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 structured programming.

Good assembly programmers enforce structured 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:

```c
static int a = 10;
static int b = 4;
static int x;

int main()
{
    if ( a < b )
        x = 1;
    else
        x = 0;

If-Then-Else with Conditional Execution

.data
a: .word 10 @ static int a=10;
b: .word 4 @ static int b=4;
x: .word 0 @ static int x;

.text
.globl main
main: ldr r0, =a @ load pointer to ‘a’
    ldr r1, =b @ load pointer to ‘b’
    ldr r0, [r0] @ load ‘a’
    ldr r1, [r1] @ load ‘b’
    cmp r0, r1 @ compare them
    movlt r0, #1 @ THEN section – load 1 into r0
    movge r0, #0 @ ELSE section – load 0 into r0
    ldr r1, =x @ load pointer to ‘x’
    str r0, [r1] @ store r0 in ‘x’
If-Then-Else with Branch Instructions

.data
a: .word 10 @ static int a=10;
b: .word 4  @ static int b=4;
x: .word 0  @ static int x;

.text
.globl main
main: ldr r0, =a @ load address of 'a'
     ldr r1, =b @ load address of 'b'
     ldr r0, [r0] @ load 'a'
     ldr r1, [r1] @ load 'b'
     cmp r0, r1 @ compare them
     bge else @ if a >= b then goto else_code
     mov r0, #1 @ THEN section - load 1 into r0
     b after @ skip the else section
else: mov r0, #0 @ ELSE section - load 0 into r0
after: ldr r1, =x @ load pointer to 'x'
     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

.data
str: .asciz "Hello World - %d\n"

.text
.globl main
main: @ We are going to use r4 and make a function call, so
    stmfd sp!,{r4,lr} @ push lr and r4 onto stack
    mov r4, #0 @ use r4 for i; i=0

loop: cmp r4, #10 @ perform comparison
    bge done @ end loop if i >= 10
    ldr r0, =str @ load pointer to format string
    mov r1, r4 @ copy i into r1
    bl printf @ printf("Hello World - %d\n",i);
    add r4, r4, #1 @ i++
    b loop @ repeat loop test

done: mov r0, #0 @ move return code into r0
    ldmfd sp!,{r4,lr} @ pop lr and r4 from stack
    mov pc, lr @ return from main
.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 loop.

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

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

.data
str: .asciz "Hello World - %d\n"

.text
.globl main

main:

@ We are going to use r4 and make a function call, so
stmfd sp!,{r4, lr} @ push lr and r4 onto stack

1dr r4, #0 @ use r4 for i; i=0

loop: 1dr r0, =str @ load pointer to format string

mov r1, r4 @ copy i into r1

bl printf @ printf("Hello World - %d\n",i);

add r4, r4, #1 @ i++

cmp r4, #10 @ perform comparison

blt loop @ end loop if i >= 10

mov r0, #0 @ move return code into r0

ldmfd sp!,{r4, lr} @ pop lr and r4 from stack

mov pc, lr @ return from main

.end @ tell assembler that we are done
Calling Standard C Library Functions

```
.data
str1: .asciz "\%d"
str2: .asciz "You entered %d\n"
n: .word 0

.text
.globl main
main: stmfd sp!,{lr} @ push link register onto stack
     ldr r0, =str1 @ load address of format string
     ldr r1, =n @ load address of int variable
     bl scanf @ call scanf("%d", &n)
     ldr r0, =str2 @ load address of format string
     ldr r1, =n @ load address of int variable
     ldr r1, [r1] @ load int variable
     bl printf @ call printf("You entered %d\n", n)
     mov r0, #0 @ load return value
     ldmfd sp!,{lr} @ pop link register from stack
     mov pc, lr @ return from main
```
# ARM Function Calling Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0 (a1)</td>
<td>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.</td>
</tr>
<tr>
<td>r1 (a2)</td>
<td></td>
</tr>
<tr>
<td>r2 (a3)</td>
<td></td>
</tr>
<tr>
<td>r3 (a4)</td>
<td></td>
</tr>
<tr>
<td>r4 (v1)</td>
<td>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.</td>
</tr>
<tr>
<td>r5 (v2)</td>
<td></td>
</tr>
<tr>
<td>r6 (v3)</td>
<td></td>
</tr>
<tr>
<td>r7 (v4)</td>
<td></td>
</tr>
<tr>
<td>r8 (v5)</td>
<td></td>
</tr>
<tr>
<td>r9 (v6)</td>
<td></td>
</tr>
<tr>
<td>r10 (v7)</td>
<td></td>
</tr>
<tr>
<td>r11 (fp) (v8)</td>
<td>Intra-procedure scratch register. May be modified.</td>
</tr>
<tr>
<td>r12 (ip)</td>
<td>Program stack pointer.</td>
</tr>
<tr>
<td>r13 (sp)</td>
<td>Link Register (return address). See <code>bl</code> instruction.</td>
</tr>
<tr>
<td>r14 (lr)</td>
<td>Program Counter. Changing this causes a branch.</td>
</tr>
<tr>
<td>r15 (pc)</td>
<td></td>
</tr>
</tbody>
</table>

CPSR
Passing One Argument

Passing a pointer to a string.

```
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.

```c
int printf("The results are: %d %d %d\n",i,j,k);
```
Passing More Than Four Arguments

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

```assembly
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 \( \text{ldr \ rd, [sp, #offset]} \),
- the *calling* function will remove parameters from the stack, if necessary,
- if the function return type is not \text{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

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

```
myfun:   add       r0,r0,r1   @ r0 = a + b
        add       r0,r0,r2   @ r0 = a + b + c
        add       r0,r0,r3   @ r0 = a + b + c + d
        ldr       r1,[sp,#0] @ load e from stack
        add       r0,r0,r1   @ r0 = a + b + c + d + e
        ldr       r1,[sp,#4] @ load f from stack
        add       r0,r0,r1   @ r0 = a + b + c + d + e + f
        mov       pc,lr      @ return from function
```
Automatic Variables

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

```c
int doit()
{
    int x[20];
    register int i; /* try to keep i in a register */
    for(i=0;i<20;i++) x[i] = i;
    return i;
}
```

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

```c
void reverse(char*a,int left, int right)
{
    char tmp;
    if(left<right)
    {
        tmp=a[left];
        a[left]=a[right];
        a[right]=tmp;
        reverse(a,left+1,right-1);
    }
}

int main()
{
    char *str="This is the string to reverse";
    printf(str);
    reverse(str,0,strlen(str)-1);
    printf(str);
    return 0;
}
```
Recursion in Assembly

reverse:

1. `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

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

```c
void reverse(char *left, char *right)
{
    char tmp;
    if(left<=right)
    {
        tmp=*left;
        *left=*right;
        *right=tmp;
        reverse(left+1,right-1);
    }
}

int main()
{
    char *str="This is the string to reverse";
    printf(str);
    reverse(str,str+strlen(str)-1);
    printf(str);
    return 0;
}
```
Using Pointers in Assembly

```assembly
reverse: cmp r0, r1 @ if(left>=right)
   bge exit @ then return
   stmfd sp!, {lr} @ I WILL call myself−save lr
   ldrb r3, [r0] @ load character at *left
   ldrb ip, [r1] @ load character at *right
   strb ip, [r0] @ store in *left
   strb r3, [r1] @ store in *right
   add r0, r0, #1 @ calculate left+1
   sub r1, r1, #1 @ calculate right−1
   bl reverse @ make recursive call
   ldmfd sp!, {lr} @ pop lr from the stack
   exit: mov pc, lr @ return from function
```
Arrays

```c
int x[100];
int i;

for(i=0;i<100;i++)
{x[i] = 0;
}
```

```
sub    sp, sp, #400    @ allocate 400 bytes in stack
mov    r0, #0         @ use r0 to hold the index
mov    r1, #0         @ value to initialize with
loop:  str    r1, [sp,r0,lsl #2] @ set array element to zero
cmp    r0, #100       @ loop test
add    r0, r0, #1    @ increment index
blt    loop          @ loop while index < 100
```
Using a C struct

```c
struct student {
    char first_name[30];
    char last_name[30];
    unsigned char class;
    int grade;
};

struct student newstudent; /* allocate on the stack */
strcpy(newstudent.first_name,"Sam");
strcpy(newstudent.last_name,"Smith");
newstudent.class = 2;
newstudent.grade = 88;
...```
Equivalent in Assembly

.data
.equ s_first_name, 0
.equ s_last_name, 30
.equ s_class, 60
.equ s_grade, 64
.equ s_size, 68

sam: .asciz "Sam"
smith: .asciz "Smith"
sub    sp, sp, #s_size                @ allocate struct on the stack
mov    r0, sp                        @ put pointer to struct in r0
add    r0, r0, #s_first_name        @ offset to first name field
ldr    r1, =sam                      @ load pointer to "Sam"
bl     strcpy                        @ copy the string
mov    r0, sp                        @ put pointer to struct in r0
add    r0, r0, #s_last_name         @ offset to last name field
ldr    r1, =smith                    @ load pointer to "Smith"
bl     strcpy                        @ copy the string
mov    r1, #2                        @ load constant value of 2
strb   r1, [r0, #s_class]           @ store with offset
mov    r1, #88                       @ load constant value of 88
str    r1, [r0, #s_grade]           @ store with offset