CMSC 216
Introduction to Computer Systems
Lecture 16
Assembly Langage, cont.

Larry Herman & Alan Sussman

Notes

- Project 2 grades visible, secret tests posted, reports sent out
- Project 3 secret tests posted, results visible on submit server
- Project 4 due Monday
  - public tests posted and submit server open

Chapters 3 and 4.1, Bryant and O'Hallaron

ASSEMBLY LANGUAGE

Branch example 1

- Assembler output:

```
0x000: f208           |           rdint  %eax
0x002: 308700000000   |           irmovl $0,%edi     # consistent zero
0x008: 308600000000   |           irmovl $0,%esi     # sum = 0
0x00e: 6070           |           addl   %edi,%eax
0x010: 7320000000     |           je     EndLoop
                      |
0x015: 6006           | Loop:     addl   %eax,%esi   # sum += n
0x017: f208           |           rdint  %eax
0x019: 6070           |           addl   %edi,%eax
0x01b: 7415000000     |           jne    Loop
                      |
0x020: f368           | EndLoop:  wrint  %esi
0x022: 30830a000000   |           irmovl $10,%ebx
0x028: f138           |           wrch   %ebx
0x02a: 10             |           halt
```

- Simulator output:

```
$ echo 1 4 9 16 25 0 | yis io.yo
55
Stopped in 29 steps at PC = 0x2b.  Exception 'HLT', CC Z=1 S=0 O=0
...
```

Branch example 2

- Assembler output:

```
0x000: 308003000000 |         irmovl $3,%eax     # a = 3
0x006: 308304000000 |         irmovl $4,%ebx     # b = 4
0x00c: 2006         |         rrmovl %eax,%esi   # s = a
0x00e: 6136         |         subl   %ebx,%esi   # s = s - b
0x010: 751c000000   |         jge    Else        # if s >= 0 jump
                    |
0x015: f308         |         wrint  %eax        # printf("%d", a)
0x017: 701e000000   |         jmp    Endif       # jump
                    |
0x01c: f338         | Else:   wrint  %ebx        # printf("%d", b)
                    |
0x01e: 30860a000000 | Endif:  irmovl $10,%esi
0x024: f168         |         wrch   %esi        # printf("\n")
0x026: 10           |         halt
```

- Simulator output:

```
3
Stopped in 10 steps at PC = 0x27.  Exception 'HLT', CC Z=0 S=1 O=0
...
```


Other Y86 instructions

Instruction	Effect	Description
`halt`	Ends program	Program halt
`pushl R`	Reg[`%esp`] ← Reg[`%esp`] - 4; Mem[Reg[`%esp`]] ← Reg[R]	Push on to stack
`popl R`	Reg[R] ← Mem[Reg[`%esp`]] ; Reg[`%esp`] ← Reg[`%esp`] + 4	Pop off of stack

- Without a **`halt`** instruction, the simulator will attempt to read in possibly invalid instructions
- **`pushl`** and **`popl`** provide quick ways to work with the program stack
  - what would you have to do if you didn't have access to these?
  - why shouldn't you use **`%esp`** as a general purpose register?

Assembler directives

Directive	Effect
`.pos number`	Subsequent lines of code start at address **number**
`.align number`	Align the next line to a **number**-byte boundary
`.long number`	Put **number** at the current address in memory

- These can be used to set up memory in various places in the address space
- **`.pos`** can put sections of code in different places in memory
- **`.align`** should be used before setting up a static variable
- **`.long`** can be used to initialize a static variable

Translating C to Y86

- This process is not always straightforward
- Even simple statements like "**`a = b * c;`**" can require several instructions:

```
irmovl B,%eax
mrmovl 0(%eax),%ebx
irmovl C,%eax
mrmovl 0(%eax),%ecx
multl  %ebx,%ecx
irmovl A,%eax
rmmovl %ecx,0(%eax)
```

Register spilling

- We only have a limited number of registers
- What happens if we need to keep more than 8 values around at once?
- If we run out of registers to store our data, we need to use memory to store values
- We can use the stack or define static arrays (as we'll see later)

Translating branches

- Consider the following C code:

```
if (i == j)
  printf("=");
else
  printf("X");
printf("\n");
```

- How would we begin to translate this into assembly?

Translating branches, cont.

- We can model this conditional structure using branches and labels

```
  if (i == j)
    goto Equal;
  printf("X");
  goto EndIf;
Equal:
  printf("=");
EndIf:
  printf("\n");
```


Aside: why we use `goto` here

- As you can see, the use of `goto` can make your code very difficult to read
- Imagine if you had `goto` statements strewn throughout a 500 line program; would you be able to debug it?
- We show it here only to show how control flow must be represented in assembly; you are not allowed to use it in your C code
- Ever.

Translating branches, cont.

- We can then take the labeled C code and translate it in a fairly straightforward fashion:

```
        subl %eax,%ebx   # i:%eax,j:%ebx
        je Equal
        irmovl $88,%ecx # else block
        wrch %ecx
        jmp EndIf
Equal:  irmovl $61,%ecx # true block
        wrch %ecx
EndIf:  irmovl $10,%ecx # after if/else
        wrch %ecx
```


Translating loops

- What about C code like this?

```
do {
   ...
} while (condition);
```

- This can be translated simply:

```
Loop: ... # begin loop body
          # evaluate condition
          je Loop # jump back if true
```

- Note that you can use whatever jump instruction/condition is appropriate

Translating while loops

- while loops are a bit different; but these can be written as do-while loops, with a little modification

```
while (condition)
  do_something();

if (condition) {
  do {
    do_something();
  } while(condition);
}
```


Translating while loops, cont.

- We know how to handle do-while and if statements!

```
if (! condition)
  goto EndWhile;
do {
  do_something();
} while (condition);
EndWhile:
```

Translating while loops, cont.

- Assuming **%eax** always holds the value of our condition (and we keep 0 stored in **%edi**):

```
        irmovl $0,%edi
        addl %edi,%eax
        je EndWhile
Loop:   # code for do_something();
        addl %edi,%eax
        jne Loop
EndWhile:
```


Translating for loops

- For loops are very similar to while loops; we can use this fact to convert a for loop into a form we know how to work with
- How do we convert this into a while loop?

```
for (init; cond; incr)
  body;

init;
while (cond) {
  body;
  incr;
}
```


Factorial example

- Consider the following program:

```
#include <stdio.h>

int main() {
  int i, f, n;
  scanf("%d", &n);
  f = 1;
  i = 1;
  while (i <= n)
    f *= i++;
  printf("%d\n", f);
  return 0;
}
```

- How would we convert this to a Y86 program?

Translating our factorial example

- Because we only have three variables, we can get away with doing all our work in registers, rather than storing things in memory
- The **scanf()** and **printf()** calls can be easily replaced by **rdint** and **wrint**/**wrch** instructions
- The main work involved here is just translating the while loop to the do-while format we need to convert to assembly

Translated into goto-code

```
#include <stdio.h>

int main() {
  int i, f, n;
  scanf("%d", &n);
  f = 1;
  i = 1;
  if (i > n)
    goto EndWhile;
 Loop:
  f *= i++;
  if (i <= n)
    goto Loop;
 EndWhile:
  printf("%d\n", f);
  return 0;
}
```


Breaking apart compound statements

```
#include <stdio.h>

int main() {
  int i, f, n;
  scanf("%d", &n);
  f = 1;
  i = 1;
  if (i > n)
    goto EndWhile;
 Loop:
  f *= i++;
  if (i <= n)
    goto Loop;
 EndWhile:
  printf("%d\n", f);
  return 0;
}
```


Breaking apart compound statements

```
#include <stdio.h>

int main() {
  int i, f, n;
  scanf("%d", &n);
  f = 1;
  i = 1;
  if (i > n)
    goto EndWhile;
 Loop:
  f = f * i++;
  if (i <= n)
    goto Loop;
 EndWhile:
  printf("%d\n", f);
  return 0;
}
```


Breaking apart compound statements

```
#include <stdio.h>

int main() {
  int i, f, n;
  scanf("%d", &n);
  f = 1;
  i = 1;
  if (i > n)
    goto EndWhile;
 Loop:
  f = f * i;
  i = i + 1;
  if (i <= n)
    goto Loop;
 EndWhile:
  printf("%d\n", f);
  return 0;
}
```

Beginning "compilation"

- We'll map registers to variables: %eax for i, %ebx for f, and %ecx for n
  - Remember why we can do this here?
- Now, let's translate all the non-branches into the assembly code we know.

Starting compilation

```
#include <stdio.h>

int main() {
  int i, f, n;
  scanf("%d", &n);
  f = 1;
  i = 1;
  if (i > n)
    goto EndWhile;
 Loop:
  f = f * i;
  i = i + 1;
  if (i <= n)
    goto Loop;
 EndWhile:
  printf("%d\n", f);
  return 0;
}
```

```
           rdint   %ecx
           irmovl  $1,%ebx
           irmovl  $1,%eax
           if (%eax > %ecx)
             goto EndWhile;

Loop:      multl   %eax,%ebx
           irmovl  $1,%edi
           addl    %edi,%eax
           if (%eax <= %ecx)
             goto Loop;

EndWhile:  wrint   %ebx
           irmovl  $10,%edi
           wrch    %edi
           halt
```


Writing branches

- To change the conditional jumps, we'll have to remember a few things:
  - we'll need to operate on copies of our data; this means we'll need to use an extra register
  - we'll need to perform an arithmetic operation before the jump
  - we need to select the correct jump instruction to operate correctly

Starting compilation

```
           rdint   %ecx
           irmovl  $1,%ebx
           irmovl  $1,%eax
           if (%eax > %ecx)
             goto EndWhile;

Loop:      multl   %eax,%ebx
           irmovl  $1,%edi
           addl    %edi,%eax
           if (%eax <= %ecx)
             goto Loop;

EndWhile:  wrint   %ebx
           irmovl  $10,%edi
           wrch    %edi
           halt
```

```
           rdint   %ecx
           irmovl  $1,%ebx
           irmovl  $1,%eax
           rrmovl  %eax,%edi
           subl    %ecx,%edi
           jg      EndWhile

Loop:      multl   %eax,%ebx
           irmovl  $1,%edi
           addl    %edi,%eax
           if (%eax <= %ecx)
             goto Loop;

EndWhile:  wrint   %ebx
           irmovl  $10,%edi
           wrch    %edi
           halt
```

Starting compilation

```
           rdint  %ecx
           irmovl $1,%ebx
           irmovl $1,%eax
           if (%eax > %ecx)
             goto EndWhile;

Loop:      multl  %eax,%ebx
           irmovl $1,%edi
           addl   %edi,%eax
           if (%eax <= %ecx)
             goto Loop;

EndWhile:  wrint  %ebx
           irmovl $10,%edi
           wrch   %edi
           halt
```

```
           rdint  %ecx
           irmovl $1,%ebx
           irmovl $1,%eax
           rrmovl %eax,%edi
           subl   %ecx,%edi
           jg     EndWhile

Loop:      multl  %eax,%ebx
           irmovl $1,%edi
           addl   %edi,%eax
           rrmovl %eax,%edi
           subl   %ecx,%edi
           jle    Loop

EndWhile:  wrint  %ebx
           irmovl $10,%edi
           wrch   %edi
           halt
```


A working Y86 program

```
           rdint  %ecx         # scanf("%d", &n);
           irmovl $1,%ebx      # f = 1;
           irmovl $1,%eax      # i = 1;
           rrmovl %eax,%edi    # tmp = i;
           subl   %ecx,%edi    # tmp -= n;
           jg     EndWhile     # if (tmp > 0) goto EndWhile;

Loop:      multl  %eax,%ebx    # f *= i;
           irmovl $1,%edi      # tmp = 1;
           addl   %edi,%eax    # i += tmp;
           rrmovl %eax,%edi    # tmp = i;
           subl   %ecx,%edi    # tmp -= n;
           jle    Loop         # if (tmp <= 0) goto Loop;

EndWhile:  wrint  %ebx         # printf("%d", f);
           irmovl $10,%edi     # tmp = '\n';
           wrch   %edi         # printf("%c", tmp);
           halt                # return 0;
```


Assembly

```
0x000: f218          |            rdint  %ecx
0x002: 308301000000  |            irmovl $1,%ebx
0x008: 308001000000  |            irmovl $1,%eax
0x00e: 2007          |            rrmovl %eax,%edi
0x010: 6117          |            subl   %ecx,%edi
0x012: 762a000000    |            jg     EndWhile
                     |
0x017: 6403          | Loop:      multl  %eax,%ebx
0x019: 308701000000  |            irmovl $1,%edi
0x01f: 6070          |            addl   %edi,%eax
0x021: 2007          |            rrmovl %eax,%edi
0x023: 6117          |            subl   %ecx,%edi
0x025: 7117000000    |            jle    Loop
                     |
0x02a: f338          | EndWhile:  wrint  %ebx
0x02c: 30870a000000  |            irmovl $10,%edi
0x032: f178          |            wrch   %edi
0x034: 10            |            halt
```


The program in memory

```
0x00: f2 18 30 83 01 00 00 00
0x08: 30 80 01 00 00 00 20 07
0x10: 61 17 76 2a 00 00 00 64
0x18: 03 30 87 01 00 00 00 60
0x20: 70 20 07 61 17 71 17 00
0x28: 00 00 f3 38 30 87 0a 00
0x30: 00 00 f1 78 10 00 00 00
```