Generic Assembler
HelloWorld Tutorial

In this tutorial, I will create a “Hello World” source program written in the MIPS assembly language. Then I will create a specification file describing the MIPS architecture and assembly language. I will then assemble it using the Generic Assembler to generate the object code for the program.

Lets begin by creating the assembly source file. I have created file “HelloWorld.txt”. Using a text editor I have inputted the “skeleton” for the source file:

```
.data

; data here

.text

; assembly code here
```

We need to initialise the “Hello World” string within the .data section. I will declare this as “helloStr”:

```
.data

helloStr .ascii "Hello World"

.text

; assembly code here
```

The assembly instructions needed for the “Hello World” program will now be placed within the .text section:

```
.data

helloStr .ascii "Hello World"

.text
```

```
ADDIU $v0, $1, 4
LUI $at, 0000
ORI $a0, $at, helloStr
SYSCALL
```

The assembly source file (HelloWorld.txt) is now complete. We need to create a specification file which describes the MIPS architecture and its assembly language so that these instructions can be successfully assembled.

I have created a specification file (specMIPS.txt). And have inputted the “skeleton” for the specification file:

```
Architecture:

; architecture data here

Registers:

; registers data here

InstructionFormat:

; instruction format data here

AssemblyOpTree:

; assemblyoptree data here

MnemonicData:

; mnemonic data here

Endian:

; endian data here

MinAddressableUnit:

; minimum addressable unit data here
```

Within the Architecture section, I will name the architecture we are describing:

```
Architecture:

MIPS
```

Now we need to specify the registers and their respective encodings which are needed for this assembly program:

```
Registers:

$1   0H   ; register $1, hex value 0 (0 in binary)
$at  1H   ; register $at, hex value 1 (1 in binary)
$v0  2H   ; register $v0, hex value 2 (10 in binary)
$a0  4H   ; register $a0, hex value 4 (100 in binary)
```

Now I will specify the instruction formats which are needed to encode the assembly instructions. There are only two needed, as 3 of the 4 assembly instructions use the same instruction format:

```
InstructionFormat:

I-type: opcode(6) rs(5) rt(5) immediate(16)
syscall: call(32)
```

The first instruction specified has been named “I-type”, and it has 4 fields. The first field “opcode” is of bit length 6; the second field “rs” is of bit length 5 etc.

Now we will create the “AssemblyOpTree” which describes the arrangement of operands and names the various types of operands.

As you can see from the source instructions, each assembly instruction starts with a mnemonic and has zero or more operands. An operand can be a register, an immediate variable or a label (which refers to the initialised “Hello World” string in the .data section). An immediate value is a hexadecimal variable. A register can be $1, $at, $a0 or $v0. The mnemonic can be ADDIU, LUI, ORI or SYSCALL.

```
AssemblyOpTree:

statement : mnem op*

op : reg
op : imm
op : LABEL

imm : HEX

reg : "$1"
reg : "$at"
reg : "$a0"
reg : "$v0"

mnem : "ADDIU"
mnem : "LUI"
mnem : "ORI"
mnem : "SYSCALL"
```

Now that the AssemblyOpTree is complete we need to complete the MnemonicData section. In this section we need to specify the relevant field encoding and instruction formats which are needed to encode each instruction. Lets start with the ADDIU mnemonic. First lets simply declare it in the MnemonicData section:

```
MnemonicData:

ADDIU
     ; TODO
```

ADDIU is encoded using the I-type instruction that we have already defined in the InstructionFormat section. Here we need to define what the encoding values for all the fields within the I-type instruction are. These fields were named opcode, rs, rt and immediate.

All ADDIU instructions have an opcode of 001001 (in binary). So lets specify this as a global opcode:

```
ADDIU
     opcode=001001B          ; tab

     ; TODO
```

Next we need to specify the expected operand format/arrangement for this mnemonic. If we look at the source instruction from within the assembly file (ADDIU $v0, $1, 4), we can see that it takes the form **mnem reg reg imm** (these tokens we specified within the AssemblyOpTree). The instruction should also have a comma after the first and second registers (i.e., **mnem reg, reg, imm)**. So lets specify this as an acceptable operand format of the ADDIU mnemonic:

```
ADDIU
     opcode = 001001B

     mnem reg, reg, imm
          ; TODO
```

Next, we must specify which fields (within the I-type instruction) are used for encoding the values for each register and the immediate. The encoding value for the first reg populates the rt field; the second reg the rs field and the imm populates the immediate field, so lets specify this:

```
ADDIU
     opcode = 001001B

     mnem reg, reg, imm
          mnem rt rs immediate     ; mnem is a placeholder
          ; TODO
```

We now need to specify if ADDI format `mnem reg, reg, imm` has any local field encodings which are unique to this operand format. It does not, and all fields within the I-type instruction have already been defined. So we just put "`--`" to indicate to the assembler that no local opcodes exist for this format:

```
ADDIU
     opcode = 001001B

     mnem reg, reg, imm
          mnem rt rs immediate
          --
```

Lastly we need to specify that ADDI format `mnem reg, reg, imm` is encoded using instruction I-type:

```
ADDIU
     opcode = 001001B

     mnem reg, reg, imm
          mnem rt rs immediate
          --
          I-type
```

The ADDIU instruction is now complete! Now lets complete the other 3 remaining instructions from the source assembly code.

Next is the LUI instruction (LUI $at, 0000). So lets declare it underneath the one we have just completed for ADDIU:

```
ADDIU
     opcode = 001001B

     mnem reg, reg, imm
          mnem rt rs immediate
          --
          I-type
                                        ; empty line
LUI
     ; TODO
```

All LUI instructions have an opcode of 001111 (in binary). It has an operand format/arrangement of mnem reg, imm. It uses the I-type instruction for encoding. The reg field populates the rt field within the I-type instruction and the imm populates the immediate field. It has local encoding rs which is encoded 00000 (in binary):

```
LUI
     opcode =001111B

     mnem reg, imm
          mnem rt immediate
          rs=00000B
          I-type
```

Next is the ORI instruction (ORI $a0, $at, helloStr). All ORI instructions have an opcode of 001101 (in binary). It has an operand format/arrangement of mnem reg, reg, LABEL. **Because the last operand refers to a variable name we defined in the .data section, and that this was defined in the AssemblyOpTree using the keyword LABEL, we specify this operand as LABEL**. It uses the I-type instruction for encoding. The first reg field populates the rt field within the I-type instruction; the second reg populates the rs field and the LABEL (which returns the offset of the address) populates the immediate field. It has no local encodings:

```
ORI
     opcode = 001101B

     mnem reg, reg, LABEL
          mnem rt rs immediate
          --
          I-type
```

We only have one more instruction to define (SYSCALL)! All SYSCALL instructions have an opcode of 1100 (in binary). It has an operand format/arrangement of just mnem. It uses the syscall instruction (which only has one field call(32)) for encoding. It has no operands and so no fields within the instructions are encoded from any of the instructions operands:

```
SYSCALL
     call = 1100B

     mnem
          --
          --
          syscall
```

Although field call is 32 bits long, and we have only defined it as 1100, the remaining 28 bits will be padded with 0's, so this is not important.

We have completed MnemonicData section! Now we need to specify the endian of the architecture. MIPS uses big endian.

```
Endian:

big
```

All there is left to do is the specify the minimum addressable unit, we will assume this is a byte (8 bits):

```
MinAddressableUnit:

8
```

All we need to do now is to execute the program with the 2 .txt files we have just created.

Put both “specMIPS.txt” and “HelloWorld.txt” in the software’s root directory (i.e., **Generic-Assembler/**). Then execute the program using the following command:

java -cp bin/ Main specMIPS.txt HelloWorld.txt

In this same directory you should now see “object_code.txt” and “spec_error_report.txt”. spec_error_report.txt should be empty (assuming no mistakes made), and the object code should be visible within object_code.txt. If successful you should see:

```
0:          48 65 6C 6C 6F 20 57 6F 72 6C 64
b:          24 02 00 04
f:          3C 01 00 00
13:         34 24 00 00
17:         00 00 00 0C
```