Lecture Topics

- Today: Assembly Language Functions (H&H 6.3)
- Next: continued

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.

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:
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.

```
count = 0
loop
  read one character
  when (end-of-file) exit loop
  count = count + 1
endloop
display count
```

Example #17

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

```plaintext
loop:

if:
then:
endif:
end:
```
Subprograms

Useful to divide programs into subprograms (functions, subroutines, procedures)

Issues:

- Source code organization
- Passing control into and out of subprograms
- Sharing information

Source Code Organization

```
.globl whatever

.text
whatever:
push {lr}

! Function body

pop {lr}
bx lr
```
Passing Control

Pass control to function whatever:

```assembly
bl whatever
```

Return control from function whatever:

```assembly
bx lr
```

Sharing Information

Subprograms share information in several different ways:

- Shared memory
- Parameters
- Return values

The ARM uses variations on those general themes.
Shared Memory

On the ARM, the "data" section is available throughout program execution (the data objects have program lifetime).

Functions within one module have access to all labels in the module.

Functions in different modules have access to labels declared "global".

Parameters and Return Values

The ARM uses the following convention:

- up to four arguments may be passed into a function using r0, r1, r2 and r3
- the return value is passed out of a function in r0

If there are more then four arguments, the "extra" arguments (beyond the fourth) will be stored in memory
Register Conventions

By convention, the 16 general-purpose registers are broken into three groups:

- **r0-r3**: parameters (and return value)
- **r4-r11**: intermediate results
- **r12-r15**: used by the system

Register Conventions (2)

Parameters and return value:

- **r0**: first argument (also return value)
- **r1**: second argument
- **r2**: third argument
- **r3**: fourth argument
Intermediate results:

- Eight registers: r4-r11
- The contents of those registers must be unaltered by calling another function (the contents of each register is the same before and after calling a function).

System registers:

- r12 (ip): intra-procedure register
- r13 (sp): stack pointer
- r14 (lr): link register
- r15 (pc): program counter
Example #19

Simple example to illustrate parameter passing and control flow. Function “main” calls function “add_three” to add three integer values and return the sum.

Course website:

~cse320/Examples/example19.pdf

The fetch/execute cycle

Normally, execution proceeds sequentially (the next instruction is in the next 4 bytes)

Branch instructions alter the order in which instructions are executed – allows us to jump to a different location in the program
Stored program

```
000103e8:  e52de004   push    (lr)
000103ec:  e59f4038   ldr     r4, =x
000103f0:  e5940000   ldr     r0, [r4]
000103f4:  e59f5034   ldr     r5, =y
000103fc:  e5951000   ldr     r1, [r5]
00010400:  e59f6030   ldr     r6, =z
00010404:  e5962000   ldr     r2, [r6]
00010408:  eb000003   bl      add_three
0001040c:  e59f7028   ldr     r7, =sum
00010410:  e5870000   str     r0, [r7]
00010414:  e49de004   bx      lr
00010418:  e52de004   push    (lr)
0001041c:  e0800001   add     r0, r0, r1
00010420:  e0800002   add     r0, r0, r2
00010424:  e49de004   pop     (lr)
00010428:  e12fff1e   bx      lr
```

Execution trace (in temporal order)

```
T+00  000103e8:  e52de004   push    (lr)
T+01  000103ec:  e59f4038   ldr     r4, =x
T+02  000103f0:  e5940000   ldr     r0, [r4]
T+03  000103f4:  e59f5034   ldr     r5, =y
T+04  000103fc:  e5951000   ldr     r1, [r5]
T+05  00010400:  e59f6030   ldr     r6, =z
T+06  00010404:  eb000003   bl      add_three
T+07  00010408:  e59f7028   ldr     r7, =sum
T+08  00010410:  e49de004   pop     (lr)
T+09  00010414:  e0800001   add     r0, r0, r1
T+10  00010418:  e52de004   push    (lr)
T+11  00010420:  e0800002   add     r0, r0, r2
T+12  00010424:  e49de004   pop     (lr)
T+13  00010428:  e12fff1e   bx      lr
T+14  00010408:  e59f7028   ldr     r7, =sum
T+15  0001040c:  e5870000   str     r0, [r7]
T+16  00010410:  e49de004   pop     (lr)
```

23

24
Recap: ARM Subprograms

- Transfer control into callee with BL (copies return address into r14 (lr))
- Transfer control back to caller with BX (uses return address in r14 (lr))
- Use stack to save r14 (lr) so that the return address is not lost if there are subsequent function calls

Recap: ARM Control Flow

- BL transfer control to subprogram
- BX returns control from subprogram
Recap: ARM Register Conventions

- r0-r3: parameters (and return value)
- r4-r11: intermediate results
- r12-r15: used by the system

Caller:
- Place arguments in the appropriate registers
- BL
  - Place return address in r14
  - Jump to callee

Callee:
- Save r14 in the stack
- Restore r14 from the stack
- BX
  - Get return address from register r14
  - Jump back to caller
Nest Function Calls

- OK to have nested function calls
- Note: r14 (lr) must be saved in the stack

Example #20

Same as Example #19, except more efficient: function "add_three" written as a leaf function (function which calls no other functions).

Course website:

~cse320/Examples/example20.pdf
Preserving Registers

- Recall the convention: registers r4-r11 cannot be altered by calling a function
- If a function uses any of those registers, it should save and restore them:
  
  \[
  \text{push} \quad \{r4, r5, r6, r7, lr\} \\
  \text{pop} \quad \{r4, r5, r6, r7, lr\}
  \]

Stack: Last-In, First-Out

- A stack is a data structure intended for inserting and removing items in a LIFO discipline
- The most recently inserted item will be the one that will be the soonest to be removed
- To ‘push’ an item means to insert it, and to ‘pop’ an item means to remove it
Example

- Start with empty stack
- Push first item
- Push second item

Example (continued)

- Pop (returns most recent item)
Example #21

Function “main” written in C, function “add” written in ARM assembly language.

Function “add” handles 32-bit addition and checks for overflow, displays results.

Function “main” calls function “add”, displays the return value.

Course website:
~cse320/Examples/example21.pdf

Interfacing with C

We’ve been following the conventions used by C for functions:

- All parameters passed by value
- One return value

The conventions aren’t much different for C++ (add reference parameters, name mangling).
Example #21

Function “main” written in C, function “add” written in ARM assembly language.

Function “add” handles 32-bit addition and checks for overflow, displays results.

Function “main” calls function “add”, displays the return value.

Course website:
~cse320/Examples/example21.pdf

```c
#include <stdio.h>

/* three value params: two ints, one address */

int add( int, int, int* );

int main()
{
  int A = 0x7fffffff, B = 0x00000002;
  int C = 0xffffffff, D = 0x00000001;
  int R;
  int flag;
```
/* Place value of A in %o0
Place value of B in %o1
Place address of R in %o2
Call add
Store %o0 into memory at Flag */

flag = add( A, B, &R );

printf( "Results: %08x", R );
if (flag == 1)
{
    printf( " (not valid -- overflow)" );
}
printf( "\n\n" );

flag = add( C, D, &R );

printf( "Results: %08x", R );
if (flag == 1)
{
    printf( " (not valid -- overflow)" );
}
printf( "\n\n" );

return 0;
}

<2 lemon:~/Examples > gcc ex21.driver.c ex21.support.s
<3 lemon:~/Examples > a.out

7fffffff
+ 00000002
----------
= 80000001  (** overflow **)

Results: 80000001 (not valid -- overflow)

fffffff
+ 00000001
----------
= 00000000

Results: 00000000

<1 lemon:~/Examples > cat example21.support.s

.global add

.text
add:
    push {r4,r5,r6,r7,lr}
    mov r4, r0  @ Save first argument
    mov r5, r1  @ Save first argument
    mov r6, r2  @ Save third argument
ldr    r0, =fmt1
mov    r1, r4
bl     printf  @ Display first 4-byte op

ldr    r0, =fmt2
mov    r1, r5
bl     printf  @ Display second 4-byte op

adds   r7, r4, r5  @ Add operands, set NZCV
str    r7, [r6]  @ Store result at pointer

bvs    overflow  @ Check for overflow
valid:
    ldr    r0, =fmt3
    mov    r1, r7
    bl     printf  @ Display the sum
    mov    r0, #0  @ Return 0 (valid)
b     done

overflow:
    ldr    r0, =fmt4
    mov    r1, r7
    bl     printf  @ Display the sum, error
    mov    r0, #1  @ Return 1 (not valid)
Example #22

Function “main” written in ARM assembly language, function "sub" written in C.

Displays part of the run-time stack several times.

Course website:
~cse320/Examples/example22.pdf