Lecture Topics

- Today: Assembly Language Functions (H&H 6.3)
- Next: Assembly Language Data Org. (H&H 6.3)

Announcements

- Self-study Module #8
- Project #8 (due no later than 11/2)
- Project #9 (due no later than 11/9)
Subprograms

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

Issues:

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

```
global 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.
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”.

---

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 than 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

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
### Recap: ARM Control Flow

- BL transfer control to subprogram
- BX returns control from subprogram

![Recap Diagram](image.png)
Nest Function Calls

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

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:
  
  \[
  \text{push } \{r4,r5,r6,r7,lr\} \\
  \text{pop } \{r4,r5,r6,r7,lr\}
  \]
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" );

return 0;

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

Results:  80000001 (not valid -- overflow)

ffffffffff
+ 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
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, 00020000 to 0002FFFF, 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).