CS241 Lecture 5

Graham Cooper

May 20th, 2015

Assemblers

Assembly Code:
add \$1, \$2, \$3
jr \$31

which goes into an assembler that outputs machine code.

Any translation process involves 2 phases:

1. Analysis - Understanding what is meant by the source string (input)
2. Synthesis - output the equivalent target-string

assembly file → stream of characters
- first step, generall,y is to group characters into meaningful tokens, eg. label, hex number, register number, .word directive, etc...
- This is done for you

My Job: Group tokens into instructions (if possible). (Analysis, trying to figure out what instructions are meant then outputing machine code)

If tokens are not arranged into sensible instructions, output ERROR to stderr. Add more descriptive error messages if needed.

Advice: There are many more wrong configurations that right ones.
- Focus on finding all of the right ones - anything else is ERROR

Problems

Biggest problem with writing assemblers:

How do we assemble:

```
beq $0, $1, abc
...
abc: add $3, $3, $3
```

The assembler does not know what ABC is yet, so it doesn't know the value of that label. We cant assemble the beq because we do not know what abc is yet.

<u>**Standard solution:**</u>

Assemble in two passes. Pass1:

-Group the tokens into instructions

-Record the addresses of all labelled instructions (symbol table - (listof(label,address)))

Notes:

1) A line of assembly may have more than one label.

eg:

f:

g:

mult $1, $2

2) The line after the last line can be labelled

eg.

jr $31

z:

Pass2:

- Translate each instruction into machine code - if an instruction refer s to a label, look up the associated address in the symbol table.

Your assembler:

- output assemble MIPS to stdout
- output the symbol table to stderr

Example:

```
main: lis $2
.word 13
add $3, $0, $0
top:
add $3, $3, $2
lis $1
.word 1
sub $2, $2, $1
```

```
        bne $2, $0, top
        jr $31
        beyond:
```

Pass1:
- group tokens into instructions
- build symbol table

name	address
main	0x00
top	0x0c
beyond	0x24

Pass2:
Translate each instruction
eg. list 2 → 0x00001014
.word 13 → 0x0000000d
...
bne $2, $0, top
- lookup top in symbol table which is 0x0c calculate $\frac{top-PC}{4} = -5$
-get 0x1440fffb (-5 = 0xfffb)

Note: To negate a 2's compliment number, flip the bits and add 1
5 = 0000 000 000 0101
-5 = 1111 1111 1111 1010 + 1
= 1111 1111 1111 1011
= fffb

Bit level operations

To assemble bne $2, $0, top (where $\frac{top-PC}{4} = -5$)
opcode = 000101 = 5
first register $2 = 2
second register $0 = 0
offset = -5

6bits	5 bits	5 bits	16 bits

To put 000101 in the first 6 bits we need to append 26 0's to it, ie left shift by 26 bits.
C: 5 <<26 (Racket: (arithmetic shift 5 -26)) $\implies$ $000101\underbrace{00...00}_{26}$

Move $2, 21 bits to the left: 2 <<21

Move $0, 16 bits to the left: 0 <<16

-5 = 0xfffffffb, we only want the last 16 bits in order to store our instruction properly.

a	b	a AND b	a OR b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

extend these to bit-by-bit operations on full ints.

$$\begin{array}{r}11001010\\ 10010110\\ \hline 10000010\end{array}$$

And with 1, get the other bit unchanged, AND with - get 0
We can use AND to turn bits off, then we can use OR to turn bits on.

So do a bitwise AND with 0x(0000)ffff c: -5 & 0xffff
Racket: (bitwise-and -5 #xffff)

Then bitwise-or the pieces together
int x = (5 << 26)|(2 << 21)|(0 << 16)|$(-5\&0xffff)$;
= 339 804 155
cout << x << endl; BAD DO NOT DOOOOO

int x = 65;
char c = 65;

cout << x << c;
Output: 65A

int x = instruction;
char c = x >> 24;
cout ¡¡ c;
c = x >> 16;
cout ¡¡ c;
c = x >> 8;
cout ¡¡ c;
c = x;

cout ¡¡ c;