

198:211 Computer Architecture

Lecture 10 Fall 2010

- Topics:
 - Integer Arithmetic Chapter 2.3
 - Overflow
 - Integer Multiplication and Division

1

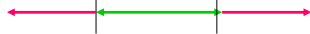
Integer Arithmetic

- In a computer, all numbers are represented with finite number of bits
 - This determines maximum value that can be represented
- `int x, short int sx;`
- What is `sizeof(x)`, `sizeof(sx)`?
 - 2 bytes = 16 bits, 4 bytes = 32 bits
 - Max integer value is given by 2^{16} or 2^{32}
- Lets look at unsigned numbers
- Range of values for `unsigned short int` is
 - 0 to 65 535
- Range of values for `short int` is (2 bytes)
 - -32768 to +32767
- Range of values for `unsigned int` is (4 bytes)
 - 0 to 4 294 967 296
- Range of values for `int` is
 - -2 147 483 648 to 2 147 483 647

2

Implication of finite number of bits

- All numbers have range $\pm 2^W$, including answers



- When two (+ve or -ve) numbers are added result may end up in the red region
- The answer will be wrong. Need to detect
- On occasions left to the programmer to detect
- On other occasions, hardware will provide the necessary information

3

Unsigned Binary addition

```

  5 0101
+ 6 0110
-----
11 1011

```

```

  6 0110
- 5 0101
-----
 1 0001

```

```

  8 1000
+10 1010
-----
18 1 0010
 2  0010

```

Sum of 4-bit operands may require 5 bits .. **Carry out**
 In general when adding 2^w bits may require 2^{w+1} bits
 We need to place a limit on all integers
 If we place a limit on all integers, we need to ignore the carry out
 If we ignore the carry out then it is modular arithmetic modulo 2^w
 (10+8) modulo 16 is 2 !!!

4

Unsigned binary addition

```
unsigned short int ux;
unsigned short int uy;
unsigned short int ans;
ux=6;uy=5;
ans=uy-ux;
printf ("Value of answer is %d\n", ans);
```

- What should happen?
 - Should this operation even be allowed?
- What will happen?
 - Guess
- Why does this happen?
 - In C, $-y$ is represented as 2s complement

5

overflow

- An overflow occurs when the result cannot fit within the word-size limits of the data type
- When executing C programs, overflows are not signaled as errors!
- It is onus on the programmer to determine if an overflow has occurred
- When two unsigned numbers s , x are added
 - $s+x \geq s$ and $s+x \geq x$
- Overflow has occurred if the
 - sum is $s+x < s$ or $s+x < x$

6

2's complement addition

- Recall, 2s complement $-ve$ numbers are represented with MSB=1
- Addition of 2s complement is just binary addition with carry ignored
- Subtraction, invert the sign of the subtrahend and add
- Carry out - a bit out of MSB
- Carry in - a bit added into MSB

-7 1001	-5 1011	4 0100
+ 5 0101	+ -2 1110	- (-2) 0010
-----	-----	-----
-2 1110	-7 1 1001 ignore carry out	6 0110

7

overflow

- When two numbers of w bits are added, the result could need $w+1$ bits.
- Since, MSB of 1 represents negative numbers, a carry into the MSB creates problems

<p>Example 1</p> <table style="margin-left: 20px;"> <tr><td style="text-align: right;">6 0110</td></tr> <tr><td style="text-align: right;">+ 5 0101</td></tr> <tr><td style="text-align: right;">-----</td></tr> <tr><td style="text-align: right;">-5 1011</td></tr> <tr><td style="text-align: right;">Carry out 0</td></tr> <tr><td style="text-align: right;">Carry in 1</td></tr> </table>	6 0110	+ 5 0101	-----	-5 1011	Carry out 0	Carry in 1	<p>Example 2</p> <table style="margin-left: 20px;"> <tr><td style="text-align: right;">- 6 1 0 1 0</td></tr> <tr><td style="text-align: right;">+ (-6) 1 0 1 0</td></tr> <tr><td style="text-align: right;">-----</td></tr> <tr><td style="text-align: right;">4 1 0 1 0 0</td></tr> <tr><td style="text-align: right;">Carry out 1</td></tr> <tr><td style="text-align: right;">Carry in 0</td></tr> </table>	- 6 1 0 1 0	+ (-6) 1 0 1 0	-----	4 1 0 1 0 0	Carry out 1	Carry in 0
6 0110													
+ 5 0101													

-5 1011													
Carry out 0													
Carry in 1													
- 6 1 0 1 0													
+ (-6) 1 0 1 0													

4 1 0 1 0 0													
Carry out 1													
Carry in 0													

8

overflow

- In example 1, we added two positive numbers, that resulted a carry into the MSB
 - (in 2's complement), MSB of 1 means negative number
 - Positive overflow
- In example 2, we added two negative numbers, that resulted a carry out of MSB (ignore)
 - A carry out of MSB leaves MSB bit as 0. A MSB of 0 means a positive number in 2's complement
 - Negative overflow

9

Normal case

3 0011	5 0101	-4 1100	-2 1110
+2 0010	-3 1101	5 0101	-3 1101
+5 0101	+2 1 0010	1 1 0001	-5 1 1011
Carry out 0	Carry out 1	Carry out 1	Carry out 1
Carry in 0	Carry in 1	Carry in 1	Carry in 1

- Adding a +ve number and a -ve number will not cause a problem
- Adding two +ve numbers without carry in and carry out is also OK
- Adding two -ve numbers with carry in and carry out is also OK

10

Detecting overflow

- Overflow occurs when
 - Adding two positive numbers and the result is negative and vice versa
 - Subtracting a negative number from a positive number and the result is negative
 - Subtracting a positive number from a negative number and the result is positive

11

Detecting overflow

Operation	First operand	Second operand	Overflow condition
X+Y	≥ 0	≥ 0	< 0
X+Y	< 0	< 0	≥ 0
X-Y	≥ 0	< 0	< 0
X-Y	< 0	≥ 0	≥ 0

12

Actions on overflow

- Detect in hardware
- Raise flags or special bits to indicate overflow
 - Compiler generated code for a Language can make use of these flags to inform programmer
 - Ignore overflow and leave it to programmer to implement checks to detect overflow

13

Unsigned multiplication

- When two numbers, each w bits long are multiplied, the result can be 2^*w bits long
- If the result also needs to be w bits long, then w bits need to be discarded
- The w Most Significant Bits (MSBs) are discarded
- Unsigned Multiplication

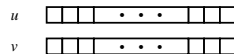
```
unsigned int ux, uy, up;
up = ux * uy;
```

- Truncates product to w -bit number $up = \mathbf{UMult}_w(ux, uy)$
- Modular arithmetic: $up = (ux \cdot uy) \bmod 2^w$
- Else, the result needs to be 2^*w bits long

14

Unsigned Multiplication in C

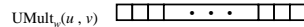
Operands: w bits



True Product: 2^*w bits



Discard w bits: w bits



- Standard Multiplication Function
 - Store true product in 2^*w bits
 - Or if product is also w bits long, Ignore high order w bits
- Implements Modular Arithmetic

$$\mathbf{UMult}_w(u, v) = u \cdot v \bmod 2^w$$

15

Multiplication terms (recall grade school)

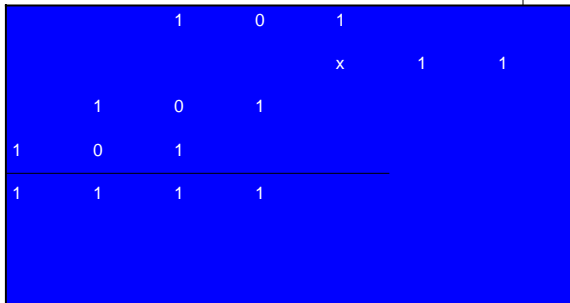
- Decimal Multiplication
- Multiplicand
- Multiplier
- Partial products
- Final sum
- Each digit in the multiplier has different place values
- The partial product is shifted left for each consecutive digit in the multiplier

```

123 x 101
 123
000
123
-----
12423
```

16

Binary (unsigned) Multiplication



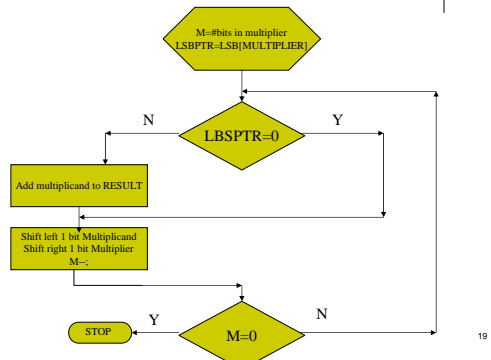
17

Algorithm

- Step1: Check LSB of multiplier
- Decision Step 2: If 1, then add multiplicand to result
If 0, add 0 to result
- Step 3: Shift multiplicand left by 1 bit
- Step 4: Shift multiplier right by 1 bit
- Stop Condition: Repeat for number of bits in multiplier

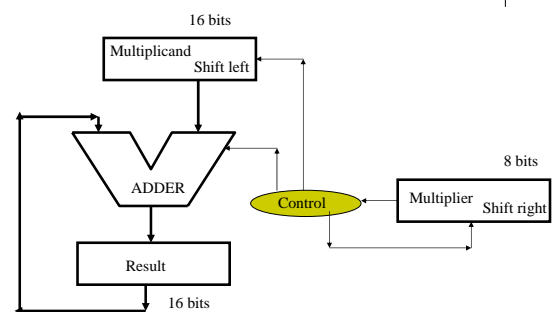
18

Binary multiplication



19

Binary multiplier



20

Signed Multiplication

- Determine sign of product based on sign of multiplicand and multiplier
 - $ANS = X * Y$
 - ANS is +ve if X and Y have the same sign else ANS is -ve
- Hence
 - Convert multiplier, multiplicand to positive numbers
 - Multiply unsigned numbers as before
 - Compute sign, convert product accordingly
- Next, 2s complement multiplication

21

Sign extension

- For +ve numbers, you can pad as many 0s to the MSB as you want without changing value
- For 2's complement, in order to change from 4 bits to 8 bits or from 8 bits to 16 bits, append additional bits on the left side of the number. Fill each extra bit with the value of the number's most significant bit (the sign bit).

Signed integer	4-bit representation	8-bit representation	16-bit representation
+1	0001	00000001	0000000000000001
-1	1111	11111111	1111111111111111

22

2s complement multiplication

- With sign extension
- $-3 * -3$
- 111101 x 111101
- Ans = 2^w LSBs
- Takes the first 6 Least Significant digits

- Note
 - $1 + 1 = 10$
 - $1 + 1 + 1 = 11$
 - $1 + 1 + 1 + 1 = 100$
 - $1 + 1 + 1 + 1 + 1 = 101$

111101 x 111101

						1	1	1	1	0	1	
					0	0	0	0	0	0		
			1	1	1	1	0	1				
		1	1	1	1	0	1					
	1	1	1	1	0	1						
1	1	1	1	0	1							
.	0	0	0	1	0	0	1

23

Fast Multipliers

- Number of tricks used to improve multiplication
- Booth multiplication algorithm

24

Integer division

$$\begin{array}{r}
 21 \\
 12 \overline{) 262} \\
 \underline{240} \\
 22 \\
 \underline{12} \\
 10
 \end{array}$$

Initially: dividend has 3 digits, divisor has 2 digits

- #times=#of digits(DIVIDEND) - #ofdigits(DIVISOR) + 1 → # of iterations
- Align divisor to dividend (by shifting left), so that MSB of dividend and divisor has the same place value
- STEP: Find the largest integer k s.t. k*divisor <= dividend
- Make k LSB of quotient
- Remainder = dividend - k*divisor
- Shift left 1 bit quotient (multiply by 10)
- Shift right 1 bit divisor (divide by 10)
- #times--
- Repeat Until #times=0
- Answer is Q, Remainder

25

Binary Division

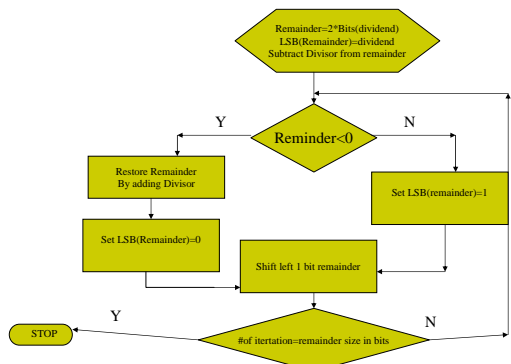
- $53 \div 7 = 7 \text{ rem } 4$

$$\begin{array}{r}
 0111 \\
 111 \overline{) 110101} \\
 \underline{111000} \\
 011100 \\
 \underline{011001} \\
 001110 \\
 \underline{001011} \\
 000111 \\
 \underline{000100} \\
 000100
 \end{array}$$

<0, so shift divisor to right, q=0
 >0 subtract, shift left q and q=1
 shift divisor to right
 >0 subtract, shift left q and q=1
 shift divisor to right
 >0 subtract, shift left q and q=1
 Stop: rem = 100

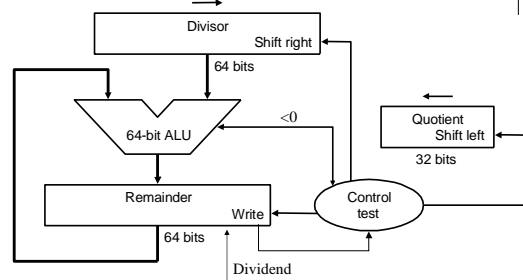
26

Binary division



27

Integer Division



Subtract divisor from remainder (initially rem = dividend)
 Result ≥ 0 , remainder ← result, 1 into Q LSB
 Result < 0 , write 0 into Q LSB
 Shift divisor to right, shift Q left

28