CC2511 Week 11

Calendar

Week	Lecture	Lab
11 (you are here)	Assembly part 2	Assembly part 2
12	Revision & Exam prep	None (open session for assignment work)
13	None (work on your assignment)	Demonstrate your assignment

- You’re nearly at the end of the subject.
- **This is the last week with new content.**

Today: More on assembly language

- Calling conventions
- Defining functions in assembly language
- Array indexing
- Mixing C and assembly code

Revision: what do these instructions do?

```
start:
    mov r0, #0
    ldr r1, =0x40048080
    ldr r2, [r1,#4]
    ldr r3, [r1,r0]
    b start
```

A historical note

- The GNU assembler defaults to an older syntax in which ARM and Thumb modes were written differently.
- Some Thumb2 instructions cannot be expressed in the older syntax.
- To use the new syntax, place the following directive at the top of the assembly file:

```
.syntax unified
```

- Without this command, some instructions become unavailable, e.g. many that manipulate the higher registers.

How to implement functions in assembly?

- Call a function by using one of the **branch** instructions.
- Questions:
  - How to pass arguments?
  - How to communicate the return value?
  - Is a function permitted to modify the registers?
  - How does the function know where to return to (branch to) when it’s done?

- Clearly, the caller and the function must agree on these points.
- The answers to these questions define a “**calling convention**”.

ARM calling convention: Passing arguments

- The first four function arguments are placed in r0, r1, r2, and r3. Subsequent arguments (if any) are placed on the stack.
- The return value is placed in r0 when the function ends.

```
int f(int a, int b) {
      // a is in r0
      // b is in r1
      ...
      // the return value is in r0
}
```

ARM calling convention: The link register

- The return address is stored in the **link register** (lr).
- To **call** a function, use bl ("branch and link")

```
/* move arguments into r0, r1, ... */
bl func
```

- The bl instruction:
  1. Places the address of the next instruction (after the bl) into the link register (lr).
  2. Jumps to the specified label (i.e. loads its address into pc)

Returning from a function

- To return from a function, branch to the address stored in the link register:

```
    bx lr   /* branch to the address in lr */

/* This is equivalent to: */
    mov pc, lr
/* but generally prefer the bx syntax */
```

ARM calling convention: Preserving registers

- Functions can freely modify r0-r3.
- Functions must preserve r4-r11.
- r12 can be freely modified on Kinetis.
  - (R12 is also called "IP" and has a special meaning on certain systems, where it is used for dynamic linking. Small microcontrollers do not have dynamic linking, so here r12 is a general purpose register.)

- In other words, the caller can assume its data in r4-r11 is retained.
- Any time a subfunction is called, r0-r3 and r12 may be overwritten.

Preserving registers

- If a function uses any of r4 – r11, it must restore the original values before returning.
- Can do this by pushing the old values to the stack.
- The ARM push instruction can move multiple registers at once.
  - The op-code uses a bitfield to specify which registers are to be pushed.

```
Myfunc:
        push {r4-r11}
        …
        pop {r4-r11}
        bx lr
```

The link register and nested function calls

- The link register tells a function where to return to.
- If a function calls another, then the link register will be overwritten!
- Need to save the link register before bl (“branch and link”) changes it.
- The standard approach is to push it to the stack at the start of the function.

```
push {lr}
...
pop {pc} /* replace pc with the previous lr */
```

Complying with the calling convention

```
func:
    /* Save r4-r11 and the link register: */
    push {r4-r11, lr}

    /* Function code here */

    /* Restore r4-r11, and set pc to the old lr */
    pop {r4-r11, pc} // replaces “bx lr”
```

Complying with the calling convention

```
Simple_func:
/* This fn returns 0 */
/* Not saving registers
because they aren’t
modified. */
    mov r0, #0
    bx lr
```

- Technically, only need to save r4-r11 and lr if they are modified.
- Very simple functions might not need to use these registers.
- Be careful: If you later change the function and forget to save the registers, you introduce a hard-to-find bug.

Exercise: write in assembly

```
int add(int a, int b)
{
    return a + b;
}

int main()
{
    add(1, 2);
}
```

Solution

```c
int add(int a, int b)
{
    return a + b;
}

int main()
{
    add(1, 2);
}
```

```asm
add:
    add r0, r1
    bx lr

main:
    mov r0, #1
    mov r1, #2
    bl add
```

The ARM stack

- ARM CPUs use a “full, descending” stack.
- “**Full**” means the stack pointer points to the **last item pushed**.
- “**Descending**” means that addresses **decrement** when items are pushed.
- The stack pointer is called “sp”.

Allocating memory on the stack: directly manipulate the stack pointer

```
func:
  push {r4-r11, lr} /* start of function */
  sub sp, #8        /* reserve space for two ints */

  str r0, [sp]      /* example of writing to 1st variable */
  str r0, [sp,#4]   /* example of writing to 2nd variable */

  add sp, #8        /* restore the stack pointer */
  pop {r4-r11, pc}  /* end of function */
```

Sections

- The current "section" is indicated by an assembly directive such as
  `.text`

- The "text" section:
  - Is read-only during execution (i.e. flash memory)
  - Contains executable code and constants

- The "data" section:
  - Is read-and-write during execution (i.e. RAM)
  - Contains variables

Statically allocating variables

```
/* At the top of the file, above the code */
      .data
      .align 2
variable_name:
      .skip 4*10  /* leave 40 bytes, i.e. 10 ints */
```

Labels evaluate as addresses

```
    .data
    .align 2
variable_name:
    .skip 4*10  /* leave 40 bytes, i.e. 10 ints */
/* ... */
    .text
    .align 2
main:
    ldr r0, =variable_name
```

Allocation of constants

- Constants can be placed in the text section:

```
        .text
        .align 2
str1:
        .ascii "Hello from ASM\0"

        .align 2
const1:
        .word 0x11112222
```

Loading constants

```
        .text
        .align 2
const1:
        .word 0x11112222
```

```
        .text
        .align 2
main:
// Load address of const1
        ldr r0, =const1
// Load value of const1
        ldr r0, [r0]
```

Calling assembly functions from C

1. Define a function prototype telling the C compiler the data types to pass to the function:

```
// In a header file:
int add(int a, int b);
```

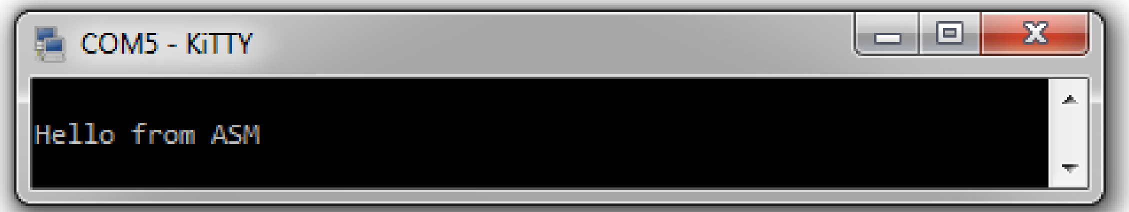
2. Write the function in assembly:

```
    .global add
add:
    add r0, r1 /* place the return value in r0 */
    bx lr
```

Calling C functions from assembly: complete example

```
    .syntax unified
/* Constants */
    .text
msg:
    .ascii "Hello from ASM\r\n\0" /* notice the explicit trailing NULL */

/* Function */
    .text
    .align 2
    .global hello
hello: /* the function name */
    push {r4-r11, lr}
    ldr r0, =msg /* first and only argument: pointer to the string */
    bl Term1_SendStr
    pop {r4-r11, pc}
```



Functions with more than 4 arguments

```
int asm_func(int a, int b, int c, int d, int e, int f, int g);

asm_func:
/* a-d are in r0-r3 */
/* e is at [sp] but we need to use the stack to save registers: */
      push {r4-r7,lr} /* 5*4=20 bytes */
/* As a result of this push, arg4 is now at [sp+20]
      ldr r4, [sp,#20] /* argument e */
      ldr r5, [sp,#24] /* argument f */
      ldr r6, [sp,#28] /* argument g */
/* Notice that we don't remove the arguments from the stack! */
      pop {r4-r7,pc} /* return */
```

Common design patterns: array indexing

```c
void zeroArray(int *a)
{
	int i;
	for (i=0;i<10;i++) {
		a[i] = 0;
	}
}
```

```
zeroArray:
	mov r1, #0 // value to store
	mov r2, #0 // array index (i)
loop1:
	str r1, [r0,r2]
	add r2, #4
	cmp r2, #40
	blt loop1
	bx pc
```

The adds and subs instructions

- Consider:

```
sub r1, #4
cmp r1, #0
bne my_loop /* branch if r1 is nonzero */
```

- The case of comparing with zero is so common that it was added to the subtract instruction:

```
subs r1, #4 /* notice s on subs */
bne my_loop /* branch if r1 is nonzero */
```

- This suffix also exists on other instructions (such as add).

Assembly language reference page 3

Where does assembly code provide benefits over C?

- Explicit choice of instruction to use, e.g. the size of the multiply instruction.
- Detection of integer overflows.
- Multiply and accumulate, which performs x = x + a*b in one step.
- Parallel operations, performing the same instruction on multiple data values.

Multiply into a 64 bit result

- A 32-bit x 32-bit multiplication in general produces a 64-bit result.
- Expressing this in C is awkward because the cast must be done before the multiply:

```
long long int result = (long long int)a * (long long int)b;
```

- ARM assembly has UMULL and SMULL instructions for unsigned and signed integers, respectively:

```
smull RdLo, RdHi, Rn, Rm
```

RdLo contains the least significant 32 bits of the result, RdHi contains the most significant 32 bits of the result, and Rn and Rm are the operands.

Detecting overflows

- The C standard says that overflows result in “undefined behaviour”.
- Most implementations wrap around, i.e. INT_MAX+1 = INT_MIN but this is not guaranteed!
- Assembly code allows overflows to be detected using the **v** condition flag.

```
ldr r0, =0x7FFFFFFF /* max two's complement integer */
adds r0, #1 /* will set flag v on overflow */
bvs overflow /* branch if v is set */
```

- There is also bvc (branch if v is cleared) for detecting no overflow.

Multiply and accumulate

- Multiply and accumulate is a common operating in digital signal processing (DSP) applications.
- The operation is x = x + a*b.

```
umlal RdLo, RdHi, Rn, Rm
```

1. Multiplies Rn and Rm to produce a 64-bit result.
2. Adds that result with the 64-bit number already in RdLo, RdHi.
3. Stores the new result back into RdLo, RdHi.

Summary

- Function arguments are in r0-r3.
- Must preserve r4-r11 and lr by pushing these to the stack and popping them at the end of the function (pop the old lr into pc).
- The return value is placed in r0.

- The task in this week's lab is to implement a high-performance assembly code function (and try to be faster than a C implementation).