Set Less Than (slt) Function

- slt function is defined as:
  
  \[
  A \text{ slt } B = \begin{cases} 
  000 \ldots 001 & \text{if } A < B, \text{i.e. if } A - B < 0 \\
  000 \ldots 000 & \text{if } A \geq B, \text{i.e. if } A - B \geq 0 
  \end{cases}
  \]

- Thus, each 1-bit ALU should have an additional input (called "Less"), that will provide results for slt function. This input has value 0 for all but 1-bit ALU for the least significant bit.
- For the least significant bit Less value should be sign of \( A - B \)

### 32-bit ALU With 5 Functions

- slt function: Operation = 3  Binvert = 1
32-bit ALU with 5 Functions and Zero

<table>
<thead>
<tr>
<th>Function</th>
<th>Binvert</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>subtract</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>slt</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

32-bit ALU with 6 Functions

A nor B = A and B

Figure B.5.10 (Top)

<table>
<thead>
<tr>
<th>Function</th>
<th>Ainvert</th>
<th>Binvert</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>subtract</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>slt</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>nor</td>
<td>1</td>
<td>1</td>
<td>00</td>
</tr>
</tbody>
</table>
32-bit ALU Elaboration

- We have (so far) designed an ALU for most (integer) arithmetic and logic functions required by the core MIPS ISA.
- 32-bit ALU with 6 functions omits support for:
  - shift instructions
  - XOR logic instruction
  - integer multiply and divide instructions.
- Shift instructions:
  - It would be possible to widen 1-bit ALU multiplexer to include 1-bit shift left and/or 1-bit shift right.
  - Hardware designers created the circuit called a barrel shifter, which can shift from 1 to 31 bits in less time than it takes to add two 32-bit numbers. Thus, shifting is normally done outside the ALU.
- Integer multiply/divide is also usually done outside the ALU.
- We will next consider integer multiplication.

Multiplication

- Multiplication is more complicated than addition:
  - accomplished via shifting and addition
- More time and more area required
- We shall look at 3 hardware design versions based on an elementary school algorithm
- Example of unsigned multiplication:
  5-bit multiplicand \( \begin{array}{c} 10001 \end{array} \) = \( 17_{10} \)
  5-bit multiplier \( \begin{array}{c} 10011 \end{array} \) = \( 19_{10} \)
  \( \begin{array}{c} 10001 \\
 10001 \\
00000 \\
00000 \\
10001 \\
\end{array} \)
  \( \begin{array}{c} 10100011 \end{array} = 323_{10} \)
- But, this algorithm is very impractical to implement in hardware

Reading Assignment: 3.4
Multiplication: Improved Algorithm

- The multiplication can be done with intermediate additions.
- The same example:

<table>
<thead>
<tr>
<th>multiplicand</th>
<th>10001</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplier</td>
<td>1011</td>
</tr>
</tbody>
</table>

intermediate product
0000000000
add since multiplier bit=1
10001
intermediate product
0000100001
shift multiplicand and add since multiplier bit=1
10001
intermediate product
0000110011
shift multiplicand and no addition since multiplier bit=0
shift multiplicand and no addition since multiplier bit=0
shift multiplicand and add multiplier since bit=1
10001
final result
0101000011

Multiplication Hardware: 1st Version

Figure 3.5

Figure 3.6
Figure 3.7
Real Numbers

- Representing real numbers
  - $3.14159256_{10}$
  - $3.155,760_{10}$
  - $315.576_{10} \times 10^4$

- Scientific notation:
  - Single digit to the left of digital point:
    - $3.15576_{10} \times 10^6$

- Normalized scientific notation:
  - No leading zeros: $1.0 \times 10^9$, but not $0.1 \times 10^8$

- Similar for binary:
  - $00101101_2 = 1.0 \times 2^5$ or $1.0 \times 2^{101}$ – normalized notation

Reading Assignment: 3.6
Floating Point Number Formats

- The term floating point number refers to representation of real binary numbers in computers.
- IEEE 754 standard defines standards for floating point representations
- Single precision:

\[
\begin{array}{cccccc}
31 & 30 & 22 & 23 & 0 \\
\hline
s & E & \text{Fraction} \\
\end{array}
\]

- Double precision:

\[
\begin{array}{cccccc}
63 & 62 & 52 & 51 & 32 \\
\hline
s & E & \text{Fraction} \\
\end{array}
\]

Converting to Floating Point

1. Normalize binary real number i.e. put it into the normalized form:

\[(-1)^s \times 1.\text{Fraction} \times 2^{\text{Exp}}\]

-1101.1011101\text{2} = (-1)^1 \times 1.1011011 \times 2^3

+1100111110111.000\text{0} = (-1)^0 \times 1.110011111101110 \times 2^9

2. Load fields of single or double precision format with values from normalized form, but with the adjustment for E field.

\[E = \text{Exp} + 127_{10} = \text{Exp} + 01111111_2 \text{ for single precision}\]

\[E = \text{Exp} + 1023_{10} = \text{Exp} + 01111111111_2 \text{ for double precision}\]

- E is called a biased exponent - \((-1)^s \times 1.\text{Fraction} \times 2^{(\text{Exp}-\text{Bias})}\)
Floating Point: Example 1

- Find single and double precision of $-13.6875_{10}$

Normalized form: $(-1)^1 \times 1.1011011 \times 2^3$

- single precision:
  \[ E = 11_2 + 01111111_2 = 10000010_2 \]
  \[ |1|10000010|101101100000000000000000| \]

- double precision
  \[ E = 11_2 + 01111111111_2 = 100000000010_2 \]
  \[ |1|100000000010|101101100000000000000000| \]
  \[ |00000000000000000000000000000000| \]

Floating Point: Example 2

- Find single and double precision of $+927.45_{10}$

Normalized form: $(-1)^0 \times 1.110011111101100 \times 2^9$

- single precision
  \[ E = 1001_2 + 01111111_2 = 10001000_2 \]
  \[ |0|10001000|1100111110110011001100| \]
  \[ \text{truncation} \quad |0|10001000|1100111110110011001101| \]
  \[ \text{rounding} \quad |0|10001000|1100111110110011001101| \]

- double precision
  \[ E = 1001_2 + 01111111111_2 = 10000001000_2 \]
  \[ |0|10000001000|1100111110110011001110| \]
  \[ \text{truncation} \quad |10011001100110011001100110011101| \]
  \[ \text{rounding} \quad |10011001100110011001100110011101| \]
### Converting to Floating Point: Conclusion

- Rules for biased exponents in single precision apply only for real exponents in the range [-126,127], thus we can have biased exponents only in the range [1,254].
- The number 0.0 is represented as S=0, E=0 and Fraction=0. The infinite number is represented with E=255. There are some additional rules that are outside our scope.
- Find the largest (non-infinite) real binary number (by magnitude) which can be represented in a single precision.
  - Floating point overflow
- Find the smallest (non-zero) real binary number (by magnitude) which can be represented in a single precision.
  - Floating point underflow

---

### Floating Point Addition

```
Start

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

   - Overflow or underflow?
     - Yes
     - No

   - Exception

4. Round the significant to the appropriate number of bits

   - Still normalized?
     - Yes
     - No

   - Done
```

---

Figure 3.16
Arithmetic Unit for Floating Point Addition

![Diagram](image)

Figure 3.17

Booth’s Algorithm

![Diagram](image)

10
Booth’s Algorithm: Example

<table>
<thead>
<tr>
<th>Product register</th>
<th>assumed for step1.</th>
<th>01 – subtract (i.e. add 001011)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000000 011101</td>
<td>011010 101110</td>
</tr>
<tr>
<td>shift</td>
<td>000101 101110</td>
<td>111010 101110</td>
</tr>
<tr>
<td></td>
<td>01 – add</td>
<td>111010 101110</td>
</tr>
<tr>
<td>shift</td>
<td>001011</td>
<td>001000 010111</td>
</tr>
<tr>
<td></td>
<td>10 – subtract</td>
<td>000001 000010</td>
</tr>
<tr>
<td>shift</td>
<td>001000 010111</td>
<td>000001 000010</td>
</tr>
</tbody>
</table>

Step 1. ends

Step 2. ends

Step 3. ends

Step 4. ends

Step 5. ends

Step 6. ends

Result: 111011 000012 = -000100 1111112 = -256+63 = -31910