Lecture Topics

- Today: Assembly Language Subprograms (H&H 6.3)
- Next: Assembly Language Data Organization (H&H 6.3)

Announcements

- Self-study Module #7 (control constructs)
- Self-study Module #8 (subprograms)
- Project #6 (due no later than 6/9)
- Project #7 (due no later than 6/13)
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

```
.global whatever

.text
whatever:
push  {lr}

! Function body

pop   {lr}
bx     lr
```
Passing Control

Pass control to function `whatever`:

```
bl whatever
```

Return control from function `whatever`:

```
bx lr
```
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, must be preserved across function calls
- r12-r15: used by the system

Nest Function Calls

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

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

```
<1 lemon:~/Examples > cat example21.driver.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

23

<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

24
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)
done:
    pop   {r4,r5,r6,r7,lr}
    bx    lr

fmt1: .asciz "   %08x\n"
fmt2: .asciz "+ %08x\n"
fmt3: .ascii "--------\n".asciz "= %08x\n"
fmt4: .ascii "--------\n".asciz "= %08x (*** overflow ***)\n"

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
ARM Memory Model

Memory is viewed as a linear sequence of bytes (flat memory model).

Addresses are 32 bits wide, so there are $2^{32}$ bytes (4 Gigabytes).

Addresses range from 00000000 to FFFFFFFF.

Manage as “segments” of 64 Kilobytes:
(00000000 to 0000FFFF, 00010000 to 0001FFFF, 00002000 to 0000FFFF, etc.)
### Relationship to C/C++ data objects:

- **program lifetime – data section**
  - global variables
  - static variables

- **block lifetime – stack**
  - local variables
  - parameters

- **programmer-defined lifetime – heap**
  - dynamically allocated memory

### Operating system controls access to segments (equivalent of file permissions):

- **OS segment** – no user access
- **text segment** – read-only access
- **data segment** – read-write access
- **heap** – read-write access
- **stack** – read-write access

Invalid access: segmentation fault
Assembler Directives

Some assembler directives are used to control the assembly process (take some action during assembly, as opposed to generating machine language which will take some action during execution).

Some assembler directives are used to reserve memory (and generate initial values).

```assembly
.text
.align 2
.global main
main:  push {lr}
   ...
   pop {lr}
   bx lr

.data
.align 2
total: .word 0
```
Switch between sections (starts in .text by default). Force next address to be a multiple of 4 (machine language instructions are 4 bytes, PC used to access RAM).

```
.text
.align 2
```

Switch between sections. Force next address to be a multiple of 4 (LDR or STR will be used to access total).

```
data
.align 2
total: .word 0
```

---

Reserve Space

Directives for integer-like objects:

```plaintext
blank: .byte 0x20
max: .short 500
sum: .word 0
list: .word 5, 10, -6, 25
```
Reserve Space

Directives for string objects:

```
str:   .ascii   "First"
      .asciz   "Second"
```

Directive for uninitialized space:

```
room:   .skip    100
```

---

Bytes Within A Word

- **Little-endian**: byte numbers start at the little (least significant) end
- **Big-endian**: byte numbers start at the big (most significant) end
- **Swift’s Gulliver’s Travels**: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- **It doesn’t really matter** which addressing type used – except when two systems share data
### Textbook's convention

Memory locations shown from largest address to smallest address:

<table>
<thead>
<tr>
<th>Big-Endian</th>
<th>Little-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte Address</strong></td>
<td><strong>Word Address</strong></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>C D E F</td>
<td>C F E D</td>
</tr>
<tr>
<td>8 9 A B</td>
<td>8 B A 9</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4 7 6 5</td>
</tr>
<tr>
<td>MSB</td>
<td>MSB</td>
</tr>
<tr>
<td>LSB</td>
<td>LSB</td>
</tr>
</tbody>
</table>