Lecture Topics

- Today: Assembly Language Basics
  (H&H 6.1-6.2)
- Next: Assembly Language Functions
  (H&H 6.3)

Announcements

- Self-study Module #7
- Project #7 (due no later than 10/26)
- Project #8 (due no later than 11/2)
Flow of Control

Three forms of control flow:

- Sequence
- Selection
- Repetition

Sequence is the default; the other two must be constructed using branches and labels.

Example: check for overflow

/* Detect overflow after addition */

```
add r7, r0, r0 @ update NZCV
bvs found_overflow @ examine NZVC
```

```
found_overflow:
```

```
Bits 31:28 are the integer condition code bits:

The icc bits are changed by "S" instructions; they are examined by "Branch" instructions.

---

Branch Instructions

Machine language format:

Execution: check NZCV bits; if condition is true, branch to a different location in the program (continue sequentially if condition is false).

PC-relative branching: new location is some displacement (+ or -) from the current location:
Recap: Example #16

Outline:

Read one character from standard input
Classify that character
dollar sign?
first or second half of ASCII table?
decimal digit?
// if( character is equal to '$' )then
// print "Found a dollar sign"
// endif

if1:
    cmp   r4, #'$
    bne   endif1

then1:
    ldr   r0, =fmt1
    bl    printf

endif1:

// if( char in first half of ASCII set )then
// print "Found ... first half of ASCII set"
// else
// print "Found ... second half of ASCII set"
// endif

if2:
    ...

then2:
    ...

else2:
    ...

endif2:
if2:
  cmp r4, #0x40
  bge else2
then2:
  ldr r0, =fmt2
  bl printf
  b endif2
else2:
  ldr r0, =fmt3
  bl printf
endif2:

// if( character is between '0' and '9' )then
//   print "Found a decimal digit"
// endif

if3:
  cmp r4, #'0'
  blt endif3
  cmp r4, #'9'
  bgt endif3
then3:
  ldr r0, =fmt4
  bl printf
endif3:
Using function printf()

Function printf() expects at least one argument: a character string (perhaps with formatting specs).

Print a message:

```c
printf( "Help!\n" );
```

Equivalent:

```c
char msg[] = "Help!\n";
printf( msg );
```

Using function printf()

Print a message and a variable:

```c
int val = 25;
printf( "Value: %d\n", val );
```

Equivalent:

```c
int val = 25;
char msg[] = " Value: %d\n ";
printf( msg, val );
```
Labels

In assembly language, labels are actually symbolic addresses – they refer to specific locations in memory.

Common uses:

- location of a data object (variable or constant)
- target for a BL instruction (function entry point)
- target for a Branch instruction (control construct)

Labels in Example #16

Ex: location of a constant data object

```assembly
fmt1:
    .asciz "\nFound a dollar sign\n"
```

Ex: target for a BL (function entry point)

```assembly
main:
    push {lr}
```

Ex: target for a Branch (control construct)

```assembly
else2:
```
Which labels are not really necessary?

// if( character is equal to '$' )then
// print "Found a dollar sign"
// endif

if1:
    cmp r4, #'$'
    bne endif1

then1:
    ldr r0, =fmt1
    bl printf

endif1:

if2:
    cmp r4, #0x40
    bge else2

then2:
    ldr r0, =fmt2
    bl printf
    b endif2

else2:
    ldr r0, =fmt3
    bl printf

endif2:
// if( character is between '0' and '9' )then
//    print "Found a decimal digit"
// endif

if3:
    cmp    r4, #'0'
    blt    endif3
    cmp    r4, #'9'
    bgt    endif3
then3:
    ldr     r0, =fmt4
    bl      printf
endif3:

Example #17
Demystifies repetitive execution

Course website:
~cse320/Examples/example17.pdf
Example #17

The program reads zero or more characters from standard input. When the end of the input stream is found, the program displays the character count.

\[
\begin{align*}
\text{count} &= 0 \\
\text{loop} & \quad \text{read one character} \\
& \quad \text{when (end-of-file) exit loop} \\
& \quad \text{count} = \text{count} + 1 \\
\text{endloop} & \quad \text{display count}
\end{align*}
\]

Example #17

The program reads zero or more characters from standard input. When the end of the input stream is found, the program displays the character count.

```
/* Example #17 -- ARM Repetitive Control Constructs */

.global main

.text

main:    push   {lr}
```

mov r4, #0
loop:
    bl getchar
    cmp r0, #-1
    beq end
    add r4, r4, #1
    b loop
end:

end:  ldr r0, =fmt
      mov r1, r4
      bl printf
      pop {lr}
      bx lr

fmt:  .asciz "\nCharacters = %d\n"
Example #18

Demonstrates nested control constructs

Course website:

~cse320/Examples/example18.pdf
Example #18

The program reads one line of text from standard input, then displays the total number of characters and the number of blanks found.

- display prompt (ask user to enter a line of text)
- initialize counts (total characters, total blanks)
- loop once for each character in the line of text
  - increment character count
  - if character is a blank, increment blank count
- when (newline) exit loop
- display counts

Example #18 -- ARM Nested Control Constructs

The program reads one line of text from standard input, then displays the total number of characters and the number of blanks found.

```
/* Example #18 -- ARM Nested Control Constructs */

.global main

.text

main: push {lr}
```
ldr r0, =fmt1
bl printf

mov r4, #0
mov r5, #0

loop: bl getchar
add r4, r4, #1

if:
  cmp r0, #0x20
  bne endif

then:
  add r5, r5, #1

endif:
  cmp r0, #0x0a
  beq end

b loop

d
end:
```
end:   ldr  r0, =fmt2
      mov  r1, r4
      mov  r2, r5
      bl   printf

      pop   {lr}
      bx   lr

fmt1:  .asciz  \nEnter a line of text: 
fmt2:  .asciz  \nThe line has %d characters,
            including %d blanks\n"
```

```
<2 lemon:~/Examples > gcc example18.s

<3 lemon:~/Examples > a.out

Enter a line of text: A short line!
The line has 14 characters, including 2 blanks

<4 lemon:~/Examples > a.out

Enter a line of text: A line with more characters.
The line has 29 characters, including 4 blanks
```
Recap: control constructs

- Selective control constructs (ifs) and repetitive control constructs (loops) must be built out of branch instructions and labels.
- Use the one entry, one exit rule: there is exactly one location where execution of a control construct starts, and exactly one location where it ends.

From Example #18:

```
loop:

if:
then:
endif:
end:
```
Efficiency Considerations

- Each machine language instruction requires one clock cycle to execute (note: there are a few instructions which require more than one clock cycle).

- One thing we can do to make a program run faster is to eliminate unnecessary machine language instructions.

Efficiency Considerations

- In Example #18, there are several places where we can eliminate instructions to improve the program:
  - revise the loop structure
  - allocate registers wisely
Example #18 loop structure:

```
loop:   bl   getchar
       add   r4, r4, #1
       .
       .
       cmp   r0, #0x0a
       beq   end

b     loop

end:
```

Revised for efficiency:

```
loop:   bl   getchar
       add   r4, r4, #1
       .
       .
       cmp   r0, #0x0a

beq   end   <= delete
b     loop  <= delete
bne   loop  <= insert

end:    <= delete
```
Revised for efficiency (final):

```
loop:  bl  getchar
       add  r4, r4, #1
       .
       .
       cmp  r0, #0x0a
       bne  loop
```

Example #18 register allocation:

@ Registers r4 and r5 used as counters

```
mov  r4, #0
mov  r5, #0
```

@ Registers r4 and r5 copied to r1 and r2
@ (arguments to function printf)

```
mov  r1, r4
mov  r2, r5
```
Example #18 register allocation (2):

```assembly
mov r1, #0  <= revise
mov r2, #0  <= revise

loop:  bl getchar
       add r1, r1, #1  <= revise

if:    cmp r0, #0x20
       bne endif

then:  add r2, r2, #1  <= revise

endif:
```

Example #18 register allocation (3):

```assembly
endif: cmp r0, #0x0a
        beq end

b loop

end:   ldr r0, =fmt2
       mov r1, r4  <= delete
       mov r2, r5  <= delete
       bl printf

@ Counts already in r1 and r2, no need to copy
```