# ECS 50: Programming Assignment #6

Instructor: Aaron Kaloti

Winter 2021

## Contents

## 1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Fixed small typo in output for example #2.
- v.3: Assembler must complain if a label is defined multiple times *but also* if a label is never defined at all. If multiple assembler errors could be generated by the input file, then you just need to generate at least one of the messages. (The autograder will be mindful of this.)
- v.4: A label's name must start with a letter.
- v.5: Added Gradescope submission entry for task #2.
- v.6: Autograder details.

## 2 General Submission Details

**Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.**

This assignment is due the night of Saturday, 03/13. Gradescope will say 12:30 AM on Sunday, 03/14, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

## 3 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 13% of your final grade. Task #1 is worth 11%, whereas task #2 is worth 2%.

## 4 Submitting on Gradescope

*Do not submit* the caller code files, i.e. `call_check_bits.s`, etc.

During the 01/12 lecture, I talked about how to manually change the active submission, just in case that is something that you find yourself needing to do.

**Once the deadline occurs, whatever score the autograder has for your active submission (your last submission, unless you manually change it) is your *final* score** (unless you are penalized for violating a restriction, as mentioned below).

### 4.1 Regarding Autograder

**Files to submit**:

- `assembler.cpp`
- (optional) `my_instructions.txt`

**The reference environment is the CSIF.**

Unless stated otherwise, your output must match mine *exactly*.

There are 20 test cases, each worth 5.5 points (for a total of 110 points), and 6 of those test cases are hidden cases. Each test case used the input file that had its case number; you can find these input files (for the visible test cases) on Canvas.

For cases in which it is expected that something is printed to standard error, the contents of the output file (if it was created) are not printed.

## 5 The MiniCUSP ISA

### 5.1 Introduction: From CUSP to MiniCUSP

Some previous ECS 50 instructors in our department would teach a fake ISA called CUSP[1] and then switch to a real ISA (e.g. x86-32, MIPS32) for the last one or two weeks of the quarter. The idea behind teaching this fake ISA was that the assembly language was free of complications that help real-world architectures but that are burdensome from an educational standpoint. As stated later in this document, **your task in this assignment is to write an assembler for a subset of the CUSP ISA that I dub MiniCUSP**[2].

Below, I first describe the MiniCUSP architecture and assembly language, and then I talk about the assembler.

### 5.2 The MiniCUSP Architecture

The MiniCUSP architecture consists of a central processing unit (CPU) and memory. There is no I/O mechanism, because who needs I/O anyways?

The CPU contains the following registers:

[1]The book that brought this fake ISA about is *Principles of Computer Systems* by Gerald M. Karam and John C. Bryant.

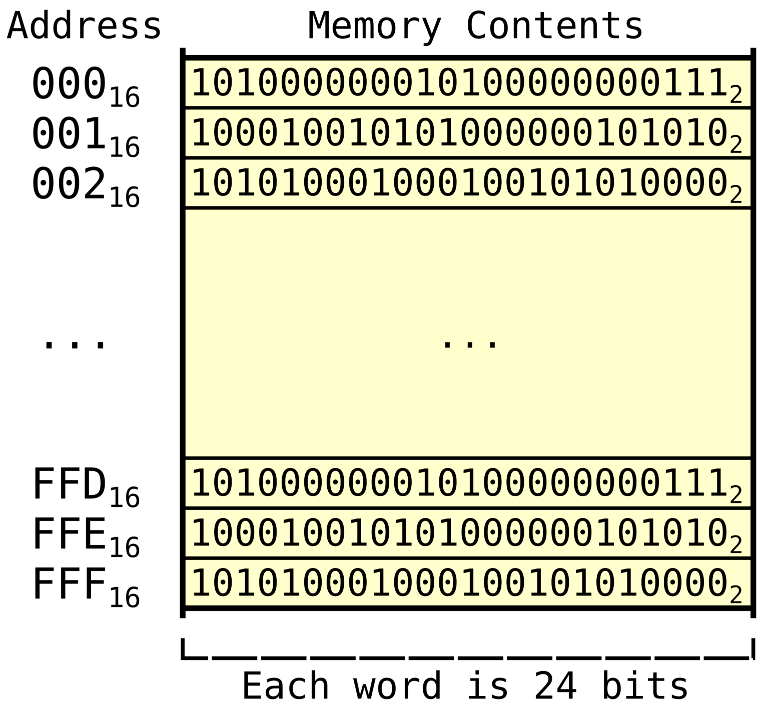
[2]I think that CUSP itself is based on an old Motorola ISA, or at least the assembly languages look quite similar. Thus, it may be the case that you are writing an assembler for a subset of a subset of a real ISA.

| Register | Full Name | Size | Use |
|---|---|---|---|
| ACC | accumulator | 24 bits | Arithmetic/logical computations |
| XR | index register | 12 bits | Indexing into an array |
| SP | stack pointer | 12 bits | Points to the top of the stack |
| FP | frame pointer | 12 bits | Stack frame |
| PC | program counter (instruction pointer) | 12 bits | Address of next instruction to fetch |
| IR | instruction register | 24 bits | Contains fetched instruction |

The IR is not accessible by the programmer, and the PC is only indirectly changeable by the programmer through the execution of jump/branching instructions. Thus, you should not need to care about these registers while creating your assembler.

The CPU also contains flags (two of them being the LT and EQ flags), but you do not need to worry about those in this assignment.

Main memory consists of $4096_{10}$ ($1000_{16}$) 24-bit words having addresses $000_{16}$ through $\text{FFF}_{16}$. This means that a MiniCUSP machine is *word-addressable* and not *byte-addressable*. (We talked about these concepts on slide #25 of slide deck #2.) Endianess is irrelevant to a word-addressable machine. Below is a visualization of main memory.



The MiniCUSP CPU also has an arithmetic logic unit (ALU) and a control unit (CU), although those are not important in this assignment.

## 5.3 The MiniCUSP Assembly Language

### 5.3.1 All Instructions

Each instruction in MiniCUSP follows one of two formats:

1. **Operand instruction**: consists of an opcode, an addressing mode, and an explicitly specified operand.
2. **Operate instruction**: consists solely of an opcode, with no (explicit) operand.

The opcode (operation code) is the portion of the instruction that identifies the operation to be performed by the CPU.

Like RISC-V, MiniCUSP uses fixed-length instructions. Each instruction – when encoded in its machine language form (ones and zeros) – is the size of a word, i.e. 24 bits. For each *operand* instruction, the opcode is 8 bits, the addressing mode is represented using 4 bits, and the operand is specified using 12 bits. Each *operate* instruction has a 24-bit opcode that starts with $\text{FFF}_{16}$ in machine language.

Below is a list of instructions supported by MiniCUSP. For each instruction, I describe what it does, list the supported addressing modes (see the next subsection), and provide the representation of the opcode in machine language. Note that the description of what the instruction does doesn't matter for the purposes of your implementation of the assembler; I only include the descriptions because I think it makes the assignment more enjoyable.

| Mnemonic | Representation | Description | Addressing Modes |
|---|---|---|---|
| LDA operand | 00 | ACC = operand | 0 through 9 |
| LDX operand | 01 | XR = operand | 0 through 9 |
| LDS operand | 02 | SP = operand | 0 through 9 |
| LDF operand | 03 | FP = operand | 0 through 9 |
| STA operand | 04 | operand = ACC | 2 through 9 |
| STX operand | 05 | XR = operand | 2 through 9 |
| STS operand | 06 | SP = operand | 2 through 9 |
| STF operand | 07 | FP = operand | 2 through 9 |
| PSH operand | 08 | Push operand onto stack and decrement SP | 0 through 9 |
| POP operand | 09 | Pop operand from stack into operand and increment SP | 2 through 9 |
| CLR operand | 0A | operand = all zeros | 2 through 9 |
| SET operand | 0B | operand = all ones | 2 through 9 |
| PSHA | FFF010 | Push ACC onto stack and decrement SP | N/A |
| PSHX | FFF011 | Push XR onto stack and decrement SP | N/A |
| PSHF | FFF012 | Push FR onto stack and decrement SP | N/A |
| POPA | FFF013 | Pop stack into ACC and increment SP | N/A |
| POPX | FFF014 | Pop stack into XR and increment SP | N/A |
| POPF | FFF015 | Pop stack into FP and increment SP | N/A |
| ADA operand | 10 | ACC += operand | 0 through 9 |
| ADX operand | 11 | XR += operand | 0 through 9 |
| ADS operand | 12 | SP += operand | 0 through 9 |
| ADF operand | 13 | FP += operand | 0 through 9 |
| SBA operand | 14 | ACC -= operand | 0 through 9 |
| SBX operand | 15 | XR -= operand | 0 through 9 |
| SBS operand | 16 | SP -= operand | 0 through 9 |
| SBF operand | 17 | FP -= operand | 0 through 9 |
| MUL operand | 18 | ACC *= operand | 0 through 9 |
| DIV operand | 19 | ACC /= operand | 0 through 9 |
| MOD operand | 1A | ACC %= operand | 0 through 9 |
| CMA operand | 20 | Computes ACC - operand | 0 through 9 |
| CMX operand | 21 | Computes XR - operand | 0 through 9 |
| CMS operand | 22 | Computes SP - operand | 0 through 9 |
| CMF operand | 23 | Computes FP - operand | 0 through 9 |
| SHRA | FFF022 | ACC >>= 1 | N/A |
| SHLA | FFF023 | ACC <<= 1 | N/A |
| JSR operand | 41 | Push PC onto stack, decrement SP, and set PC = operand | 2 through 9 |
| RTN | FFF040 | Pop stack into PC and increment SP | N/A |
| JEQ operand | 48 | if (EQ flag == 1) PC = operand | 2 through 9 |
| JLT operand | 4A | if (LT flag == 1) PC = operand | 2 through 9 |
| JGE operand | 4B | if (LT flag == 0) PC = operand | 2 through 9 |
| NOP | FFF038 | Does nothing | N/A |
| HLT | FFFFFF | Stop the program | N/A |

Each operand instruction supports addressing modes 2 through 9, but not all operand instructions support addressing modes 0 and 1.

In MiniCUSP, the jump/branching instructions do not support immediate mode. This is counterintuitive to me, given that – for example – `JSR 15` sets the PC to 15 and not to `Memory[15]`, but that's the way it was in CUSP, so I've kept it that way. It doesn't have significant implications for your assembler anyways.

The mnemonic opcodes are always uppercase, i.e. you may assume that you will never see something like `nop` or `lDa# 5`, only `NOP` and `LDA# 5`.

### 5.3.2 Specifying the Operand: Addressing Modes

Here, I talk about how to specify the operand for an operand instruction. None of what I mention in this section applies to operate instructions.

The first thing to note is that a register cannot be an operand. Movement of data into and out of registers is done through specific instructions such as `LDA` and `STF`.

Below is a list of addressing modes that MiniCUSP supports. For each addressing mode, I specify its name, how it syntactically appears in MiniCUSP assembly language code, and how it is represented with one base 16 digit in machine language. I also specify what each addressing mode does, but as is the case with the descriptions of each instruction, the effect of each addressing mode should not be important to you as you implement the assembler; it just might help you to enjoy or understand the context of what you're doing. In the assembly representation column, `inst` is the instruction, and `XXX` is the operand. In the behavior column, memory is treated as an array.

| Name | In Assembly | Representation in Base 16 | Behavior (what the operand evaluates to) |
|---|---|---|---|
| Immediate | inst # XXX | 0 | XXX |
| Direct | inst XXX | 2 | Memory[XXX] |
| Indexed | inst + XXX | 4 | Memory[XXX + XR] |
| Indirect | inst * XXX | 6 | Memory[Memory[XXX]] |
| Indirect Indexed | inst & XXX | 8 | Memory[Memory[XXX] + XR] |
| Frame Immediate | inst # ! XXX | 1 | XXX + FP |
| Frame Direct | inst ! XXX | 3 | Memory[XXX + FP] |
| Frame Indexed | inst + ! XXX | 5 | Memory[XXX + FP + XR] |
| Frame Indirect | inst * ! XXX | 7 | Memory[Memory[XXX + FP]] |
| Frame Indirect Indexed | inst & ! XXX | 9 | Memory[Memory[XXX + FP] + XR] |

For the addressing modes that use multiple symbols, the order is mandatory. For example, frame indirect mode cannot have the `!` before the `*`. You may assume that each of `inst`, the addressing mode symbol(s) (if any), and the operand `XXX` are one space apart, but it probably shouldn't matter if you are reading from the input file with `std::ifstream` like you ought to.

Note that if an operand is a constant, that does not necessarily mean that the operand is in immediate mode. It depends on the addressing mode. For example `LDA# 18` loads $18_{10}$ into the ACC. In contrast, `LDA 18` loads whatever is at address $18_{10}$ into memory.

### 5.3.3 Constant Operands

As operands, constant literals (i.e. not labels; see later section) can be specified in one of two ways:

1. Base 10: the constant is specified alone.
2. Base 16: the constant is specified with a dollar sign ($) immediately preceding it. Any hexadecimal digit is upper-case, e.g. B instead of b.

Below are examples. I denote comments with ; for clarity, but your assembler may assume that the input file has no comments, i.e. MiniCUSP doesn't actually support comments.

```
LDX # 18        ; XR = 18
LDX # $18       ; XR = 16 + 8 = 24
LDX   18        ; XR = Memory[18]
LDX   $18       ; XR = Memory[16 + 8] = Memory[24]
```

You may assume that:

- All constants are positive/unsigned, i.e. MiniCUSP does not support negative integers[3].
- No constant is larger than $FFF_{16} = 4095_{10}$
- Leading zeros will never be specified, e.g. you'll never see `LDX# 0318` or `LDA $053`.

### 5.3.4 Examples of Converting Instructions to Machine Code

Below are examples of MiniCUSP assembly instructions and their machine code representation. As a reminder, MiniCUSP uses fixed-length instructions, where each instruction is 24 bits long. Note that constants are encoded in base 16, even if they are specified in base 10 (i.e. without a leading dollar sign). As was the case in the example above, I denote comments here with ; for simplicity, but your assembler will not be given input files that have comments.

```
LDA $35         ; 002035 <== LDA is 00, addr mode is direct (2), and operand is 035
LDA # $35       ; 000035 <== same as above, except that addr mode is immediate (0)
CMA $9          ; 202009 <== CMA is 20, addr mode is direct (2), and operand is 009
JLT $6          ; 4A2006 <== JLT is 4A, addr mode is direct (2), and operand is 006
STA $A          ; 04200A <== STA is 04, addr mode is direct (2), and operand is 00A
LDF # $90       ; 030090 <== LDF is 03, addr mode is immediate (0), and operand is 90
LDF # 90        ; 03005A <== LDF is 03, addr mode is immediate (0), and operand is 5A in base 16
```

[3]On a related note, I don't talk about real numbers in this document. Those are not supported either.

```
LDF + $90       ; 034090 <== LDF is 03, addr mode is indexed (4), and operand is 90
LDS # ! $90     ; 021090 <== LDS is 02, addr mode is frame immediate (1), and operand is 90
LDS & ! 90      ; 02905A <== LDS is 02, addr mode is frame indirect indexed (9), operand is 5A in base 16
```

### 5.3.5 Examples of Illegal Instructions

As discussed later, your assembler will need to reject instructions that are used with addressing modes that they do not support. Below are two examples.

```
STA # 85        ; ILLEGAL: STA does not support immediate addressing
JSR # ! 900     ; ILLEGAL: JSR does not support frame immediate addressing
```

### 5.3.6 Directives: Creating Variables

The only directive that MiniCUSP supports is `.word`, which allocates a word and requires a value used to initialize that word. Any value from $000000_{16}$ to $FFFFFF_{16}$ is supported, and the given value is assumed to be in base 16. Leading zeros will never be specified, e.g. you'll never see `.word 00035B`, but you do need the leading zeros in the machine code. Below are examples followed by their representations in machine code.

```
.word 58        ; 000058
.word 9BE       ; 0009BE
.word FEB89C    ; FEB89C
.word FFFFA     ; 0FFFFA
```

### 5.3.7 Labels

These are pretty much the same as in x86-64 and RISC-V. The label will always be immediately followed by a colon. Below are examples.

```
bar:
        PSHX
blah18:
hello:
        LDA # 1
var:
        .word 18
```

Below are details on the naming of labels:

- You may assume that a label's name only consists of *lowercase* letters and digits.
- A label's name must start with a letter.
- A label's name never consists of more than 8 characters, not counting the colon (which should not be treated as part of the label's name anyways).

For simplicity, we will assume that labels are never followed by an instruction or data on the same line. As examples, the following will never occur.

```
foo:    PSHA          ; NEVER OCCURS

var:    .word 18      ; NEVER OCCURS
```

A label's value is the address of the word containing the instruction or data that immediately follows the creation of the label. Below is an example.

```
label1:                ; Value of ‘label1‘ is 000.
    POPA               ; 1st instruction/data   ==> address is 000.
label2:                ; Value of ‘label2‘ is 001.
label3:                ; Value of ‘label3‘ is also 001.
    LDA # label4       ; 2nd instruction/data   ==> address is 001.
    LDA  label4        ; 3rd instruction/data   ==> address is 002.
    JLT  label1        ; 4th instruction/data   ==> address is 003.
    HLT                ; 5th instruction/data   ==> address is 004.
label4:                ; Value of ‘label4‘ is 005.
    .word 53           ; 6th instruction/data   ==> address is 005.
    .word 9            ; 7th instruction/data   ==> address is 006.
    .word 9            ; 8th instruction/data   ==> address is 007.
    .word 9            ; 9th instruction/data   ==> address is 008.
    .word 9            ; 10th instruction/data  ==> address is 009.
    .word 9            ; 11th instruction/data  ==> address is 00A.
    .word 9            ; 12th instruction/data  ==> address is 00B.
label5:                ; Value of ‘label5‘ is 00C (although nothing follows it).
```

As is the case in x86-64, labels *do not exist* at the machine code level in MiniCUSP. Any label must be replaced with its value when creating the corresponding machine code representation. Below is the above example repeated, except with the machine code representations.

```
label1:                 ; Value of 'label1' is 000.
    POPA                ; FFF013
label2:                 ; Value of 'label2' is 001.
label3:                 ; Value of 'label3' is also 001.
    LDA # label4        ; 000005
    LDA  label4         ; 002005
    JLT  label1         ; 4A2000
    HLT                 ; FFFFFF
label4:                 ; Value of 'label4' is 005.
    .word 53            ; 000053
    .word 9             ; 000009
    .word 9             ; 000009
    .word 9             ; 000009
    .word 9             ; 000009
    .word 9             ; 000009
    .word 9             ; 000009
label5:                 ; Value of 'label5' is 00C (although nothing follows it).
```

Typically, an assembler keeps track of the values with a **symbol table**. This is discussed below.

## 6 Task #1 - MICA: A MiniCUSP Assembler

For the programming portion of this assignment, you will implement a MiniCUSP assembler called MICA (MiniCUSP Assembler).

***Where appropriate, you should ask for clarification on what to do in certain scenarios or what the specifications are. There are so many specifications in this assignment that it is likely I missed something. As is always the case, you should refer to the latest version of this document on Canvas.***

### 6.1 Requirements

Your program will always be given two command-line arguments:

1. The name of the input MiniCUSP assembly code file, whose file extension will always be `.csp`.
2. The name of the output file for the machine code, whose file extension will be `.txt` (see below). If this file already exists, then your program should erase its old contents, which should be easy to do if you simply open the file for writing.

**Submitted Files**: Although some may argue that using multiple files may be more convenient or "better form", **I am requiring that your code be in a single file called `assembler.cpp`. If you wish, you can also submit a text file called `my_instructions.txt` whose contents you get to choose and that you can have your program read from.** (Your program would need to assume that this text file is in the same directory.) The purpose of such a file is to make it less tedious to manage the different characteristics of the instructions, e.g. the machine code opcodes. Again, you do not need to create/submit `my_instructions.txt` if you do not want to.

**Compilation**: The autograder will compile your code with the following command:

```
g++ -Wall -Werror -std=c++14 -g assembler.cpp -o assembler
```

#### 6.1.1 Examples

A real assembler would output a binary file, but in your case, you will output what some would call a text/ascii file[4].

**Example #1**: Below is an example that shows what the expected output looks like and that uses one of the example files from earlier.

```
$ cat example_inputs/example1.csp
label1:
    POPA
label2:
label3:
    LDA # label4
    LDA label4
```

[4]As I'm sure I've said from time to time, "text file" (in contrast to "binary file") is a strange term, considering that all files are stored in binary... (i.e. 1s and 0s)

```
    JLT label1
    HLT
label4:
    .word 53
    .word 9
    .word 9
    .word 9
    .word 9
    .word 9
    .word 9
label5:
$ ./assembler example_inputs/example1.csp output.txt
$ cat output.txt
FFF013
000005
002005
4A2000
FFFFFF
000053
000009
000009
000009
000009
000009
000009
```

**Example #2**: This also uses one of the example files shown earlier.

```
$ cat example_inputs/example2.csp
LDA $35
LDA # $35
CMA $9
JLT $6
STA $A
LDF # $90
LDF # 90
LDF + $90
LDS # ! $90
LDS & ! 90
$ ./assembler example_inputs/example2.csp output.txt
$ cat output.txt
002035
000035
202009
4A2006
04200A
030090
03005A
034090
021090
02905A
```

**Example #3**:

```
$ cat example_inputs/example3.csp
.word 100000
.word 10000
.word 1000
.word 100
.word 10
.word 1
.word F00000
.word F0000
.word F000
.word F00
.word F0
.word F
label1:
    LDA # label1
$ ./assembler example_inputs/example3.csp output.txt
$ cat output.txt
100000
010000
001000
000100
```

```
000010
000001
F00000
0F0000
00F000
000F00
0000F0
00000F
00000C
```

**Example #4**: This example is also used earlier.

```
$ cat example_inputs/example4.csp
.word 58
.word 9BE
.word FEB89C
.word FFFFA
$ ./assembler example_inputs/example4.csp blah.txt
$ cat blah.txt
000058
0009BE
FEB89C
0FFFFA
```

### 6.1.2 Assembler Errors

**In the three scenarios listed below, your assembler should generate an assembler error.** ***There is no other scenario under which an assembler error should be raised. There is no other input validation that you need to do.*** If an assembler error occurs, your program can stop immediately (the autograder won't check the output file) and print the appropriate error message from the examples below *to standard error, not to standard output.* If multiple assembler errors could occur due to a given input file, the autograder will just check that your program printed *at least one* assembler error message (that starts with "ASSEMBLER ERROR") in standard error.

- **Scenario #1 - Label Defined Multiple Times**: In the below example, each of `foo` and `bar` is defined more than once. Note that using a label multiple times is OK, e.g. having `LDA# foo` and then `LDX# foo` would have been fine; using `foo` twice differs from *defining* `foo` twice.

```
$ cat example_inputs/invalid_labels.csp
    PSHA
foo:
    PSHX
bar:
    PSHF
foo:
    HLT
bar:
    HLT
$ ./assembler example_inputs/invalid_labels.csp output.txt
ASSEMBLER ERROR: label defined multiple times.
```

- **Scenario #2 - Label Undefined**: In the below example, `blah` is never defined.

```
$ cat example_inputs/invalid_undefined_label.csp
LDA blah
$ ./assembler example_inputs/invalid_undefined_label.csp output.txt
ASSEMBLER ERROR: undefined label.
```

- **Scenario #3 - Operand Instruction Using Unsupported Addressing Mode**: In the example below, `STA` is using frame immediate mode even though it does not support frame immediate mode.

```
$ cat example_inputs/invalid_addr_mode.csp
.word 58
STA # ! 900
LDF + $90
$ ./assembler example_inputs/invalid_addr_mode.csp output.txt
ASSEMBLER ERROR: instruction using unsupported addressing mode.
```

## 6.2 Miscellaneous Assumptions You Get to Make

You may assume that the program will never require more than $FFF_{16}$ words[5].

[5]Having a program this large would be unrealistic, given that MiniCUSP only has $FFF_{16}$ words for its entire memory, but MiniCUSP isn't supposed to be realistic in the first place.

You may assume that the input assembly code file contains no empty lines[6].

You may assume that the input assembly code file contains no made-up instructions, e.g. `PSHS`.

### 6.3 Advice

There are a few places in this program where an intelligent approach can result in a significantly smaller program, e.g. save you from having to do something ridiculous like one conditional statement per instruction whenever you must check certain attributes about an instruction. My program was around 250 lines (this counts numerous comments), although my approach was horribly inefficient (in terms of lines of code) in some areas.

As implied above, you are encouraged to create a **symbol table** to keep track of which **symbols (i.e. labels)** have and have not been defined. A symbol table is a hash table used to map symbols to what is thought to be their values. In Python, sets and dictionaries are implemented with hash tables. In C++, the `unordered_set` and `unordered_map` types are implemented with hash tables[7]. Because **forward references** (i.e. an instruction uses a label that is defined after the instruction) are possible in MiniCUSP, a single pass is not possible. Instead, it is encouraged that you do **two-pass assembly** and take two passes through the lines of the assembly code file. The first pass should determine all symbols/labels and their values. The second pass should generate the machine code.

The (fictional) CUSP assembler used a **location counter** to faciliate the generation of the symbol table. Initially $000_{16}$, the location counter would be incremented as the assembler traversed through the input assembly code file's contents. Specifically, the location counter would be incremented by 1 *only when* an instruction or data was encountered. The effect of this was that the location counter would always contain the address of the instruction or data when it was encountered. This made it much easier to determine the value of labels.

I really don't want the file reading and parsing to be a significant obstacle in this assignment. On that note: unless you're aiming for extensible code[8], you should take advantage of "cheap" ways of checking for certain things. For example, in my code, I check if I am at a label by checking if the first character of whatever string I just read is a lowercase letter[9].

## 7 Task #2 - Some History of Architectures

For this part, you will read section 2.21 of *Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition)* by David A. Patterson and John L. Hennessy. This section is freely available here and is also posted to Canvas. Read the entirety of this section, answer the below questions, and then submit your answers in a PDF to Gradescope. **The Gradescope submission entry is titled, "P6: Task 2 (Some History of Architectures)" and is on Gradescope now.**

Some of the questions ask about basic facts in the reading and are intended to make sure you read the section. Other questions are intended to make you think a bit about the reading. In your answers, you may quote from the reading if you wish.

While doing the reading, pay attention to the role of economic factors as well, as economic factors are still important today in deciding which architectures maintain dominance, which decline, and which remain relevant only in academic institutions.

### 7.1 Questions

1. In the early days of computers, why did computer architectures not have the high number of registers (e.g. 16, 32) that they did today?
2. According to the reading, what is considered by many to be the first supercomputer? When was it released?
3. According to the reading, what kind of architectures were most popular in the 1960s?
4. Why did Intel x86 have variable-length instructions?
5. Why do ARM, MIPS, and RISC-V have versions that offer 16-bit and 32-bit instruction formats?
6. Name two benefits of stack architectures.
7. State (without justification) if the following statement is true or false: The introduction of RISC assembly languages meant that instruction sets could be measured by how skillfully assembly language programmers used them rather than how well compilers used them.
8. Whenever the reading mentions ARM, "ARMv7" and "ARMv8" are listed separately rather than treated as one. Why is this?

[6]If you open the example input assembly code files provided on Canvas in some GUI text editor, such as Sublime Text, it may appear as if the last line is an empty line, but that is because of the text editor. The file does not end in an empty line.

[7]The `set` and `map` types are implemented with self-balancing binary search trees, most likely red-black trees but I don't know if that's changed recently.

[8]This is not a bad idea if you have the time.

[9]You should learn about sophisticated and real-world parsing approaches for more complicated languages in ECS 140A. I think you learn about them a bit in ECS 120 too, but I never took that course and am thus not completely sure.

9. Describe one factor that contributed to the success of the IBM PC over the Macintosh (Mac).
10. Of the kinds of architectures (e.g. stack architecture, reduced instruction set architecture) mentioned in the reading, which would you say MiniCUSP is? Defend your answer.
11. What kind of architecture would you say that x86-64 is?
12. Is LISP a compiled language? Or an interpreted one? Explain.
13. According to the reading, which language replaced Algol-60 as the most popular language for academics to teach programming?
14. According to p.162.e7, "[Alan] Kay and his colleagues argued that processors were getting faster, and that we must eventually be willing to sacrifice some performance to improve program development." How does this argument apply today when one tries to choose which programming language (e.g. C++ vs. Python) to use for a project?
15. Which famous $\mathcal{NP}$-complete graph problem[10] was relevant to a breakthrough in register allocation for compiler optimizations?

UCDAVIS
COMPUTER SCIENCE

[10] You will learn about complexity theory in ECS 120 or ECS 122A. It is still possible to answer this question based on the reading, even if you do not know what "$\mathcal{NP}$-complete" means.