University of Waterloo

CS241 - Winter 2024 - Course Notes

Author: Brandon Zhou

Course Code: CS241

Course Name: Foundations of Sequential Programs

Instructor: Chengnian Sun

Section: 001

Date Created: January 09, 2024

Final Exam Date: TBA

Disclamer: These course notes are intended to supplement primary instructional materials and facilitate learning. It's worth mentioning that some sections of these notes might have been influenced by ChatGPT, an OpenAI product. Segments sourced or influenced by ChatGPT, where present, will be clearly indicated for reference.

While I have made diligent efforts to ensure the accuracy of the content, there is a potential for errors, outdated information, or inaccuracies, especially in sections sourced from ChatGPT. I make no warranties regarding the completeness, reliability or accuracy of the notes contained in this notebook. It's crucial to view these notes as a supplementary reference and not a primary source.

Should any uncertainties or ambiguities arise from the material, I strongly advise consulting with your course instructors or the relevant course staff for comprehensive understanding. I apologize for any potential discrepancies or oversights.

Any alterations or modifications made to this notebook after its initial creation are neither endorsed nor recognized by me. For any doubts, always cross-reference with trusted academic resources.

Table of Contents

Lecture 1

- Definition: A bit is a binary digit. 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
- in C/C++:
  - Char: 8 bits
  - Unsigned char: 8 bits
  - Short: 2 bytes/16 bits
  - int: 4 bytes
  - longlong: 16 bytes
- 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.
- Definition (Hexadecimal Notation): 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, f (which convert to the numbers from 10 to 15 in decimal notation)
  - 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
  - Note: upper case letters are also used for hexadecimal notation. Context should make things clear.

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

- Notation:
  - Binary: 0, 1
  - Decimal(Base 10): 0, ..., 9
  - Hexadecimal: 0, ..., 9, A(10), B(11), C(12), D(13), E(14), F(15)
- Examples:
  - 0000(base 2) -> 0x0(base 16)
  - 1111(base 2) -> 0xf(base 16)
- What do bytes represent?
  - Numbers
  - Characters
  - Garbage in memory
  - Instructions (Words, or 4 bytes, will correspond to a compute instruction in our computer system)
- Bytes as Binary Numbers
  - Unsigned (non-negative integers)
    - $b_7 ... b_0$ (base 2) = $b_7 \times 2^7 + ... + b_0 \times 2^0$ (base 10)
    - Example: $01010101 = 0 \times 2^7 + ... + 1 \times 2^0$
    - Converting to Binary:
      - One way: Take the largest power of 2 less than the unsigned integer, subtract and repeat
      - Another way is to constantly divide by 2, get the remainer for each division, reading *from the bottom to up* at the end, and that will be the binary representation of this unsigned integer

- Example: 38

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

- Brief Explanation: Consider

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

The remainder when dividing N by 2 gives the b_0 value. After doing $\frac{N-b_0}{2}$, we end of with

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

and we can repeat the process. (This is why we have to read bottom-up as we get b_0 first, then b_1 ...)

- Signed 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 number and a negative number positive or negative? It depends!
  - Attempt 2: Two's complement form
    - Similar to "sign-magnitude" representation in spirit, first bit is 0 if non-negative, 1 if negative
    - Negate a value by just subtracting from zero and *letting it overflow*.
    - Decimal to Two's Compliment:
      - A trick to the same thing of negating a value:
        - Take the complement of all bits (flip the 0 bits to 1 and 1 bits to 0)
        - Add 1
      - A slightly faster trick is to locate the rightmost 1 bit and flip all the bits to the left of it
        - Example: 11011010 Negating: 00100110 = 00100101 + 1
        - Note: Flipping the bits and adding 1 is the same as
          - subtracting 1 and flipping the bits for non-zero numbers
          - subtracting from 0
        - Example: compute -38_{10} using this notation in one byte of space:
          - Step 1: $38_{10} = 00100110_2$
          - Step 2: take the complment of all the bits: 11011001_2
          - Step 3: plus 1: 11011010_2
    - Two's Compliment to Decimal
      - Let's compute -38_{10} using one-byte Two's complement. 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}.
      - To convert 11011010_2, a number in Two's complement representation, 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 28 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

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

- The idea behind [one byte] Two's Complement notation is based on the following observations:
  - The range for unsigned integers is 0 to 255. Recall that 255 is 11111111_2. If we add 1 to 255, then, after discarding overflow bits, we get the number 0.
  - Thus, let's treat 2^8 as 0, i.e., let's work modulo $2^8 = 256$. In this vein, we set up a correspondence between the positive integer k and the unsigned integer $2^8 - k$. Since we are working modulo 2^8, subtracting a positive integer k from 0 is the same as subtracting it from 2^8.
  - In this cse, note that $255 = 2^8 - 1 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ and in general

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

  - 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? Well, note that we've said subtracting a positive integer k from 0 is the same as subtracting it from 2^8 :

$$\begin{aligned} 2^8 - 1 &= 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\ 2^8 - 1 &= 38 + 2^7 + 2^6 + 2^4 + 2^3 + 2^0 \\ 2^8 - 38 &= 2^7 + 2^6 + 2^4 + 2^3 + 2^0 + 1 \qquad \text{(flip the bits and add 1)} \end{aligned}$$

  - We mentioned that another method of negating a two's complement number is to flip the bits to the left of the rightmost 1. Justify why this technique works.
    - Every bit to the right of the rightmost 1 is a 0. When we "flip the bits", these become 1s. When we "add 1", the carry propagates up until the position of the "rightmost 1" (the "rightmost 1" is 0 after flip and will stop propagating when the carry reaches this point, and everything on the left of the "rightmost 1" is flipped).

- The main difference between signed and unsigned binary arithmetic is that we are now working modulo 256 (or, more generally, 2n in the case of n-bit Two's complement numbers)
- When working in Two's complement, overflow occurs when adding numbers if the original two numbers have the same sign, but the result has a different sign.
- Arithmetic of Signed Integers
  - All of the arithmetic works by ignoring overflow precisely because arithmetic works in Z_{256}!
- What is the range of numbers expressible in one-byte Two's Complement notation?
  - $-128 \sim 127$
- 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).

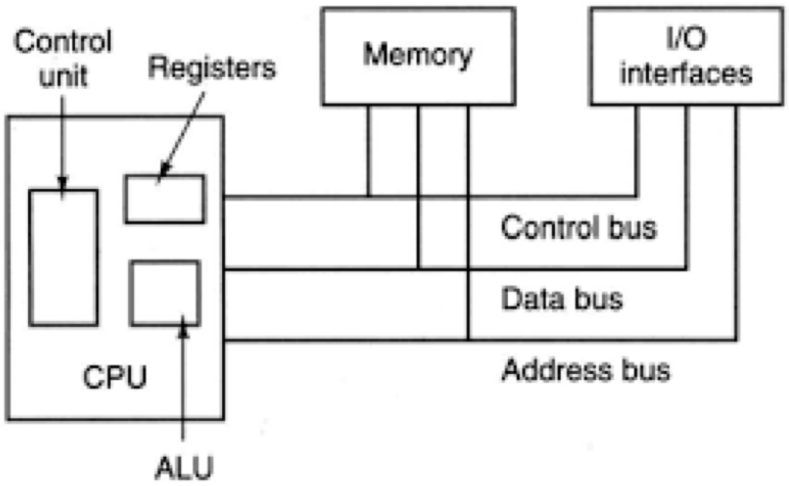
Lecture 2

- ASCII (American Standard Code for Information Interchange) uses 7 bits to represent characters.

```
char c = '0';
printf("%c", c);
```

```
// stdout: 0
printf("%d", (int)c);
// stdout: 48
```

- Bit-Wise Operators
  - Example: suppose we have `unsigned char a=5, b=3;`, which means $a = 00000101, b = 00000011$
    - Bitwise not $\sim a$, for example $c = \sim a;$ gives $c = 11111010$
    - Bitwise and $\boxed{\&}$, for example $\boxed{\texttt{c=a\&b;}}$ gives $c = 00000001$
    - Bitwise or $|$, for example $c = a|b;$ gives $c = 00000111$
    - Bitwise exclusive$\hat{}$(only true if the bit in a and b is different), for example $c = a\hat{}b;$ gives $c = 00000110$
    - Bitwise shift right or left $>>$ and $<<$, for example
      - $c = a >> 2;$ gives $c = 00000001$ and
      - $c = a << 3;$ gives $c = 00101000$
    - $a << 1$ equivalent to $a * 2$
    - $a >> 1$ equivalent to $a/2$
    - These can even be combined with the assignment operator: `c &= 5;`



- CPU with Memory
- MIPS has 32 registers that are called “general purpose”
  - Some general-purpose registers are special:
    - `$0` is always 0
    - `$31` is for return address
    - `$30` is our stack pointer
    - `$29` is our frame pointer
- Problem: We only know from context what bits have what meaning, and in particular, which are instructions.
  - Solution: Convention is to set memory address 0 in RAM to be an instruction. `0x0: instruction i1`
- Problem: How does MIPS know what to do next?
  - Solution: Have a special register called the Program Counter (or PC for short) to tell us what instruction to do next.
- Problem: How do we put our program into RAM?
  - Solution: A program called a loader puts our program into memory and sets the PC to be the first address.
- Algorithm 1 Fetch-Execute Cycle

```
PC=0
while true do
    IR = MEM[PC]
    PC += 4
    Decode and execute instruction in IR
endwhile
```

- Write a program in MIPS that adds the values of registers `$8` and `$9` and stores the result in register `$3`.

```
add $d, $s, $t
0000 00ss ssst tttt dddd d000 0010 0000
```

- Why 5 bits for each? Because there are 32 registers `$0 - $31` and $2^5 = 32$
- Adds registers `$s` and `$t` and stores the sum in register `$d`. Important! The order of `$d`, `$s` and `$t` are shifted in the encoding.

Lecture 3

Lecture 4

Lecture 5

Lecture 6

Lecture 7

Lecture 8

Lecture 9

Lecture 10

Lecture 11

Lecture 12

Lecture 13

Lecture 14

Lecture 15

Lecture 16

Lecture 17

Lecture 18

Lecture 19

Lecture 20

Lecture 21

Lecture 22

Lecture 23

Lecture 24

Final Review