ECS 50: Programming Assignment #5

Instructor: Aaron Kaloti

Winter Quarter 2022

Contents

1 Changelog

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

- v.1: Initial version.

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 Tuesday, 03/01. Gradescope will say 12:30 AM on Wednesday, 03/02, 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 Purpose of This Assignment

- To familiarize you with the functionality of an assembler.
- To familiarize you with machine code.
- To give you experience with a decently sized assignment that tests your implementation skills and your ability to handle a lot of details/specifications and separate useful information from irrelevant information.

4 Reference Environment

The autograder, which is talked about at the end of this document, will compile and run your code in a Linux environment, specifically Ubuntu 18. That means that you should make sure your code compiles and behaves properly in a sufficiently similar environment. **The CSIF is one such environment.** Each student can remotely access the CSIF, and now that learning is back in person, the CSIF computers can be physically accessed in the Kemper basement. I talk more about the CSIF in the syllabus.

Do not assume that because your code compiles on an *insufficiently* similar environment (e.g. directly on your Mac laptop), it will compile on the Linux environment used by the CSIF.

You should avoid causes of undefined behavior in your code (i.e. you should avoid things that make your code behave differently in one environment/context vs. another), such as uninitialized variables. If you have things like this in your code, then your code could end up generating the wrong answer when autograded, even if it generated the right answer when you tested it in a sufficiently similar environment.

5 General Details about this Assignment

Filename: `assembler.cpp`

You are required to write C++ code for this assignment, *not* C code.

In this assignment, you will write an assembler that converts assembly code to machine code. Doing this for a real ISA such as x64 or RISC-V would be immensely difficult, so instead, you will work with a fake ISA called Ahrin that I completely made up. That is, you will write an assembler that converts Ahrin assembly code to Ahrin machine code.

The next section solely provides ISA-specific details; the one after that provides assembler-specific details. As you will come to understand as you learn more about how an assembler works (as you read through these specifications and do this assignment), not all details of the ISA are relevant. For instance, the sizes of the registers won't affect your assembler, and – perhaps surprisingly – neither will the behavior of the instructions, although I did bother to make the instructions somewhat realistic.

There are some details that will not make sense until you read a later part. For instance, the purpose of the registers' base-2 representations will not be immediately clear.

As with any assignment, if something about the specifications is ambiguous (e.g. if it is unclear what to do in a certain scenario), then you should ask me for clarification. There are so many specifications in this assignment that I likely missed some important detail. As always, you should refer to the latest version of this document on Canvas.

Your output must match mine *exactly*.

6 The Ahrin ISA

6.1 The Ahrin Architecture

The CPU contains eight 32-bit general-purpose registers (shown with their machine code representations):

- R1: 000_2.
- R2: 001_2.
- R3: 010_2.
- R4: 011_2.

- R5: 100_2.
- R6: 101_2.
- RS (stack pointer): 110_2.
- RB (base/frame pointer): 111_2.

The program counter / instruction pointer is called RP and is not directly accessible to the assembly language programmer, so its representation is not given.

The CPU contains the same flags that x64 does: ZF, SF, CF, and OF. (We discussed these while going through slide deck #3.) These flags are stored in a register called RF that is not directly accessible to the asesmbly language programmer, so its representation is not given.

I do not talk about the details of main memory.

6.2 The Ahrin Assembly/Machine Language

Each line in an Ahrin assembly language program does one (*and only one*) of three things:

1. Contain an instruction.
2. Create a label.
3. Create a variable.

6.2.1 Instructions

The Ahrin assembly language supports twelve instructions. They are listed below, categorized by how many operands they have.

- Two operands: (first operand is "dst", second is "src")
  - `mov`: dst = src
    - Opcode: 0000_2
  - `add`: dst += src
    - Opcode: 0001_2
  - `cmp`: dst - src
    - Opcode: 0010_2
- One operand: (the operand is referred to as "op")
  - `push`: push op onto stack.
    - Opcode: 0011_2
  - `pop`: pop from stack into op.
    - Opcode: 0100_2
  - `call`: call function called op.
    - Opcode: 0101_2
  - `je`: jump to op if ZF flag is set.
    - Opcode: 0110_2
  - `jge`: jump to op if SF flag equals OF flag.
    - Opcode: 0111_2
  - `jl`: jump to op if SF flag doesn't equal OF flag.
    - Opcode: 1000_2
  - `j`: unconditional jump.
    - Opcode: 1001_2
- Zero operands:
  - `ret`: return from function.
    - Opcode: 1010_2
  - `nop`: do nothing ("no operation").
    - Opcode: 1011_2

There are no pseudoinstructions.

6.2.2 Addressing Modes

Any use of the word "constant" or "index" refers to a 32-bit signed integer.
Each operand is given in one of four addressing modes:

1. Immediate mode: a dollar sign ($) followed by a constant or label.
   - Representation: 00_2
2. Register mode: a register, given by name (e.g. `R3`) with no percent sign (%) or any other preceding character.
   - Representation: 01_2
3. Direct mode: a label. This mode is helpful for accessing a variable's value.
   - Representation: 10_2
4. Indexed mode: an index followed by a register in parentheses. This mode is helpful for accessing an element of an array or data on the stack.
   - Representation: 11_2

Restrictions on addressing modes:

- The first operand of a `mov` or `add` instruction cannot be in immediate mode.
- The operand of a `pop` instruction cannot be in immediate mode.
- The operand of a jump instruction (including `call`) must be in direct mode.

Unlike in x64, it is possible for an Ahrin instruction to have two operands that access main memory. This is probably the only thing that Ahrin does better than x64.

6.2.3 Global Variables

The Ahrin assembly language only supports the creation of 32-bit unsigned integer global variables. A global variable is created with the word "var" followed by the variable's initial value.

There is no distinction between data and text sections. It is possible to create a global variable between instructions, even if that would cause chaos.

6.2.4 Labels

As far as Ahrin assembly code goes, labels are the same as in x64 and RISC-V assembly code, except that the syntax is a bit different.

Things become more complicated once we consider machine code. When a label is used in an operand, the label's value must be stored in the machine code representation. This is talked about more in the examples below.

It is NOT an error if an Ahrin assembly language file does not define the `main` label.

6.2.5 Encoding an Instruction in Machine Code

An instruction's machine code representation consists of the following components concatenated together:

- Opcode.
- First operand's addressing mode.
- First operand's representation.
- Second operand's addressing mode.
- Second operand's representation.

If the operand is in register mode, then its representation is just the appropriate sequence of three bits mentioned above. If the operand is in direct mode and is a constant (e.g. `$5`), then its representation is the constant's base-2 representation (using 32 bits). If the operand is a label (whether in immeidate mode like `$sum` or direct mode like `sum`), then its representation is the label's value in base-2 (using 32 bits). (Labels' values are talked about more in the assembler section.) If the operand is in indexed addressing mode, then its representation is the base-2 representation of the index (using 32 bits) followed by the three bit sequence of the register that's in the parentheses.

If the instruction only has one operand, then ignore the last two parts. If the instruction has zero operands, then the representation is just the opcode.

The Ahrin CPU is byte-addressable, so although instructions are variable-length, we don't want any instruction's length to be a non-integer number of bytes / a number of bits that is not a multiple of eight. Thus, each instruction's machine code representation should end in whatever number of zeros is needed to make the instruction's length in bits be a multiple of eight.

Examples of machine code are elaborated on in the assembler section.

6.2.6 Miscellaneous

Comments are not supported by the assembly language.

6.2.7 Example #1

Below is an example of an Ahrin assembly language program. You can find this program (`sum_arr.s`) on Canvas. This program computes the sum of the elements in the array whose starting address is given by the label `arr`. This sum is then placed in the variable whose address is given by the label `sum`.

```
arr
    var 10
    var 20
    var 30
len
    var 3
sum
    var 0
main
    mov R1 $0
    mov R2 $arr
loop
    cmp R1 len
    jge endLoop
    add sum 0(R2)
    add R1 $1
    add R2 $4
    j loop
endLoop
    ret
uselessLabel
```

Each instruction has the instruction name followed by anywhere from zero to two operands. Each operand follows one of the addressing modes described earlier. For instance, the instruction `add sum 0(R2)` has one operand in direct mode (`sum`) and another operand in indexed mode (`0(R2)`). The instruction is saying to load the 32-bit value at the address given by 0 + R2 and add it to the variable whose address is given by the `sum` label.

6.2.8 Example #2

Below is another example of an Ahrin assembly language program. You can find this program (`nonsense.s`) on Canvas. Even though the program makes absolutely no sense, your assembler must be able to support it. After all, it's not like compilers/assemblers tell you if the code you write is nonsensical.

```
    var 19
    var 15
blah
hello
    call func
func
    push $18
    ret
    pop RB
    j blah
    var 45
    nop
    add 4(R3) 7(R2)
```

7 AHA, an Ahrin Assembler

7.1 Overview

Your ultimate goal in this assignment is to implement an assembler. Thus, your program (`assembler.cpp`) will take as its only command-line argument the name of an Ahrin assembly language program. Your program must print the 1s and 0s that make up each byte of the corresponding machine code.

7.2 Assembler Concepts

As the `nonsense.s` example suggests, your assembler need not care about the behavior of the assembly language program. It only needs to convert said program to machine code. Specifically, the variables and instructions must be converted to machine code. The creation of a label does not itself result in machine code, although the use of labels (in operands) affects machine code.

As mentioned above, each global variable is a 32-bit unsigned integer, so a variable's representation in machine code should simply be the base-2 representation of the variable's value.

Converting an instruction to machine code is, for the most part, as simple as concatenating the appropriate sequences of bits. For instance, the machine code representation of `push $18` is the concatenation of the opcode for `push`, two bits to indicate the immediate addressing mode, the 32-bit representation of 18, and whatever number of zeros is needed to achieve a number of bits that is a multiple of eight.

To encode operands that use labels, you must be able to determine the value of a label. As has been mentioned sometimes in lecture, a label contains the value of whatever comes after it, whether an instruction or a variable, and that is basically true here. Since an Ahrin computer, like an x64 computer or a RISC-V computer, is byte-addressable, each address corresponds to a byte. Thus, a label's value is the address of the first byte of whatever follows. For instance, in the below program, the `arr` label contains the address of the first byte belonging to the variable created by `var 10`, the `main` label contains the address of the first byte belonging to the instruction `mov R1 $arr`, and `uselessLabel` contains what *would* have been the address of the first byte of whatever came after it, if there had been anything. The first variable or instruction in the program is always at address #0.

```
arr
    var 10
    var 20
    mov R2 uselessLabel
main
    mov R1 $arr
uselessLabel
```

It would be great if an assembler could do its job in one *pass* through the contents of the file. Unfortunately, because of **forward references**, this is not possible. In the above snippet, the instruction `mov R2 uselessLabel` uses the label `uselessLabel`, which will not be seen until the end of the file is reached. Forward references necessitate **two-pass assembly**. The first pass should determine all labels' values, and the second pass should generate the machine code. Typically, there is a structure called a **symbol table** that maps each symbol/label to its value.

7.3 Example #1

Below is one example of how your program should behave, followed by a breakdown of much of the output and how the labels' values are determined. The file `sum_arr_output.txt` on Canvas contains the output of `./assembler sum_arr.s`, just in case you want it side-by-side with the explanations in this PDF.

```
$ cat sum_arr.s
arr
    var 10
    var 20
    var 30
len
    var 3
sum
    var 0
main
    mov R1 $0
    mov R2 $arr
loop
    cmp R1 len
    jge endLoop
    add sum 0(R2)
    add R1 $1
    add R2 $4
    j loop
endLoop
    ret
uselessLabel
$ ./assembler sum_arr.s
Byte  0: 00000000
Byte  1: 00000000
Byte  2: 00000000
Byte  3: 00001010
```

```
Byte  4: 00000000
Byte  5: 00000000
Byte  6: 00000000
Byte  7: 00010100
Byte  8: 00000000
Byte  9: 00000000
Byte 10: 00000000
Byte 11: 00011110
Byte 12: 00000000
Byte 13: 00000000
Byte 14: 00000000
Byte 15: 00000011
Byte 16: 00000000
Byte 17: 00000000
Byte 18: 00000000
Byte 19: 00000000
Byte 20: 00000100
Byte 21: 00000000
Byte 22: 00000000
Byte 23: 00000000
Byte 24: 00000000
Byte 25: 00000000
Byte 26: 00000100
Byte 27: 10000000
Byte 28: 00000000
Byte 29: 00000000
Byte 30: 00000000
Byte 31: 00000000
Byte 32: 00100100
Byte 33: 01000000
Byte 34: 00000000
Byte 35: 00000000
Byte 36: 00000001
Byte 37: 10000000
Byte 38: 01111000
Byte 39: 00000000
Byte 40: 00000000
Byte 41: 00000001
Byte 42: 00011000
Byte 43: 00011000
Byte 44: 00000000
Byte 45: 00000000
Byte 46: 00000000
Byte 47: 01000011
Byte 48: 00000000
Byte 49: 00000000
Byte 50: 00000000
Byte 51: 00000000
Byte 52: 00100000
Byte 53: 00010100
Byte 54: 00000000
Byte 55: 00000000
Byte 56: 00000000
Byte 57: 00000000
Byte 58: 00100000
Byte 59: 00010100
Byte 60: 10000000
Byte 61: 00000000
Byte 62: 00000000
Byte 63: 00000000
Byte 64: 10000000
Byte 65: 10011000
Byte 66: 00000000
Byte 67: 00000000
Byte 68: 00000000
Byte 69: 10000000
Byte 70: 10100000
```

- Bytes 0-3 (`var 10`): 32-bit representation of 10.
- Since the `arr` label comes before the creation of the aforementioned variable, the value of the `arr` label is 0, which is the address of the first byte of this variable.
- Bytes 4-7 (`var 20`): 32-bit representation of 20.

- Bytes 8-11 (`var 30`): 32-bit representation of 30.
- Bytes 12-15 (`var 3`): 32-bit representation of 3.
- Since the `len` label comes before the creation of the aforementioned variable, the value of the `len` label is 12, which is the address of the first byte of this variable.
- Bytes 16-19 (`var 0`): 32-bit representation of 0.
- Since the `sum` label comes before the creation of the aforementioned variable, the value of the `sum` label is 16, which is the address of the first byte of this variable.
- Bytes 20-25 (`mov R1 $0`): The full representation of this instruction (with underscores separating each byte for readability) is 00000100_00000000_00000000_00000000_00000000_00000000. The first four bits are 0000_2, the representation of `mov`. The next two bits are 01_2, indicating that the first operand is in register mode. The three bits after that are 000_2, indicating that the first operand is R1. The two bits after that are 00_2, indicating that the second operand is in immediate mode. The 32 bits after that are the binary representation of 0_{10}, the second operand. There are then five zeros for padding, in order to ensure that the number of bits in the instruction is a multiple of eight.
- Since the `main` label comes before the above instruction, the value of the `main` label is 20, which is the address of the first byte of the instruction.
- Bytes 26-31 (`mov R2 $arr`): Since the `arr` label has the value 0, the representation of this instruction ends up identical to that of the previous instruction, except that the first operand – which is R2 instead of R1 – is represented with 001_2 instead of 000_2.
- Bytes 32-37 (`cmp R1 len`): The full representation of this instruction (with underscores separating each byte for readability) is 00100100_01000000_00000000_00000000_00000001_10000000. The first four bits are 0010_2, the representation of `cmp`. The next two bits are 01_2, indicating that the first operand is in register mode. The three bits after that are 000_2, indicating that the first operand is R1. The two bits after that are 10_2, indicating that the second operand is in direct mode. The 32 bits after that are the binary representation of 12_{10}, which – as mentioned above – is the value of the `len` label. There are then five zeros for padding, in order to ensure that the number of bits in the instruction is a multiple of eight.
- Since the `loop` label comes before the above instruction, the value of the `loop` label is 32, which is the address of the first byte of the instruction.
- Bytes 38-42 (`jge endLoop`): The full representation of this instruction (with underscores separating each byte for readability) is 01111000_00000000_00000000_00000001_00011000. The first four bits are 0111_2, the representation of `jge`. The next two bits are 10_2, indicating that the operand's addressing mode is direct mode. The 32 bits after that are the binary representation of 70_{10}, the value of the `endLoop` label. There are then two zeros for padding, in order to ensure that the number of bits in the instruction is a multiple of eight.
- Bytes 43-52 (`add sum 0(R2)`): The first four bits of this instruction's representation are 0001_2, the representation of `add`. The next two bits are 10_2, indicating that the first operand is in direct mode. The next 32 bits are the base-2 representation of the value of the `sum` label, which is 16_{10}. The next two bits (which are the last two bits of byte 47) are 11_2, indicating that the second operand is in indexed mode. The next 32 bits are the base-2 representation of 0_{10}, since the index in the second operand (`0(R2)` is 0_{10}. The next three bits are 001_2, indicating that the register in the parentheses is R2. There are then five zeros for padding.
- Bytes 53-58 (`add R1 $1`): The representation of this instruction follows logic similar to that of the representation of `mov R1 $0` above, except that the representation of `add` and the second operand are different.
- Bytes 59-64 (`add R2 $4`): The representation of this instruction follows logic similar to the instruction immediately above this one.
- Bytes 65-69 (`j loop`): The representation of this instruction follows logic similar to that of the representation of `jge endLoop`.
- Byte 70 (`ret`): The representation of this instruction (which has no operands) is simply 1010_2 (the representation of `ret`) followed by four zeros for padding.
- Since the `endLoop` label comes before the above instruction, the value of the `endLoop` label is 70, which is the address of the first byte of the instruction.
- Since the `uselessLabel` label comes immediately after the aforementioned instruction, the value of the `uselessLabel` label is 71, which is the address of the first byte after the instruction.

7.4 Example #2

Below is another example of how your program should behave.

```
$ cat nonsense.s
    var 19
    var 15
blah
hello
```

```
    call func
func
    push $18
    ret
    pop RB
    j blah
    var 45
    nop
    add 4(R3) 7(R2)
$ ./assembler nonsense.s
Byte  0: 00000000
Byte  1: 00000000
Byte  2: 00000000
Byte  3: 00010011
Byte  4: 00000000
Byte  5: 00000000
Byte  6: 00000000
Byte  7: 00001111
Byte  8: 01011000
Byte  9: 00000000
Byte 10: 00000000
Byte 11: 00000000
Byte 12: 00110100
Byte 13: 00110000
Byte 14: 00000000
Byte 15: 00000000
Byte 16: 00000000
Byte 17: 01001000
Byte 18: 10100000
Byte 19: 01000111
Byte 20: 10000000
Byte 21: 10011000
Byte 22: 00000000
Byte 23: 00000000
Byte 24: 00000000
Byte 25: 00100000
Byte 26: 00000000
Byte 27: 00000000
Byte 28: 00000000
Byte 29: 00101101
Byte 30: 10110000
Byte 31: 00011100
Byte 32: 00000000
Byte 33: 00000000
Byte 34: 00000000
Byte 35: 00010001
Byte 36: 01100000
Byte 37: 00000000
Byte 38: 00000000
Byte 39: 00000000
Byte 40: 11100100
```

7.5 Input Validation

Unless explicitly specified (e.g. in the cases below), do not worry about input validation. For instance, do not worry about the autograder doing something like providing three operands to the `add` instruction or making up an addressing mode.

If the wrong number of command-line arguments are provided, then the error message shown below must be printed to standard error, and your program must exit with exit code 1.

```
$ ./assembler
Wrong number of arguments.
$ echo $?
1
$ ./assembler a b
Wrong number of arguments.
$ echo $?
1
```

If the assembly language file cannot be opened, then your program must report this in the way shown below.

```
$ ./assembler nonexistent_file
Failed to open file.
$ echo $?
```

```
2
$ ./assembler nonexistent_file 2> /dev/null
$
```

If the assembly language program contains a duplicate label, then your program must report this in the way shown below.

```
$ cat duplicate_labels.s
foo
bar
foo
    j bar
$ ./assembler duplicate_labels.s
At least one duplicate label found.
$ ./assembler duplicate_labels.s 2> /dev/null
$ echo $?
3
```

7.6 Parsing Assembly Code

There are many challenges in parsing a language, whether a high-level language or an assembly language. For instance, a C++ programmer can have randomly placed whitespace, and if the body of an `if` statement only has one statement, the programmer may omit curly braces. Such situations make parsing a challenging, well-studied subject. You will learn more about parsing, grammars, and other related subjects in ECS 140A and perhaps in other courses like ECS 120.

For this assignment, I do not want parsing to be a big challenge[1], so in this subsection, I talk about details of the assembly language or C++ tips that should make parsing easier. It is OK to take advantage of some of these details in ways that make you feel as if you are writing code of less-than-ideal quality. For instance, the next paragraph implies that you could check if a line in an Ahrin assembly language program contains a label simply by checking if the first character is not a whitespace. Of course, if you have time on your hands and want to write high quality, extensible code[2], then please feel free to do so.

Blank lines are prohibited in an Ahrin assembly language program. If a line contains a label, then the label will always be at the start of the line. Otherwise, the line will begin with exactly four whitespaces before the word "var" or an instruction is encountered.

Each line will contain only one of a variable creation, label creation, or an instruction. There will never be a mix. For instance, there will not be a line that contains two instructions or a line that contains an instruction and the creation of a variable.

A label's name only consists of letters and is at most 15 characters.

You may assume that an instruction will never use an undefined/uncreated label in an operand[3].

The instruction name (e.g. `mov`), first operand, and second operand will always be separated by one whitespace. There will never be trailing whitespaces. There will not be whitespace within an operand, e.g. you will never see something like `R 1` instead of `R1` or `0 ( R2 )` instead of `0(R2)`.

Make sure you are somewhat familiar with the many helpful features C++ provides you when it comes to parsing strings. Don't be afraid to make use of C++ STL containers.

8 Epilogue: Additional Features and Further Challenges

Do not add anything from this subsection into what you end up submitting to Gradescope.

Below is a non-exhaustive list of things you could do with this assignment *on your own time* (again, not for your submission to Gradescope), in order to make it more complex and, perhaps, more résumé-worthy.

- Implement the assignment in C.
- Output a binary file. That is, write the 1s and 0s as bits to a file (which will create a file that appears to contain garbage, if you were to view its contents, just like what happens with an actual executable or object file) instead of printing out the 1s and 0s. This is NOT the same as a text file that contains 1s and 0s (i.e. a text file that contains the ASCII values of a bunch of '1' and '0' characters).
- More validation / assembler errors.
- Detection of undefined labels.
- Support constants in other bases. Decide on some prefix (e.g. `0x` for base 16) to distinguish between different bases.

[1] This is a big part of the reason that I decided to have you do this assignment in C++ instead of C. Doing anything with strings, including parsing, is painful in C.

[2] "Extensible code" refers to code that was written in a way that makes it easy to add features, e.g. if you wanted to extend the ISA to support eight more registers. Being able to write extensible code is a useful skill.

[3] A real assembler would leave a "note" (in some format) indicating that a label is undefined. There would not be an assembler error, because the label could be defined in another file. The linker – which brings together multiple files – figures this out.

- Implement *relative addressing*. In our assembler here, whenever a label was used, the operand required 32 bits when stored in machine code. (The problem would be even worse if addresses were 48 or 64 bits.) Relative addressing significantly decreases the number of bits needed to represent such an operand.

9 Autograder Details

Once the autograder is set up on Gradescope, I will send a Canvas announcement and add additional details below.

If you haven't already, you should read what I say in the syllabus about the Gradescope autograder.

UCDAVIS
COMPUTER SCIENCE