The Assembler

Goal: Automate the process of translating ASM to ML.
Input: Assembly source code
Output: Machine code

Translation has 2 phases:

1. Analysis: Understand the meaning of source string
2. Synthesis: Output the equivalent target string
Assembly Translation

Read the input one ASCII char at a time; i.e. as a stream of char.
The first step is to group characters into meaningful tokens:
  • labels, register #, hex #, .word, etc
  • Note: This is done for you in asm.rkt and asm.cc
Assembly Translation

Read the input one ASCII char at a time; i.e. as a stream of char. The first step is to group characters into meaningful \textit{tokens}:

- labels, register \#, hex \#, \texttt{.word}, etc
- Note: This is done for you in \texttt{asm.rkt} and \texttt{asm.cc}

Your job:

1. Analysis: Check sequence of tokens is a valid program
2. Synthesis: Output equivalent machine code

Focus on checking if the sequence of tokens is valid; anything else, output an error message containing the word \texttt{ERROR} to \texttt{stderr}.
Assembler Challenges

Most of the process is straightforward since 1 assembly instruction translates to exactly 1 machine language instruction.

Challenge (the extra things your Assembler does):

- Comments and whitespace are simply discarded.
- Labels are used to compute memory addresses for jumps and branch offsets.

Remember labels, comments, whitespace are there to help programmers. MIPS machine code is simply a sequence of 32-bit binary instructions (no comments, whitespace, labels).
Assembler Challenges - Labels

We want to read 1 assembly instruction and directly output its encoded machine instruction.

How to assemble:

beq $0, $1, label
...
label: add $22, $10, $31

Problem:
Assembler Challenges - Labels

We want to read 1 assembly instruction and directly output its encoded machine instruction.

How to assemble:

beq $0, $1, label

...  

label: add $22, $10, $31

Problem: To encode beq we need the memory address of label, but we haven’t encountered this label yet! Fix?
2-Pass Assembler

Pass 1:

- Group tokens into instructions, verifying instructions are valid.
- Keep track of the memory address (starting at 0x0) each instruction will be given when loaded into memory.
- Build a **symbol table** for (label, address) pairs (use map).
- **Note**: multiple labels may have the same address.
2-Pass Assembler

Pass 1:
- Group tokens into instructions, verifying instructions are valid.
- Keep track of the memory address (starting at \(0x0\)) each instruction will be given when loaded into memory.
- Build a **symbol table** for (label, address) pairs (use `map`).
- **Note**: multiple labels may have the same address.

Pass 2:
- Translate each instructions into machine code.
- If a label is encountered, look up associated address - compute branch offset if necessary.

Output translated, assembled MIPS to `stdout`. 
Symbol Table Example

0x00  main:  lis $2
0x04  .word 20
0x08  lis $1
0x0c  .word 2
0x10  add $3, $0, $0

  top:
0x14  add $3, $3, $2
0x18  sub $2, $2, $1
0x1c  bne $2, $0, top
0x20  jr $31
0x24  beyond:

Recall, offset in bne: (top − PC)/4 = (0x14 − 0x20)/4 = -3
Encoding Instructions into Binary

Translate each assembly instruction into its binary encoding.

Avengers: lis $2
          .word Avengers

Assemble!
Encoding Instructions into Binary

Translate each assembly instruction into its binary encoding.

Avengers: lis $2
    .word Avengers

Assemble!
lis $2 ⇒ 0x00001014
.word 0x0 ⇒ 0x00000000
bne $2, $0, top
Encoding Instructions into Binary

Translate each assembly instruction into its binary encoding.

Avengers: lis $2
          .word Avengers

Assemble!

lis $2 ⇒ 0x00001014
.word 0x0 ⇒ 0x00000000

bne $2, $0, top ⇒ 0x1440fffd
  • bne has opcode 000101
  • 2 ⇒ 00010
  • 0 ⇒ 00000
  • top = -3 ⇒ 1111111111111101 = 0xffffd
Obtain pieces from the sequence of tokens, then assemble!

Assembly: `bne $2, $0, -3`

Binary:

```
0001 01 00 0100 00000 1111 1111 1111 1101
```

Can we simply print out each piece, token by token?

- `printf("000101"); printf("00010");...`
- `printf("0x"); printf("1"); printf("4");...`
Assemblying the Pieces

Obtain pieces from the sequence of tokens, then assemble!

Assembly: `bne $2, $0, -3`

Binary:

```
0001  01  00  010  0  0000
1111  1111  1111  1101
```

6 bits opcode 5 bits reg s 5 bits reg t 16 bits offset

Can we simply print out each piece, token by token?

- `printf("000101"); printf("00010");...`
- `printf("0x"); printf("1"); printf("4");...`

NO!
We need to build and store the encoded instruction using 32 bits, then output the result.

What type in C++ can we use that has 32 bits?
Assemblying the Pieces

We need to build and store the encoded instruction using 32 bits, then output the result.

What type in C++ can we use that has 32 bits? int

How do we put the first piece into place?

The first 6 bits should be 000101 = 5.
We need to build and store the encoded instruction using 32 bits, then output the result.

What type in C++ can we use that has 32 bits? **int**

How do we put the first piece into place?

The first 6 bits should be **000101 = 5**.

**Bitwise operators!**

How far do we need to **shift**?

(int) 5 is **0000 0000 0000 0000 0000 0000 0000 0101**

We want: **0001 0100 0000 0000 0000 0000 0000 0000**
To shift into place, need to append 26 zeros ⇒ left-shift by 26 bits:

- C++: \(5 << 26\)
- Racket: `(arithmetic-shift 5 -26)`

Move $2, 21 bits left:

- C++: \(2 << 21\)
- Racket: `(arithmetic-shift 2 -21)`

Move $0, 16 bits left:

- C++: \(0 << 16\)
- Racket: `(arithmetic-shift 0 -16)`

Result so far is: \(0x14400000\)
Negative offsets are tricky.

We currently have: 0x14400000 from the first 3 pieces
and ultimately want: 0x1440fff

How do put the last piece into place?

(int) -3 is 1111 1111 1111 1111 1111 1111 1111 1101
Or, in 32-bit hexadecimal: 0xfffffffff
Negative offsets are tricky.

We currently have: 0x14400000 from the first 3 pieces
and ultimately want: 0x1440ffffd
How do put the last piece into place?

(int) -3 is 1111 1111 1111 1111 1111 1111 1111 1101
Or, in 32-bit hexadecimal: 0xffffffffd

Only want last 16 bits ⇒ bitwise AND with 0x0000ffff:
  • 0xffffffffd AND 0x0000ffff ⇒ 0x0000fffd
  • C++: -3 & 0xffff
  • Racket: (bitwise-and -3 #xffff)
Final Assembly and Output

As a single statement, bitwise OR all the pieces:

```cpp
int instr = (5 << 26) | (2 << 21) | (0 << 16) |
            (-3 & 0xffff);
```

\[
\text{(bitwise-or (arithmetic-shift 5 -26) ...}
\text{(bitwise-and -3 \#xffff))}
\]

Final value of `instr` is 339804157 (in decimal).

Output: `cout << instr`?
Final Assembly and Output

As a single statement, bitwise OR all the pieces:

```c
int instr = (5 << 26) | (2 << 21) | (0 << 16) |
            (-3 & 0xffff);
```

```
(bitwise-or (arithmetic-shift 5 -26) ... 
            (bitwise-and -3 \#xffff))
```

Final value of `instr` is 339804157 (in decimal).

Output: `cout << instr`?

**No!** This prints 339804157 - 9 ASCII characters.

We need to output 4 bytes!
What gets Output?

What does the following print?

```cpp
char c = 97;
int x = 97;
cout << x << c;
```
What gets Output?

What does the following print?

```cpp
char c = 97;
int x = 97;
cout << x << c;
```

⇒ 97a

**Note:** x printed 2 ASCII characters and c printed 1.

Based on the type, C++ displays the format you expect to see. Although we see ‘a’ on the screen, we know the 1-byte ASCII value was output.
Output Byte by Byte

int instr = 339804157; is the 4 bytes:

00010100 01000000 11111111 11111101
1st byte 2nd byte 3rd byte 4th byte

We want to print the ASCII char for each byte. When printed, it may look strange, i.e. **the correct output may look like garbage!**

- ASCII code 20 ⇒ [Device Control 4]
- ASCII code 64 ⇒ @
- ASCII code 255 ⇒ ???
- ASCII code 253 ⇒ ???

Some characters may also not visibly print anything (ASCII 7)!
Output Byte by Byte in C++

Output the int byte by byte using a char.

```cpp
int instr = 339804157;
char c = instr >> 24;
cout << c;
c = instr >> 16;
cout << c;
c = instr >> 8;
cout << c;
c = instr;
cout << c;
```