CS 241 Lecture 1

Introduction and Binary Encoding

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

About the course

- www.student.cs.uwaterloo.ca/~cs241
  - **Read the outline!** (policies, due dates, etc.)
  - Read the Announcements
  - Read everything else on the main webpage
  - Make sure you can get on Piazza!

About the course

- Assignments
  - Start assignments early and don't fall behind!
  - 8 assignments in total with many subparts.
  - See outline for due dates
  - Slip days available: see outline

About the course

- Content of this course in *course notes*, posted to web site
- Course = notes. Anything I mention here that isn't in the notes isn't in the course.
- These slides will be posted on the course web site along with the module text (in the "course notes" tab)
- Please report any typos to me!

Marking

- Assignments: 40%
- Midterm: 20%
- Final Exam: 40%

- You must pass the weighted exam average to pass the course otherwise your final average is your exam average.

Marmoset

- Public tests (aka “sanity tests”)
- Release tokens
  - Reset once per hour
- Starting early maximizes your chances of success on your assignments!

- Your program must run correctly on the linux.student.cs environment.

Personnel

- Instructors:
  - Sylvie Davies <sylvie.davies@uwaterloo.ca>
  - **Gregor Richards** <gregor.richards@uwaterloo.ca>

- ISAs and IAs (See the outline for more details):
  - Quan Cheng Taian <cs241@uwaterloo.ca>
  - Hassan Hashmi <cs241@uwaterloo.ca>
  - Pedro Oliveira <pjmcoliveira@uwaterloo.ca>

- Instructional Support Coordinator:
  Gang Lu <glu@uwaterloo.ca>

Other Resources

- Textbooks: None required. Get the "dragon book" if you want another perspective.
- Discussion Forum: Piazza
  - Rule 1: Piazza is not Reddit. Be courteous.
  - Rule 2: Post questions in the appropriate folders.
  - Rule 3: Read first, search second, post last

Purpose of the course

- Assemble a compiler for a fairly complete (but small) language
- Learn MIPS (assembly and machine code)
- Write a program that reads a program and outputs a program
- Most fundamentally, this course is about *abstraction*

What’s in a name?

Foundations of Sequential Programs

- What is a sequential program? (single-threaded; not concurrent or parallel)
- What really happens when I compile and run a program?
- How does a computer take code and turn it into something it can utilize?
- By the end of the course, there should be very little mystery left about computers or computer programs.

Basic Definitions

Definition

A **bit** is a **bi**nary digi**t**. That is, a 0 or 1 (off or on)

Definition

A **nibble** is 4 bits.

Example: 1001.

Definition

A **byte** is 8 bits.

Example: 10011101.

Hexadecimal Notation

Definition

A word is a machine-specific grouping of bytes. For us, a word will be 4 bytes (32-bit architecture) though 8-byte (or 64-bit architectures) words are more common now.

- Example: 10011101100111011001110110011101.
- It can be hard to read words in binary. Can we make the notation more compact? Yes!

Hexadecimal Notation

Definition

The base-16 representation system is called the **hexadecimal** system. It consists of the numbers from 0 to 9 and the letters a, b, c, d, e and f (which convert to the numbers from 10 to 15 in decimal notation.

Example: The binary number 10011101 will convert to 9d in hexadecimal.

- Sometimes we denote the base with a subscript like 10011101_2 and $9d_{16}$.
- Also, for hexadecimal, you will routinely see the notation 0x9d. (The 0x denotes a hexadecimal representation in computer science).
- Note that each hexadecimal character is a nibble (4 bits).

Conversion Table

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binary	Decimal	Hex
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

Note: Upper case letters are also used for hexadecimal notation.
Context should make things clear.

Binary Numbers, What are they Good For?

What do bytes represent?

- Numbers (but **what** number?)
- Characters (but **what** character?)
- Garbage in memory
- Instructions! (Parts of instructions in our case. Words, or 4 bytes, will correspond to a complete instruction for our computer system).

Bytes as Binary Numbers

- We will discuss two types:
  - Unsigned (non-negative integers)
  - Signed integers

- However, there are many others (floating point, algebraic, etc.)

Unsigned Integers

This is a positional number system that works like a normal binary system.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

The value of a number stored in this system is the binary sum, that is

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0$$

For example,

$$01010101_2 = 2^6 + 2^4 + 2^2 + 2^0 = 64 + 16 + 4 + 1 = 85_{10}$$

or

$$\begin{aligned} 11111111_2 &= 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 \\ &= 255_{10} \end{aligned}$$

Unsigned Integers

Arithmetic is done in the ordinary way:

```
  1111 111
   01001001
+  01111111
-----------
   11001000
```

Watch out for overflow errors!

Converting to Binary

Goal: Write 38 in binary.

- One way: Take the largest power of 2 less than 38, subtract and repeat.
- For example, 32 is the largest power of two less than 38, subtracting gives 6. Next, 4 is the largest power of two less than 6 and subtracting gives 2. This is a power of two hence $38 = 32 + 4 + 2 = (100110)_2$.
- Another way is to constantly divide by 2:

Number	Quotient	Remainder
38	19	0
19	9	1
9	4	1
4	2	0
2	1	0
1	0	1

- ..and in binary (reading bottom to top) this is $(100110)_2$.

Brief Explanation

Consider:

$$N = b_0 + 2b_1 + 2^2 b_2 + \ldots$$

The remainder when dividing N by 2 gives the b_0 value. After doing $(N - b_0)/2$, we end up with

$$\frac{N - b_0}{2} = b_1 + 2b_2 + 2^2 b_3 + \ldots$$

and we can repeat the process. (This is why we have to read bottom-up).

Signed Integers

How to we represent negative integers?

Attempt 1: Make the first bit a signed bit. This is called the “sign-magnitude” representation.

- Problems:
  - Two representations of 0 (wasteful and awkward).
  - Arithmetic is tricky. Is the sum of a positive and negative number positive or negative? It depends!

Signed Integers

Attempt 2: **Two's complement form**

- Similar to sign-magnitude in spirit. First bit is 0 if non-negative, 1 if negative.
- Negate a value by just subtracting from zero and *letting it overflow*
- A trick to get the same thing:
  1. Take the complement of all bits
  2. Add 1
- A slightly faster trick is to locate the rightmost 1 bit and flip all the bits to the left of it.

11011010 Negating 00100110

Note: Flipping the bits and adding 1 is the same as subtracting 1 and flipping the bits for non-zero numbers (exercise), or subtracting from 0.

Decimal to Two's Compliment

Let's compute -38_{10} using this notation in one byte of space. First, write 38 in binary:

$$38_{10} = 00100110_2$$

Next, take the complement of all the bits

$$11011001_2$$

Finally, add 1:

$$11011010_2$$

This last value is -38_{10}.

Two's Compliment to Decimal

To convert 11011010_2 to decimal, one method is to flip the bits and add 1:

$$00100110_2 = 2^5 + 2^2 + 2^1 = 38$$

Thus, the corresponding positive number is 38 and so the original number is -38. Another way to do this computation is to treat the

original number 11011010_2 as an unsigned number, convert to decimal and subtract 2^8 from it (since we have 8 bits and the first bit is a 1 meaning it should be a negative value). This also gives -38:

$$\begin{aligned} 11011010_2 &= 2^7 + 2^6 + 2^4 + 2^3 + 2^1 - 2^8 \\ &= 128 + 64 + 16 + 8 + 2 - 256 \\ &= 218 - 256 = -38 \end{aligned}$$

What is Happening

The idea behind [one byte] Two's Compliment notation is based on the following observations:

- Range for unsigned integers is 0 to 255. The number after $255 = 11111111_2$ is in some sense 0 if we ignore overflow.
- Thus, let's make $2^8 = 0$ (i.e., we work modulo $2^8 = 256$). In this vein, we set up a correspondence to the positive integer k with the unsigned integer $2^8 - k$.
- In this case, note that $2^8 - 1 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ and in general:

$$2^n = \sum_{i=0}^{n-1} 2^i$$

As an explicit example (which can be generalized naturally) take a number, say $38_{10} = 00100110_2 = 2^5 + 2^2 + 2^1$. What should the corresponding negative number be? (See next slide).

What is Happening

As an explicit example (which can be generalized naturally) take a number, say $38_{10} = 00100110_2 = 2^5 + 2^2 + 2^1$. What should the corresponding negative number be? Well,

$$2^8 - 1 = 2^7 + 2^6 + \mathbf{2^5} + 2^4 + 2^3 + \mathbf{2^2} + \mathbf{2^1} + 2^0$$

$$2^8 - 1 = \mathbf{38} + 2^7 + 2^6 + 2^4 + 2^3 + 2^0$$

$$2^8 - \mathbf{38} = \underbrace{2^7 + 2^6 + 2^4 + 2^3 + 2^0}_{\text{Flip the bits ...}} + \underbrace{1}_{\text{and add one}}$$

Arithmetic of Signed Integers

- All of the arithmetic works by ignoring overflow precisely because arithmetic works in Z_{256}!
- Arithmetic works naturally except that any final carry overs are ignored (see the two examples below).
- For a few examples, to add 4 and −3 on the left in a 4 bit system or adding −4 and −3 on the right, we have

$$\begin{array}{rl} 0000\,0100 & (+4) \\ +\ 1111\,1101 & (-3) \\ \hline 0000\,0001 & \end{array} \qquad \begin{array}{rl} 1111\,1100 & (-4) \\ +\ 1111\,1101 & (-3) \\ \hline 1111\,1001 & \end{array}$$

- Overflow occurs when you add two numbers of the same sign but get a different sign.

Natural Definition

Definitions

- The Most Significant Bit (MSB) is the left-most bit (highest value/sign bit)
- The Least Significant Bit (LSB) is the right-most bit (lowest value)

Bytes as Characters

- ASCII (American Standard Code for Information Interchange) uses 7 bits to represent characters (see next table)
- Note that 'a' is different than 0xa: the former is the decimal number 97 in ASCII, and the latter is just the number 10 in decimal.
- Unicode extends ASCII in a backwards-compatible way. We will use ASCII throughout.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Highlights

- Characters 0–31 are control characters
- Characters 48–57 are the numbers 0 to 9
- Characters 65–90 are the letters A to Z
- Characters 97–122 are the letters a to z
- Note that `'A'` and `'a'` are 32 away