Garrett Rademacher

Professor Pfaffmann

Computer Science 203

22 October 2017

Project One Writeup

The goal of this project was to create a toll-chain of three separate programs: an assembler to compile our version of the assembly language called “little-finger” into machine code, a visualizer to examine and debug the produced machine code, and a processor to run the produced machine code. The major challenge of this assignment was designing essentially our own programming language, granted we were able to reference and pull from assembly. The little-finger language that I ended up producing is very similar to standard assembly with some restrictions and other liberties taken. The commands that were implemented in the Assembler as well as their argument formats can be seen in figure 1. To able to translate these into machine code, I took a similar approach to assembly in which each instruction has a format code that dictates how it is translated into machine code. The format codes were largely dictated by the number of arguments that each instruction accepted followed by the type of instructions. For example, even though both ADD and ADDI both accept three arguments, three registers for ADD and 2 registers plus a number for ADDI, because they have different types of arguments, they receive different format codes. This was done to be able to better consolidate the functions required for translating the instructions and their arguments into machine code. In figure 2, the arguments and their format codes as well as their operation codes can be seen. Figure 3 displays the format of the different types of machine codes. Both the Assembler and the Processor convert instructions based on these formats as they are hard coded into both programs. The other big instruction categories that are in little-finger, are the labels (tags), and directives. Labels are

mainly used for branching, however are also used to denote the start of the program, and the stack. Both the main label and the stack label are required in my program as part of my design decisions. Another label, directive related design decision is that labels, directives, and instructions can only occupy one line and there is only one command is allowed per line, i.e. there cannot be a label and an instruction on the same line. This was done to make parsing the .as file much easier as the program only has to worry about parsing one command at a time and not splitting a line up with multiple commands. Furthermore, there are three required directives, they are: wordsize, maxmem, and regcount. These three directives are required as they are all integral to the running of the Processor and the Processor cannot function without either of these three items being specified. Examples of the labels and directives can be found in figure 4. The following should be used as a general guideline when designed programs in my version of little-finger as they pertain to the general accepted structure and format of the programs that will be run successfully:

- The first three lines of the .as file are intended for the maxmem, regcount, and wordsize directives, while they can theoretically be placed anywhere in the .as file, the first three lines are the best place for them.
- The default word size is 32 as that is the size of commands and the most used data sizes. This specification means less interpretation and work that is required by the assembler and processor.
- The maximum number of registers is 32. This is due to the rm, rn, rd, and rt fields in the machine codes are limited to 5 bits and the largest register that can be specified successfully is 31. Additionally, register number run from zero to the specified register count minus one. This means no negative register numbers.

- Little-Finger that should be run in the .as file should be placed between the ‘main’ label and the HALT command as the simulator starts running from the location of the ‘main’ label and stops running at the halt command. Failure to do this will result in general data being interpreted as instructions.
- When writing instructions, instruction arguments (registers, numbers, etc.) must be separated by comas. The use of ‘X’ before a register and ‘#’ before a number are optional but are recommended as they make the little-finger code easier to read. Additionally, the use of brackets ‘[]’ in instructions such as LDUR and STUR are optional but once again recommended.
- When defining and using labels, the label must be defined on its own line using the syntax ‘label:’ before the label can be used in branching statement. This is because the assembler is a one pass compiler.
- All data directives, double, single, half, and byte, except the single should be proceeded by an align command with size eight so they can be properly read and processed by the processor when dealing with memory as the default word size is 32 bits.

The following are set of design decisions that are command specific that were all made to streamline the program, prevent overcomplicated functions, and to streamline the design:

- When using branching commands such as B, CBZ, or CBNZ (among others), the only allowed branching address is a label, and not an address in memory. This is to avoid the issue of dealing with relative branching and makes the program much simpler.
- When using commands that take an ALU Immediate, the ALU number cannot be larger than (2^16)-1. This is because the ALU gets 16 bits in the machine code and the most significant bit is used for the sign, leaving the rest to be used for any numbers.

- The LSR command will do both logical and arithmetic shifting depending on whether or not the number is signed. On the other hand, the LSL command only does logical shifting.
- The LDUR and STUR commands are the only fully functional commands implemented in the processor (the other commands will still be translated into machine codes but will not be run). This is due to the time constraints of the project and the complexity required to implement these commands. Additionally, LDUR and STUR pull single sized chunks of memory (32 bits) due to this being the default word size.
- CBNZ, CBN, and B are the only fully functional branching commands that are implemented in the processor once again due to the time constraints and complexity of the project.
- The align directive takes arguments that are in terms of bytes and not bits as in standard assembly.

The following are design decisions related to the overall design of the assembler, visualizer, and processor:

- The assembler and processor both use hexadecimal for their input and output while the visualizer supports both hexadecimal and binary notation for viewing commands. That being said, the visualizer will save the commands in hexadecimal due to that being the standard notation of both the assembler and the processor.
- When making changes to the hexadecimal code in the visualizer, the opened file will be overwritten when the save button is pressed meaning any code that the user does not want to be overwritten should be saved under a different name or moved to a different directory.

- Furthermore, if a file already exists with the name given to the assembler, it will be overwritten when creating the new .o file.
- When saving the state of the processor in the processor GUI, the memory image will always be saved with the name given with 2 appended. For example, if the file being processed was “test.o”, then the new image file would be “test2.o”. The processor status will also be saved as the file name with the addition of the string “_CPU_State”. Going back to our previous example, the output file for the processor state would be “test_CPU_state.txt”. Once again, if a file already exists with these names, they will be overwritten.

The assembler program will catch most errors in the format of commands, i.e. incorrect number of arguments, however it will not catch all errors made by the user and as such, these design decisions are specified to prevent an incorrect command.

The simulator design was relatively easy to create once the assembler was created. The goal was just to reverse engineer the structure of the machine code, decode it, and run the specified commands. The simulator begins by taking in a .o file and parsing it. First the header line that contains all of the passed arguments such as wordsize, register count, location of stack and main, etc. These arguments are saved and the general structure of the CPU is created based on these arguments. Second, the second line of the .o file, which contains the hexadecimal code, is parsed and saved into memory. After this, the GUI is built and populated with the registers, little-finger code, and memory. At this point, the program waits until the user interacts with the GUI to begin running the program either one step at a time or continuously until the program hits the HALT command. Both of these run options utilize the runStep() method which runs the command at the current command index.

When a command is run, the processor pulls the command name and format code from a hash table which uses the op code as the key. Depending on the format code, the machine code line is parsed to get the various arguments that the machine code contains. The op code, operation names, and instruction arguments are then passed to an execute method depending on the format code. Depending on the op code, the command is executed and if necessary, flags are set. After this, the command counter increases by one to keep track of the current location in the command ArrayList. Following the execution of a command, the GUI is updated and re-displayed.

The big data structures used in the program are a Hashtable, several ArrayLists, and a custom memory container. The Hashtable stores the op codes as the key and a string containing the format code and operation name as the value. This is used when deciding how a command should be decoded and what operation should be done on the arguments/registers involved. The ArrayLists are mainly used to keep track of memory directly in the processor class for easy access while the CPUMemFill object is used for filling and storing the memory temporarily while the program is staring up.

The rest of the methods, instance variables are used for general upkeep and tracking of what is happening in the program, such as the flags.

The file structure of the program is also important to discuss. All of the text files and produced or used files in the program are kept in a separate directory from the java files and the class files. This is to prevent cluttering. All files that are either produced or used in the program, such as the .as file, .o file, instruction set file, etc. are stored in the “Files” directory for easy access. The program is set up so as long as there is at least the instruction set file and one .as file, the rest of the files can be generated by the program and run in any directory

environment. The Makefile is in the main parent directory, it goes through all three program directories and compiles the files into the “compiledPrograms” files where they can be run from the command line. Each program, the visualizer, assembler, and processor, has its own directory for enhanced organization. Each of the three program directories contains its respective Javadoc.

In the Files directory, there are several example programs which demonstrate the functionality of the implemented little-finger language which can be run.

In order to run the programs, the following need to be done, after compile. The assembler program, “Assembler”, must be passed a file name with the .as extension. This a corresponding .o file will then be placed in the Files directory. The visualizer program, “Visualizer”, must be passed a file name with the .o extension. The processor, “CPU”, must be passed both a file with the .o extension as while as a second argument, either ‘true’ or ‘false’, to denote if noisy mode should be turned on or not. If any of these conditions is not met for the program, an error message will be displayed which corresponds to the argument that was passed which was incorrect or missing.

Figure 1. Instructions Implemented and example arguments.

```
ADD X1, X2, X3          ADDI X1, X2, #1
SUB X1, X2, X3          SUBI X1, X2, #1
ADDS X1, X2, X3         ADDIS X1, X2, #1
SUBS X1, X2, X3         SUBIS X1, X2, #1

AND X1, X2, X3          ANDI X1, X2, #1
ORR X1, X2, X3          ORRI X1, X2, #1
EOR X1, X2, X3          EORI X1, X2, #1
LSL X1, X2, #1          LSR X1, X2, #1

LDUR X1, [X2, #8]       LDURSW X1, [X2, #8]
STUR X1, [X2, #8]       STURW X1, [X2, #8]
LDURH X1, [X2, #8]      LDURB X1, [X2, #8]
STURH X1, [X2, #8]      STURB X1, [X2, #8]

CBZ X1, main            CBNZ X1, main
BCOND X1, main

B main                  BR main
BL main

PUSH                    POP

HALT                    NOP
```

Figure 2. Instructions, their format codes, and op codes.

```
ADD A 1          ADDI B 3
SUB A 2          SUBI B 4
ADDS A 5         ADDIS B 7
SUBS A 6         SUBIS B 8


AND A 9          ANDI B 12
ORR A 10         ORRI B 13
EOR A 11         EORI B 14

LSL B 15         LSR B 16

LDUR C 17        LDURSW C 19
STUR C 18        STURW C 20
LDURH C 21       LDURB C 23
STURH C 22       STURB C 24

CBZ D 25         CBNZ D 26
BCOND D 27

B E 28           BR E 29
BL E 30

PUSH G 31        POP G 32

HALT F 63        NOP F 0
```

Figure 3. Layout of the different machine code formats.

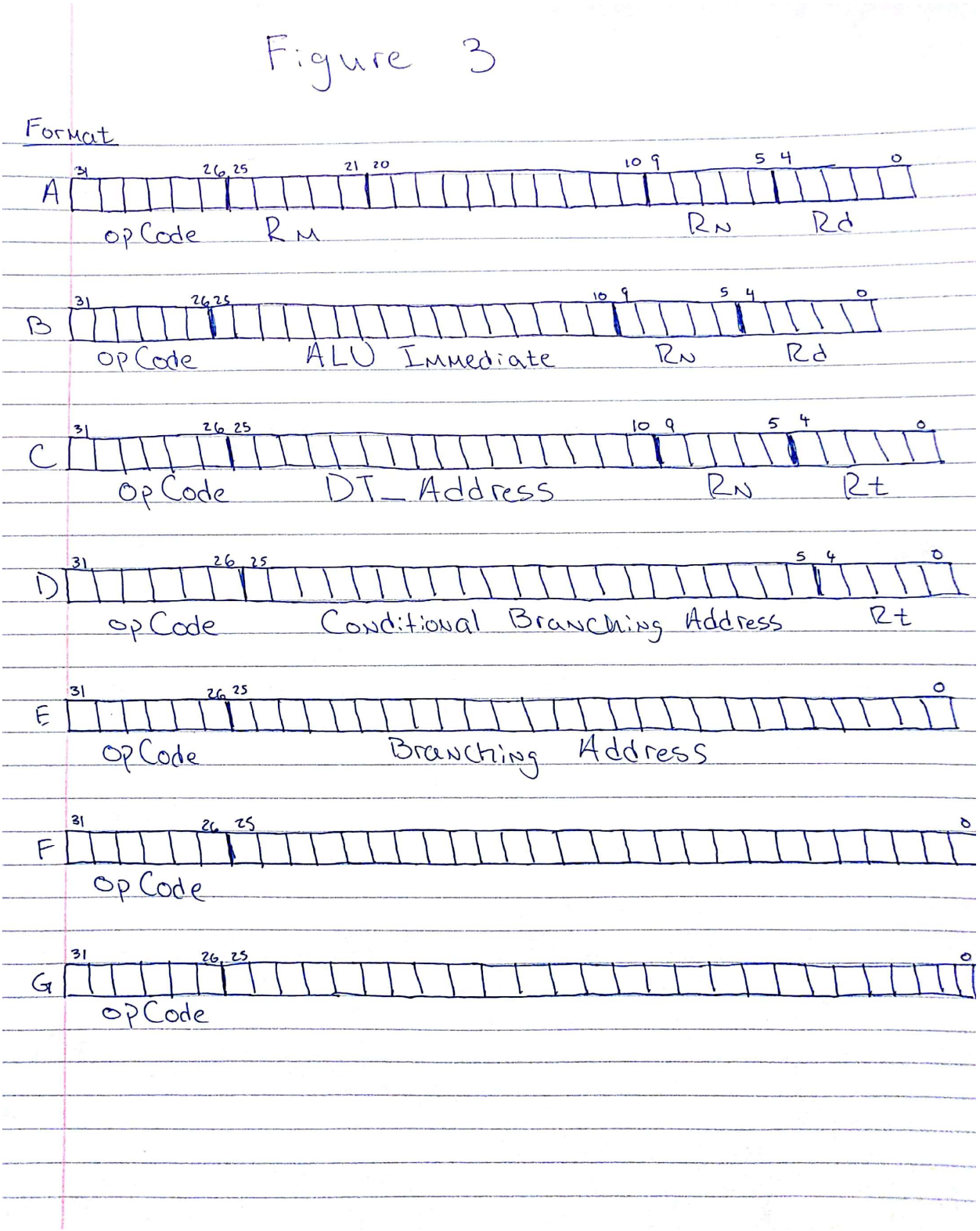

Figure 4. Examples of labels and directives

```
main:           - specifies the start of the program

stack:          - specifies the program stack

loop1           - example label which can be looped back
                to using one of the branching commands

.wordsize 32    - bit size of the registers

.regcnt 10      - number of registers
                (indexable registers run from zero to 9)

.maxmem 0x200   - number of addressable bytes in the memory

.align 8        - align the next data directive
                to 8 bytes

.double 0x0AB   - create a double in memory with the value AB

.single 0x0AB   - create a single in memory with the value AB

.half 0x0AB     - create a half in memory with the value AB

.byte 0x0AB     - create a byte in memory with the value AB

.pos 0x200      - move the current memory index to 200
```