0447 Week 2 Recitation

January 16th, 2024

Important Information

- Email: kpb42@pitt.edu
- Office Hours:
  - Mondays: 9:30am - 11:00am in SENSQ 5806
  - Thursdays: 4:30pm - 6:00pm in SENSQ 5806
  - Fridays: 12:00pm - 1:00pm in SENSQ 5806
- Announcements for this recitation will be posted on canvas

Today’s Topics

- Number Bases
  - Binary Numbers + conversions
  - Hex Numbers + conversions
- Numeric Representation
  - Signed & Unsigned Ints
  - Extension & Truncation
  - Math
- Lab 1
  - MIPS introduction
  - Lab Overview

Numeric Bases

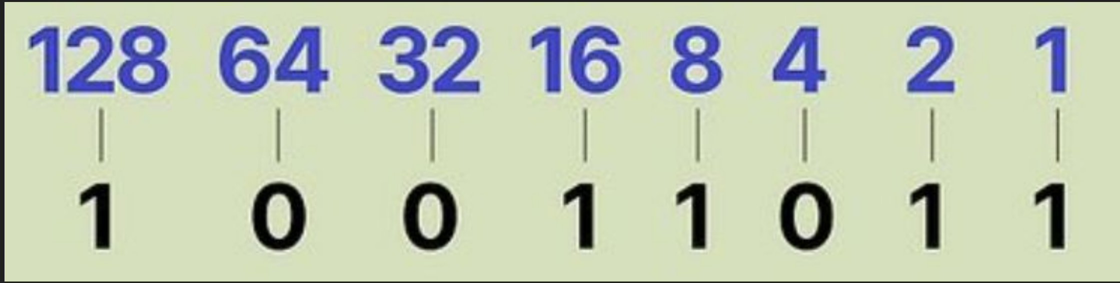
Binary Numbers

- Base 2 number system
- Each digit is 0 or 1
- You may see these numbers denoted with the prefix *0b*

```
int example = 0b1001;
```

Binary to Decimal

- Starting from the least significant digit (leftmost), each digit can be seen as a power of 2 (Starting from 2^0!)
- Then, we multiply the power of 2 by the binary digit (0 or 1) and add them together



Binary to Decimal Example

Convert 0b1001 to decimal:

1 0 0 1 => 1 0 0 1

2^3 2^2 2^1 2^0 => 8 4 2 1

(1 * 8) + (0 * 4) + (0 * 2) + (0 * 1) = 8 + 0 + 0 + 1 = ***9***

Hexadecimal Numbers

- Base 16 number system
- 16 digits from 0 to F (15)
- Denoted with the prefix *0x*

```
int theCoolerExample = 0xFFA5;
```

Hexadecimal to Decimal Table

MATH MONKS

Hexadecimal (Base 16)	Decimal (Base 10)	Binary (Base 2)
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Hexadecimal to Decimal

Convert 0xAAA to decimal

A	A	A	=>	10	10	10
16^2	16^1	16^0	=>	256	16	1

(10 * 265) + (10 * 16) + (10 * 1) = 2650 + 160 + 10 = <u>***2820***</u>

Binary to Hexadecimal

- 1 hexadecimal digit can represent 4 binary digits
- To convert from hex to binary, group 4 binary digits for each hex digit
- Example: convert 11001001 to hexadecimal

=> 1 1 0 0   1 0 0 1

=> (1 * 2^3) + (1 * 2^2) + 0 + 0 (1 * 2^3) + 0 + 0 + (1 * 2^0)

=> (8 + 4) (8 + 1)

=> C9

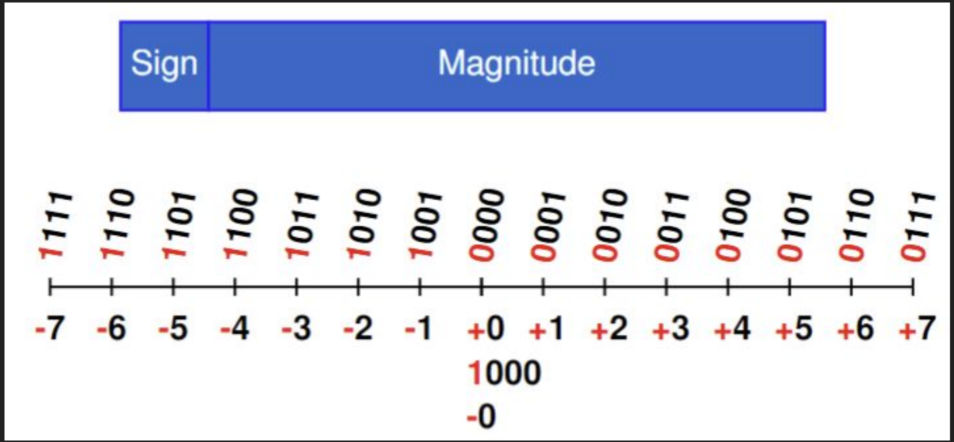
Unsigned and Signed Integers

Unsigned and Signed Integers

- Unsigned: for non-negative numbers (natural numbers) only
- Signed: for positive and negative numbers
- All integers have a range of values they can represent based on the number of bits
  - For example: a byte has 8 bits, meaning it can represent numbers from 0 (00000000) to 255 (11111111)
  - To represent 256, we would need another digit

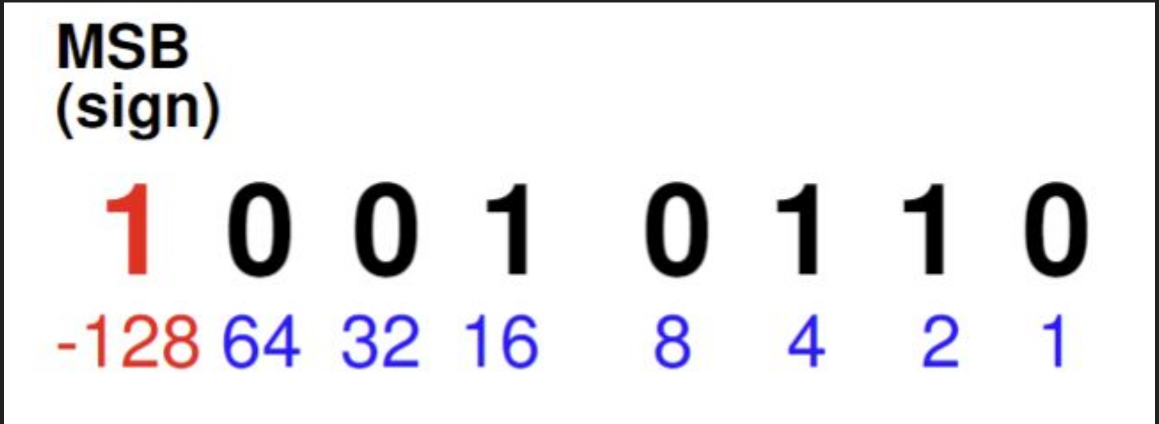
Signed Integer Representation

- Sign Magnitude: reserve leftmost bit to represent the sign of a number
  - Easy negation -> just flip the bit
  - However, this means that 0 is represented by 2 numbers
    - If we had a 4-bit number using sign magnitude, 1000 = 0 and 0000 = 0
    - Wasting space = bad!



Signed Integer Representation

- Two’s complement: give the leftmost bit negative weight
  - Slightly imbalances the number line, we get 1 more negative number
    - Positive side of the number line “loses” a number because we need to represent 0
  - For a 4-bit number, we can represent -8 (1000) to 7 (0111)



Extension & Truncation

Extension & Truncation

- Because the number of bits varies between variable types, *Things Happen*™ when converting between them
- Specifically:
  - Data may need to be resized to fit a larger size
  - Data may be lost in converting to a smaller size

Extension

- Occurs when a variable with a smaller size is set to a larger 1
- Resizing without changing the data is different for signed and unsigned integers
- Zero Extension: putting “leading 0’s” before a number to extend it for a larger data type
  - For unsigned numbers only!
- Sign Extension: extend the smaller numbers signed bit to fit a larger data type
  - For signed numbers

Extension Examples

- Convert unsigned 4-bit number 0111 (7) to 8 bit
  - To fill the 4 extra bits, zero extend

0111 => *0000* 0111 (still 7!)

- Convert signed 4-bit number 1100 (-4) to 8 bit
  - To fill the 4 extra bits, sign extend

1100 => *1111* 1100 (still 4!)

Truncation

- Cutting off the leftmost bits of a number
- Occurs when a number’s bits are greater than the max bits in a variable

```
// for example, look at this conversion
byte b = 10; // byte has 8 bits
int i = b; // int has 32 bits, no data lost
```

```
// what about the other way?
int i = 10; // 32 bits
byte b = i: // error: possible lossy conversion
```

Lab 1: Landing on MARS

Lab 1: Landing on MARS (Due 1/28)

- Make sure you install the MARS from the software section of canvas!
- This lab is mostly familiarizing you with the basics of MIPS
- Some of the components of the lab:
  - General assembly coding practices
  - Loading / storing memory into registers
  - Basic arithmetic
  - System calls: instructions that directly ask the system to do something
    - More on this in 449 & 1550

- Come see me in office hours or email me if you have any questions!

An extremely brief intro to MIPS assembly

- Assembly Language: human-readable, textual representation of machine code
- *To Do Things™*, computers use Registers: small, fast hardware memory inside the CPU
  - MIPS has 32 registers
- Register Types:
  - a0 - a3: argument variables
  - v0, v1: result variables
  - t0 - t9: temporary variables
  - s0 - s7: saved variables

register	assembly name	Comment
r0	$zero	Always 0
r1	$at	Reserved for assembler
r2-r3	$v0-$v1	Stores results
r4-r7	$a0-$a3	Stores arguments
r8-r15	$t0-$t7	Temporaries, not saved
r16-r23	$s0-$s7	Contents saved for later use
r24-r25	$t8-$t9	More temporaries, not saved
r26-r27	$k0-$k1	Reserved by operating system
r28	$gp	Global pointer
r29	$sp	Stack pointer
r30	$fp	Frame pointer
r31	$ra	Return address

There is a load more in store

- To put values into registers, we can use load instructions
- To put a value into a register, we use a load instruction
  - *li* (load immediate): used for immediate values
  - Other load instructions (like *lw*) load memory into registers

```
li v0 10 # v0 = 10
```

```
lw a0, x # a0 = x
```

- To save values into memory, we use a store instruction
  - Various types of store instructions based on the size of what is being stored
    - *sw (store word)* stores a word (32-bits) into memory

Arithmetic

- Note: when working with immediates, add an ‘i’ to the end of the instruction!
  - add => addi, etc.

```
add t2, t0, t1     # t2 = t0 + t1
sub t2, t0, t1     # t2 = t0 - t1
mul t2, t0, t1     # t2 = t0 * t1
div  t2, t0, t1    # t2 = t0 / t1
rem t2, t2, t1     # t2 = t2 % t1
```