Set Less Than (slt) Function

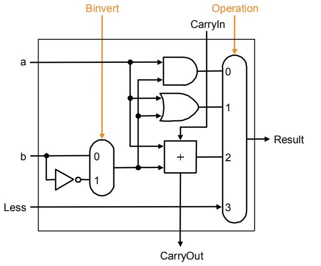
- slt function is defined as:

$$A \text{ slt } B = \begin{cases} 000 \ldots 001 & \text{if } A < B\text{, i.e. if } A - B < 0 \\ 000 \ldots 000 & \text{if } A \geq B\text{, i.e. if } A - B \geq 0 \end{cases}$$

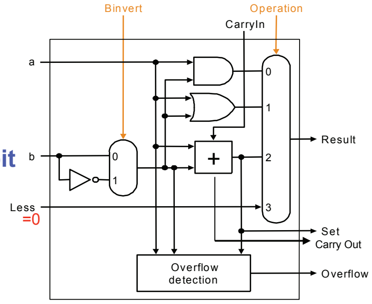
- Thus, each 1-bit ALU should have an additional input (called "Less"), that will provide results for slt function. This input has value 0 for all but 1-bit ALU for the least significant bit.
- For the least significant bit Less value should be sign of A – B

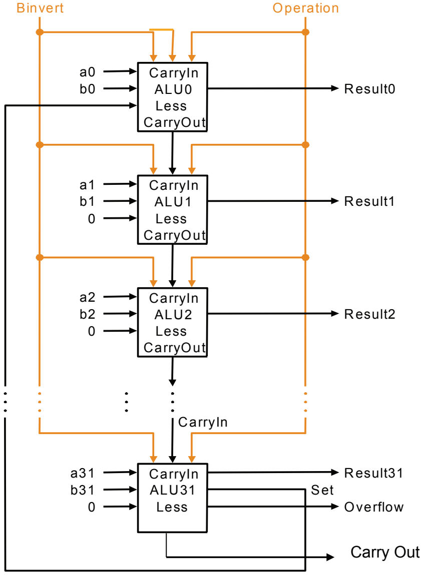
32-bit ALU With 5 Functions

1-bit ALU1-ALU30



1-bit ALU31 the most significant bit

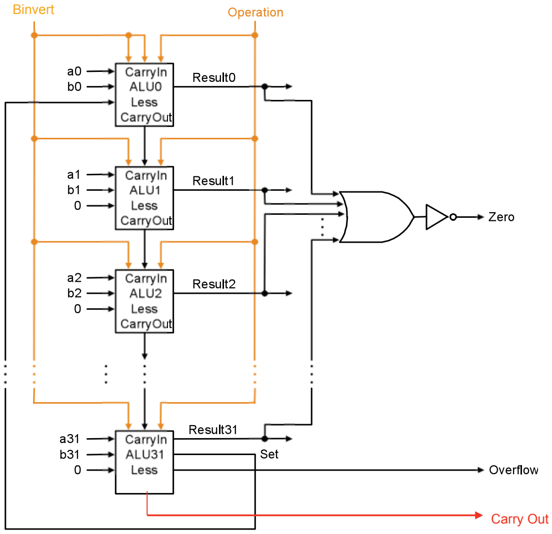




slt function: Operation = 3 Binvert =1

32-bit ALU with 5 Functions and Zero

	Control lines	
Function	**Binvert** (1 line)	**Operation** (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10
slt	1	11



32-bit ALU with 6 Functions

A nor B = $\overline{A}$ and $\overline{B}$

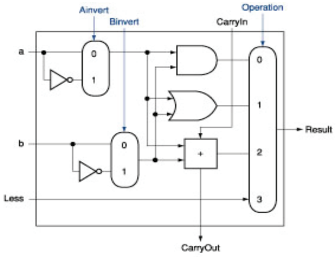


Figure B.5.10 (Top)

Function	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00

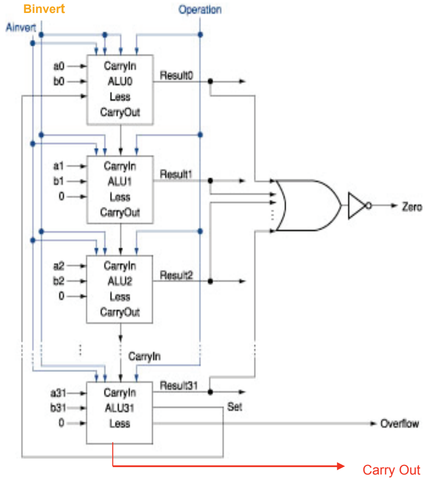


Figure B.5.12
+ Carry Out + Binvert

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 $10001_2 = 17_{10}$

5-bit multiplier $\times\ \underline{10011}_2 = 19_{10}$

$$\begin{array}{r} 10001\\ 10001\\ 00000\\ 00000\\ \underline{10001}\\ 101000011_2 = 323_{10} \end{array}$$

- **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:

multiplicand	**10001**
multiplier	**× 10011**
intermediate product	**0000000000**
add since multiplier bit=1	**10001**
intermediate product	**0000010001**
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

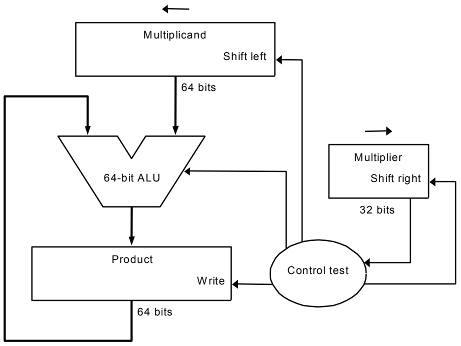


Figure 3.5

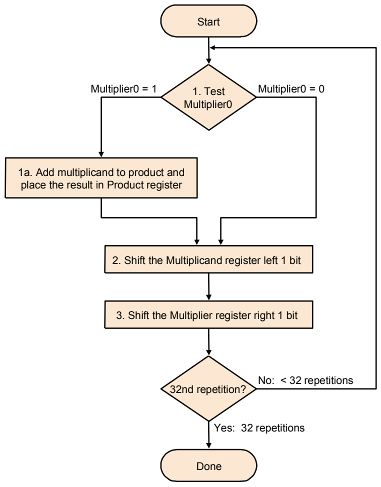
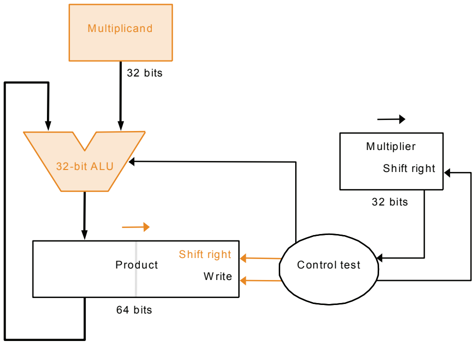
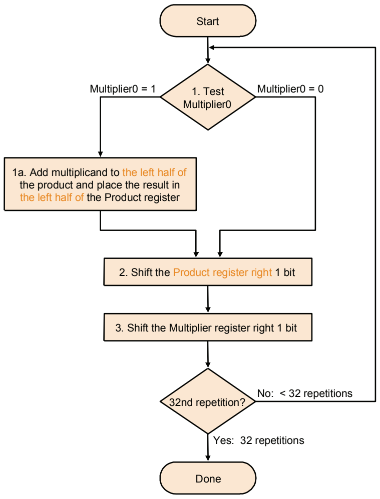


Figure 3.6

Multiplication Hardware: 2nd Version





Multiplication Hardware: 3rd Version

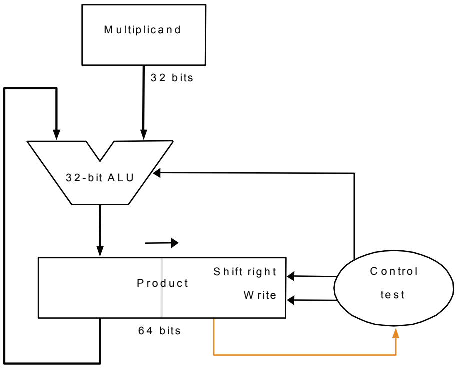
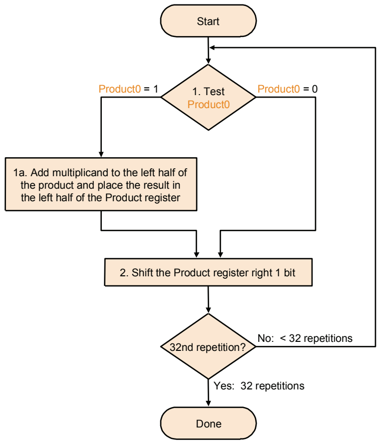


Figure 3.7



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: $1.0_{10} \times 10^{-9}$, but not $0.1_{10} \times 10^{-8}$
- **Similar for binary:**
  - $00101101_2 = 1.0 \times 2^{5}$ or 1.0×2^{101} – normalized notation

Reading Assignment: 3.6

Real Numbers

- *Conversion from real binary to real decimal*

  $-1101.1011_2 = -13.6875_{10}$

  since: $1101_2 = 2^3 + 2^2 + 2^0 = 13_{10}$ and

  $0.1011_2 = 2^{-1} + 2^{-3} + 2^{-4} = 0.5 + 0.125 + 0.0625 = 0.6875_{10}$
- Conversion from real decimal to real binary:

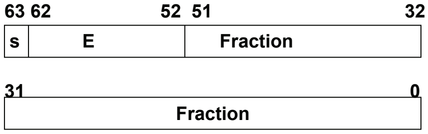
  $+927.45_{10} = +1110011111.01\ 1100\ 1100\ 1100$ …..

927/2 = 463 + ½ ← LSB	0.45 × 2 = 0.9 + 0 ← MSB
463/2 = 231 + ½	0.9 × 2 = 0.8 + 1
231/2 = 115 + ½	0.8 × 2 = 0.6 + 1
115/2 = 57 + ½	0.6 × 2 = 0.2 + 1
57/2 = 28 + ½	0.2 × 2 = 0.4 + 0
28/2 = 14 + 0	0.4 × 2 = 0.8 + 0
14/2 = 7 + 0	0.8 × 2 = 0.6 + 1
7/2 = 3 + ½	0.6 × 2 = 0.2 + 1
3/2 = 1 + ½	0.2 × 2 = 0.4 + 0
1/2 = 0 + ½	0.4 × 2 = 0.8 + 0 ……

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:

- Double precision:



Converting to Floating Point

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

   $(-1)^s \times 1.\text{Fraction} * 2^{\text{Exp}}$

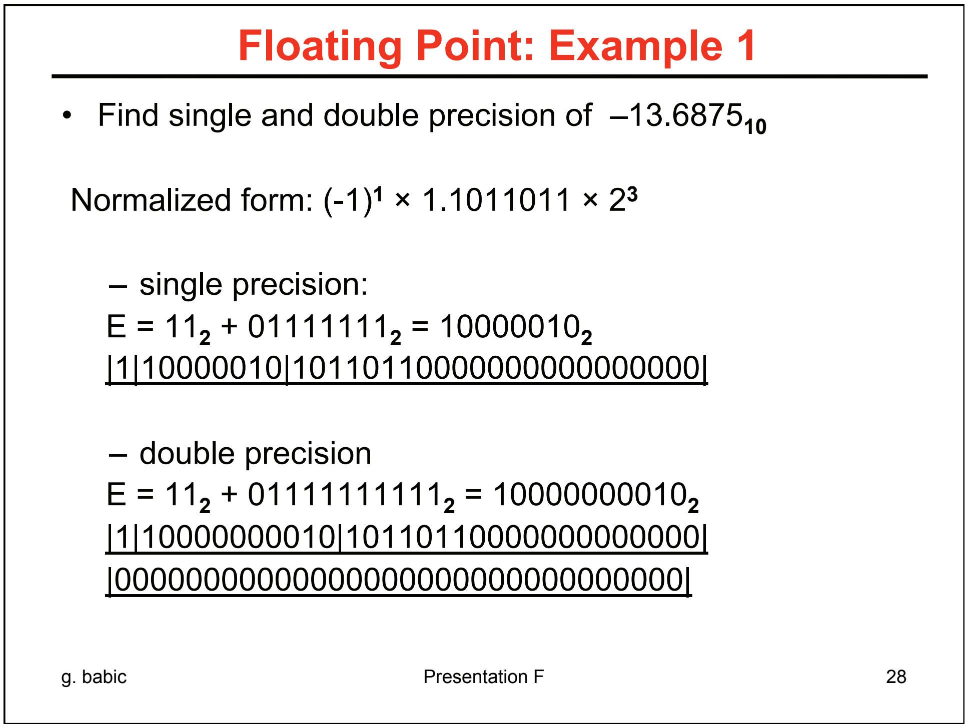
   $-1101.1011_2 = (-1)^1 \times 1.1011011 * 2^3$

   $+1110011111.01\overline{1100} = (-1)^0 \times 1.1100111110\overline{1100} * 2^9$

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

   $E = \text{Exp} + 127_{10} = \text{Exp} + 01111111_2$ for single precision
   $E = \text{Exp} + 1023_{10} = \text{Exp} + 01111111111_2$ for double precision

- E is called a biased exponent - $(-1)^s \times 1.\text{Fraction} * 2^{(\text{Exp-Bias})}$



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:

  $E = 11_2 + 01111111_2 = 10000010_2$

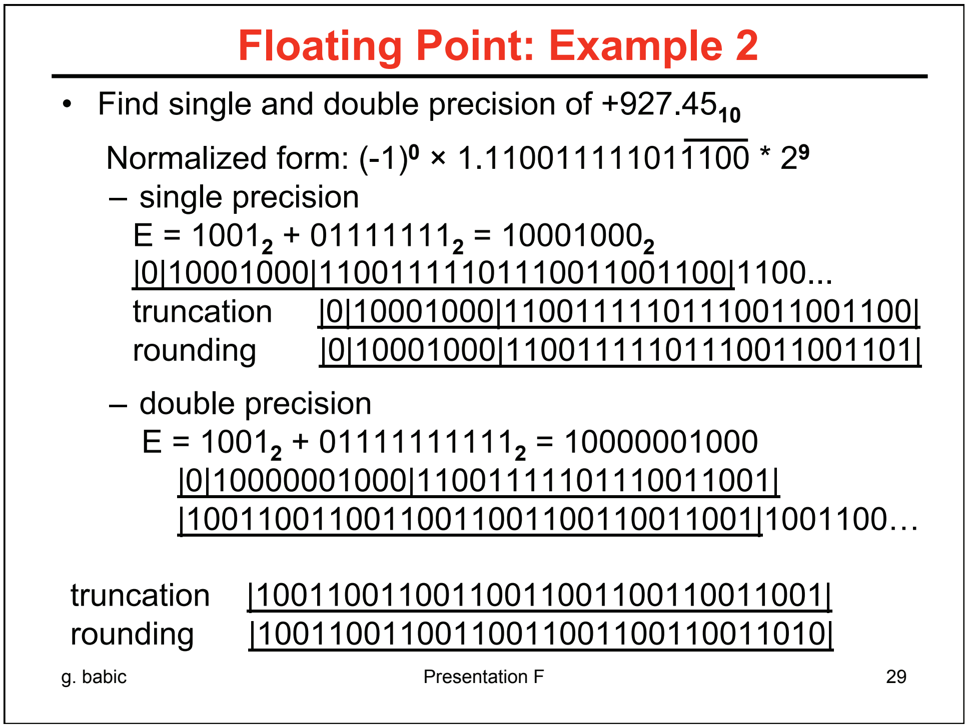
  |1|10000010|10110110000000000000000|

- double precision

  $E = 11_2 + 01111111111_2 = 10000000010_2$

  |1|10000000010|10110110000000000000|

  |00000000000000000000000000000000|



Floating Point: Example 2

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

  Normalized form: $(-1)^0 \times 1.110011111011\overline{1100} * 2^9$

  - single precision

    $E = 1001_2 + 01111111_2 = 10001000_2$

    |0|10001000|11001111101110011001100|1100...

    truncation |0|10001000|11001111101110011001100|

    rounding |0|10001000|11001111101110011001101|

  - double precision

    $E = 1001_2 + 01111111111_2 = 10000001000$

    |0|10000001000|11001111101110011001|

    |10011001100110011001100110011001|1001100…

truncation |10011001100110011001100110011001|

rounding |10011001100110011001100110011010|

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

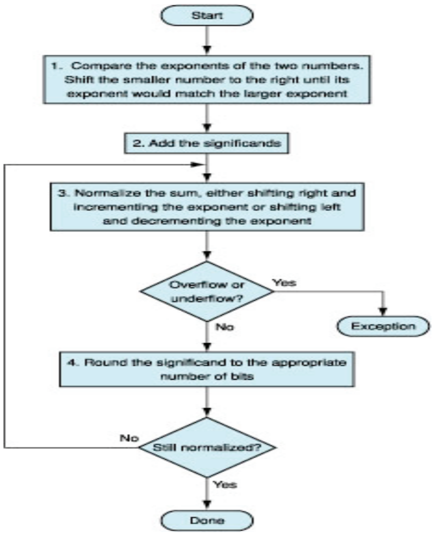


Figure 3.16

Arithmetic Unit for Floating Point Addition

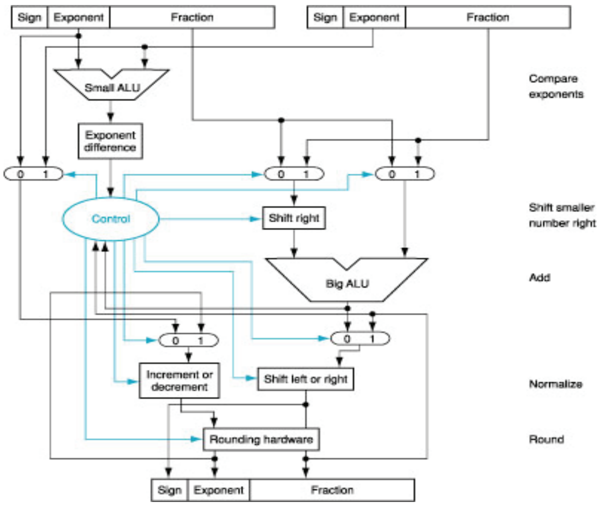
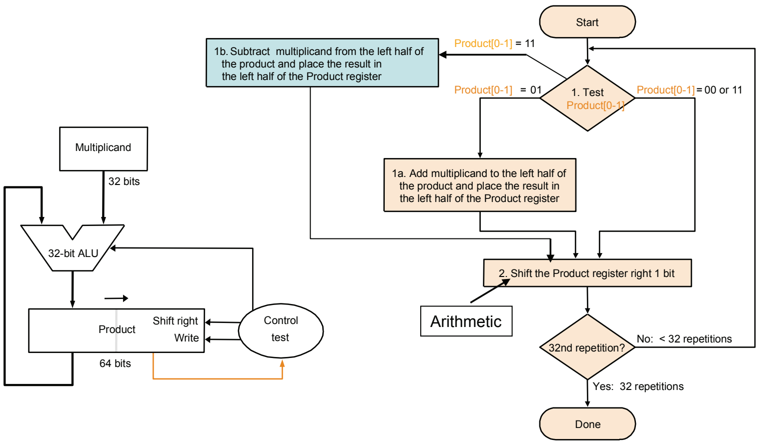


Figure 3.17

Booth’s Algorithm



Booth’s Algorithm: Example

6-bit (signed) multiplicand 110101 = -11_{10} ; Note -110101 = 001011
6-bit (signed) multiplier 011101 = $+29_{10}$

```
Product register  000000 011101  0   assumed for step1.
                  001011             10 – subtract (i.e. add 001011)
                  001011 011101
           shift  000101 101110| 1   step 1. ends
                  110101             01 - add
                  111010 101110
           shift  111101 010111| 0   step 2. ends
                  001011             10 - subtract
                  001000 010111
           shift  000100 001011| 1   step 3. ends
                                     11 - no arithmetic
           shift  000010 000101| 1   step 4. ends
                                     11 - no arithmetic
           shift  000001 000010| 1   step 5. ends
```

Booth’s Algorithm: Example (continued)

```
shift  000001 000010| 1   step 5. ended
       110101             01 - add
       110110 000010|
shift  111011 000001| 0   step 6. ends
```

Result 111011 000001_2 = - 000100 111111_2 = - (256+63) = - 319_{10}