
SI232
Set #10:
More Computer Arithmetic (Chapter 3)

1

Exercise #1

- Do the following assuming 6 bit, two's complement numbers.
When does overflow occur?

$$\begin{array}{r} 010101 \\ + 001101 \\ \hline \end{array}$$

$$\begin{array}{r} 111111 \\ + 111101 \\ \hline \end{array}$$

$$\begin{array}{r} 010011 \\ + 001110 \\ \hline \end{array}$$

$$\begin{array}{r} 010011 \\ + 111110 \\ \hline \end{array}$$

ADMIN

- Course paper topics – due Fri Feb 24 via plain text email

2

Exercise #2

- Do the following assuming 6 bit, two's complement numbers.
When does overflow occur? (note subtraction here)

$$\begin{array}{r} 011101 \\ - 100101 \\ \hline \end{array}$$

$$\begin{array}{r} 111111 \\ - 111101 \\ \hline \end{array}$$

$$\begin{array}{r} 010011 \\ - 001110 \\ \hline \end{array}$$

$$\begin{array}{r} 010011 \\ - 111110 \\ \hline \end{array}$$

3

4

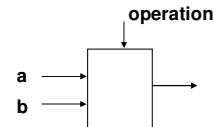
An Arithmetic Logic Unit (ALU)

The ALU is the ‘brawn’ of the computer

- What does it do?

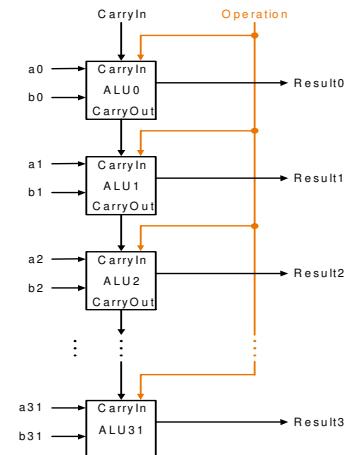
- How wide does it need to be?

- What outputs do we need for MIPS?



5

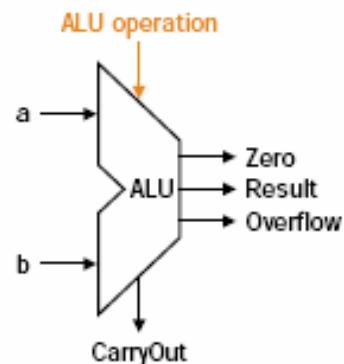
A simple 32-bit ALU



6

ALU Control and Symbol

ALU Control Lines	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR



7

Multiplication

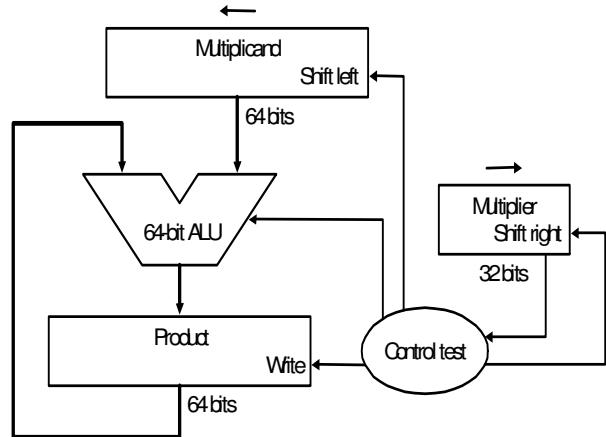
- More complicated than addition
 - accomplished via shifting and addition
- Example: grade-school algorithm

$$\begin{array}{r} 0010 \text{ (multiplicand)} \\ \times 1011 \text{ (multiplier)} \\ \hline \end{array}$$

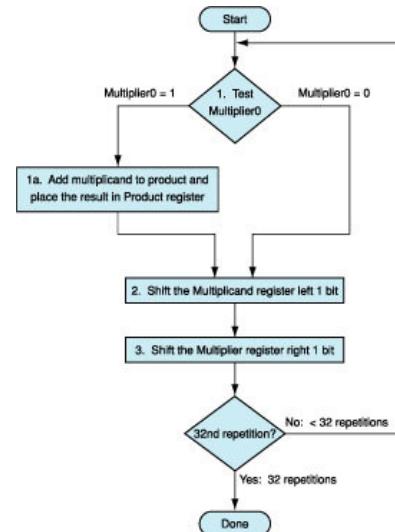
- Multiply m * n bits, How wide (in bits) should the product be?

8

Multiplication: Simple Implementation



9



Using Multiplication

- Product requires 64 bits
 - Use dedicated registers
 - HI – more significant part of product
 - LO – less significant part of product
- MIPS instructions

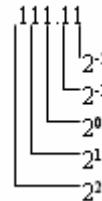

```
mult $s2, $s3
multu $s2, $s3
mfhi $t0
mflo $t1
```
- Division
 - Can perform with same hardware! (see book)

```
div $s2, $s3      Lo = $s2 / $s3
                  Hi = $s2 mod $s3
divu $s2, $s3
```

11

Floating Point

- We need a way to represent
 - numbers with fractions, e.g., 3.1416
 - very small numbers, e.g., .000000001
 - very large numbers, e.g., 3.15576×10^{23}
- Representation:
 - sign, exponent, significand:
 - $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent(some power)}}$
 - Significand always in normalized form:
 - Yes:
 - No:
 - more bits for significand gives more
 - more bits for exponent increases



12

IEEE754 Standard

Single Precision (float): 8 bit exponent, 23 bit significand

31	30	29	28	27	26	25	24	23	22	21	20	.	.	.	9	8	7	6	5	4	3	2	1	0
S	Exponent (8 Bits)												Significand (23 bits)											

Double Precision (double): 11 bit exponent, 52 bit significand

31	30	29	28	.	.	.	21	20	19	18	17	.	.	.	9	8	7	6	5	4	3	2	1	0
S	Exponent (11 Bits)												Significand (20 bits)											
31	30	29	28	.	.	.	21	20	19	18	17	.	.	.	9	8	7	6	5	4	3	2	1	0

More Significand (32 more bits)

Blank space

13

IEEE 754 – Optimizations

- **Significand**
 - What's the first bit?
 - So...
- **Exponent is “biased” to make sorting easier**
 - Smallest exponent represented by:
 - Largest exponent represented by:
 - **Bias values**
 - 127 for single precision
 - 1023 for double precision
- **Summary:** $(-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$

14

Example #1:

- Represent -5.75_{10} in binary, single precision form:
 - Strategy
 - Transfer into binary notation (fraction)
 - Normalize significand (if necessary)
 - Compute exponent
 - Apply results to formula
 - $(-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$

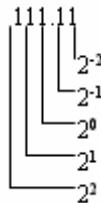
15

16

Example #1:

Represent -9.75_{10} in binary single precision:

- $9.75_{10} =$



Blank space

- Compute the exponent (-1):
 - Remember ($2^{\text{exponent} - \text{bias}}$)
 - Bias = 127
- Formula($-1^{\text{sign}} \times (1 + \text{significand}) \times 2^{\text{exponent} - \text{bias}}$)

31	30	29	28	27	26	25	24	23	22	21	20	.	.	.	9	8	7	6	5	4	3	2	1	0

18

Example #2:

- How about from a bit pattern to a single precision floating point?
1000 1100 1100 1000 0000 0000 0000 0000
- Solution:
 - Sign
 - Exponent
 - Significand
 - Final Result

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
 - IEEE 754 keeps two extra bits, guard and round
 - four rounding modes
 - positive divided by zero yields “infinity”
 - zero divide by zero yields “not a number”
 - other complexities
- Implementing the standard can be tricky

19

20