that cannot be combined with another implicant to eliminate a term

*essential prime implicant*: if an element of the ON-set is covered by a single prime implicant, it is an essential prime

Objective:

- grow implicants into prime implicants
- cover the ON-set with as few prime implicants as possible
- essential primes participate in ALL possible covers
Examples to Illustrate Terms

6 Prime Implicants:  
A' B' D, B C', A C, A' C' D, A B, B' C D

Minimum cover = B C' + A C + A' B' D

5 Prime Implicants:  
B D, A B C', A C D, A' B C, A' C' D

Essential implicants form minimum cover
Examples to Illustrate Terms

Prime Implicants: B D, C D, A C, B’ C

Essential primes form the minimum cover
Algorithm: Minimum Sum of Products Expression from a K-Map

1. Choose an element of ON-set not already covered by an implicant
2. Find "maximal" groupings of 1's and X's adjacent to that element. Remember to consider top/bottom row, left/right column, and corner adjacencies. This forms prime implicants (always a power of 2 number of elements).

- Repeat Steps 1 and 2 to find all prime implicants

3. Revisit the 1's elements in the K-map. If covered by single prime implicant, it is essential, and participates in final cover. The 1's it covers do not need to be revisited

4. If there remain 1's not covered by essential prime implicants, then select the smallest number of prime implicants that cover the remaining 1's
Example

Initial K-map

Primes around A' B C' D'

Primes around A B C' D
Example

Primes around $A \ B \ C' \ D$

Primes around $A \ B' \ C' \ D'$

Essential Primes with Min Cover
Different Implementations

- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - For our purposes, ease of comprehension is important

- Let's look at a 1-bit ALU for addition:

  \[
  c_{\text{out}} = ab + ac_{\text{in}} + bc_{\text{in}} \\
  \text{sum} = a \oplus b \oplus c_{\text{in}}
  \]

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?
Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

Let's build our ALU using a MUX

**Note:** we call this a 2-input MUX even though it has 3 inputs!
1-bit ALU for AND & OR

\[ \text{operation} \]

result
Building a 32 bit ALU

CarryIn → operation → ALU0

a0 → c_in → ALU0
b0 → c_out → result0

ALU0

Operation

CarryIn → operation → ALU1

a1 → c_in → ALU1
b1 → c_out → result1

…

CarryIn → operation → ALU31

a31 → c_in → ALU31
b31 → c_out → result31

In

Out

result0

result1

result31
What about Subtraction?

![Diagram of subtraction circuit]

- Input: a, b
- Operation: binvert
- Carry-in (c_in)
- Result (c_out, operation)
- Output (result)
Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction ($\text{slt}$)
  - remember: $\text{slt}$ is an arithmetic instruction
  - produces a 1 if $rs < rt$ and 0 otherwise
  - use subtraction: $(a-b) < 0$ implies $a < b$

- Need to support test for equality ($\text{beq}$ $t5$, $t6$, $t7$)
  - use subtraction: $(a-b) = 0$ implies $a = b$
Supporting slt – without overflow

\[ a \rightarrow \text{binvert} \rightarrow (0, 1) \rightarrow \text{operation} \rightarrow \text{result} \]

\[ b \rightarrow \text{less} \rightarrow 0 \rightarrow \text{ operation } \rightarrow \text{result} \]

\[ c_{\text{in}} \rightarrow \text{operation} \rightarrow \text{result} \]

\[ \text{operation} \rightarrow \text{result} \]

\[ \text{result} \rightarrow \text{operation} \]

\[ \text{Cout} \]

\[ \text{operation} \rightarrow \text{result} \]
Supporting slt – with overflow & set

binvert

less

overflow detection

operation

result

set

overflow
A 32 bit ALU

1 bit ALU without overflow & set

1 bit ALU with overflow & set

1 bit ALU with overflow & set

1 bit ALU with overflow & set
Test for Equality

- control lines:
  - 000 = and
  - 001 = or
  - 010 = add
  - 110 = subtract
  - 111 = slt
  - 110 = beq

(use zero as the output)
The ALU Symbol

ALU operation

a

b

ALU

zero

result

overflow

CarryOut
Multiplication

- more complicated than addition
  - accomplished via shifting and addition
- more time and more area
- Let's look at 3 versions based on grade school algorithm

0010     (multiplicand)  
\[ \times 0011 \] (multiplier)

- negative numbers: convert and multiply
Multiplication

0010     (multiplicand)
× 0011   (multiplier)

0010
0000
0000
0000110
Multiplication Hardware: First Version

- Multiplicand shift left
- 64-bit ALU
- Product write
- 64 bits
- Control test
- Multiplier shift right
- 32 bits
Multiplication Algorithm: First Version

1. test Multiplier0
   - Multiplier0=1
     1a. add Multiplicand to Product and place the result in Product register
   - Multiplier0=0
     32nd repetition?
     - yes: 32 repetitions
     - no: < 32 repetitions
     2. shift the Multiplicand register left 1 bit
     3. shift the Multiplier register right 1 bit
     done
# Multiplication Example: First Version

<table>
<thead>
<tr>
<th>iteration</th>
<th>step</th>
<th>Multiplier</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>initial value</td>
<td>0011</td>
<td>0000 0010</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>1a: 1 ⇒ Prod = Prod + Mcand</td>
<td>0011</td>
<td>0000 0010</td>
<td>0000 0010</td>
</tr>
<tr>
<td></td>
<td>2: shift left Multiplicand</td>
<td>0011</td>
<td>0000 0100</td>
<td>0000 0010</td>
</tr>
<tr>
<td></td>
<td>3: shift right Multiplier</td>
<td>0001</td>
<td>0000 0100</td>
<td>0000 0110</td>
</tr>
<tr>
<td>2</td>
<td>1a: 1 ⇒ Prod = Prod + Mcand</td>
<td>0001</td>
<td>0000 1000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>2: shift left Multiplicand</td>
<td>0001</td>
<td>0000 1000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>3: shift right Multiplier</td>
<td>0001</td>
<td>0000 1000</td>
<td>0000 0110</td>
</tr>
<tr>
<td>3</td>
<td>1: 0 ⇒ no operation</td>
<td>0000</td>
<td>0001 0000</td>
<td>0000 1110</td>
</tr>
<tr>
<td></td>
<td>2: shift left Multiplicand</td>
<td>0000</td>
<td>0001 0000</td>
<td>0000 1110</td>
</tr>
<tr>
<td></td>
<td>3: shift right Multiplier</td>
<td>0000</td>
<td>0001 0000</td>
<td>0000 1110</td>
</tr>
<tr>
<td>4</td>
<td>1: 0 ⇒ no operation</td>
<td>0000</td>
<td>0010 0000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>2: shift left Multiplicand</td>
<td>0000</td>
<td>0010 0000</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>3: shift right Multiplier</td>
<td>0000</td>
<td>0010 0000</td>
<td>0000 0110</td>
</tr>
</tbody>
</table>
Multiplication Hardware: Second Version

- **Multiplicand**: 32 bits
- **32-bit ALU**: 64 bits
- **Multiplier**: 32 bits
- **Product**: shift right, write
Multiplication Algorithm: Second Version

start

1. test Multiplier0

- Multiplier0=1
  - 1a. 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?
      - yes: 32 repetitions
        - done
      - no: < 32 repetitions
        - Multiplier0=0

# Multiplication Example: Second Version

<table>
<thead>
<tr>
<th>iteration</th>
<th>step</th>
<th>Multiplier</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>initial value</td>
<td>001()</td>
<td>0010</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>1a: Prod = Prod + Mcand</td>
<td>0011</td>
<td>0010</td>
<td>0010 0000</td>
</tr>
<tr>
<td></td>
<td>2: shift right Product</td>
<td>0011</td>
<td>0010</td>
<td>0001 0000</td>
</tr>
<tr>
<td></td>
<td>3: shift right Multiplier</td>
<td>000()</td>
<td>0010</td>
<td>0001 0000</td>
</tr>
<tr>
<td>2</td>
<td>1a: Prod = Prod + Mcand</td>
<td>0001</td>
<td>0010</td>
<td>0011 0000</td>
</tr>
<tr>
<td></td>
<td>2: shift right Product</td>
<td>0001</td>
<td>0010</td>
<td>0001 1000</td>
</tr>
<tr>
<td></td>
<td>3: shift right Multiplier</td>
<td>000()</td>
<td>0010</td>
<td>0001 1000</td>
</tr>
<tr>
<td>3</td>
<td>1: no operation</td>
<td>0000</td>
<td>0010</td>
<td>0001 1000</td>
</tr>
<tr>
<td></td>
<td>2: shift right Product</td>
<td>0000</td>
<td>0010</td>
<td>0000 1100</td>
</tr>
<tr>
<td></td>
<td>3: shift right Multiplier</td>
<td>000()</td>
<td>0010</td>
<td>0000 1100</td>
</tr>
<tr>
<td>4</td>
<td>1: no operation</td>
<td>0000</td>
<td>0010</td>
<td>0000 1100</td>
</tr>
<tr>
<td></td>
<td>2: shift right Product</td>
<td>0000</td>
<td>0010</td>
<td>0000 0110</td>
</tr>
<tr>
<td></td>
<td>3: shift right Multiplier</td>
<td>0000</td>
<td>0010</td>
<td>0000 0110</td>
</tr>
</tbody>
</table>
Multiplication Hardware: Third Version

- Multiplicand
- 32-bit ALU
- 64 bits
- Product
- shift right
- write
- control test
Multiplication Algorithm: Third Version

1. test $Product0$
   - $Product0 = 1$
     - 1a. add Multiplicand to the left half of the Product and place the result in the left half of the Product register
   - $Product0 = 0$
   2. shift the Product register right 1 bit
      - 32nd repetition? (yes: 32 repetitions, no: < 32 repetitions)
      - done
# Multiplication Example: Third Version

<table>
<thead>
<tr>
<th>iteration</th>
<th>step</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>initial value</td>
<td>0010</td>
<td>0000 0011</td>
</tr>
<tr>
<td>1</td>
<td>1a: 1 ⇒ Prod = Prod + Mcand</td>
<td>0010</td>
<td>0010 0011</td>
</tr>
<tr>
<td></td>
<td>2: shift right Product</td>
<td>0010</td>
<td>0011 0001</td>
</tr>
<tr>
<td>2</td>
<td>1a: 1 ⇒ Prod = Prod + Mcand</td>
<td>0010</td>
<td>0011 0001</td>
</tr>
<tr>
<td></td>
<td>2: shift right Product</td>
<td>0010</td>
<td>0001 1000</td>
</tr>
<tr>
<td>3</td>
<td>1: 0 ⇒ no operation</td>
<td>0010</td>
<td>0000 1100</td>
</tr>
<tr>
<td></td>
<td>2: shift right Product</td>
<td>0010</td>
<td>0000 0110</td>
</tr>
</tbody>
</table>
MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits

- Instructions
  - `$\text{mult\ } rs, \ rt$ / $\text{multu\ } rs, \ rt$`
  - 64-bit product in HI/LO
  - `$\text{mfhi\ rd}$ / $\text{mflo\ rd}$`
  - Move from HI/LO to rd
  - Can test HI value to see if product overflows 32 bits
  - `$\text{mul\ rd,\ rs,\ rt}$`
  - Least-significant 32 bits of product $\rightarrow$ rd
Division

- Dividend = Quotient x Divisor + Remainder

\[
\begin{array}{c}
1000 \\
-1000 \\
\hline
1010 \\
-1000 \\
\hline
10
\end{array}
\]
Division Hardware: First Version

- Divisor shift right
- 64-bit ALU
- Remainder write
- 64 bits
- Quotient shift left
- 32 bits
- control test
Division Algorithm: First Version

1. subtract the Divisor register from the Remainder register and place the result in the Remainder register

2a. shift the Quotient register to the left, setting the new rightmost bit to 1

2b. restore the original value by adding the Divisor register to the Remainder register and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. shift the Divisor register right 1 bit

33rd repetition?

yes: 33 repetitions

no: < 33 repetitions

done
## Division Example: First Version

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Quotient</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>initial values</td>
<td>0000</td>
<td>0010 0000</td>
<td>0000 0111</td>
</tr>
<tr>
<td>1</td>
<td>1: Rem = Rem - Div</td>
<td>0000</td>
<td>0010 0000</td>
<td>110 0111</td>
</tr>
<tr>
<td></td>
<td>2b: Rem &lt; 0 ⇒ +Div, sll Q, Q0=0</td>
<td>0000</td>
<td>0010 0000</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td>3: shift Div right</td>
<td>0000</td>
<td>0001 0000</td>
<td>0000 0111</td>
</tr>
<tr>
<td>2</td>
<td>1: Rem = Rem - Div</td>
<td>0000</td>
<td>0001 0000</td>
<td>111 0111</td>
</tr>
<tr>
<td></td>
<td>2b: Rem &lt; 0 ⇒ +Div, sll Q, Q0=0</td>
<td>0000</td>
<td>0001 0000</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td>3: shift Div right</td>
<td>0000</td>
<td>0000 1000</td>
<td>0000 0111</td>
</tr>
<tr>
<td>3</td>
<td>1: Rem = Rem - Div</td>
<td>0000</td>
<td>0000 1000</td>
<td>111 1111</td>
</tr>
<tr>
<td></td>
<td>2b: Rem &lt; 0 ⇒ +Div, sll Q, Q0=0</td>
<td>0000</td>
<td>0000 1000</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td>3: shift Div right</td>
<td>0000</td>
<td>0000 0100</td>
<td>0000 0111</td>
</tr>
<tr>
<td>4</td>
<td>1: Rem = Rem - Div</td>
<td>0001</td>
<td>0000 0100</td>
<td>0000 0011</td>
</tr>
<tr>
<td></td>
<td>2a: Rem ≥ 0 ⇒ sll Q, Q0=1</td>
<td>0001</td>
<td>0000 0100</td>
<td>0000 0011</td>
</tr>
<tr>
<td></td>
<td>3: shift Div right</td>
<td>0001</td>
<td>0000 0010</td>
<td>0000 0011</td>
</tr>
<tr>
<td>5</td>
<td>1: Rem = Rem - Div</td>
<td>0011</td>
<td>0000 0010</td>
<td>0000 0011</td>
</tr>
<tr>
<td></td>
<td>2a: Rem ≥ 0 ⇒ sll Q, Q0=1</td>
<td>0011</td>
<td>0000 0010</td>
<td>0000 0011</td>
</tr>
<tr>
<td></td>
<td>3: shift Div right</td>
<td>0011</td>
<td>0000 0001</td>
<td>0000 0001</td>
</tr>
</tbody>
</table>
Division Hardware: Second Version

- 32-bit ALU
- Divisor 32 bits
- Remainder 64 bits
- Quotient shift left 32 bits
- control test
- shift left write
- Remainder
Division Hardware: Third Version

32-bit ALU

Remainder shift right shift left write

Divisor

32 bits

64 bits

control test
Division Algorithm: Third Version

1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

Test Remainder

Remainder >= 0

Remainder < 0

33rd repetition?

Yes: 33 repetitions

No: < 33 repetitions

Done. Shift left half of Remainder right 1 bit
Division Example: Third Version

<table>
<thead>
<tr>
<th>iteration</th>
<th>step</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>initial value</td>
<td>0010</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td>shift Rem left 1</td>
<td>0010</td>
<td>0000 1110</td>
</tr>
<tr>
<td>1</td>
<td>2: Rem = Rem – Div</td>
<td>0010</td>
<td>0001 1100</td>
</tr>
<tr>
<td></td>
<td>3b: Rem &lt; 0 ⇒ +Div, sll R, R0 = 0</td>
<td>0010</td>
<td>0011 1000</td>
</tr>
<tr>
<td>2</td>
<td>2: Rem = Rem – Div</td>
<td>0010</td>
<td>0001 1000</td>
</tr>
<tr>
<td></td>
<td>3b: Rem &lt; 0 ⇒ +Div, sll R, R0 = 0</td>
<td>0010</td>
<td>0011 0001</td>
</tr>
<tr>
<td>3</td>
<td>2: Rem = Rem – Div</td>
<td>0010</td>
<td>0010 0011</td>
</tr>
<tr>
<td></td>
<td>3a: Rem ≥ 0 ⇒ +Div, sll R, R0 = 1</td>
<td>0010</td>
<td>0001 0011</td>
</tr>
<tr>
<td>4</td>
<td>shift left half of Rem right 1</td>
<td>0010</td>
<td>0001 0011</td>
</tr>
</tbody>
</table>
MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient

- Instructions
  - `div rs, rt` / `divu rs, rt`
  - No overflow or divide-by-0 checking
  - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result
Right Shift and Division

- Left shift by \( i \) places multiplies an integer by \( 2^i \)
- Right shift divides by \( 2^i \)?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., \(-5 / 4\)
    - \( \text{11111011}_2 >> 2 = 11111110_2 = -2 \)
    - Rounds toward \(-\infty\)
  - c.f. \( \text{11111011}_2 >>> 2 = 00111110_2 = +62 \)
Floating Point

- We need a way to represent
  - numbers with fractions, e.g., 3.14159265
  - very small numbers, e.g., 0.000000001
  - very large numbers, e.g., $3.15576 \times 10^9$

- Like scientific notation
  - $-2.34 \times 10^{56}$
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^9$

- In binary
  - $\pm1.xxxxxxxxx_2 \times 2^{yyyy}$

- Types float and double in C
Floating Point Standard

- Defined by IEEE Std. 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)
IEEE Floating-Point Format

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation:
  - actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$
Single-Precision Range

- Exponents 00000000 and 11111111 reserved

- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 - 127 = -126
  - Fraction: 000...00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

- Largest value
  - Exponent: 11111110
    $\Rightarrow$ actual exponent = 254 - 127 = +127
  - Fraction: 111...11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$
Double-Precision Range

- Exponents 0000...00 and 1111...11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 - 1023 = -1022
  - Fraction: 000...00 ⇒ significand = 1.0
    \[ \pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308} \]
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 - 1023 = +1023
  - Fraction: 111...11 ⇒ significand ≈ 2.0
    \[ \pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308} \]
Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx $2^{-23}$
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx $2^{-52}$
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision
Floating-Point Example

- Represent $-0.75$
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - $S = 1$
  - Fraction $= 1000...00_2$
  - Exponent $= -1 + \text{Bias}$
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 0111111110_2$

- Single: $1011111101000...00$
- Double: $1011111111101000...00$
What number is represented by the single-precision float

\[ \begin{array}{c}
\text{1100000101000...00} \\
\text{S = 1} \\
\text{Fraction = 01000...00_2} \\
\text{Exponent = 1000001_2 = 129} \\
\end{array} \]

\[ x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)} \\
= (-1) \times 1.25 \times 2^2 \\
= -5.0 \]
Floating-Point Addition

9.999 \times 10^1 + 1.610 \times 10^{-1} = ?

1. \quad 1.610 \times 10^{-1} = 0.0161 \times 10^1
   - align decimal points
   - shift number with smaller exponent

2. \quad 9.999 + 0.016 = 10.015
   - add significands

3. \quad 10.015 \times 10^1 = 1.0015 \times 10^2
   - normalize result & check for over/underflow

4. \quad 1.0015 \times 10^2 = 1.002 \times 10^2
   - round and renormalize if necessary
Floating-Point Addition

Now consider a 4-digit binary example
- $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2} (0.5 + -0.4375)$

1. Align binary points
- Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2. Add significands
- $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

3. Normalize result & check for over/underflow
- $1.000_2 \times 2^{-4}$, with no over/underflow

4. Round and renormalize if necessary
- $1.000_2 \times 2^{-4}$ (no change) = 0.0625
Floating-Point Addition

1. compare the exponents of the two numbers. shift the smaller number to the right until its exponent would match the larger exponent

2. add the significands

3. normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent

4. round the significand to the appropriate number of bits

overflow or underflow?

yes

exception

no

still normalized?

yes
done

no

no
Floating-Point Addition

- Sign → Exponent → Fraction
- small ALU
- exponent difference
- control
- big ALU
- 0 1
- incrementor difference
- rounding hardware
- 0 1
- shift right
- shift left or right
- Sign → Exponent → Fraction
Floating-Point Multiplication

\[ 1.110 \times 10^{10} \times 9.200 \times 10^{-5} = ? \]

1. addition of exponents
   \[ 10 + -5 = 5 \]

2. multiplication of significands
   \[ 1.110 \times 9.200 = 10.212000 \]

3. \[ 10.212 \times 10^{5} = 1.0212 \times 10^{6} \]

4. \[ 1.0212 \times 10^{6} = 1.021 \times 10^{6} \]

5. determination of sign
   +
Floating-Point Multiplication

- Now consider a 4-digit binary example
  \[ 1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} \ (0.5 \times -0.4375) \]

1. Add exponents
   \[-1 + -2 = -3\]

2. Multiply significands
   \[ 1.000_2 \times 1.110_2 = 1.110_2 \implies 1.110_2 \times 2^{-3} \]

3. Normalize result & check for over/underflow
   \[ 1.110_2 \times 2^{-3} \ (no \ change) \ with \ no \ over/underflow \]

4. Round and renormalize if necessary
   \[ 1.110_2 \times 2^{-3} \ (no \ change) \]

5. Determine sign: \(+ve \times -ve \Rightarrow -ve\)
   \[ -1.110_2 \times 2^{-3} = -0.21875 \]
Floating-Point Multiplication

start

1. add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent

2. multiply the significands

3. normalize the product if necessary, shifting if right and incrementing the exponent

overflow or underflow?

no

4. round the significand to the appropriate number of bits

still normalized?

no

5. set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative

exception

done
Rounding with Guard Digits

$2.56 \times 10^0 + 2.34 \times 10^2 = ?$

- with Guard Digits (2 extra bits)
  
  $0.0256 \times 10^2 + 2.3400 \times 10^2$
  
  $= 2.3656 \times 10^2 = 2.37 \times 10^2$

- without Guard Digits
  
  $0.02 \times 10^2 + 2.34 \times 10^2$
  
  $= 2.36 \times 10^2$
FP Instructions in MIPS

- Separate FP registers
  - 32 single-precision: $f0, $f1, ... $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, ...

- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact

- FP load and store instructions
  - lwcl, ldcl, swcl, sdcl
  - e.g., ldcl $f8, 32($sp)
FP Instructions in MIPS

- Single-precision arithmetic
  - add.s, sub.s, mul.s, div.s
  - e.g., add.s $f0, $f1, $f6

- Double-precision arithmetic
  - add.d, sub.d, mul.d, div.d
  - e.g., mul.d $f4, $f4, $f6

- Single- and double-precision comparison
  - c.xx.s, c.xx.d (xx is eq, lt, le, ...)

- Sets or clears FP condition-code bit
  - e.g., c.lt.s $f3, $f4

- Branch on FP condition code true or false
  - bclt, bclf
  - e.g., bclt TargetLabel
FP Example: °F to °C

- C code:
  ```c
  float f2c (float fahr) {
    return ((5.0/9.0)*(fahr - 32.0));
  }
  ```
  - fahr in $f12, result in $f0, literals in global memory space

- MIPS code:
  ```mips
  f2c: lwc1 $f16, const5($gp)
lwc2 $f18, const9($gp)
div.s $f16, $f16, $f18
lwc1 $f18, const32($gp)
sub.s $f18, $f12, $f18
mul.s $f0,  $f16, $f18
jr  $ra
  ```