Outline

- **Representing Negative Numbers**
  - **Overflow**
- **Shift and multiplication**

Sign/Magnitude Representation

- So far, we've dealt with unsigned numbers
  - How are negative numbers represented on a computer?
- One way to represent negative number is called sign/magnitude.
- For an N-bit binary number:

1 sign bit, N-1 magnitude bits

Sign bit is the most significant bit (leftmost)

  - Negative number: sign bit = 1
  - Positive number: sign bit = 0

- Rest of the bits are numerical value of the number

Sign/Magnitude Representation

1 sign bit, N-1 magnitude bits

Sign bit is the most significant bit (leftmost)

- Rest of the bits are numerical value of the number
- Hence, in a binary number with eight bits, the magnitude can range from 0000000 (0) to 1111111 (127)
- Thus numbers ranging from -127_{10} to $+127_{10}$ can be represented, once the sign bit (the eighth bit) is added

Problems

- Two representations of zero [0000, 1000]
- Difficulties in arithmetic operations:
  - 1101 (-5) + 0011 (3) = 0000

2's Complement Representation

The most significant bit still indicates the sign

Same as unsigned binary, but the value of the most significant bit is -2^{N-1}

Example: What is the value of 10110

- Unsigned: 10110 = 16 + 4 + 2 = 22
- Twos complement 10110 = -16 + 4 + 2 = -10

Single representation of zero (0000)

Arithmetic works fine

- 1011 (-5) + 0011 (3) = 1110 (-8 + 4 + 2) = -2

2's Complement Representation

How to convert a 2's complement binary to its decimal value?

- Assume in an 8-bit two's-complement numeral system

8-Bit Two's Complement ($-128 \leq x < 127$)

	MSB							LSB
Bit	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	0^{th}
Weight	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- What does 0000 0001 represent? 1
- What does 1111 1111 represent? -1
- What does 0101 1011 represent? 91

2's Complement Representation

How to convert a positive 2's complement binary to its negative value?

E.g., what is the 2's complement of the binary number 011010

First method	**Complement each digit to get the first complement**	100101
	Add one to the first complement.	100110
Second Method	Start from the least significant bit, locate the first digit with value 1	011010
	Complement each digit after the first digit with value 1.	100110

2's Complement Representation

- The two's-complement numeral system is the most common method of representing signed integers on computers
  - Advantages: The fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those for unsigned binary numbers
  - This property makes the system both simpler to implement and capable of easily handling higher precision arithmetic
  - Also, zero has only a single representation 0, eliminating the subtleties associated with negative zero
- All modern processors primarily use two's complement
  - Includes all the MIPS's integer arithmetic instructions
  - Different instruction variants for signed and unsigned

Sign Extension

- Sign extension is the operation of increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value
  - This is done by appending digits to the most significant side of the number, following a procedure dependent on the particular signed number representation used
- In MIPS, all arithmetic immediate values are sign-extended
- e.g., If six bits are used to represent the number "00 1010" (decimal +10) and the sign extension operation increases the word length to 16 bits, then the new representation is simply "0000 0000 0000 1010“ – padding the left side with 0s

Sign Extension

- Solution: If ten bits are used to represent the value "11 1111 0001" (decimal -15) in two's complement, and the sign extension operation increases the word length to 16 bits, the new representation is "1111 1111 1111 0001" – padding the left side with 1s

Long Addition

Long Addition in Decimal

$$
\begin{array}{rcccccccc}
 & 6 & 2 & 9 & 5 & 1 & 4 & 1 & 3 \\
+ & & 2 & 8 & 1 & 8 & 2 & 8 & 1 & 7 \\
\hline
 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & \text{Carry} \\
\hline
 & & 9 & 1 & 1 & 3 & 4 & 2 & 3 & 0
\end{array}
$$

0111 0110 + 1101 0101

Long Addition in Binary

$$
\begin{array}{rcccccccc}
 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & = 118 \\
+ & & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & = 213 \\
 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & \text{Carry} \\
\hline
 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & = 331
\end{array}
$$

Overflow

- One issue in computer arithmetic is dealing with finite amounts of storage, such as 32-bit MIPS registers
- Overflow occurs when the result of an operation is too large to be stored
- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated. E.g. 1111+0001

Overflow for addition

- For a signed number, overflow occurs when
  - Adding two positives yields a negative. E.g. (0111 + 0001)
  - Or, adding two negatives gives a positive. E.g. (1000 + 1000)
  - Or, subtract a negative from a positive gives a negative
  - Or, subtract a positive from a negative gives a positive

- One way to detect overflow is to check whether the sign bit is consistent with the sign of the inputs when the two inputs are of the same sign – if you added two positive numbers and got a negative number, something is wrong, and vice versa

Outline

- **Representing Negative Numbers**
  - **Overflow**
- **Shift and multiplication**

Shift Operations

- Shift operations shift a word a number of places to the left or right
- Bits which are shifted out, just disappear
- E.g. on 4 bits: 1011 shifted left 1 bit, results in 0110.

- Logical shift: in right shift, 0s are filled in empty positions.
  - E.g., 1011 shift right 1 bit => 0101
- Arithmetic shift: in right shift, sign bits are filled in empty positions.
  - E.g., 1011 shift right 1 bit => 1101

Shift Operations

- For unsigned and 2's complement numbers, logical/arithmetic shift left by 1 is multiplication by 2 if there is no overflow
  - e.g. 0011 (3) shifted left 1 bit results in 0110 (6)
  - e.g. 1100 (-4) shifted left 1 bit results in 1000 (-8)

- For unsigned numbers, logical shift right by 1 is division by 2, ignoring remainder.
  - e.g. 0101 (5) shifted right 1 bit results in 0010 (2)
  - For 2's complement numbers, logical shift right by 1 does not correspond to dividing 2.
  - e.g. 1100 (-4) shifted right results in 0110 (6)

- For 2's complement numbers, arithmetic shift right by 1 performs division by 2.
  - e.g. 1100 (-4) shifted right arithmetical results in 1110 (-2)

Multiplication

- Binary multiplication similar to decimal multiplication – multiplying with each bit and add the (shifted) results together.
- If we multiply an m-bit with an n-bit number, we need $m+n$ bits to store the result
  - Multiplying 4-digit numbers needs 8 digits
  - e.g. 1101_2 x 1011_2

Long Multiplication in Binary

				1	1	0	1	$a = 13_{10}$
$\times$				1	0	1	1	$b = 11_{10}$
				1	1	0	1	
			1	1	0	1		
		0	0	0	0			
$+$	1	1	0	1				Partial Sums
$=$	1	0	0	0	1	1	1	$c = 143_{10}$

Multiplication Instructions

- MIPS stores the 64 bit result of the multiplication of two 32 bit registers in two special 32-bit registers Hi and Lo
  - Bits 32 to 63 are stored in Hi
  - Bits 0 to 31 are stored in Lo

Find a similar question on division from our Week3's lab

- There are two instructions: multiply (`mult`) and unsigned multiply (`multu`)
- e.g. `mult $s1,$s2`
  - Performs a signed multiplication of the registers $s1 and $s2, storing the result in Hi and Lo
- We can use the instructions: move from low (`mflo`) and move from high (`mfhi`), e.g. `mflo $s0`
  - Store the lower 32-bit of the result of the previous multiplication operation in register $s0

Multiplication Instructions

- Two small integers are multiplied. Where is the result?
- If the result is small enough all the significant bits will be contained in Lo and Hi will contain all zeros
- Since it is common that we are only interested in the 32 bit result of a multiplication, the sequence

```
mult $s1,$s2
mflo $s0
```

  can be encoded as one operation:

```
mul $s0,$s1,$s2
```

- Note `mul` does not check for overflow, but a pseudo-instruction multiply with overflow `mulo` does
- Think about it：How to check overflow for multiplications?