Subroutines

- Programs can be divided into blocks of instructions that each perform some specific task
  - generate a hamming code
  - find the length of a string
  - convert a string from UPPER CASE to lower case
  - play a sound

- Would like to avoid repeating the same set of operations throughout our programs
  - write the instructions to perform some specific task once
  - invoke the set of instructions many times to perform the same task
Example: upr\text{case}

- Write a pseudo-code program that converts two strings stored in memory to UPPER CASE

```c
address = address of first character of string1

while ((char = Memory.byte[address]) != NULL) {
    if (char ≥ ‘a’ AND char ≤ ‘z’) {
        char = char AND 0xFFFFFFDF
        Memory.byte[address] = char
    }
    address = address + 1
}

address = address of first character of string2

while ((char = Memory.byte[address]) != NULL) {
    if (char ≥ ‘a’ AND char ≤ ‘z’) {
        char = char AND 0xFFFFFFDF
        Memory.byte[address] = char
    }
    address = address + 1
}
```

Repetition
Repetition

- Repetition of identical code leads to problems ...
  - harder to follow and understand
  - prone to mistakes
  - difficult to modify (modifications need to be repeated)
  - wasteful of memory

- Package the repeated code into a subroutine
  - changes only need to be made in one location
  - shorter
  - easier to follow and understand
  - less prone to mistakes
Example: uprcase

- UPPER CASE subroutine (pseudo-code)

```c
uprcase (address) {
    while ((char = Memory.byte[address]) != NULL) {
        if (char ≥ ‘a’ AND char ≤ ‘z’) {
            char = char AND 0xFFFFFFDF
            Memory.byte[address] = char
        }
        address = address + 1
    }
}

address = address of first character of string1
uprcase (address)

address = address of first character of string2
uprcase (address)
```
Subroutines

- When designing a program, it should be structured into **logical blocks (subroutines)**, each of which corresponds to a **specific task**

- Each subroutine can be programmed, tested and debugged independently of other subroutines

- Subroutines ...
  - facilitate good design
  - facilitate reuse
  - can be invoked (executed) many times
  - can invoke other subroutines (and themselves!)
  - correspond to procedures/functions/methods in high-level languages
Subroutine implementation

- **Call** and **Return** mechanism
- Invoking *(calling)* a subroutine requires a deviation from the default sequential execution of instructions
  - Flow control again, like selection and iteration
- **Branch to the subroutine by modifying the Program Counter (PC) using a PC Modifying Instruction**
  - e.g. B, BEQ, BNE
  - When a program branches to a subroutine, the processor begins execution of the instructions that make up the subroutine
- When the subroutine has completed its task, the processor must be able to branch back *(return)* to the instruction immediately following the branch instruction that invoked the subroutine
Consider the flow of execution of the program to fill memory with a specified value

Branching to a subroutine: branch to the address (or label) of the first instruction in the subroutine (simple flow control)

Returning from a subroutine: must have remembered the address of the instruction immediately following the branch instruction that invoked the subroutine
Implementing subroutine call/return

- Calling the subroutine (the long way!)
  - Save address of instruction immediately following branch instruction (**return address**) in a register (e.g. R14)
  - Branch (B) to the subroutine

```
... ...
MOV R14, pc ; save return address
B sub1 ; branch to the subroutine (labelled sub1)
??? ???, ???, ??? ; some random instruction after the branch
...
```

- Returning from the subroutine
  - Copy (**MOV**) the return address from R14 into the Program Counter

```
... ...
??? ???, ???, ??? ; last subroutine instruction
MOV pc, R14 ; return to the calling program
```
Branch and Link Instruction – BL

- Saving the return address before branching to a subroutine is a common operation

```
... ... ;
MOV R14, pc ; save return address
B sub1 ; branch to the subroutine (labelled sub1)
??? ???, ???, ??? ; some random instruction after the branch
... ... ;
```

- The Branch and Link instruction (BL) is provided to perform the same task in a single instruction

```
... ... ;
BL sub1 ; branch to the sub1, saving return address
??? ???, ???, ??? ; some random instruction after the branch
... ... ;
```

- BL always saves the return address (PC – 4) in R14, called the link register (LR)
Returning from a subroutine

- Having called the subroutine using BL, we can return in the same way as before ...

  ```
  ... ... 
  ??? ???, ???, ??? ; last subroutine instruction
  MOV pc, lr ; return to the calling program
  ```

- Note use of lr as a synonym for R14

- Above return method works but assembler will produce a warning and recommend the following ...

  ```
  ... ... 
  ??? ???, ???, ??? ; last subroutine instruction
  BX lr ; return to the calling program
  ```

- Branch and eXchange (BX) loads the contents of the link register (lr) into the program counter
Implementing the UPPER CASE subroutine

- Top level program

Branch and Link instruction saves address of the next instruction before branching to the instruction referred to by label uppercase

```
start
    LDR r1, =str1            ; load address of first string
    BL uprcase              ; invoke uprcase subroutine
    LDR r1, =str2           ; load address of second string
    BL uprcase              ; invoke uprcase subroutine
stop  B       stop

; Define strings to test program

AREA  TestData, DATA, READWRITE
str1  DCB  "motor",0; NULL terminated test string
str2  DCB  "zero",0 ; NULL terminated test string
```
Implementing UPPER CASE subroutine

- Subroutine

```assembly
B testwh1 ; while ( (char = Memory.byte[address]) != 0 ) {
  wh1 CMP r0, #'a' ; if (char >= 'a')
  BCC endif1 ; AND
  CMP r0, #'z' ; char <= 'z')
  BHI endif1 ; {
  BIC r0, #0x00000020 ; char = char AND NOT 0x00000020
  STRB r0, [r1, #-1] ; Memory.byte[address - 1] = char
  endif1 ; }
  testwh1 LDRB r0, [r1], #1 ; }
  CMP r0, #0 ;
  BNE wh1 ;
  BX lr ; return
```

Label for 1st instruction in subroutine

Branch and eXchange instruction causes execution to continue at the instruction immediately following the instruction that branched to the subroutine.
Consider the following program

```assembly
; Top level program
start
    BL sub1 ; call sub1
stop    B    stop

; sub1 subroutine
sub1
    BL sub2 ; call sub2
    BX lr   ; return from sub1

; sub2 subroutine
sub2
    BX lr   ; return from sub2
```

- Top level program calls sub1, which in turn calls sub2
- What happens when we execute this program?
### Saving the link register

#### Solution

- Save the contents of the link register before a subroutine invokes a further nested subroutine.
- Restore the contents of the link register when the nested subroutine returns.
- Where should we save the contents of the link register?

#### Revised sub1 from the previous example ...

```assembly
; sub1 subroutine
sub1

STMFD sp!, {lr} ; save link register
BL sub2 ; call sub2
LDMFD sp!, {lr} ; restore link register
BX lr ; return from sub1
```

- Using this approach we can call as many nested subroutines as we need (almost ...)

Saving the link register

- A more general and efficient solution
  - Save the contents of the link register on the system stack at the start of every subroutine
  - Restore the contents of the link register immediately before returning at the end of every subroutine

```assembly
; subx subroutine
subx
  STMFD   sp!, {lr} ; save link register
  ...  ...  
  ...  ...
  LDMFD   sp!, {lr} ; restore link register
  BX      lr  ; return from sub1
```

- More efficiently, we could restore the saved $lr$ to the $pc$, avoiding the need for the $BX$ instruction (preferred)

```assembly
...  ...
  LDMFD  sp!, {pc} ; restore link register
```
Multiple return points

- Single or multiple return points (e.g. BX lr) in a single subroutine?
- Good programming practice to have exactly one return point from every subroutine
  - i.e. only one BX lr instruction
  - return point should be at the end of the subroutine
- Your program will assemble and run with more than one return point or with return points in places other than the end of a subroutine ...
  - .. but is this desirable?
Consider the following program which converts a string to UPPER CASE before making a copy of it in memory

```
start
  LDR  r0, =deststr  ; ptr1 = address of deststr
  LDR  r1, =teststr  ; ptr2 = address of teststr
  BL   uprcase       ; uprcase(ptr2)

  LDRB r2, [r1], #1  ; ch = Memory.Byte[ptr1++]
  STRB r2, [r0], #1  ; Memory.Byte[ptr2++] = ch
  CMP  r2, #0        ; } while (ch != NULL);
  BNE  do1

stop  B   stop

... ... 

teststr  DCB  "xerox",0
deststr  SPACE  256

END
```
Unintended side effects

- ... implementation of uprcase subroutine (as before)

```
; UPPER CASE subroutine
uprcase
STMFD sp!, {lr}
B    testwh2    ; while ( (char = Memory.byte[address++])
          != 0) {

wh2   CMP    r0, #'a'    ; if (char >= 'a'
BCC   endif1    ;    AND
CMP    r0, #'z'    ;    char <= 'z')
BHI   endif1    ;    {
BIC    r0, #0x00000020    ;    char = char AND NOT 0x00000020
STRB   r0, [r1, #1]    ;    Memory.byte[address - 1] = char
endif1 }
endif1
testwh2 LDRB    r0, [r1], #1    ;    }
CMP    r0, #0    ;
BNE   wh2    ;
LDMFD sp!, {pc}    ; return
```

- Why won’t this program work when uprcase is used?
Unintended side effects

- When designing and writing subroutines, clearly and precisely define what effect the subroutine has.
- Effects outside this definition should be considered unintended and should be hidden by the subroutine.
- In the previous example, the calling top level program should not be affected by modifications to r0 and r1 made by the `uppercase` subroutine.
- In general, subroutines should save the contents of the registers they use at the start of the subroutine and should restore the saved contents before returning.
- **Save register contents on the system stack**
Avoiding unintended side effects

- Example: modified uprcase subroutine

```assembly
; UPPER CASE subroutine
uprcase

STMFD sp!, {r0-r1,lr}
B testwh2 ; while ( (char = Memory.byte[address++]) != 0) {

wh2
    CMP r0, #'a' ; if (char >= 'a'
    BCC endif1 ; AND
    CMP r0, #'z' ; char <= 'z')
    BHI endif1 ;
    BIC r0, #0x00000020 ; char = char AND NOT 0x00000020
    STRB r0, [r1, #-1] ; Memory.byte[address - 1] = char
    endif1

testwh2
    LDRB r0, [r1], #1 ; }
    CMP r0, #0 ; }
    BNE wh2 ; return
    LDMFD sp!, {r0-r1,pc} ; return
```

- Any registers used are saved on the stack (along with the link register) at the start of the subroutine
- These are restored before returning
Passing parameters

- Information must be passed to and from a subroutine using a fixed and well defined interface, known to both the subroutine and calling programs.

- `toupper` subroutine had single address parameter.

```c
toupper (address)
{
    ...
}

address = address of first character of string1
toupper (address)
...
```

- Simplest way to pass parameters to/from a subroutine is to use well defined registers, e.g. for `toupper`:
  - `address ↔ r1`
Example: fillmem

- Design and write an ARM Assembly Language subroutine that fills a sequence of words in memory with the same 32-bit value

- Pseudo-code solution

```c
fillmem (address, length, value) {
    count = 0;
    while (count < length) {
        Memory.Word[address] = value;
        address = address + 4;
        count = count + 1;
    }
}
```

- 3 parameters
  - address – start address in memory
  - length – number of words to store
  - value – value to store
Example: fillmem Version 1

- First version

```
; fillmem subroutine
; Fills a contiguous sequence of words in memory with the same value
; parameters  r0: address - address of first word to be filled
;            r1: length - number of words to be filled
;            r2: value - value to store in each word
fillmem
  STMFD  sp!, {r0-r2,r4,lr} ; save registers
  MOV r4, #0 ; count = 0;
wh1
  CMP r4, r1 ; while (count < length)
  BHS endwh1 ; {
  STR r2, [r0, r4, LSL #2] ; Memory.Word[address] = value;
  ADD r4, #1 ; count = count + 1;
  B   wh1 ; }
endwh1
  LDMFD sp!, {r0-r2,r4,pc} ; restore registers
```

- Could be more efficient ...
**Example: fillmem Version 2**

- **Second version**

```asm
; fillmem subroutine
; Fills a contiguous sequence of words in memory with the same value
; parameters
; r0: address - address of first word to be filled
; r1: length - number of words to be filled
; r2: value - value to store in each word

fillmem
STMFD sp!, {r0-r1,lr} ; save registers
CMP r1, #0 ; while (length != 0)
B testwh1 ; {
wh1 STR r2, [r0], #4 ; Memory.Word[address] = value;
; address = address + 4;
SUBS r1, #1 ; length = length - 1;
testwh1 BNE wh1 ; }
LDMFD sp!, {r0-r1,pc} ; restore registers
```
Parameters

- In high level languages, the interface is defined by the programmer and the compiler enforces it.

- In assembly language, the interface must be both defined and enforced by the programmer.

- Parameter types
  - **Variable parameters:** Changes made to the parameter by the subroutine *should* be visible to the caller.
  - **Value parameters:** The subroutine *must not* change the value of the parameter (or, any changes must not be visible outside the subroutine).
Example: count1s

- Design and write an ARM Assembly Language subroutine that counts the number of set bits in a word

```assembly
; count1s subroutine
; Counts the number of set bits in a word
; parameters
;   r0: count (var) - count of set bits
;   r1: wordval (val) - word in which 1s will be counted

count1s

STMFD sp!, {r1,lr} ; save registers
MOV r0, #0 ; count = 0;

wh1

CMP r1, #0 ; while (wordval != 0)
BEQ endwh1 ; {
MOVS r1, r1, LSR #1 ; wordval = wordval >> 1; (update carry)
ADC r0, r0, #0 ; count = count + 0 + carry;
B wh1 ; }

endwh1

LDMFD sp!, {r1,pc} ; restore registers
```
Example: count1s

- **count** Parameter
  - *variable* parameter
  - used to return the count of 1s to the calling program
  - changes made by the subroutine should be visible to the calling program
  - should not be saved (and restored) on the stack

- **wordval** parameter
  - *value* parameter
  - used to pass the word in which 1s are to be counted to the count1s subroutine
  - should be saved / restored on the stack at the start / end of the subroutine to hide any modifications
Parameters

- It is good programming practice to save ...
  - any value parameters
  - any registers used internally by the subroutine
  - (and the link register!)

... on the system stack at the start of a subroutine and restore them before returning to the calling program

- Avoids unexpected side-effects

- Also remember: a subroutine should pop off everything that it pushed onto the stack
  - Not doing this is like to cause errors that may be difficult to correct
### Passing parameters by reference

- Often parameters passed to/from a subroutine are too large to be stored in registers
  - e.g. 128-bit integer, ASCII string, image, list of integers
- **Solution**: the calling program ...
  - stores the parameter in memory
  - uses a register to pass a pointer to the parameter to the subroutine (an address)

### Example
- Design and write an ARM Assembly Language subroutine that will add two 128-bit integers
  - Require 4 words for each operand and 4 words for the result
Example: add128

- Begin with a program and data to test the subroutine ...

```assembly
start
LDR r1, =val1 ; load 1st 128bit value
LDR r2, =val2 ; load 2nd 128bit value
LDR r0, =result ; load address for 128bit result

BL add128

stop
B stop

; ... ... ...
; <the subroutine will go here>
; ... ... ...

AREA TestData, DATA, READWRITE

val1 DCD 0x57FD30C2,0x387156F3,0xFE4D6750,0x037CB1A0
val2 DCD 0x02BA862D,0x298B3AD4,0x213CF1D2,0xFD00357C
result SPACE 16
```

- ... design and write the subroutine ...
Example: add128

; add128 subroutine
; Adds two 128-bit integers
; Parameters
; r0: pResult (val) - result
; r1: pVal1 (val) - first integer
; r2: pVal2 (val) - second integer
add128

STMFD sp!, {r0-r2,r5-r7,lr} ; save registers
LDR r5, [r1], #4 ; tmp1 = Memory.Word[pVal1]; pVal1 = pVal1 + 4;
LDR r6, [r2], #4 ; tmp2 = Memory.Word[pVal2]; pVal2 = pVal2 + 4;
ADDS r7, r5, r6 ; tmpResult = tmp1 + tmp2; (update C flag)
STR r7, [r0], #4 ; Memory.Word[pResult]; pResult = pResult + 4;

LDR r5, [r1], #4 ; tmp1 = Memory.Word[pVal1]; pVal1 = pVal1 + 4;
LDR r6, [r2], #4 ; tmp2 = Memory.Word[pVal2]; pVal2 = pVal2 + 4;
ADCS r7, r5, r6 ; tmpResult = tmp1 + tmp2; (update C flag)
STR r7, [r0], #4 ; Memory.Word[pResult]; pResult = pResult + 4;

LDR r5, [r1], #4 ; tmp1 = Memory.Word[pVal1]; pVal1 = pVal1 + 4;
LDR r6, [r2], #4 ; tmp2 = Memory.Word[pVal2]; pVal2 = pVal2 + 4;
ADCS r7, r5, r6 ; tmpResult = tmp1 + tmp2; (update C flag)
STR r7, [r0], #4 ; Memory.Word[pResult]; pResult = pResult + 4;

LDR r5, [r1], #4 ; tmp1 = Memory.Word[pVal1]; pVal1 = pVal1 + 4;
LDR r6, [r2], #4 ; tmp2 = Memory.Word[pVal2]; pVal2 = pVal2 + 4;
ADCS r7, r5, r6 ; tmpResult = tmp1 + tmp2; (update C flag)
STR r7, [r0], #4 ; Memory.Word[pResult]; pResult = pResult + 4;

LDMFD sp!, {r0-r2,r5-r7,pc} ; restore registers
Although the subroutine was modifying the 12-bit result, the pResult parameter passed to it (in r0) should be treated as a value parameter.

Remember, modifications to value parameters should be hidden from code external to subroutine.

The four repeated blocks of code could be replaced with a single block in a loop:
- Left as an exercise
- Iterate over each of the 4 words in the 128-bit value
- Pay attention to propagation of C flag from each iteration to the next (a CMP will overwrite any C-out from ADD/ADC)
Passing parameters on the stack

- If there are insufficient registers to pass parameters to a subroutine, the system stack can be used in its place
  - Commonly used by high-level languages
  - Similar to passing parameters by reference but using the stack pointer instead of a dedicated pointer

- General approach
  - Calling program pushes parameters onto the stack
  - Subroutine accesses parameters on the stack, relative to the stack pointer
  - Calling program pops parameters off the stack after the subroutine has returned
Example: fillmem

- Re-write the `fillmem` subroutine to pass parameters on the stack (instead of registers)

- Pseudo-code reminder

```plaintext
fillmem (address, length, value)
{
    count = 0;
    while (count < length)
    {
        Memory.Word[address] = value;
        address = address + 4;
        count = count + 1;
    }
}
```
First, write a program to test the subroutine:

```assembly
start
    LDR r0, =tstarea ; Load address to be filled
    LDR r1, =32      ; Load number of words to be filled
    LDR r2, =0xC0C0C0C0 ; Load value to fill
    STR r0, [sp, #-4]! ; Push address parameter on stack
    STR r1, [sp, #-4]! ; Push length parameter on stack
    STR r2, [sp, #-4]! ; Push value parameter on stack
    BL fillmem       ; Call fillmem subroutine
    ADD sp, sp, #12   ; Efficiently pop parameters off stack

stop  B  stop

AREA   Test, DATA, READWRITE

tstarea SPACE 256

END
```
Example: fillmem Version 3

- fillmem subroutine with stack parameters

```assembly
; fillmem subroutine
; Fills a contiguous sequence of words in memory with the same value
; parameters
;   [sp+0]: value - value to store in each word
;   [sp+4]: length - number of words to be filled
;   [sp+8]: address - address of first word to be filled
fillmem
    STMFD sp!, {r0-r2,r4,lr} ; save registers
LDR     r0, [sp, #8+20]   ; load address parameter
LDR     r1, [sp, #4+20]   ; load length parameter
LDR     r2, [sp, #0+20]   ; load value parameter
    MOV     r4, #0           ; count = 0;
wh1
    CMP     r4, r1           ; while (count < length)
    BHS     endwh1           ; {
    STR     r2, [r0, r4, LSL #2] ; Memory.Word[address] = value;
    ADD     r4, #1           ; count = count + 1;
    B       wh1              ; }
endwh1
    LDMFD sp!, {r0-r2,r4,pc} ; restore registers
```
Could push the parameters onto the stack more efficiently with a single STMFD instruction
  • But we’re being explicit – subroutines will usually specify order for operands on stack

Important that calling program restores the system stack to its original state
  • Pop off the three parameters
  • Quickly and simply done by adding 12 to sp.

Subroutine doesn’t pop parameters off the stack
  • Accesses them in-place, using offsets relative to the stack pointer.

Subroutine saves some registers to the stack
  • compensate by adding addition offset (+20) to parameter offsets

Example: fillmem
Example: fillmem

- **Stack state**

Before pushing parameters:
- 0xA1001024
- 0xA1001020
- 0xA100101C
- 0xA1001018
- 0xA1001014
- 0xA1001010
- 0xA100100C
- 0xA1001008
- 0xA1001004
- 0xA1001000
- 0xA100FFFC
- 0xA100FF8

Before calling subroutine:
- 0xA1001024
- 0xA1001020
- 0xA100101C
- 0xA1001018
- 0xA1001014
- 0xA1001010
- 0xA100100C
- 0xA1001008
- 0xA1001004
- 0xA1001000
- 0xA100FFFC
- 0xA100FF8

Subroutine after saving registers:
- 0xA1001024
- 0xA1001020
- 0xA100101C
- 0xA1001018
- 0xA1001014
- 0xA1001010
- 0xA100100C
- 0xA1001008
- 0xA1001004
- 0xA1001000
- 0xA100FFFC
- 0xA100FF8
What happens the fillmem example if we change the list of registers that we save? (Or worse manipulate the stack during the execution of the subroutine)

```
fillmem
STMFD sp!, {r0-r4,lr} ; save registers
LDR r0, [sp, #8+20] ; load address parameter
LDR r1, [sp, #4+20] ; load length parameter
LDR r2, [sp, #0+20] ; load value parameter
MOV r4, #0 ; count = 0;
CMP r4, r1 ; while (count < length)
BHS endwh1 ; {
STR r2, [r0, r4, LSL #2] ; Memory.Word[address] = value;
ADD r4, #1 ; count = count + 1;
B wh1 ; }
endwh1
LDMFD sp!, {r0-r4,pc} ; restore registers
```

Offsets to parameters on the stack will change at design time (and perhaps at runtime)
Workaround – at start of subroutine

- Save contents of a “scratch” register and lr
- Copy sp + 8 to “scratch” register
- Continue to push data onto the stack as required
- Access parameters relative to “scratch” register

Example: fillmem Version 4

```
STMFD sp!, {r12, lr} ; save r12, lr
ADD r12, sp, #8 ; scratch = sp + 8
STMFD sp!, {r0-r4} ; save registers

LDR r0, [r12, #8] ; load address parameter
LDR r1, [r12, #4] ; load length parameter
LDR r2, [r12, #0] ; load value parameter

<remainder of subroutine as before>

LDMFD sp!, {r0-r4} ; restore registers
LDMFD sp!, {r12, pc} ; restore r12, pc
```
Example: `fillmem` Version 4

- **Stack state**

Before pushing parameters:
- 0xA1001024
- 0xA1001020
- 0xA100101C
- 0xA1001018
- 0xA1001014
- 0xA1001010
- 0xA100100C
- 0xA1001008
- 0xA1001004
- 0xA1001000
- 0xA100FFC
- 0xA100FF8

Before calling subroutine:
- 0xA1001024
- 0xA1001020
- 0xA100101C
- 0xA1001018
- 0xA1001014
- 0xA1001010
- 0xA100100C
- 0xA1001008
- 0xA1001004
- 0xA1001000
- 0xA100FFC
- 0xA100FF8

Subroutine after saving registers:
- 0xA1001024
- 0xA1001020
- address
- length
- value
- saved r12
- saved lr
- saved r4
- saved r2
- saved r1
- saved r0
- ??
ARM APCS

- AAPCS – ARM Application Procedure Call Standard
  [Link](http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf)

- Writing subroutines that adhere to this standard allows subroutines to be separately written and assembled

- Contract between subroutine callers and callees

- Standard specifies
  - how parameters must be passed to subroutines
  - which registers must have their contents preserved across subroutine invocations (and which are corruptible)
  - special roles for certain registers
  - a Full Descending stack pointed to by R13 (sp)
  - etc.
## Simplified AAPCS register specification

<table>
<thead>
<tr>
<th>Register</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>Parameters to and results from subroutine</td>
</tr>
<tr>
<td>r1</td>
<td>Otherwise may be corrupted</td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td>Variables</td>
</tr>
<tr>
<td>r8</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>r9</td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td>Scratch register (corruptible)</td>
</tr>
<tr>
<td>r13</td>
<td>Stack Pointer (SP)</td>
</tr>
<tr>
<td>r14</td>
<td>Link Register (LR)</td>
</tr>
<tr>
<td>r15</td>
<td>Program Counter (PC)</td>
</tr>
</tbody>
</table>
Subroutines can invoke themselves – recursion

Example: Design, write and test a subroutine to compute $x^n$

$$x^n = \begin{cases} 
1 & \text{if } n = 0 \\
x & \text{if } n = 1 \\
\left(x^2\right)^{n/2} & \text{if } n \text{ is even} \\
x.(x^2)^{n/2} & \text{if } n \text{ is odd}
\end{cases}$$
Example: power

- Pseudo-code solution

```c
power (x, n) {
    if (n == 0) {
        result = 1;
    }
    else if (n == 1) {
        result = x;
    }
    else if (n & 1 == 0) {
        result = power (x . x, n >> 1);
    }
    else {
        result = x . power (x . x, (n - 1) >> 1);
    }
}
```
Example: power

```
; power subroutine
; Compute x^n
; Parameters:  
; r0: result (variable) - x^n
; r1: x (value) - x
; r2: n (value) - n

power
  STMFD sp!, {r1-r2,r4,lr} ; save registers
  CMP   r2, #0            ; if (n == 0)
  BNE   else11           ; }
  MOV   r0, #1           ;  result = 1;
  B     endif1           ; }

else11
  CMP   r2, #1           ; else if (n == 1)
  BNE   else12          ; }
  MOV   r0, r1          ;  result = x;
  B     endif1          ; }

else12
  TST   r2, #1           ; else if (n & 1 == 0)
  BNE   else13          ; }
  MOV   r4, r1          ;  tmpx = x;
  MUL   r1, r4, r1      ;  x = tmpx * x;
  MOV   r2, r2, LSR #1  ;  n = n / 2;
  BL    power           ;  result = power (x, n);
  B     endif1        ; }
```

... continued ...
Example: power

... continued ...

```assembly
else13
    MOV r4, r1
    MUL r1, r4, r1
    SUB r2, r2, #1
    MOV r2, r2, LSR #1
    BL power
    MUL r0, r4, r0
endif1

    ; else {
    ;    tmpx = x;
    ;    x = x * tmpx;
    ;    n = n - 1;
    ;    n = n / 2;
    ;    result = power (x, n);
    ;    result = tmpx * result;
    ; }

    LDMFD sp!, {r1-r2,r4,pc}; restore registers
```

... continued ...