Topic 4 – Implementing an Assembler

Key Ideas

- pseudocode for pass 1
- calculating addresses of instructions
- dealing with labels
- pseudocode for pass 2
- bitwise operations
  - bitwise and
  - bitwise or
  - shift left

Implementing an Assembler

General Strategy

- *test every detail on the reference sheets*
  - e.g. test that ints are in the proper range
- must know the language better than a programmer
- error reporting can be unsophisticated
  - just report ERROR, meaningful details are optional
- don't try to think about all possible errors just *be very specific about what you are expecting,* i.e.
  - the opcode **`add`** is followed by exactly 3 registers,
  - the opcode **`mult`** is followed by exactly 2 registers,
  - the opcode **`jr`** is followed by exactly 1 register.

Implementing an Assembler

Recall: Format of Input

- each line of assembly language is of the format

  label(s) *instruction* *comments*

```
main:    lis $1        ; $1 = 1
         .word 0x1
```

  - each of these three components are optional
  - a line may have 0 , 1, 2 or all 3 of them
- Lines without an *instruction* are called *null lines* and do not specify an instruction word.

Implementing an Assembler

Calculating the Locations for Instructions

- ignore all (labels, comments, blank lines) but the instructions to track the address of each instruction
- each instruction is 4 bytes long

Location	Input
0x00	`; my prog`
0x00	`start:`
0x00	`add $1, $2, $3`
0x04	`middle: centre: ; important`
0x04	`lw $2, 0($1)`
0x08	`add $2, $2, $4`
0x0C	`jr $31`

Implementing Pass 1

Pseudocode for Pass 1: Analysis

```
PC= 0                                       // program counter
for each line of input {
    scan line

    for each LABEL {                        // process labels
        if already in symbol table
            report ERROR and exit
        add (label, PC) pair to symbol table

        if next token is an OPCODE {        // process instructions
            if remaining tokens are not what is expected
                report ERROR and exit
            create intermediate representation of instruction
            PC += 4
        }
    }
}
```

Implementing Pass 1

Pseudocode for Pass 1: Analysis

PC= 0 *// program counter*
for each line of input
    scan line *// ⇐ we'll help you here*

- You should use asm.rkt or the C++ code in asm.zip in order to help you identify tokens
- You may use C++, Racket or Scala
- Later on, in A5 and A6, you will learn how to identify tokens yourself.
- Typically you use another program (such as lex or flex) to create this file.

Implementing a Symbol Table

Input

```
a:      lis $1
        .word 0x1
        beq $0,$0,b
a:      add $1,$0,$0
        bne $2,$0,b
        ...
        beq $2,$0,a
        ...
b:      sub $2,$2,$1
```

Resolving Labels

- Which location does the label **a** refer to?
- Labels can
  - only be *defined once*
  - be *used many times* as a operand
- Your assembler needs the ability to *add* and *lookup* (string, number) pairs

Implementing a Symbol Table

In C++

- *could use a map*

```
using namespace std;
#include <map>
#include <string>

map<string, int> st;

st["foo"] = 42;
```

Implementing a Symbol Table

In C++

- an *incorrect* way of accessing elements:

```
x = st["foo"]; // x gets 42
y = st["bar"]; // y gets 0, and (bar, 0)
               // gets added to st.
```

- a *correct* way of accessing elements:

```
if (st.find("biff") != st.end()) {
    ... not found ...
}
```

Implementing an Assembler

Pseudocode for Pass 2: Synthesis

for each OPCODE in the intermediate representation
    translate to MIPS machine code
        look up any labels in the symbol table
    output the instruction (as 4 bytes)

Caution

For each instruction, the output is

- 32 bits (i.e. 4 bytes)
- *not 32 ASCII characters* (i.e. 32 bytes)

Implementing an Assembler

Translating Instructions

- *Use the MIPS reference sheet as your guide*
- e.g. for the command `lis $2` the format is

  0000 0000 0000 0000 dddd d000 0001 0100

  where ddddd is 00010 (binary for 2)
- this step is very similar to Assignment 1
- *but now you've got to encode this in four bytes* which involves dealing with, and shifting around, bits
- we'll look at `bne $2,$0,top` in detail …

Sample Input

PC Labels Instructions

```
00  main:    lis $2
04           .word 0xd
08           add $3,$0,$0
0C  top:     add $3,$3,$2
10           lis $1
14           .word 1
18           sub $2,$2,$1
1C           bne $2,$0,top
20           jr $31
24  beyond:
```

Symbol Table

Label	Address
main	0x00
top	0x0C
beyond	0x24

Implementing an Assembler

Building up a Instruction

- for **`bne $2,$0,top`**
- first look up **`top`** in the symbol table
- it corresponds to address 0x0C
- but *we need a number of instructions to jump* back or forward not an address
- (top – PC) / 4 = (0x0C – 0x20) / 4 = (12-32)/ 4 = -5
- recall that the PC gets incremented by 4 (i.e. 0x1c + 4 = 0x20)
- so now the instruction becomes **`bne $2,$0,-5`**
- the format the **`bne`** instructions is

  `0001 01ss ssst tttt iiii iiii iiii iiii`

  so we must build up each component of thus instruction…

Implementing an Assembler

Bitwise Operations

- typically the smallest unit of data that can be assigned directly is a single byte (i.e. a char)
- to manipulate anything smaller, we must use *bitwise operations* (operations that act on a single bit of a char or int)
- *bitwise and*, a & b, performs the *and* operation on *individual bits*, e.g. for 8 bit values, it would be …

a =	0	1	0	0	1	0	1	1
b =	1	1	0	0	0	1	0	1
a & b =	0	1	0	0	0	0	0	1

a	*b*	*a* & *b*
0	0	0
0	1	0
1	0	0
1	1	1

Implementing an Assembler

Bitwise Operations

- This operation is used to *mask off* bits (i.e. change a portion of the bits to all 0's), e.g. for an 8-bit value

a =	1 1 0 1 0 1 0 1
bit_mask (0x0F)	0 0 0 0 1 1 1 1
a & bit_mask =	0 0 0 0 0 1 0 1

- Here the most significant nibble (half byte) of *a* has been masked off (set to 0).
- If *a* is a 32-bit number, 0xffff would mask off the most significant 2 bytes, e.g.

a	= 1101 0011 1010 1000 1101 1010 1101 1111
0xffff	= 0000 0000 0000 0000 1111 1111 1111 1111
a & 0xffff	= 0000 0000 0000 0000 1101 1010 1101 1111

Implementing an Assembler

Bitwise Operations

- *bitwise or*, a | b, performs the *or* operation on *individual bits*, e.g. for 8 bit values it would be

a =	0	1	0	0	1	0	1	1
b =	1	1	0	0	0	1	0	1
a | *b* =	1	1	0	0	1	1	1	1

a	*b*	*a* | *b*
0	0	0
0	1	1
1	0	1
1	1	1

- the *shift left operator*, <<, shifts bits left, introducing 0's on the right hand side, e.g. for 8 bit values it would be …

a =	0	1	1	0	1	0	0	1
a << 1 =	1	1	0	1	0	0	1	0
a << 2 =	1	0	1	0	0	1	0	0
a << 3 =	0	1	0	0	1	0	0	0
a << 4 =	1	0	0	1	0	0	0	0
a << 5 =	0	0	1	0	0	0	0	0
a << 6 =	0	1	0	0	0	0	0	0
a << 7 =	1	0	0	0	0	0	0	0

Implementing an Assembler

Translating Instructions

- recall that the format of the `bne $2,$0,-5` instructions is

```
0001 01ss ssst tttt iiii iiii iiii iiii
↑      ↑     ↑     ↑                  ↑
32     26    21    16                 1
```

where the opcode is 5 = 000101_2 shifted 26 bits left

5	0000 0000 0000 0000 0000 0000 0000 0101
5 << 26	0001 0100 0000 0000 0000 0000 0000 0000

s is 2 = 00101_2 shifted 21 bits left

2	0000 0000 0000 0000 0000 0000 0000 0010
2 << 21	0000 0000 0100 0000 0000 0000 0000 0000

t is 0 = 00000_2 shifted 16 bits left

0	0000 0000 0000 0000 0000 0000 0000 0000
0 << 16	0000 0000 0000 0000 0000 0000 0000 0000

Implementing an Assembler

Translating Instructions

i is -5 in 16-bit two's complement notation

-5	1111 1111 1111 1111 1111 1111 1111 1011
0xffff	0000 0000 0000 0000 1111 1111 1111 1111
-5 & 0xffff	0000 0000 0000 0000 1111 1111 1111 1011

or'ing these all together we have

instr = (5 << 26) | (2 << 21) | (0 << 16) | (-5 & 0xffff)

(5 << 26) |	0001 0100 0000 0000 0000 0000 0000 0000
(2 << 21) |	0000 0000 0100 0000 0000 0000 0000 0000
(0 << 16) |	0000 0000 0000 0000 0000 0000 0000 0000
(-5 & 0xffff)	0000 0000 0000 0000 1111 1111 1111 1011
= instr	0001 0100 0100 0000 1111 1111 1111 1011

Implementing an Assembler

Translating Instructions

- In C++ the instruction `bne $2,$0,-5` becomes
  int instr;
  instr = (5 << 26) | (2 << 21) | (0 << 16) | (-5 & 0xffff);
- However if you try `cout << instr;` you will get it represented as an integer, e.g. 339804155 which is not what we want so *we must write out each byte as a char,* i.e. …

Implementing an Assembler

Translating Instructions

- write out each byte as a char
- *do not add newlines anywhere*

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

Implementing an Assembler

Translating Instructions

- The C function **`putchar`** takes an int as an argument, converts it into a char internally, and outputs that char to stdout

```
#include <cstdio>

void output_instr(int instr) {
   putchar(instr >> 24);
   putchar(instr >> 16);
   putchar(instr >> 8);
   putchar(instr);
}
```

- note we write out the most significant byte of the word first (called *big endian* format)
- other processors use *little endian* format, in which case we would write out the least significant byte of the word first.

Implementing an Assembler

Hint for Translating Instructions

- CS 241’s subset of MIPs assembly language, instructions only come in *a few different formats*
  1. add, sub, slt, sltu
  2. mult, div, multu, divu
  3. mfhi, mflo, lis
  4. lw, sw
  5. beq, bne
  6. jr, jalr
  7. .word

  Hint: you might consider a function for each format rather than one function for each instruction.