ECE 350

Assembler Documentation

Matthew Dickson
Last Revision: February 13, 2019

Contents

1 Overview

The assembler is designed to take in one or more files containing the modified MIPS-like ISA for ECE 350 and output a .MIF file of the assembled code while running in django server in a Docker container. This allows instructors and other maintainers to ensure continuous deployment and potentially expand the assembler's functionality without students needing to worry about re-downloading any software. The assembler can also take in new instructions, instruction types, and named registers from students on a per-run basis, a large improvement over the previous Java version.

2 Usage

2.1 Students

NOTE: This section was last updated on 28 January 2019. Please consult a TA to ensure the given ISA is in-date if taking the class after 2019S.

2.1.1 Process

1. Write your code in any file, named `<foo>.s` or `<bar>.asm`
2. Goto 350assembler.colab.duke.edu
3. Usining the onscreen GUI, upload your code file(s), as well as JSON files containing additional instructions, instruction types, or named registers (see JSON Resources in this document for formatting information). You can pass as many assembly files at once as desired, but only one JSON file per field.
4. Save the returned zip file containing your code!

2.1.2 Allowed Syntax within MIPS

Without any additional student resources, any MIPS command is legal syntax. As of writing, those are:

```
add   $rd,   $rs,   $rt
addi  $rd,   $rs,   N
sub   $rd,   $rs,   $rt
or    $rd,   $rs,   $rt
and   $rd,   $rs,   $rt
sll   $rd,   $rs,   shamt
sra   $rd,   $rs,   shamt
mul   $rd,   $rd,   $rt
div   $rd,   $rs,   $rt
sw    $rd,   N($rs)
lw    $rd,   N($rs)
j     T
bne   $rd,   $rs,   N
jal   T
jr    $rd
blt   $rd,   $rs,   N
bex   T
setx  T
```

Additionally, the assembler supports `nop` and `mult` as keywords by default, so students can use them in as syntax as well.

2.1.3 General Syntax Notes

Just as in standard MIPS, the comment start character is `#`, and there's no block comment command. Students can make comments in line (ie, `addi $1, $0, 10 # r1 is now 10`) or on a line of its own. All comment only lines get removed when assembling code, so they cannot be used as separators of code sections. For example, consider the below snippet:

```
1 addi $r2, $r0, 5
2 loop:
3   addi $r1, $r1, 1
4   # Only a comment here
5   # Maybe I'm explaining how loops work?
6   bne $r1, $r2, loop
```

The assembler will remove lines 4 and 5 before parsing the actual assembly, so the `bne` will get an immediate value of -2, since it will only see the `addi` between it and loop. The same is true for lines that are either blank or have just a jump target, like line 2 of the above example.
To specify a register, you can use either standard MIPS naming, like `$1`, or a modified version seen in previous semesters of this class, like `$r1`. You can mix these syntaxes within a file if you really hate maintainable code. The assembler recognizes named registers as well. By default, it only understands `$ra` and `$rstatus`, which map to register 31 and 30, respectively. If you choose to add more named registers they *must* start with the $ character. See the Restrictions and Illegal Syntax section for more information.
All parts of an instruction *must* be whitespace delimited. While the assembler does its best to be whitespace agnostic, try to keep these breaks to spaces (or tabs if you have not culture) since only those two have been tested. You can optionally also add commas between pieces and end each line with a semicolon. Basically, all of the below will compile to set register 1 to 10:

```
1 addi $1, $0, 10
2 addi $1  $0  10
3 addi $r1  $r0 10
4 addi $1, $r0  10;
```

For a few examples of what a full file should look like, see the Examples section in the appendix.
If there's a basic MIPS syntax error on a line, the assembler will replace the line in the output with an error message for what went wrong. If unable to load the MIF file into memory, check that the file assembled correctly and there's no error messages. Note that the syntax checking is far from exhaustive, so don't rely on this for error checking.

2.1.4 Restrictions and Illegal Syntax

All of the below are illegal, and will cause the assembler to either act unexpectedly or replace the entry with an error message.

- Giving an immediate not representable with the provided bits
- Having more than one label on a line
- Having too many or too few arguments for the given instructions
- Labels cannot contain any of these characters: `( ) , ; $`
- Named registers cannot contain any of these characters: `( ) , ;`
  Additionally, the `$` character can only be used for the first character of the register name.

2.2 Process

For the curious, the assembler broadly works like this:

1. Take all the received files and dump them into their own unique temp disk files to save on memory.

2. Make a zip archive to contain all return files, and list to hold all the `File-Like` objects while the the program runs
3. Run the parser over the firs file to find jump targets and filter out lines without instructions
4. Go through the file, line by line, and make an Instruction object for that line
5. Convert all the Instructions to strings and dump them into a `ByteIO`, stored in the temporary list
6. Reset the parser to clear local jump targets
7. Repeat 2-5 until there are no more files
8. Put the contents of all the `ByteIO`s into their own files within the zip achieve

3 JSON Resources

3.1 Instruction File

The instruction file tells the assembler what instruction are allowed, as well as how to parse their arguments. Critically, this file doesn't specify how the instruction is formatted as a binary string, so there's no need to note how a given instruction type is assembled in this file. Below is an excerpt from the default file as of 29 Jan 2019 to show how to format a file to declare instructions.

```
{
  "add":  {"type": "R",   "aluop":  "00000", "syntax": ["rd", "rs", "rt"]    },
  "addi": {"type": "I",   "opcode": "00101", "syntax": ["rd", "rs", "immed"]},
  "sll":  {"type": "R",   "aluop":  "00100", "syntax": ["rd", "rs", "shamt"]},
  "lw":   {"type": "I",   "opcode": "01000", "syntax": ["rd", "immed", "rs"]},
  "j":    {"type": "JI",  "opcode": "00001", "syntax": ["T"]                 },
  "bne":  {"type": "I",   "opcode": "00010", "syntax": ["rd", "rs", "immed"]}
}
```

Each instruction name is a key in the outermost dictionary. The inner dictionary specifies the instruction type, the opcode or aluop, and then the "syntax" field. The syntax field lists the components *as they appear in a command*, **not** as they are stored in the final binary string. For example, look at `lw`, which is written as `lw $rd N($rs)`. Hence, the order of the arguments on the line are `$rd N $rs`, so that's the order of the syntax list. Changing the order will change what the assembler thinks it sees at each step, but it will not change the order of these parts in the resultant binary string.
All the names in the syntax list must also be present int the list specified by the instruction type. For example, if name if the field `shamt` in `sll` was changed to `shift_amount`, it must also be changed in the Instruction Type json, which specifies what fields an `R-type` instruction can have. See the next section for more details on the this second field.
The distinction between `immed` and `T` in the syntax field is that `immed` is signed, while `T` is not. As discussed above, simple switching out `immed` for `T` to get a signed value will fail, as `JI` types expect the field to be called `T`.

3.2 Instruction Type File

The Instruction Type JSON file tells the assembler what each instruction should look like after processing, as well as what instructions are branches. Below is the base file:

```
{
  "types":
  {
    "R":   {"opcode": 5, "rd": 5, "rs": 5, "rt": 5, "shamt": 5, "aluop": 5, "zeroes": 2},
    "I":   {"opcode": 5, "rd": 5, "rs": 5, "immed": 17},
    "JI":  {"opcode": 5, "T": 27},
    "JII": {"opcode": 5, "rd": 5, "zeroes": 22},
    "E":   {"err": 32}
  },
```

```
  "branches": ["blt", "bne"]
}
```

The `types` key maps to a dictionary, where the keys specify instruction types and the values are the number of bits that the field uses. The field names must also exist in the `syntax` array from the `instruction` JSON, specified in the section above. The key order also represents the instruction structure. For example, the assignment spec says that `JII` type instructions look like this:

Opcode [31:27]	$rd [26:22]	Zeroes [21:0]

Consequently, the dictionary for `JII` is ordered `opcode`, `rd`, `zeros` to tell the assembler the required order. This ordering is divorced from the order that arguments appear on a line of MIPS code, so the ISA could be updated later with a different component order while preserving the traditional MIPS syntax.
The second key is `branches`, used in label replacement. Since the next PC after a taken branch is

$$PC + N + 1,$$

N can be found as

$$PC_{next} - PC - 1.$$

This differs from jumps, for which the next `PC` is simply the specified immediate, so the assembler needs to know whether to replace the text label with just the instruction number of the target or `target - PC - 1`. If the instruction name is in this list, the assembler will treat it like branch, while all others will be treated like jumps.

Note: the close reader will observe that the fields are stored in a dictionary, which may not preserve order. Since Python 3.6, `dicts` maintain their insert order, and are read in element by element from the JSON file, so this shouldn't be an issue.

3.3 Value Mappings file

This file allows users to add special names and a numeric representation for them. By default, it looks like this:

```
{
  "$rstatus": 30,
  "$ra"     : 31
}
```

Whenever the assembler finds a string token in the MIPS, it will attempt to replace it with the value specified in this file.
Note: This dictionary is collated with the jump/branch labels, which is why a custom value cannot share a name with a jump/branch target. Be careful with your named values.

4 Known Issues

- Cannot use `setx` with a negative immediate. This will likely never be fixed because of how the assembler handlers binary conversions.

A Example MIPS File

A.1 Example 1

```
loop:
  sub $1 $2 $3
  bne $1, $0, loop

bne $2, $1, T
addi $1, $1, 1
sll $1 $1 4
T: sub $1 $2 $3
```

A.2 Example 2

```
lw $8, 1($0)
lw $9, 2($0)
lw $10, 3($0)
lw $11, 4($0)
lw $12, 5($0)
lw $13, 6($0)
lw $14, 7($0)

add $23, $1, $0
add $24, $8, $0
add $25, $15, $0
jal check
add $22, $22, $26
add $15, $25, $0

add $23, $2, $0
add $24, $9, $0
add $25, $16, $0
jal check
add $22, $22, $26
add $16, $25, $0

add $23, $3, $0
add $24, $10, $0
add $25, $17, $0
jal check
add $22, $22, $26
add $17, $25, $0

add $23, $4, $0
add $24, $11, $0
add $25, $18, $0
jal check
add $22, $22, $26
add $18, $25, $0

add $23, $5, $0
add $24, $12, $0
add $25, $19, $0
jal check
add $22, $22, $26
add $19, $25, $0

add $23, $6, $0
add $24, $13, $0
add $25, $20, $0
jal check
add $22, $22, $26
add $20, $25, $0

add $23, $7, $0
add $24, $14, $0
add $25, $21, $0
jal check
add $22, $22, $26
add $21, $25, $0

j 0

check:
#is the key supposed to be down?
bne $0, $24, sk1
addi $25, $0, 0
addi $26, $0, 0
jr $31
#is the key down?
sk1: bne $23, $0, sk2
addi $26, $0, 0
jr $31
#has the score been incremented already?
sk2: bne $25, $23, sk3
addi $26, $0, 0
jr $31
sk3: addi $25, $0, 1
addi $26, $0, 1
jr $31
```