MIPS Integer 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 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu$2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>
### 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>

#### Type 0:

<table>
<thead>
<tr>
<th>Type</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>53</td>
<td></td>
</tr>
</tbody>
</table>
MIPS Integer ALU Requirements

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

(2) Block Diagram:

10 operations thus 4 control bits

<table>
<thead>
<tr>
<th>Code</th>
<th>Operation</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>
Building Block: 1-bit Full Adder

A → 1-bit Full Adder → Sum
B → CarryIn

<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>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</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>

2 gate delay for sum
3 gate delay for carry out

2 gate delay version for carry out
Building Block: 1-bit ALU

Performs: AND, OR, addition on A, B or A, B inverted

Diagram:
- Input: A, B, invertB, CarryIn
- Operations: AND, OR, add
- Output: Result, CarryOut
32-Bit ALU Using 32 1-Bit ALUs

32-bit rippled-carry adder
(operation/invertB lines not shown)

Addition/Subtraction Performance:
Total delay = 32 x (1-Bit ALU Delay)
= 32 x 2 x gate delay
= 64 x gate delay
Adding Overflow/Zero Detection Logic

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

- In SLT if $A < B$, the least significant result bit is set to 1.
- Perform $A - B$, $A < B$ if sign bit is 1
  - Use sign bit as Result0 setting all other result bits to zero.

Modified 1-Bit ALU

Control values:
- $000 = \text{and}$
- $001 = \text{or}$
- $010 = \text{add}$
- $110 = \text{subtract}$
- $111 = \text{slt}$

- $\text{invertB}$
- $\text{Operation}$
- $\text{MUX select}$
- $\text{Less}$

position 0: connected to sign bit, Result31
positions 1-31: set to 0

```
<table>
<thead>
<tr>
<th>Control Value</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>and</td>
</tr>
<tr>
<td>001</td>
<td>or</td>
</tr>
<tr>
<td>010</td>
<td>add</td>
</tr>
<tr>
<td>110</td>
<td>subtract</td>
</tr>
<tr>
<td>111</td>
<td>slt</td>
</tr>
</tbody>
</table>
```
MIPS ALU With SLT Support Added

CarryIn0
A0
B0
Less
1-bit ALU
Result0

CarryIn1
A1
B1
Less = 0
1-bit ALU
Result1

CarryIn2
A2
B2
Less = 0
1-bit ALU
Result2

CarryIn3
...

CarryIn31
A31
B31
Less = 0
1-bit ALU
Result31

C

CarryOut0
CarryOut1
CarryOut2
CarryOut30

Zero

Overflow
Improving ALU Performance: Carry Look Ahead (CLA)

\[ C1 = G_0 + C_0 \cdot P_0 \]

\[ C2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \]

\[ C3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \]

G = A and B

P = A xor B

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C-in</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>C-in</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

“kill”

“propagate”

“generate”
Cascaded Carry Look-ahead
16-Bit Example

\[ C_1 = G_0 + C_0 \cdot P_0 \]
\[ C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \]
\[ C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \]

Delay = 2 + 2 + 1 = 5 gate delays

Assuming all gates have equal delay
Additional MIPS ALU requirements

- **Mult, MultU, Div, DivU:**
  
  => Need 32-bit multiply and divide, signed and unsigned.

- **Sll, Srl, Sra:**
  
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits.

- **Nor:**
  
  => logical NOR to be added.
Unsigned Multiplication Example

• Paper and pencil example (unsigned):

| Multiplicand | 1000 |
| Multiplier   | 1001 |
|             | 1000 |
|             | 0000 |
|             | 0000 |
|             | 1000 |
| Product     | 01001000 |

• \( m \) bits \( \times n \) bits = \( m + n \) bit product, \( m = 32, n = 32, 64 \text{ bit product.} \)

• The binary number system simplifies multiplication:

  0 => place 0 \( (0 \times \text{multiplicand}). \)
  1 => place a copy \( (1 \times \text{multiplicand}). \)

• We will examine 4 versions of multiplication hardware & algorithm:

  –Successive refinement of design.
An Unsigned Combinational Multiplier

- Stage $i$ accumulates $A \times 2^i$ if $B_i == 1$
- How much hardware for a 32-bit multiplier? Critical path?
Operation of Combinational Multiplier

- At each stage shift A left (x 2).
- Use next bit of B to determine whether to add in shifted multiplicand.
- Accumulate 2n bit partial product at each stage.
Unsigned Shift-Add Multiplier (version 1)

- 64-bit Multiplicand register.
- 64-bit ALU.
- 64-bit Product register.
- 32-bit multiplier register.

Multiplier = datapath + control
Multiply Algorithm
Version 1

Start

1. Test Multiplier0

- Multiplier0 = 1
  - 1a. Add multiplicand to product & place the result in Product register

- Multiplier0 = 0

2. Shift the Multiplicand register left 1 bit.

3. Shift the Multiplier register right 1 bit.

32nd repetition? [Flowchart]

- No: < 32 repetitions
- Yes: 32 repetitions

Done

<table>
<thead>
<tr>
<th>Product</th>
<th>Multiplier</th>
<th>Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0011</td>
<td>0000 0010</td>
</tr>
<tr>
<td>0000 0010</td>
<td>0001</td>
<td>0000 0100</td>
</tr>
<tr>
<td>0000 0110</td>
<td>0000</td>
<td>0000 1000</td>
</tr>
<tr>
<td>0000 0110</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MULTIPLY HARDWARE Version 2

- Instead of shifting multiplicand to left, shift product to right:
  - 32-bit Multiplicand register.
  - 32-bit ALU.
  - 64-bit Product register.
  - 32-bit Multiplier register.

![Diagram of multiply hardware process]

- 32-bit Multiplicand register.
- 32-bit ALU.
- 64-bit Product register.
- 32-bit Multiplier register.
**Multiply Algorithm**

**Version 2**

1. Test Multiplier0
   - Multiplier0 = 1
     - 1a. Add multiplicand to **the left half of** product & place the result in **the left half of** Product register
   - Multiplier0 = 0

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
Multiplication Version 2 Operation

0 0 0 0

A_{3} A_{2} A_{1} A_{0}

• Multiplicand stays still and product moves right.
MULTIPLY HARDWARE Version 3

- Combine Multiplier register and Product register:
  - 32-bit Multiplicand register.
  - 32-bit ALU.
  - 64-bit Product register, (0-bit Multiplier register).
Multiply Algorithm
Version 3

Start

1. Test Product0
   - Product0 = 1
   - Product0 = 0

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

2. Shift the Product register right 1 bit.

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

Done
Observations on Multiply Version 3

• 2 steps per bit because Multiplier & Product are combined.
• MIPS registers Hi and Lo are left and right halves of Product.
• Provides the MIPS instruction MultU.

• What about signed multiplication?
  – The easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps).
  – Apply definition of 2’s complement:
    • Need to sign-extend partial products and subtract at the end.
  – Booth’s Algorithm is an elegant way to multiply signed numbers using the same hardware as before and save cycles:
    • Can handle multiple bits at a time.
Motivation for Booth’s Algorithm

- Example $2 \times 6 = 0010 \times 0110$:

```
  0010
x 0110
+ 0000  shift (0 in multiplier)
+ 0010  add (1 in multiplier)
+ 0100  add (1 in multiplier)
+ 0000  shift (0 in multiplier)
```

$00001100$

- An ALU with add or subtract gets the same result in more than one way:

$6 = -2 + 8$

$0110 = -00010 + 01000 = 11110 + 01000$

- For example:

```
  0010
x 0110
+ 0000  shift (0 in multiplier)
- 0010   sub (first 1 in multiplier)
  0000  shift (mid string of 1s)
+ 0010   add (prior step had last 1)
```

$00001100$
Booth’s Algorithm

<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>00011111000</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>00011111000</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>00011111000</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>00011111000</td>
<td>none</td>
</tr>
</tbody>
</table>

- Originally designed for Speed (when shift was faster than add).
- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one.
# Booth Example (2 x 7)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Multiplicand</th>
<th>Product</th>
<th>next?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. initial value</td>
<td>0010</td>
<td>0000 0111 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td>1a. $P = P - m$</td>
<td>1110</td>
<td>+ 1110</td>
<td>shift P (sign ext)</td>
</tr>
<tr>
<td>1b.</td>
<td>0010</td>
<td>1111 0011 1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>2.</td>
<td>0010</td>
<td>1111 1001 1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>3.</td>
<td>0010</td>
<td>1111 1100 1</td>
<td>01 -&gt; add</td>
</tr>
<tr>
<td>4a.</td>
<td>0010</td>
<td>+ 0010</td>
<td>shift</td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>0000 1110 0</td>
<td>done</td>
</tr>
</tbody>
</table>
## Booth Example (2 x -3)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Multiplicand</th>
<th>Product</th>
<th>next?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. initial value</td>
<td>0010 1110</td>
<td>0000 1101 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td>1a. P = P - m</td>
<td>1101 0</td>
<td>+ 1110</td>
<td>shift P (sign ext)</td>
</tr>
<tr>
<td>1b.</td>
<td>0010</td>
<td>1111 0110 1</td>
<td>01 -&gt; add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ 0010</td>
<td></td>
</tr>
<tr>
<td>2a.</td>
<td></td>
<td>0001 0110 1</td>
<td>shift P</td>
</tr>
<tr>
<td>2b.</td>
<td>0010</td>
<td>0000 1011 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ 1110</td>
<td></td>
</tr>
<tr>
<td>3a.</td>
<td>0010</td>
<td>1110 1011 0</td>
<td>shift</td>
</tr>
<tr>
<td>3b.</td>
<td>0010</td>
<td>1111 0101 1</td>
<td>11 -&gt; nop</td>
</tr>
<tr>
<td>4a</td>
<td>1111 0101 1</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>1111 1010 1</td>
<td>done</td>
</tr>
</tbody>
</table>
# MIPS Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ⊕ $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~($2</td>
<td>$3)</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1, $2,10</td>
<td>$1 = ~$2</td>
<td>~10</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>rl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slv $1,$2,$3</td>
<td>$1 = $2 &lt;&lt; $3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srlv $1,$2, $3</td>
<td>$1 = $2</td>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>srav $1,$2, $3</td>
<td>$1 = $2</td>
<td>&gt;&gt;</td>
</tr>
</tbody>
</table>
Combinational Shifter from MUXes

Basic Building Block

\[ \text{sel} \rightarrow 1 \ 0 \rightarrow D \]

8-bit right shifter

- What comes in the MSBs?
- How many levels for 32-bit shifter?
If added Right-to-left connections could support Rotate (not in MIPS but found in ISAs)
Barrel Shifter

Technology-dependent solution: a transistor per switch

SR3  SR2  SR1  SR0
D3  D2  D1  D0

A6  A5  A4  A3  A2  A1  A0
Division

\[
\begin{array}{c|c|c}
\text{Divisor} & 1000 & \text{Quotient} \\
\hline
\text{Dividend} & 1001010 & 1001 \\
-1000 & & \\
\hline
10 & & \\
101 & & \\
1010 & & \\
-1000 & & \\
10 & & \\
\end{array}
\]

- See how big a number can be subtracted, creating quotient bit on each step:

\[
\text{Binary} \Rightarrow 1 \times \text{divisor} \text{ or } 0 \times \text{divisor}
\]

\[
\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}
\]

\[
\Rightarrow |\text{Dividend}| = |\text{Quotient}| + |\text{Divisor}|
\]

- 3 versions of divide, successive refinement
DIVIDE HARDWARE Version 1

- 64-bit Divisor register.
- 64-bit ALU.
- 64-bit Remainder register.
- 32-bit Quotient register.
Divide Algorithm
Version 1
Takes \( n+1 \) steps for \( n \)-bit Quotient & Rem.

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, & 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.

\[ \text{Remainder} \geq 0 \quad \text{Test} \quad \text{Remainder} \leq 0 \]

- \( \text{Remainder} \geq 0 \):
  - \( \text{Remainder} \) = \( \text{Remainder} \) - \( \text{Divisor} \)
  - \( \text{Remainder} \) = \( \text{Remainder} \) + \( \text{Divisor} \)

- \( \text{Remainder} < 0 \):
  - \( \text{Remainder} \) = \( \text{Remainder} \) - \( \text{Divisor} \)

\( n+1 \) repetition?

- Yes: \( n+1 \) repetitions (\( n = 4 \) here)
- No: < \( n+1 \) repetitions

Done
Observations on Divide Version 1

• 1/2 bits in divisor are always 0.
  => 1/2 of 64-bit adder is wasted.
  => 1/2 of divisor is wasted.

• Instead of shifting divisor to right, shift remainder to left?

• 1st step cannot produce a 1 in quotient bit (otherwise too big).
  => Switch order to shift first and then subtract, can save 1 iteration.
DIVIDE HARDWARE Version 2

- 32-bit Divisor register.
- 32-bit ALU.
- 64-bit Remainder register.
- 32-bit Quotient register.
Divide Algorithm Version 2

1. Shift the Remainder register left 1 bit.

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

Remainder >= 0 Test Remainder

Remainder < 0

3a. Shift the Quotient 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, & place the sum in the left half of the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

n
repetition?

No: < n repetitions

Yes: n repetitions (n = 4 here)

Done
Observations on Divide Version 2

• Eliminate Quotient register by combining with Remainder as shifted left:
  – Start by shifting the Remainder left as before.
  – Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half.
  – The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  – Thus the final correction step must shift back only the remainder in the left half of the register.
DIVIDE HARDWARE Version 3

- 32-bit Divisor register.
- 32-bit ALU.
- 64-bit Remainder register (0-bit Quotient register).
Divide Algorithm
Version 3

Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit.

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

Remainder \( \geq 0 \)

Test Remainder

Remainder \( < 0 \)

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, & place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.

nth repetition?

No: \( < n \) repetitions

Yes: \( n \) repetitions (\( n = 4 \) here)

Done. Shift left half of Remainder right 1 bit.
Observations on Divide Version 3

- Same Hardware as Multiply: Just requires an ALU to add or subtract, and 64-bit register to shift left or shift right.

- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide.

- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary.
  - **Note:**
    - Dividend and Remainder must have same sign.
    - Quotient negated if Divisor sign & Dividend sign disagree.
    - e.g., \(-7 \div 2 = -3\), remainder = \(-1\)

- Possible for quotient to be too large: If dividing a 64-bit integer by 1, quotient is 64 bits (“called saturation”).
Scientific Notation

- Decimal point
- Exponent
- Mantissa
- Radix (base)

Sign, Magnitude

5.04 x 10^25
- 1.673 x 10^-24
Representation of Floating Point Numbers in Single Precision *IEEE 754 Standard*

Value = \( N = (-1)^S \times 2^{E-127} \times (1.M) \)

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exponent: excess 127 binary integer added</td>
<td>mantissa: sign + magnitude, normalized binary significand with a hidden integer bit: 1.M</td>
</tr>
</tbody>
</table>

0 < E < 255
Actual exponent is: \( e = E - 127 \)

Example: 0 = 0 00000000 0...0
-1.5 = 1 01111111 10...0

Magnitude of numbers that can be represented is in the range:

\[
2^{-126} \text{ (1.0)} \quad \text{to} \quad 2^{127} \left( 2 - 2^{-23} \right)
\]

Which is approximately:

\[
1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}
\]
Representation of Floating Point Numbers in Double Precision **IEEE 754 Standard**

Value = N = \((-1)^S \times 2^{E-1023} \times (1.M)\)

0 < E < 2047
Actual exponent is: e = E - 1023

Example: 0 = 0 00000000000 0 . . . 0

-1.5 = 1 01111111111 10 . . . 0

Magnitude of numbers that can be represented is in the range: $2^{-1022}$ ($1.0$) to $2^{1023} (2 - 2^{-52})$

Which is approximately: $2.23 \times 10^{308}$ to $1.8 \times 10^{308}$
## IEEE 754 Special Number Representation

<table>
<thead>
<tr>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Number Represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent</td>
<td>Significand</td>
<td>Exponent</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>0</td>
</tr>
<tr>
<td>1 to 254</td>
<td>anything</td>
<td>1 to 2046</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>2047</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>2047</td>
</tr>
</tbody>
</table>

\(^1\) May be returned as a result of underflow in multiplication

\(^2\) Positive divided by zero yields “infinity”

\(^3\) Zero divide by zero yields NaN “not a number”
Floating Point Conversion Example

- The decimal number \(0.75_{10}\) is to be represented in the IEEE 754 32-bit single precision format:

\[ 0.75_{10} = 0.11_2 \] (converted to a binary number)

\[ = 1.1 \times 2^{-1} \] (normalized a binary number)

- The mantissa is positive so the sign \(S\) is given by:

\[ S = 0 \]

- The biased exponent \(E\) is given by \(E = e + 127\)

\[ E = -1 + 127 = 126_{10} = 01111110_2 \]

- Fractional part of mantissa \(M\):

\[ M = .10000000000000000000000 \] (in 23 bits)

The IEEE 754 single precision representation is given by:

\[ \begin{array}{ccc}
S & E & M \\
0 & 01111110 & 10000000000000000000000000000000 \\
1 & \text{bit} & 8 \text{bits} & 23 \text{bits}
\end{array} \]
Floating Point Conversion Example

- The decimal number \(-2345.125_{10}\) is to be represented in the **IEEE 754 32-bit single precision format**:

  \[-2345.125_{10} = -100100101001.001_2 \quad \text{(converted to binary)}\]
  \[= -1.00100101001001 \times 2^{11} \quad \text{(normalized binary)}\]

- The mantissa is negative so the sign \(S\) is given by:
  \[S = 1\]

- The biased exponent \(E\) is given by \(E = e + 127\)
  \[E = 11 + 127 = 138_{10} = 10001010_2\]

- Fractional part of mantissa \(M\):
  \[M = .0010010100100100000000 (\text{in 23 bits})\]

The **IEEE 754 single precision representation** is given by:

1 bit | 8 bits | 23 bits
--- | --- | ---
1 | 10001010 | 0010010100100100000000000
Basic Floating Point Addition Algorithm

Assuming that the operands are already in the IEEE 754 format, performing floating point addition: \[ \text{Result} = X + Y = (X_m \times 2^{X_e}) + (Y_m \times 2^{Y_e}) \]

involves the following steps:

1. **Align binary point:**
   - Initial result exponent: the larger of \( X_e, Y_e \)
   - Compute exponent difference: \( Y_e - X_e \)
   - If \( Y_e > X_e \) Right shift \( X_m \) that many positions to form \( X_m \times 2^{X_e-Y_e} \)
   - If \( X_e > Y_e \) Right shift \( Y_m \) that many positions to form \( Y_m \times 2^{Y_e-X_e} \)

2. **Compute sum of aligned mantissas:**
   - i.e \( X_m \times 2^{X_e-Y_e} + Y_m \) or \( X_m + X_m \times 2^{Y_e-X_e} \)

3. **If normalization of result is needed, then a normalization step follows:**
   - Left shift result, decrement result exponent (e.g., if result is 0.001xx…) or
   - Right shift result, increment result exponent (e.g., if result is 10.1xx…) Continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard).

4. **Doubly biased exponent must be corrected:** extra subtraction step of the bias amount.

5. **Check result exponent:**
   - If larger than maximum exponent allowed return exponent overflow
   - If smaller than minimum exponent allowed return exponent underflow

6. **Round the significand and re-normalize if needed.** If result mantissa is 0, may need to set the exponent to zero by a special step to return a proper zero.
Start

(1) Compare the exponents of the two numbers shift the smaller number to the right until its exponent matches the larger exponent

(2) Add the significands (mantissas)

(3) Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent

(4) Overflow or Underflow?

Yes
Generate exception or return error

No
Stil normalized?

Yes
Done

No
Round the significand to the appropriate number of bits
If mantissa = 0, set exponent to 0

(5)
Floating Point Addition Example

- Add the following two numbers represented in the IEEE 754 single precision format: \( X = 2345.125_{10} \) represented as:

\[
0 \quad 10001010 \quad 00100101001001000000000000
\]

to \( Y = .75_{10} \) represented as:

\[
0 \quad 01111110 \quad 10000000000000000000000000
\]

1. Align binary point:
   - \( X_e > Y_e \) initial result exponent = \( Y_e = 10001010 = 138_{10} \)
   - \( X_e - Y_e = 10001010 - 01111110 = 00000110 = 12_{10} \)
   - Shift \( Y_m \) \( 12_{10} \) positions to the right to form
     \[
     Y_m 2^{Y_e - X_e} = Y_m 2^{-12} = 0.000000000001100000000000000
     \]

2. Add mantissas:
   \[
   X_m + Y_m 2^{-12} = 1.00100101001001000000000000
   \]

\[
+ 0.0000000000011000000000000000 = 1.00100101001111000000000
\]

3. Normalized? Yes

Result \[
0 \quad 10001010 \quad 00100101001111000000000000
\]
IEEE 754 Single precision Addition Notes

• If the exponents differ by more than 24, the smaller number will be shifted right entirely out of the mantissa field, producing a zero mantissa.
  – The sum will then equal the larger number.
  – Such truncation errors occur when the numbers differ by a factor of more than $2^{24}$, which is approximately $1.6 \times 10^7$.
  – Thus, the precision of IEEE single precision floating point arithmetic is approximately 7 decimal digits.

• Negative mantissas are handled by first converting to 2's complement and then performing the addition.
  – After the addition is performed, the result is converted back to sign-magnitude form.

• When adding numbers of opposite sign, cancellation may occur, resulting in a sum which is arbitrarily small, or even zero if the numbers are equal in magnitude.
  – Normalization in this case may require shifting by the total number of bits in the mantissa, resulting in a large loss of accuracy.

• Floating point subtraction is achieved simply by inverting the sign bit and performing addition of signed mantissas as outlined above.
Basic Floating Point Multiplication Algorithm

Assuming that the operands are already in the IEEE 754 format, performing floating point multiplication:

\[ \text{Result} = R = X \times Y = (-1)^{Xs} (Xm \times 2^{Xe}) \times (-1)^{Ys} (Ym \times 2^{Ye}) \]

involves the following steps:

1. If one or both operands is equal to zero, return the result as zero, otherwise:
2. Compute the exponent of the result:
   \[ \text{Result exponent} = \text{biased exponent (X)} + \text{biased exponent (Y)} - \text{bias} \]
3. Compute the sign of the result \( Xs \ XOR \ Ys \)
4. Compute the mantissa of the result:
   - Multiply the mantissas: \( Xm \times Ym \)
5. Normalize if needed, by shifting mantissa right, incrementing result exponent.
6. Check result exponent for overflow/underflow:
   - If larger than maximum exponent allowed return exponent overflow
   - If smaller than minimum exponent allowed return exponent underflow
7. Round the result to the allowed number of mantissa bits; normalize if needed.
Overflow or Underflow?

Floating Point Multiplication Flowchart

1. Is one/both operands = 0?
   - Set the result to zero: exponent = 0

2. Compute exponent: biased exp.(X) + biased exp.(Y) - bias

3. Compute sign of result: Xs XOR Ys

4. Multiply the mantissas

5. Normalize mantissa if needed

6. Overflow or Underflow?
   - Yes: Generate exception or return error
   - No: Round or truncate the result mantissa

7. StillNormalized?
   - Yes: Done
   - No: Round or truncate the result mantissa
Floating Point Multiplication Example

- Multiply the following two numbers represented in the IEEE 754 single precision format:  \( X = -18_{10} \) represented as:

<table>
<thead>
<tr>
<th></th>
<th>01000011</th>
<th>00100000000000000000000</th>
</tr>
</thead>
</table>

and \( Y = 9.5_{10} \) represented as:

<table>
<thead>
<tr>
<th></th>
<th>01000010</th>
<th>00110000000000000000000</th>
</tr>
</thead>
</table>

1. Value of one or both operands = 0? No, continue with step 2
2. Compute the sign: \( S = X_s \ XOR \ Y_s = 1 \ XOR \ 0 = 1 \)
3. Multiply the mantissas: The product of the 24 bit mantissas is 48 bits with two bits to the left of the binary point:

\( (01).01010110000000000000000 \)

Truncate to 24 bits:

hidden \( \rightarrow (1).01010110000000000000000 \)

4. Compute exponent of result:
\( X_e + Y_e - 127_{10} = 1000 \ 0011 \ + \ 1000 \ 0010 \ - \ 0111111 \ = \ 1000 \ 0110 \)

5. Result mantissa needs normalization? No

Result

<table>
<thead>
<tr>
<th></th>
<th>10000110</th>
<th>01010101100000000000000</th>
</tr>
</thead>
</table>
IEEE 754 Single precision Multiplication Notes

• Rounding occurs in floating point multiplication when the mantissa of the product is reduced from 48 bits to 24 bits.
  – The least significant 24 bits are discarded.

• Overflow occurs when the sum of the exponents exceeds 127, the largest value which is defined in bias-127 exponent representation.
  – When this occurs, the exponent is set to 128 (E = 255) and the mantissa is set to zero indicating + or - infinity.

• Underflow occurs when the sum of the exponents is more negative than -126, the most negative value which is defined in bias-127 exponent representation.
  – When this occurs, the exponent is set to -127 (E = 0).
  – If M = 0, the number is exactly zero.
  – If M is not zero, then a denormalized number is indicated which has an exponent of -127 and a hidden bit of 0.
  – The smallest such number which is not zero is \(2^{-149}\). This number retains only a single bit of precision in the rightmost bit of the mantissa.
Basic Floating Point Division Algorithm

Assuming that the operands are already in the IEEE 754 format, performing floating point multiplication:

\[
\text{Result} = R = \frac{X}{Y} = (-1)^{X_s} (X_m \times 2^{X_e}) / (-1)^{Y_s} (Y_m \times 2^{Y_e})
\]

involves the following steps:

1. If the divisor Y is zero return “Infinity”, if both are zero return “NaN”
2. Compute the sign of the result \( X_s \ XOR \ Y_s \)
3. Compute the mantissa of the result:
   - The dividend mantissa is extended to 48 bits by adding 0's to the right of the least significant bit.
   - When divided by a 24 bit divisor \( Y_m \), a 24 bit quotient is produced.
4. Compute the exponent of the result:
   \[
   \text{Result exponent} = [\text{biased exponent} (X) - \text{biased exponent} (Y)] + \text{bias}
   \]
5. Normalize if needed, by shifting mantissa left, decrementing result exponent.
6. Check result exponent for overflow/underflow:
   - If larger than maximum exponent allowed return exponent overflow
   - If smaller than minimum exponent allowed return exponent underflow
Extra Bits for Rounding

Extra bits used to prevent or minimize rounding errors.

How many extra bits?
IEEE: As if computed the result exactly and rounded.

Addition:

\[
\begin{align*}
1.xxxxx & \quad 1.xxxxx & \quad 1.xxxxx \\
+ 1.xxxxx & \quad 0.001xxxx & \quad 0.01xxxx \\
1x.xxxxxy & \quad 1.xxxxxxyyy & \quad 1x.xxxxxyyy \\
\end{align*}
\]

post-normalization \quad pre-normalization \quad pre and post

- **Guard Digits**: digits to the right of the first \( p \) digits of significand to guard against loss of digits – can later be shifted left into first \( P \) places during normalization.
- Addition: carry-out shifted in.
- Subtraction: borrow digit and guard.
- Multiplication: carry and guard. Division requires guard.
Rounding Digits

Normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

E.g., B = 10, p = 3:

\[
\begin{array}{c|c|c}
0 & 2 & 1.69 \\
\hline
- & 0 & 7.85 \\
\hline
0 & 2 & 1.61 \\
\end{array}
\]

\[= 1.6900 \times 10^{2\text{-bias}}\]

\[= -0.0785 \times 10^{2\text{-bias}}\]

\[= 1.6115 \times 10^{2\text{-bias}}\]

One round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit.

**IEEE Standard:**

- four rounding modes: round to nearest (default)
  - round towards plus infinity
  - round towards minus infinity
  - round towards 0

round to nearest:

- round digit < B/2 then truncate
- > B/2 then round up (add 1 to ULP: unit in last place)
- = B/2 then round to nearest even digit

*it can be shown that this strategy minimizes the mean error introduced by rounding.*
Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding.

\[
d_{0} \cdot d_{1} d_{2} d_{3} \ldots d_{p-1} 0 0 0
\]

<table>
<thead>
<tr>
<th>0 . 0 0 X . . . X</th>
<th>X X S</th>
</tr>
</thead>
</table>

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

\[
d_{0} \cdot d_{1} d_{2} d_{3} \ldots d_{p-1} 0 0 0
\]

| 0 . 0 0 X . . . X | X X 0 |

\[
d_{0} \cdot d_{1} d_{2} d_{3} \ldots d_{p-1} 0 0 0
\]

| 0 . 0 0 X . . . X | X X 1 |

generates a borrow

Rounding Summary:

Radix 2 minimizes wobble in precision.

Normal operations in +,-,*,/ require one carry/borrow bit + one guard digit.

One round digit needed for correct rounding.

Sticky bit needed when round digit is B/2 for max accuracy.

Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed.
Infinity and NaNs

Result of operation *overflows*, i.e., is larger than the largest number that can be represented.

Overflow is not the same as divide by zero (raises a different exception).

\[ \pm\text{ infinity } \begin{array}{c} S \end{array} \begin{array}{c} 1 \ldots 1 \end{array} \begin{array}{c} 0 \ldots 0 \end{array} \]

It may make sense to do further computations with infinity e.g., \( X/0 > Y \) may be a valid comparison

Not a number, but not infinity (e.q. \( \sqrt{-4} \)) invalid operation exception (unless operation is = or =)

\[ \text{NaN} \begin{array}{c} S \end{array} \begin{array}{c} 1 \ldots 1 \end{array} \text{ non-zero} \quad \text{HW decides what goes here} \]

NaNs propagate: \( f(\text{NaN}) = \text{NaN} \)