CSE 101: Introduction to Computational Thinking

Unit 9: Computer Architecture and Assembly Language

von Neumann Architecture

- In the beginning of the course we learned the basics of the **von Neumann architecture**, which consists of several components:
  - The **central processing unit**, or CPU, which performs computations
  - The **main memory**, which provides temporary storage of data
  - **Input/output (I/O) devices**, which enable data to enter and leave the computer

von Neumann Architecture

- Millions or billions of times per second the CPU performs the following steps (the **fetch-decode-execute** cycle):
  1. Fetch the next instruction from memory.
  2. Decode the instruction to determine what to do.
  3. If necessary, read operands (data) from main memory that are needed to perform the instruction.
  4. Execute the instruction.
  5. If necessary, write a result back to main memory.

von Neumann Architecture

- The CPU itself is composed of several sub-components:
  - The **arithmetic/logic unit**, which performs the computations
  - The **control unit**, which decodes the instructions and tells the ALU what to do
  - **Registers**, which are small memory banks that temporarily hold instruction operands and results
- Registers are needed because the CPU cannot directly perform calculations on values stored in memory
- Rather, the CPU first copies operands from memory to registers, performs the operation (the result of which is saved in a register), and copies the result from the register back to memory

Bit-level Computations

- Ultimately, all data in a computer is reduced to 1s and 0s, even images, audio, video – everything
- Therefore, all computations are performed on bits
- We already saw three bitwise operators in Python:

	AND	OR	XOR
a b	**a & b**	**a | b**	**a ^ b**
0 0	0	0	0
0 1	0	1	1
1 0	0	1	1
1 1	1	1	0

Bit-level Computations

- These bitwise operators can be used with multi-bit values
- For example, suppose we had the following **bitstrings**:
  - `0010110011001010`
  - `1001101010101010`
- The operation below:

```
  0010110011001010
& 1001101010101010
```

would perform a bitwise-AND on each pair of bits:

```
  0010110011001010
& 1001101010101010
  ----------------
  0000100010001010
  ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
```

Perform 16 **AND** operations on each "vertical" pair of bits

Bit-level Computations

- These operations can likewise be performed for bitwise **`OR`** (**`|`**) and bitwise **`XOR`** (**`^`**):

```
  0010110011001010
| 1001101010101010
------------------
  1011111011101010

  0010110011001010
^ 1001101010101010
------------------
  1011011001100000
```

Bit-level Computations

- Two other useful operators for working with bits are the **shift left** (**<<**) and **shift right** (**>>**) operators
- These operators shift bits by a given number of positions
- For a shift left, bits on the left-hand side are dropped and zeroes are *shifted in* on the right-hand side
- For a shift right, bits on the right-hand side are dropped and copies of the leftmost bit are shifted in on the left-hand side
  - For example, if the leftmost bit is 0, then zeroes are shifted in
  - If the leftmost bit is 1, then ones are shifted in

Bit-level Computations

- Some examples:

  `0010110011001010 >> 3 = 0000010110011001`

  `1010110011001010 << 2 = 1011001100101000`

- With these five bitwise operators (`&`, `|`, `^`, `>>`, `<<`), we can perform many useful operations
  - One important thing to note: the bits themselves are numbered from right-to-left starting with position #0
  - In the examples so far, which have 16 bits, the bits are numbered from 0 to 15 (right-to-left):

bit:	1	0	1	0	1	1	0	0	1	1	0	0	1	0	1	0
#	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Masking Bits

- **Masking** is the process of using bitwise operators to read out or change the bits in a bitstring
- Masking operations are used in many kinds of **systems-level** applications and scenarios, such as operating systems, network communication, computer graphics, and others

Bit-level Computations

- Example: mask some bits to 1
- Suppose we wanted to turn *on* certain bits in a particular number, leaving the other bits unaffected
  - To turn a bit *on* means to set it to 1
- We create a mask with 1s in those bit locations (0 for the other bits) and then use bitwise-OR
- If we wanted to turn on bits 5 through 9, we would use the mask **0000001111100000**

```
  0010110011001010 (input)
| 0000001111100000 (mask)
  ----------------
  0010111111101010
```

Bit-level Computations

- Example: mask some bits to 0
- Likewise, if we want to turn *off* some bits (set them to 0), leaving other bits unaffected, we create a mask consisting of all 1s except for the bits we want to turn off and use a bitwise-AND operation
- If we wanted to turn off bits 2 through 7, we would use the mask **1111111100000011**

```
  0010110011001010 (input)
& 1111111100000011 (mask)
------------------------
  0010110000000010
```

Bit-level Computations

- Example: reading the value of a bit
- Suppose we want to know if a particular bit is 0 or 1
- We create a mask of all 0s except for the bit position we want to check
- Then we perform a bitwise-AND and check if the result is zero or non-zero (non-zero means the bit is 1)
- If we wanted to check if bit #7 is turned on, we would use the mask **0000000010000000**

```
  0010110011001010 (input)
& 0000000010000000 (mask)
  ----------------
  0000000010000000
```

← non-zero result means that the bit was set to 1

Bit-level Computations

- Example: reading the values of several bits
- Suppose we want to know the values of several bits
- We create a mask of all 0s except for the bit positions we want to check
- Then we perform a bitwise-AND and then shift right by a number of bits equal the position of the lowest bit
- If we wanted to read the values of bits 4-8, we would use the mask **`0000000111110000 and then shift right by 4`**
- ```
    0010110011001010 (input)
  & 0000000010000000 (mask)
  ------------------------
    0000000010000000
  ```
  ```

Bit-level Computations

- Example: toggling bits
- Suppose we want toggle (flip) some of the bits in a number, leaving the other bits unaffected (this is done in graphics hardware sometimes)
- We create a mask of 1s in those places we want to toggle
- Then we perform a bitwise-XOR
- If we wanted to toggle bits 4 through 7, we would use the mask `0000000011110000`

```
  0010110011001010 (input)
^ 0000000011110000 (mask)
  0010110000111010
```

Bit-level Computations

- Example: consider the binary representations of several multiples of 4:
  - 8_{10}: 1000_2
  - 24_{10}: 11000_2
  - 36_{10}: 100100_2
- What they all have in common is that the rightmost two bits are both 0
- How might we write a Boolean expression that uses only bitwise operators and relational operators to set the value of a variable to **`True`** only if a number is divisible by 4?

Bit-level Computations

- Suppose we turn off bits 2 through 31, leaving bits 0 and 1 unaffected
- If the resulting number is all zeroes (i.e., the number zero), that means our original number ended in 00, which makes it divisible by 4
- If the result is not all zeroes, it means that the number ended in 01, 10 or 11

```
  0000000001100100 (input)
& 0000000000000011 (mask)
  ----------------
  0000000000000000
```

- Since the result is zero, this means that the original number is divisible by 4

Bit-level Computations

- In Python, if we prepend **`0b`** to a set of 1s and 0s, the computer interprets as a binary number, like **`0b10011`**
- Suppose we wanted to perform:

```
  0010110011001010
& 1001101010101010
```

- We would write it in Python like this:
- **`0b0010110011001010 & 0b1001101010101010`**
- See unit09/bitwise.py for examples of bitwise operations in Python

Programming the CPU

- Python is an example of a **high-level language**, which means that the instructions we write are generally easy to read
- In contrast, the CPU can process only simple instructions that consist of 0s and 1s
  - The CPU's language is called **machine language** and is specific to each product line of CPU
- Because writing instructions using only 0s and 1s is very difficult and error-prone, early computer scientists created **assembly languages** instead
  - Assembly language consists of easier-to remember *mnemonics* that map directly to machine language

Programming the CPU

- For example, the machine language instruction **`00000100011001010000000000100000`** represents an addition instruction in the **MIPS** machine language
  - MIPS is the name of a product line of CPUs
  - The same instruction written in MIPS assembly language is **`add $s0, $s1, $s2`**
  - Although this still looks a little mysterious, it's definitely less mysterious than thirty-two 0s and 1s!

MIPS Assembly Language Overview

- To get an idea of how assembly language really works, we'll look at a real assembly language: 32-bit MIPS assembly
  - A MIPS CPU has thirty-two 32-bit registers, with names like **`$s0`**, **`$t0`** and **`$a0`**
  - We will confine ourselves only to registers **`$t0`** through **`$t9`**
  - Each register can hold 32 bits of data, such as an integer
  - Our assembly language instructions will allow us to refer to them by name (e.g., **`$t0`**)

MIPS Assembly Language Overview

- Every MIPS assembly language program has two sections:
  - The **`.data`** section stores the data values we will process
  - The **`.text`** section stores the instructions themselves
- Recall from earlier in the semester the concept of the **stored-program computer**
  - The idea is that data and instructions are stored together in the main memory
  - This is done so that the computer can be easily reprogrammed to execute different tasks
  - We will see very clearly now how this is actually done in a real computer

MIPS: Loading Values

- One of the most basic operations in MIPS assembly is to **load** a register with a value
- The instruction is called **`li`**, which is short for *load immediate*
- The word **immediate** refers to a constant value that appears in an instruction
- The **`li`** instruction is similar to an assignment statement in Python (e.g., **`x = 28`**)
- For example, if we wanted to load (copy) the value 28 into register **`$t2`**, we would write this: **`li $t2, 28`**
- Note how the destination is given first, just like an assignment statement

MIPS: Loading Values

- Assembly languages don't directly support concepts like variables or functions
  - Instead, we have *labels*
- A label is a name given to a 32-bit cell of main memory in the computer. It's very similar to a Python variable.
- Sometimes the CPU needs to read a value from memory
  - We can identify this value by the label of the memory cell where the data exists
  - We will also have to tell the CPU which register to store the data in
  - The relevant MIPS assembly language instruction is **`lw`**, which is short for *load word*

MIPS: Loading Values

- Recall that a **word** is the native unit of data for a CPU. In MIPS, a word is 32 bits in size.
- Suppose we have a label called **`salary`** that currently stores the value 5000, and we want to add 200 to it
- First we load (copy) the value from memory into a register
- Let's assume we want to copy the value into register **`$t0`**
- Here's the code:

```
.data
salary: .word 5000
.text
lw $t0, salary
```

- **`.word`** means that **`salary`** is a word (i.e., an integer)

MIPS: Arithmetic

- Suppose now that we wanted to add 200 to **`salary`**
- MIPS assembly language has a variety of instructions for performing arithmetic
- One of these is **`addi`**, which is short for *add immediate*
- **`addi`** would correspond to a statement like **`new_salary = salary + 200`** in Python
- We need to provide **`addi`** with the *destination register* (where the new value will be saved), the *source register* (where the current value is saved), and an *increment* (how much to add to the source value)
- Here's the code, along with an explanatory comment:

```
addi $t1, $t0, 200   # $t1 = $t0 + 200
```

MIPS: Arithmetic

- In Python, we know that we can use notation like **`+=`** to increment a variable by some amount
- For example, we could write **`salary += 200`** to add **`200`** to the variable **`salary`**
- We can do the same thing in MIPS assembly of we use the same register for the source and destination
- In the example below, 200 is added to the value in **`$t0`**, overwriting the current value with the new sum

```
addi $t0, $t0, 200   # $t0 += 200
```

- There is no **`subi`** instruction in MIPS assembly, but we can simulate it if we use a negative immediate value:

```
addi $t0, $t1, -50   # $t0 = $t1 - 50
```

MIPS: Storing Values

- To copy data from a register into a memory cell we can use a label
- This operation is known as a **store** because we are storing a piece of data in the main memory
- The instruction is **`sw`**, which is short for *store word*
- First we give the register that has the value we want to copy, and then we give the label that says where the value should be copied to
- Suppose we wanted to copy the value in **`$t0`** to the memory cell called **`salary`**
- The code below will accomplish this task:

```
sw $t0, salary   # salary = $t0
```

MIPS: More Arithmetic

- Sometimes we need to perform arithmetic using only registers (no immediate values)
- Consider the Python statement **`salary += raise`**
- Perhaps **`raise`** is determined by some calculation, and is not a constant
- Let's assume that register **`$t0`** contains **`salary`** for someone and **`$t1`** contains **`raise`**
- We can use the **`add`** instruction to add the contents of two registers and store the result
- The destination register is given first, followed by the two registers that hold the operands, as in this example:

```
add $t0, $t0, $t1   # $t0 = $t0 + $t1
```

MIPS: More Arithmetic

- MIPS assembly also supports **`sub`** for subtraction, **`mul`** for multiplication and **`div`** for integer division
- All instructions have the same format:
  `[instruction] [destination] [operand1] [operand2]`
- Note that none of these instructions uses immediates, so any constants must first be loaded into registers using **`li`**
- The code below doubles the value stored in **`$t3`** and divides that result by 7, storing the quotient in **`$t4`**

```
li $t0, 2                # $t0 = 2
mul $t3, $t3, $t0        # $t3 = $t3 * 2
li $t1, 7                # $t1 = 7
div $t4, $t3, $t1        # $t4 = $t3 // 7
```

Compute $f(x) = Ax^2 + Bx + C$

- Suppose we wanted to write a program that computes the value of a quadratic polynomial $f(x) = Ax^2 + Bx + C$
- First we would need a way to store the values of A, B, C and x
- In the **`.data`** section of our program we will create a label for each of these values:

```
.data
x:  .word 6
A:  .word 3
B:  .word 4
C:  .word 5
```

Compute $f(x) = Ax^2 + Bx + C$

- Now we can create a **`.text`** section where we will write the operations themselves
- Let's load these four values into registers so that we can use them in calculations:

```
.text
lw $t0, x
lw $t1, A
lw $t2, B
lw $t3, C
```

Compute $f(x) = Ax^2 + Bx + C$

- Let's consider what arithmetic we need to perform to compute $Ax^2 + Bx + C$
- Looking closely we see we will need 5 operations: three multiplications and two additions
- We can perform these in any order, provided that we implement the correct order of operations
- To make things manageable, let's do the multiplications first:

```
mul $t4, $t0, $t0    # $t4 = x^2
mul $t5, $t1, $t4    # $t5 = A*x^2
mul $t6, $t2, $t0    # $t6 = B*x
```

Compute $f(x) = Ax^2 + Bx + C$

- With most of the hard work done we can now perform the additions
- However, we didn't decide earlier where we want to save the final result, so let's decide now to save it in **`$t8`**

```
add $t7, $t5, $t6  # $t7 = A*x^2 + B*x
add $t8, $t7, $t3  # $t8 = A*x^2 + B*x + C
```

Compute $f(x) = Ax^2 + Bx + C$

```
.data
x:  .word 6
A:  .word 3
B:  .word 4
C:  .word 5

.text
lw $t0, x
lw $t1, A
lw $t2, B
lw $t3, C
```

```
# $t4 = x^2
mul $t4, $t0, $t0

# $t5 = A*x^2
mul $t5, $t1, $t4

# $t2 = B*x
mul $t6, $t2, $t0

# $t0 = A*x^2 + B*x
add $t7, $t5, $t6

# $t0 = A*x^2 + B*x + C
add $t8, $t7, $t3
```

See unit09/ quadratic.asm

A MIPS Virtual Machine

- We don’t have access to a real MIPS-based computer, but we can create a Python program that emulates a MIPS computer
- Such a program is called a **virtual machine**
- Our program will need to perform the essential functions of a von Neumann machine
- We will also need to simulate the CPU's registers and main memory
- To that end we will create two dictionaries: **`registers`** and **`labels`**:

```
registers = {}
labels = {}
```

A MIPS Virtual Machine

- Now we need a way to read and write values into the registers and main memory to support the **`li`**, **`lw`** and **`sw`** instructions
- Our approach will be to write a function for each of these operations

```
def li(reg, immed):
    registers[reg] = immed

def lw(reg, label):
    registers[reg] = labels[label]

def sw(reg, label):
    labels[label] = registers[reg]
```

A MIPS Virtual Machine

- Next we have the various arithmetical operators:

```
def addi(dest, src, immed):
    registers[dest] = registers[src] +
                      immed
def add(dest, src1, src2):
    registers[dest] = registers[src1] +
                      registers[src2]
def sub(dest, src1, src2):
    registers[dest] = registers[src1] –
                      registers[src2]
```

- And likewise for **`mul`**, **`div`** and **`mod`** (for **`%`**)

A MIPS Virtual Machine

- Last, we need to write a Python function to read a MIPS assembly program from disk and execute it
- The essential elements of this function are:
  1. Open the file and read it into a list of strings
  2. For each string:
     a. Identify labels (which are basically variables) and store the associated values in labels
     b. Identify **`li/lw/sw`** instructions and perform loads and stores
     c. Identify **`addi/add/sub/mul/div/mod`** instructions and perform the arithmetical operations
- See unit09/mips_vm.py for the code and examples

MIPS Change-maker

- One of the first programs we wrote in the course was the change-making program
- Given a total number of cents, we want to know many dimes, nickels and pennies are needed to make that change while minimizing the number of coins
- Python code to solve this problem is on the right

```
cents = 138
dimes = cents // 10
cents = cents % 10
nickels = cents // 5
cents = cents % 5
pennies = cents
print(dimes, nickels,
      pennies)
```

MIPS Change-maker

- Let's translate this program into MIPS assembly
- Assume that the number of cents is stored at a label called **`cents`**
- We will store the final number of dimes in a label **`dimes`**, nickels in a label **`nickels`** and pennies in a label **`pennies`**

MIPS Change-maker

```
.data
cents: .word 138
dimes: .word 0
nickels: .word 0
pennies: .word 0
```

See unit09/change.asm

```
.text
li $t0, 10
lw $t1, cents
div $t2, $t1, $t0
sw $t2, dimes
mod $t1, $t1, $t0
li $t0, 5
div $t2, $t1, $t0
sw $t2, nickels
mod $t1, $t1, $t0
sw $t1, pennies
```

Final Note on MIPS Programming

- You will NOT be required to read or write MIPS code on any examination in this course
- The main purpose of our brief exploration of assembly language was to help you develop a general understanding and appreciation of what instructions the CPU is actually executing as it runs software