Modern Assembly Language Programming with the ARM processor

Chapter 5: Structured Programming

1. Introduction
2. Structured Programming
3. Selection
4. Iteration
5. Calling Functions
6. Writing Subroutines
7. Aggregate Data Types

Why Use Structured Programming?

Structured code is:

- easier to write,
- easier to understand,
- easier to debug, and
- easier to maintain.

Good high-level *languages* enforce strucured programming.

Good assembly *programmers* enforce strucured programming.

Blocks

A “block” of code

- contains one or more statements (instructions),
- has one entry point and one exit point,
- may contain other blocks.

Flow control structures are used to control which blocks are executed.

Flow Control

All programs can be written using only:

Sequencing Execute instructions (statement) sequentially. Blocks which contain only basic instructions (statements) which are executed sequentially, are called “basic blocks”.

Selection Execute a block of instructions, a, or a block of instructions, b, but not both. A selection structure also forms a block, but not a basic block.

Iteration Execute the same block of instructions, a, zero or more times. An iteration structure also forms a block, but not a basic block.

Blocks can be executed sequentially, selectively, or iteratively.
All programming is done with blocks. High level languages enforce the use of blocks. Assembly does not!

If-Then-Else

The following two slides show two ways to implement the following C code:

```
1   static int a = 10;
2   static int b = 4;
3   static int x;
4
5   int main()
6   {
7     if ( a < b )
8       x = 1;
9     else
10      x = 0;
11    .
12    .
13    .
```

If-Then-Else with Conditional Execution

```
1        .data
2  a:    .word   10    @ static int a=10;
3  b:    .word   4     @ static int b=4;
4  x:    .word   0     @ static int x;
5        .text
6        .globl  main
7  main: ldr     r0, =a        @ load pointer to 'a'
8        ldr     r1, =b        @ load pointer to 'b'
9        ldr     r0, [r0]      @ load 'a'
10       ldr     r1, [r1]      @ load 'b'
11       cmp     r0, r1        @ compare them
12       movlt   r0, #1        @ THEN section - load 1 into r0
13       movge   r0, #0        @ ELSE section - load 0 into r0
14       ldr     r1, =x        @ load pointer to 'x'
15       str     r0, [r1]      @ store r0 in 'x'
```

If-Then-Else with Branch Instructions

```
1         .data
2  a:     .word   10    @ static int a=10;
3  b:     .word   4     @ static int b=4;
4  x:     .word   0     @ static int x;
5         .text
6         .globl  main
7  main:  ldr     r0, =a        @ load address of 'a'
8         ldr     r1, =b        @ load address of 'b'
9         ldr     r0, [r0]      @ load 'a'
10        ldr     r1, [r1]      @ load 'b'
11        cmp     r0, r1        @ compare them
12        bge     else          @ if a >= b then goto else_code
13        mov     r0, #1        @ THEN section - load 1 into r0
14        b       after         @ skip the else section
15 else:  mov     r0, #0        @ ELSE section - load 0 into r0
16 after: ldr     r1, =x        @ load pointer to 'x'
17        str     r0, [r1]      @ store r0 in 'x'
```

For and While Loop in C

```c
int main()
{
  int i;
  for(i=0;i<10;i++)
    printf("Hello World - %d\n",i);
  return 0;
}
```

Any `for` loop can be converted to a `while` loop.

```c
 int main()
 {
   int i;
   i = 0;
   while(i<10)
   {
     printf("Hello World - %d\n",i);
     i++;
   }
   return 0;
}
```

For and While Loop in Assembly

```
1        .data
2  str:  .asciz "Hello World - %d\n"
3
4        .text
5        .globl main
6  main: @ We are going to use r4 and make a function call, so
7        stmfd  sp!,{r4,lr}  @ push lr and r4 onto stack
8        mov    r4, #0       @ use r4 for i; i=0
9  loop: cmp    r4, #10      @ perform comparison
10       bge    done         @ end loop if i >= 10
11       ldr    r0, =str     @ load pointer to format string
12       mov    r1, r4       @ copy i into r1
13       bl     printf       @ printf("Hello World - %d\n",i);
14       add    r4, r4, #1   @ i++
15       b      loop         @ repeat loop test
16 done: mov    r0, #0       @ move return code into r0
17       ldmfd  sp!,{r4,lr}  @ pop lr and r4 from stack
18       mov    pc, lr       @ return from main
19       .end
```

Do-While Loop in C

If we know for certain that the body of a `for` or `while` loop will execute at least once, then we can convert it to a (more efficient) `do-while`

```c
int main()
{
  int i;
  for(i=0;i<10;i++)
    printf("Hello World - %d\n",i);
  return 0;
}
```

```c
int main()
{
  int i = 0;
  do {
    printf("Hello World - %d\n",i);
    i++;
  } while(i<10)
  return 0;
}
```

Do-While Loop in Assembly

```
1        .data
2  str:  .asciz  "Hello World - %d\n"
3        .text
4        .globl  main
5  main:
6        @ We are going to use r4 and make a function call, so
7        stmfd   sp!,{r4,lr}  @ push lr and r4 onto stack
8        ldr     r4, #0       @ use r4 for i; i=0
9  loop: ldr     r0, =str     @ load pointer to format string
10       mov     r1, r4       @ copy i into r1
11       bl      printf       @ printf("Hello World - %d\n",i);
12       add     r4, r4, #1   @ i++
13       cmp     r4, #10      @ perform comparison
14       blt     loop         @ end loop if i >= 10
15       mov     r0, #0       @ move return code into r0
16       ldmfd   sp!,{r4,lr}  @ pop lr and r4 from stack
17       mov     pc, lr       @ return from main
18       .end                 @ tell assembler that we are done
```

Calling Standard C Library Functions

```
1         .data
2   str1: .asciz  "%d"
3   str2: .asciz  "You entered %d\n"
4   n:    .word   0
5         .text
6         .globl  main
7   main: stmfd   sp!,{lr}    @ push link register onto stack
8         ldr     r0, =str1   @ load address of format string
9         ldr     r1, =n      @ load address of int variable
10        bl      scanf       @ call scanf("%d",&n)
11        ldr     r0, =str2   @ load address of format string
12        ldr     r1, =n      @ load address of int variable
13        ldr     r1, [r1]    @ load int variable
14        bl      printf      @ call printf("You entered %d\n",n)
15        mov     r0, #0      @ load return value
16        ldmfd   sp!,{lr}    @ pop link register from stack
17        mov     pc, lr      @ return from main
```

ARM Function Calling Conventions

r0 (a1)	Used to pass argument values into a subroutine and to return a result value from a function. They may also be used to hold intermediate values within a routine. Caller assumes they will be modified.
r1 (a2)	
r2 (a3)	
r3 (a4)	
r4 (v1)	A subroutine must preserve (or save and restore) the contents of these registers. If they are used, they must be pushed to the stack at the beginning of the subroutine/function, and restored before returning.
r5 (v2)	
r6 (v3)	
r7 (v4)	
r8 (v5)	
r9 (v6)	
r10 (v7)	
r11 (fp) (v8)	
r12 (ip)	Intra-procedure scratch register. May be modified.
r13 (sp)	Program stack pointer.
r14 (lr)	Link Register (return address). See `bl` instruction.
r15 (pc)	Program Counter. Changing this causes a branch.
CPSR	

Passing One Argument

Passing a pointer to a string.

```
1    printf("Hello World");
```

```
1  @ load first param (pointer to  format string) in r0
2  ldr   r0, =hellostr  @ hellostr previously declared
3  @ call printf
4  bl    printf
```

Passing Four Arguments

Some variables may be in memory, others may be already in registers.

They all have to be copied to the correct registers before the function is called.

```
1    printf("The results are: %d %d %d\n",i,j,k);
```

```
1  @ load first param (pointer to format string) in r0
2  ldr   r0, =formatstr
3  ldr   r1, =i          @ load pointer to i in r1
4  ldr   r1, [r1]        @ load value of i in r1
5  mov   r2, r6          @ value of j was in r6. copy to r2
6  ldr   r3, =k          @ load pointer to k in r3
7  ldr   r3, [r3]        @ load value of k in r3
8  @ call printf
9  bl    printf
```

Passing More Than Four Arguments

```
printf("The results are: %d %d %d %d %d\n",i,j,k,l,m);
```

```
  ldr   r0,=m          @ load pointer to last variable 'm'
  ldr   r0,[r0]        @ load value of m
  str   r0,[sp,#-4]!   @ push it on the stack
  ldr   r0,=l          @ load pointer to variable 'l'
  ldr   r0,[r0]        @ load value of l
  str   r0,[sp,#-4]!   @ push it on the stack
  @ load first param (pointer to format string) in r0
  ldr   r0, =resultstr
  ldr   r1, =i         @ load pointer to i in r1
  ldr   r1, [r1]       @ load value of i in r1
  mov   r2, r6         @ value of j was in r6. copy to r2
  mov   r3, r7         @ value of k was in r7. copy to r3
  @ call printf
  bl    printf
  add   sp,sp,#8       @ pop 2 words from the stack
```

Rules for a Subroutine or Function

When writing a subroutine or function:

- the first four parameters are in `r0-r3`,
- any additional parameters can be accessed with `ldr rd,[sp,#offset]`,
- the *calling* function will remove parameters from the stack, if necessary,
- if the function return type is not `void`, then the return value must be placed in `r0` (and possibly `r1, r2, r3`), and
- the return address will be in `lr`.

A Simple Function

```
1 int myfun(int a, int b, int c, int d, int e, int f)
2 {
3   return a+b+c+d+e+f;
4 }
```

```
1 myfun: add     r0,r0,r1    @ r0 = a + b
2        add     r0,r0,r2    @ r0 = a + b + c
3        add     r0,r0,r3    @ r0 = a + b + c + d
4        ldr     r1,[sp,#0]  @ load e from stack
5        add     r0,r0,r1    @ r0 = a + b + c + d + e
6        ldr     r1,[sp,#4]  @ load f from stack
7        add     r0,r0,r1    @ r0 = a + b + c + d + e + f
8        mov     pc,lr       @ return from function
```

Automatic Variables

Automatic (local) variables *may* be allocated on the stack.

```
1  int doit()
2  { int x[20];
3    register int i;    /* try to keep i in a register */
4    for(i=0;i<20;i++) x[i] = i;
5    return i;
6  }
```

```
1  doit: sub    sp,sp,#80 @ Allocate  'x' on stack
2        mov    r2,#0     @ use r2 as 'i'
3  loop: cmp    r2,#20    @ pre-test loop
4        bge    done      @ quit if i >= 20
5        str    r2,[sp,r2,asl#2] @  x[i] = i;
6        add    r2,r2,#1  @ i++
7        b      loop      @ go back to loop test
8  done: mov    r0,r2     @ return i
9        add    sp,sp,#80 @ destroy automatic variable
10       mov    pc,lr     @ return from function
```

Recursion in C

```
1  void reverse(char *a,int left, int right)
2  { char tmp;
3    if(left<right)
4      {
5        tmp=a[left];
6        a[left]=a[right];
7        a[right]=tmp;
8        reverse(a,left+1,right-1);
9      }
10 }
11 int main()
12 { char *str="This is the string to reverse";
13   printf(str);
14   reverse(str,0,strlen(str)-1);
15   printf(str);
16   return 0;
17 }
```

Recursion in Assembly

```
1  reverse:stmfd  sp!,{lr}    @ I may call myself:save lr
2          sub    sp,sp,#4    @ Allocate tmp on stack
3          cmp    r1,r2       @ if(left>=right)
4          bge    exit        @ then return
5          ldrb   r3,[r0,r1]  @ load character at a[left]
6          strb   r3,[sp,#0]  @ store in tmp
7          ldrb   r3,[r0,r2]  @ load character at a[right]
8          strb   r3,[r0,r1]  @ store in a[left]
9          ldrb   r3,[sp,#0]  @ load tmp
10         strb   r3,[r0,r2]  @ store in a[right]
11         add    r1,r1,#1    @ calculate left+1
12         sub    r2,r2,#1    @ calculate right-1
13         bl     reverse     @ make recursive call
14 exit:   ldr    lr,[sp,#4]  @ get lr from 4 bytes above sp
15         add    sp,sp,#8    @ restore sp to original value
16         mov    pc,lr       @ return from function
```

Much Better Recursion in Assembly

```
1  reverse:cmp     r1,r2        @ if(left>=right)
2          bge     exit         @ then return
3          stmfd   sp!,{lr}     @ I WILL call myself-save lr
4          ldrb    r3,[r0,r1]   @ load character at a[left]
5          ldrb    ip,[r0,r2]   @ load character at a[right]
6          strb    r3,[r0,r2]   @ store in a[right]
7          strb    ip,[r0,r1]   @ store in a[left]
8          add     r1,r1,#1     @ calculate left+1
9          sub     r2,r2,#1     @ calculate right-1
10         bl      reverse      @ make recursive call
11         ldmfd   sp!,{lr}     @ pop lr from the stack
12 exit:   mov     pc,lr        @ return from function
```

Using Pointers in C

```
1  void reverse(char *left, char *right)
2  {
3    char tmp;
4    if(left<=right)
5    {
6      tmp=*left;
7      *left=*right;
8      *right=tmp;
9      reverse(left+1,right-1);
10   }
11 }
12 int main()
13 { char *str="This is the string to reverse";
14   printf(str);
15   reverse(str,str+strlen(str)-1);
16   printf(str);
17   return 0;
18 }
```

Using Pointers in Assembly

```
1  reverse:cmp     r0,r1       @ if(left>=right)
2          bge     exit        @ then return
3          stmfd   sp!,{lr}    @ I WILL call myself-save lr
4          ldrb    r3,[r0]     @ load character at *left
5          ldrb    ip,[r1]     @ load character at *right
6          strb    ip,[r0]     @ store in *left
7          strb    r3,[r1]     @ store in *right
8          add     r0,r0,#1    @ calculate left+1
9          sub     r1,r1,#1    @ calculate right-1
10         bl      reverse     @ make recursive call
11         ldmfd   sp!,{lr}    @ pop lr from the stack
12 exit:   mov     pc,lr       @ return from function
```

Arrays

```
1          :
2          int x[100];
3          int i;
4
5          for(i=0;i<100;i++)
6            x[i] = 0;
7          :
```

```
1          :
2        sub   sp, sp, #400        @ allocate 400 bytes in stack
3        mov   r0, #0              @ use r0 to hold the index
4        mov   r1, #0              @ value to initialize with
5  loop: str   r1, [sp,r0,lsl #2] @ set array element to zero
6        cmp   r0, #100            @ loop test
7        add   r0, r0, #1          @ increment index
8        blt   loop                @ loop while index < 100
9          :
```

Using a C `struct`

```
1  struct student {
2      char first_name[30];
3      char last_name[30];
4      unsigned char class;
5      int grade;
6    };
7    :
8    struct student newstudent;  /* allocate on the stack */
9    strcpy(newstudent.first_name,"Sam");
10   strcpy(newstudent.last_name,"Smith");
11   newstudent.class = 2;
12   newstudent.grade = 88;
13   :
```

Equivalent in Assembly

```
1          .data
2          .equ    s_first_name, 0
3          .equ    s_last_name, 30
4          .equ    s_class, 60
5          .equ    s_grade, 64
6          .equ    s_size, 68
7  sam:    .asciz  "Sam"
8  smith:  .asciz  "Smith"
```

Equivalent in Assembly (continued)

```
1       :
        :
2       sub   sp, sp,#s_size          @ allocate struct on the stack
3       mov   r0, sp                  @ put pointer to struct in r0
4       add   r0, r0, #s_first_name@ offset to first name field
5       ldr   r1, =sam                @ load pointer to "Sam"
6       bl    strcpy                  @ copy the string
7       mov   r0, sp                  @ put pointer to struct in r0
8       add   r0, r0, #s_last_name @ offset to last name field
9       ldr   r1, =smith              @ load pointer to "Smith"
10      bl    strcpy                  @ copy the string
11      mov   r0, sp                  @ put pointer to struct in r0
12      mov   r1, #2                  @ load constant value of 2
13      strb  r1, [r0, #s_class]    @ store with offset
14      mov   r1, #88                 @ load constant value of 88
15      str   r1, [r0, #s_grade]    @ store with offset
16      :
        :
```