Project Checkpoint 1 Specification[[1]](#footnote-1)

In the first part of your project, you will write an ***assembler*** that will process one or more .asm files containing MIPS assembly into two binary files: one containing the binary encoded instructions, and another containing the contents of static memory (static variables, etc.).

In later steps of this project, you will load these binaries into a simulated processor that will execute these instructions.

# Running your program

Your main program will be written in C++. It should compile with a MakeFile, and the final target should be an executable called “assemble”.

Your program should accept commandline arguments. The first arguments are the names of the MIPS assembly files to be read. The next is the name of the binary file that you will write the static memory binary output out to. The last is the name of the binary file that you will write the instruction binary output to.

Expected usage(s):

>> make  
>> ./assemble test1.asm test1\_static.bin test1\_inst.bin  
>> ./assemble test2a.asm test2b.asm static\_output2.bin inst\_output2.bin

Expected behavior: The first line compiles your C++ assembler into an executable called assemble. The second line runs this assembler on the input assembly file test1.asm, and outputs the assembled static binary into test1\_static.bin and the instruction binary into test1\_inst.bin. The third line does the same but with two assembly files (test2a and test2b, in that order).

# Input

The input files to be assembled will be (simplified) MIPS assembly files. This is a list of the instructions that you will have to handle:

* add, addi, sub
* mult, div, mflo, mfhi
* sll, srl
* lw, sw
* slt
* beq, bne
* j, jal, jr, jalr
* syscall

You will also have to handle the load address “la” pseudoinstruction that helps access static memory.

Registers can appear in your input in either conventional usage like “$ra” or in numerical form like “$31”. You need to handle both cases. Immediates used in addi may be negative, but will always fit in a 16 bit integer.

In addition to the above MIPS commands, there may also be labels in the MIPS input file. The labels do not count as instruction and aren’t given their own line number. The line number of a label refers to the line number of the instruction immediately following the label. There may also be comments in the MIPS file, and you should ignore those.

# Special Instructions

**jalr**

This instruction is new to you! It means **jump and link register**, and it lets you choose where to jump based on what’s in a register.

* When specifying only one register: jalr $r0
* means: store the next line number (PC + 4) into register $ra ($31), and then jump to line $r0. This would be encoded as:
* 000000 [r0] 00000 11111 00000 001001

Sources disagree about the syntax of two-variable jalr. Use the following interpretation:

* Instruction: jalr $r0, $r1
* means: store the next line number (PC+4) into register $r1, and jump to line $r0.
* encoding: 000000 [r0] 00000 [r1] 00000 001001
* For example, jalr $t0, $t1 would be  
  000000 01000 00000 01001 00000 001001
* **Syscall**

Although this isn’t quite how it works in the real world, for our convenience, we will encode syscall as:

* Instruction: syscall
* encoding: 000000 00000 00000 11010 00000 001100  
  The reason for this will make more sense later.

**Static Memory**

To load the address of something in static memory, you need to have the address of that label in static memory. You should imagine that static memory starts at address 0 (not how it works in the real world), and each integer or function address is 4 bytes.

The la instruction is a pseudoinstruction. In real assemblers, la is translated into multiple instructions to deal with constants greater than 16 bits. For this project, **translate la into an appropriate addi** (where the constant to add is determined by the assembler after determining the address of all static memory labels). You may assume the constant will not exceed 16 bits.

The static memory output should be a list of all the integers present in static memory.

# Branches vs Jumps

Make sure you understand the difference between how branches and jumps are encoded.

* Jump instruction formats with a 26 bit target address have the *line number* of the target stored in those 26 bits. Unlike real world MIPS, you should start your counting at line number 0.
* Branch instructions don’t store the whole target address in the encoding, but instead store the *offset* between the next line after the branch and the target destination. This amount is also measured in *lines*.
* Instructions like jalr, jr, and la involve jumping to/from instruction numbers that are stored in registers, not ones that are encoded in the instruction itself. These instructions will read/write line number addresses that have been placed into registers *measured in bytes*, not lines. This number will be four times larger than the line number itself.

# Code Requirements

Your project part 1 must be written in C++, have appropriate header file(s), and be compiled with a makefile. You \*may\*, but do not have to, use multiple classes/files to organize your code. You \*must\* organize your code using good design practices, including:

* Separating code into functions that have a concise, encapsulated purpose that are not too small or too large.
* Appropriate use of data structures (built in or user defined) to store information.
* Appropriate use of library functions and methods (don’t reinvent the wheel)
* Good variable names.
* Comments.

You are provided with starter code that can handle add instructions only. Understand how it works, especially the string processing utility functions provided. You will need to change the provided code only a little to handle many more arithmetic instructions (sll, srl, mult, sub, etc.). You will need to change the provided code more substantially to handle static memory and branches/jumps.

You are also provided with a readbytes file that can help you read the contents of a raw binary file. Combined with command line “diff” or VSCode’s “Select for Compare,” you can compare the binary output of your code with the given tests.

# Example Output

The lines in your output should be binary encodings of the given MIPS assembly code in the input file.

For example if the MIPS assembly input was:

.data  
 b: .word 4 f  
 array: .word 3 4 5 #This comment must be ignored  
 .text  
 .globl main  
main:  
 #This comment must be ignored  
 addi $s0, $zero, 100 #this comment needs to be ignored  
loop:  
 add $a0, $s0, $0 #call f with parameter i  
 jal f  
 addi $s0, $v0, 0 #i = f(i)  
 bne $s0, $zero, loop  
 addi $v0, $0, 10  
 syscall  
f:  
 la $t0, array  
 srl $v0, $a0, 1  
 jr $ra

Then the static memory output would be (spaces and parentheses added for clarity, not part of the binary)

00000000 00000000 00000000 00000100 (4)  
 00000000 00000000 00000000 00011100 (28 - instruction number of f (7) in bytes (\*4))  
 00000000 00000000 00000000 00000011 (3)  
 00000000 00000000 00000000 00000100 (4)  
 00000000 00000000 00000000 00000101 (5)

And the instruction output (translated into binary MIPS format with artificial spaces) would be:

001000 00000 10000 0000000001100100  
000000 00000 10000 00100 00000 100000  
000011 00000000000000000000000111 (jal f - f is on line 7)  
001000 00010 10000 0000000000000000  
000101 01000 00000 1111111111111100 (jump -4 lines back : (bne line + 1 - label line))  
001000 00000 00010 0000000000001010  
00000000000000001101000000001100 (syscall)  
001000 00000 01000 0000000000001000 (la = addi - array begins at byte #8 in static mem)  
000000 00000 00100 00010 00001 000010 (srl)  
000000 11111 000000000000000 001000 (jr $ra)

The spaces won’t be there in the final file, the spaces are to help you see the instruction format.

# Testing

Your starter file includes .asm files for five test programs and the corresponding gold output binary files. To view a binary file, compile the provided readbytes.cpp and run the result on the file you want to view; e.g.,

g++ -std=c++17 readbytes.cpp -o readbytes  
./readbytes Testcases/GoldBinaries/test1inst.bin

Probably the easiest way to use these testcases is to dump the readbytes output for the gold into one .txt file, the readbytes output for your corresponding binary into another, and diff the results. If there's no difference, then you've passed the testcase. If there's a difference, you can see what it is.

In addition, you will be generating a test suite as part of the corresponding applications & analysis assignment. **You should use that additional test suite to make sure you haven't missed any edge cases.**

# Advice

Disclaimer: You do not have to follow the following advice.

**Order that you should do work:**

* Understand the starter code as given, seeing that the code runs in three main “phases”.
* Run the assembler on demo1.asm, and use readbytes.cpp to compare your output on demo1.asm with demo1inst.bin
* Add support for the "easy" arithmetic instructions - this involves phase 3 only.
* Use readbytes.cpp to check your output on test1
* Use readbytes.cpp to check your output on test4
* Add support for labels, branches and jumps - this involves phases 1 and 3
* Use readbytes.cpp to check your output on test3
* Add support for static memory - this involves phases 1 and 2
* Use readbytes.cpp to check your output on test2 (single file)
* Use readbytes.cpp to check your output on test5 (multiple files)
* Use readbytes.cpp to check your output on all of the tests from your Application/Analysis 2a assignment.

# Interview

After you submit the first portion of your project, you **must** schedule an interview with me to discuss how your project works, what design choices you made and why, etc. I’ll also give my comments to you about your project and make recommendations that you should change/fix before part 2 of the project.

I will assign your project checkpoint grade (incomplete, retry, or achieved) based on the interview. To get an “achieved”, you need to show me that your assembler works as described in this document *and* that you fully understand how it works.

# Peer Evaluation

As explained in the Project Teams document, you must complete a peer evaluation for every checkpoint. You will not earn an “achieved” if you do not submit a peer evaluation.

# Challenges

In addition to requirements, each checkpoint includes some challenges. Each challenge has a number of "stars" associated with it approximately reflecting its difficulty. Success on a checkpoint requires that you complete at least a specified number of stars.

If you complete a total of 100 stars beyond what is required of you over the course of the project, you will get a half letter-grade increase in your grade for the course. That is, if you otherwise would earn a B+, you will receive an A-. Stars for this purpose need to be completed *by the end of the project*; however, I recommend that you work on the challenges for the current checkpoint while your code is fresh in your mind.

**Challenge Requirements for Project Checkpoint 1:**

Teams must complete at least 10 stars' worth of challenges for this checkpoint.

**Available Challenges for Project Checkpoint 1:**

You can earn a maximum of 22 stars on this assignment.

One star:

* Implement both of the following MIPS pseuodinstructions by translating them into the above “real” instructions first: mov, li. One star for implementing both.
* Implement the following MIPS pseuodinstructions by translating them into the above “real” instructions first: sge, sgt, sle, seq, sne bge, bgt, ble, blt, abs. (Hint: adding new labels is a pain. Try to figure out a way that doesn’t require new labels). One star per pseudoinstruction.
* Implement the following real MIPS instructions in your assembler: and, or, nor, xor, andi, ori, xori, lui. One star per two instructions.

Two stars:

* Handle immediates that are larger than 16 bits, but smaller than 32 bits.

Five stars:

* Handle static memory strings, which appear with the tag .asciiz. For example,

.data  
 .asciiz "Hello, world."

Although a real-world character string needs only 1 byte per character, you should use 4 bytes per character in your output with the rest of the bits being leading 0s.

The above pseudoinstructions do NOT have a MIPS encoding. You must translate them into one or more of the "standard" MIPS instructions first, potentially making use of the $at register as needed.

# What to submit

Submit all source files to Blackboard. Submit an AI log (you may submit one shared log for both team members or separate logs, whichever is easier for you).

Submit your peer evaluation using the CATME tool.

1. Many thanks to Dr. Prateek Bhakta, who designed the original version of this project. Credit for anything that works belongs to him; blame for any mistakes belongs to Dr. Dollak. [↑](#footnote-ref-1)