Lecture 9. Branches and Loops

- **Rewrite `sum.c` using labels and gotos**

```
#include <stdio.h>

int main(void) {
    int i = 1, n, sum = 0;

    printf("Enter n:\n");
    scanf("%d", &n);
    n--;
Top: if (n < 0) goto End;
        sum += i;
        i++;
        n--;
        goto Top;
End: printf("Sum from 1 to %d = %d\n", i - 1, sum);
    return 0;
}
```

- **Compilers implement C loop statements with branches and labels**

`while` (*conditional*)
**   *statement***

L_1: if (!*conditional*) goto L_2
**   *statement***
**   goto L_1**
L_2:

Ditto for do-while and for loops

Implementing Loops, cont'd

```
0E                                          starting address
0E:  B001     R0 <- 01                      R0 holds 1
0F:  B10A     R1 <- 0A                      R1 is n
10:  B201     R2 <- 01                      R2 is i
11:  B300     R3 <- 00                      R3 is sum
12:  2110     R1 <- R1 - R0                 n--
13:  6118     jump to 18 if R1 < 0          if (n < 0) goto End
14:  1332     R3 <- R3 + R2                 sum += i
15:  1220     R2 <- R2 + R0                 i++
16:  2110     R1 <- R1 - R0                 n--
17:  5013     jump to 13                    goto Top
18:  4302     print R3                      print sum
19:  0000     halt
```

```
% /u/cs126/bin/toy /u/cs126/toy/sum.toy
Toy simulator $Revision: 1.8 $
0037
PC = 001A
R0 = 0001  R1 = FFFF  R2 = 000B  R3 = 0037
R4 = 0000  R5 = 0000  R6 = 0000  R7 = 0000
0008: 0000 0000 0000 0000 0000 0000 B001 B10A
0010: B201 B300 2110 6118 1332 1220 2110 5013
0018: 4302 0000 0000 0000 0000 0000 0000 0000
%
```

Example: Computing Fibonacci Numbers

```
0F
0F:  B601     R6 <- 01
10:  B720     R7 <- 20
11:  9272     R2 <- M[R7+2] = M[22]
12:  9170     R1 <- M[R7+0] = M[20]
13:  9371     R3 <- M[R7+1] = M[21]
14:  4302     print R3
15:  1113     R1 <- R1 + R3
16:  4102     print R1
17:  1313     R3 <- R1 + R3
18:  4302     print R3
19:  2226     R2 <- R2 - R6 = R2 - 1
1A:  B000     R0 <- 0
1B:  2702     R7 <- R0 - R2 = -R2
1C:  6715     if R7 < 0 (i.e., R2 > 0) goto 15
1D:  0000     halt
20:  0000
21:  0001
22:  000B

% /u/cs126/bin/toy /u/cs126/toy/fib.toy
Toy simulator $Revision: 1.8 $
0001 0001 0002 0003 0005 0008 000D 0015 0022 0037 0059 ... 2AC2 452F 6FF1
PC = 001E ...
```

- **Each number is sum of the previous two numbers; two numbers per loop iteration**
- **Computes Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … limit given by R_2**

Manipulating Addresses, a.k.a. Pointers

- **Find the maximum value in an array of positive integers**

```
/*
Find the largest positive integer
in an array.
*/
#include <stdio.h>


short int a[15] = {
    0x0001, 0x0002, 0x0010, 0x1000, 0x7EFE,
    0x6030, 0x0040, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0010, 0x0000, 0x1000, 0x0000 };

int main(void) {
    int i = 14, max = 0;

    while (i >= 0) {
        if (a[i] > max)
            max = a[i];
        i--;
    }
    printf("%d\n", max);
    return 0;
}
```

```
% lcc max.c
% a.out
32510
```

Manipulating Addresses, cont'd

```
0E                                          starting address
0E: B000    R0 <- 0                         constant 0
0F: B101    R1 <- 1                         constant 1
10: B20E    R2 <- 0x0E                      i = 14
11: B322    R3 <- 0x22                      address of a
12: B600    R6 <- 0                         max = 0
13: 621B    if R2 < 0 goto 1B               while (i >= 0) {
14: 1723    R7 <- R2 + R3                       R4 <- a[i]
15: 9470    R4 <- M[R7+0] = M[R2+R3]
16: 2546    R5 <- R4 - R6                       compute a[i] - max
17: 6519    if R4-R6 < 0 jump to 19             if (a[i] > max)
18: 1640    R6 <- R4 + R0 = R4                  max = a[i]
19: 2221    R2 <- R2 - R1 = R2 - 1              i--
1A: 5013    jump to 13                      }
1B: 4602    print R6                        print max
1C: 0000    halt

22: 0001    a[0]        28: 0040                    2E: 0000
23: 0002                29: 0000                    2F: 1000
24: 0010                2A: 0000                    30: 0000    a[14]
25: 1000                2B: 0000
26: 7EFE                2C: 0000
27: 6030                2D: 0010
```

- **$R_2 + R_3$ is the address of `a[i]`; R_2 is decremented, so $R_2 + R_3$ walks backwards in a**
- **What happens if location 11 is loaded with B318?**

Function Linkages

- **Use jump and link/jump indirect to call/return to/from functions**

  jump and link **8562** $R_5 \leftarrow PC, PC \leftarrow 62_{16}$

  jump indirect **7500** $PC \leftarrow R_5$

- **power computes R3 $\leftarrow x^n$ where *x* is *passed* in R_1, *n* is *passed* in R_2**

```
int power(int x, int n) {
    int z = 1;

    while (--n >= 0)
        z *= x;
    return z;
}
```

```
14: B401        R4 <- 1                  constant 1
15: B301        R3 <- 1                  z = 1
16: 2224        R2 <- R2 - R4 = R2 - 1   while (--n >= 0)
17: 621A        if R2 < 0 jump to 1A
18: 3331        R3 <- R3 * R1                z *= x;
19: 5016        jump to 16
1A: 7500        jump to address in R5    return z
```

- ***Calling conventions*** **specify the locations of the actual arguments, the return value, and the return address; can vary among operating systems and languages on the *same machine***

Function Linkages, cont'd

- **To compute $3^4 + 2^5$**

```
04: B100     R0 <- 0
05: B11C     R1 <- 1C
06: 9110     R1 <- M[R1+0] = M[1C] = 0003
07: B204     R2 <- 4
08: 8514     call power, R5 <- 09
09: 1630     R6 <- R3 + R0 = R3 = 0051
0A: B11D     R1 <- 1D
0B: 9110     R1 <- M[R1+0] = M[1D] = 0002
0C: B205     R2 <- 5
0D: 8514     call power, R5 <- 0E
0E: 1663     R6 <- R6 + R3 = 0051 + 0020 = 0071
0F: 4602     print R6
10: 0000     halt
04

1C: 0003
1D: 0002

% /u/cs126/bin/toy /u/cs126/toy/power.toy
0071
PC = 0011
R0 = 0000  R1 = 0002  R2 = FFFF  R3 = 0020
R4 = 0001  R5 = 000E  R6 = 0071  R7 = 0000
```

- **Function linkages on 'real' machines usually involve a *stack* to hold some of the arguments**

Simulating TOY

- **Any modern computer can *simulate* TOY: Write a C program that executes TOY instructions exactly as a TOY machine would**
- **Simulate memory and registers with 16-bit integer arrays**

```
short int mem[256], regs[8];
```

- **Simulate the PC and the *fetch-increment-execute* cycle**

```
unsigned char pc;
do {
    int inst = mem[pc++];
    execute(inst);
} while (inst != HALT);
```

- **Switch statement — a multiway branch — decodes and 'executes' instructions**

```
void execute(int inst) {
    switch ((inst>>12)&0xF) {
    case ADD:
        regs[(inst>>8)&0F] = regs[(inst>>4)&0F] + regs[inst&0F];
        break;
    ...
    case JUMP:   pc = inst&0xFF; break;
    }
}
```

- **This is simplified slightly; see `/u/cs126/toy/toy.c` for the full story**