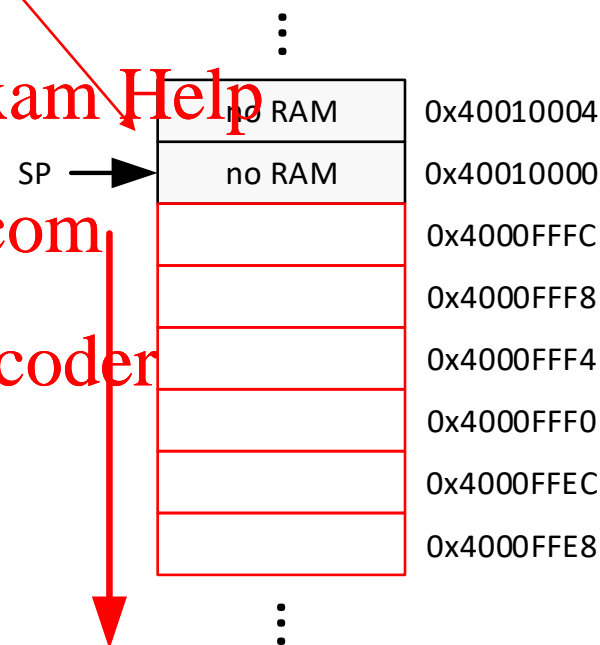## System Stack

- area of RAM used as a stack

- item(s) can be pushed onto stack

- item(s) can be popped from stack

- with ARM, item(s) means register(s)

- SP (stack pointer = R13) points to last item pushed on stack and the stack grows down in memory (RAM)

- SP initialised to 0x40010000

- why 0x40010000?

- because hardware has RAM from 0x40000000 to 0x4000FFFF

- SP initially points to word beyond top of RAM (i.e. stack is empty)

top of EMPTY stack

no RAM	0x40010004
no RAM	0x40010000
	0x4000FFFC
	0x4000FFF8
	0x4000FFF4
	0x4000FFF0
	0x4000FFEC
	0x4000FFE8

SP →

stack grows down in memory

System Stack…

- consider the stack after the following instructions
  are executed

  LDR   R1, =0x11111111
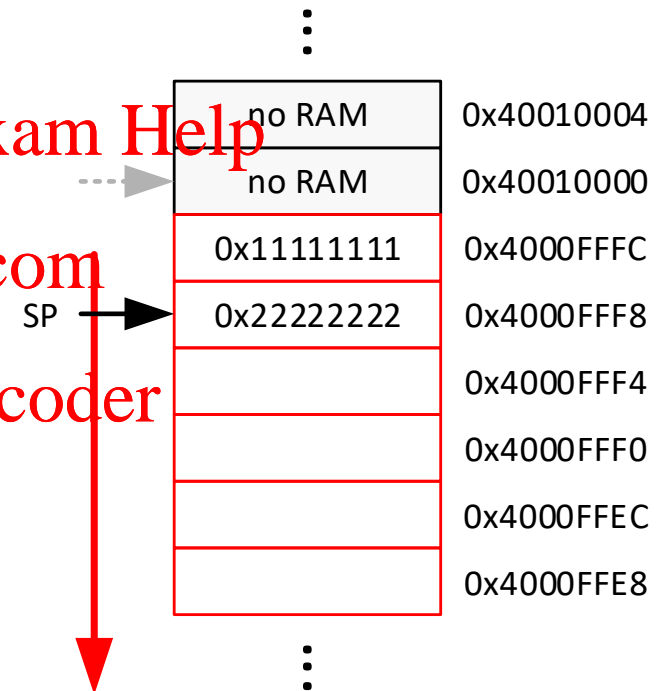
  LDR   R2, =0x22222222

  PUSH   {R1}

  PUSH   {R2}

- PUSH pre-decrements SP by 4 and saves register
  on stack at address in SP

- SP = 0x4000FFF8 (decremented by 8 as 8 bytes
  have been pushed on to the stack)

- SP (top of stack) -> pushed R2 (0x22222222)

no RAM	0x40010004
no RAM	0x40010000
0x11111111	0x4000FFFC
0x22222222	0x4000FFF8
	0x4000FFF4
	0x4000FFF0
	0x4000FFEC
	0x4000FFE8

SP →

stack grows down
in memory

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

System Stack…

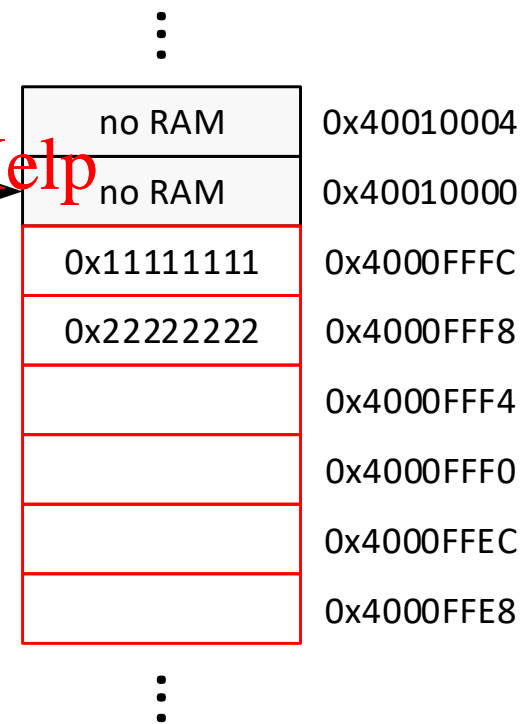- now consider the stack after the following instructions are executed

POP   {R1}

POP   {R2}

- POP reads item from address in SP into specified register and then increments SP by 4

R1 = 0x22222222

R2 = 0x11111111

- have used stack to swap contents of R1 and R2

- SP = 0x40010000 (incremented by 8 as 8 bytes have been popped from stack)

- stack is a LIFO - last in, first out data structure

no RAM	0x40010004
SP → no RAM	0x40010000
0x11111111	0x4000FFFC
0x22222222	0x4000FFF8
	0x4000FFF4
	0x4000FFF0
	0x4000FFEC
	0x4000FFE8

stack grows down in memory

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

System Stack…

- ARM instruction set allows stacks to grow up or down in memory <u>and</u> for the SP to point to the first free location or last item pushed on stack

- STMxx instructions (store multiple) used to push a list of registers onto stack

- LDMxx instructions (load multiple) used to pop a list of registers from stack

  D = decrement B = before

- much easier, **FOR OUR PURPOSES**, to use the PUSH and POP pseudonyms for STMDB and LDMIA respectively

  I = increment A = after

- example PUSH and POP instructions

```
PUSH    {R3, R4, R5, R12}          ; push R12, R5, R4 and R3
PUSH    {R0-R15}                   ; push ALL registers  R15, R14, R13 … R1 and R0
POP     {R3-R5, R12}               ; pop R3, R4, R5 and R12
```

System Stack…

* in what order are the registers pushed? and popped?

* registers pushed/popped with "*highest register number at the highest address*"

* with a stack that grows down in memory…

```
PUSH    {R4-R12}    ; registers pushed in order R12, R11, R10, … R4
                    ; R12 will be at the highest address


POP     {R4-R12}    ; registers popped in order R4, R5, R6, … R12
                    ; R12 will be at the highest address
```

* if using a stack, remember to initialise the SP

```
LDR     SP, =0x40010000         ; for CS1021 Keil uVision configuration
```

System Stack...

- note that the LDR and STR instructions can be used to push and pop a single register to and from the stack

D = decrement B = before

```
STR      R5, [SP, #-4]!     ; push R5 (pre-decrement by 4)      DB
STR      R4, [SP, #-4]!     ; push R4 (pre-decrement by 4)      DB
LDR      R4, [SP], #4       ; pop R4 (post-increment by 4)      IA
LDR      R5, [SP], #4       ; pop R5 (post-increment by 4)      IA
```

I = increment A = after

- code above equivalent to using the following PUSH and POP instructions

```
PUSH     {R4, R5}           ; push R5 and R4
POP      {R4, R5}           ; pop R4 and R5
```

Subroutines

- a **subroutine** is a sequence of instructions that performs a particular task

- subroutine called wherever task needs to be performed

    - divide Assignment Project Exam Help

    - find the length of a NUL terminated string

    - compute x^y https://powcoder.com

    - decrypt an email

    - … Add WeChat powcoder

- subdivide a program into many "short" subroutines

- write subroutines so they can be called with different parameters

- breaking a large program into many subroutines will reduce development and maintenance costs and improve code quality and reliability

Subroutines...

- facilitates good program design

- facilitates code reuse

- can be called ("executed", "invoked") whenever needed

- can be called with different parameters

- can call other subroutines (and themselves recursively)

- correspond to procedures/functions/methods in high-level languages

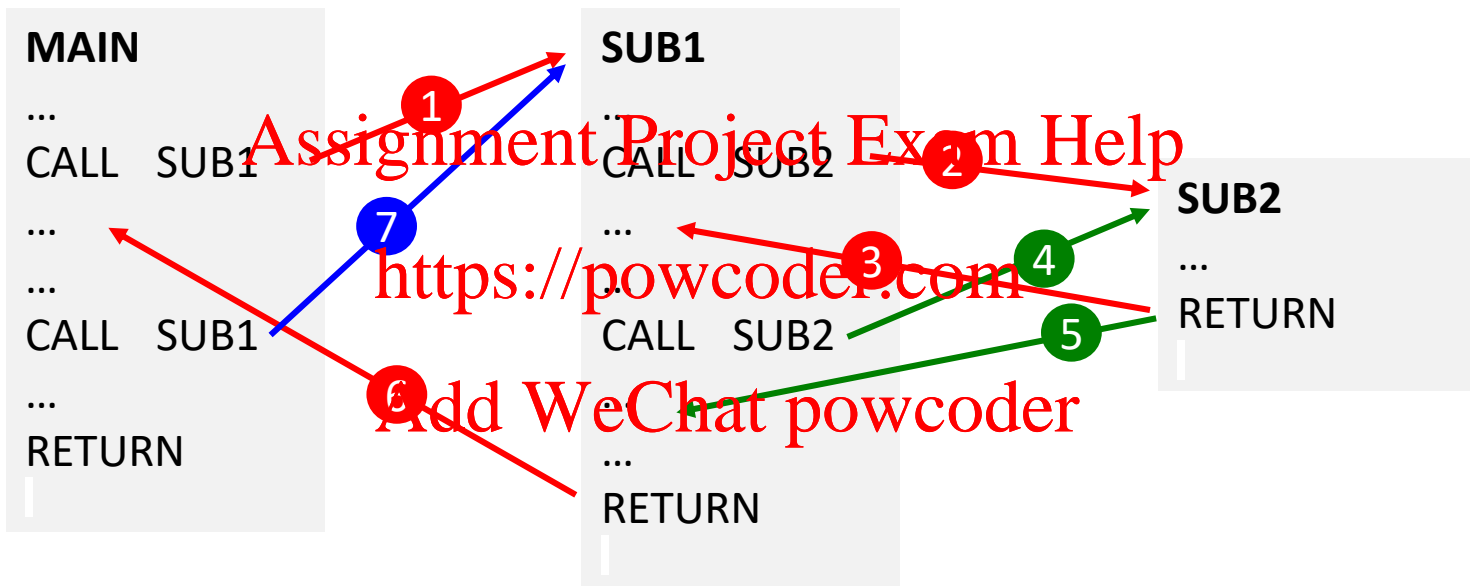- each subroutine can be programmed, tested and debugged independently

Subroutine call and return mechanism

MAIN
...
CALL SUB1
...
...
CALL SUB1
...
RETURN

SUB1
...
CALL SUB2
...
...
CALL SUB2
...
RETURN

SUB2
...
RETURN

- MAIN calls SUB1 (1), SUB1 calls SUB2 (2), SUB2 returns (3), SUB1 calls SUB2 again (4), SUB2 returns(5), SUB1 returns (6), MAIN calls SUB1 again (7), and so on

- RETURN returns to execute the instruction immediately following the call

ARM call and return mechanism

- to call a subroutine use the BL (branch and link) instruction

- saves return address in link register (LR = R14)

0x00000400   BL   SUB1   ; LR = 0x00000404 (return address)

0x00000404   ...   ;

return address = address of next instruction

- to return from a subroutine use BX (branch and exchange) specifying the link register (LR)

       BX   LR   ; PC (program counter) = LR

- works for **leaf** subroutines (subroutines which do NOT call other subroutines), but if a subroutine calls another subroutine the return address saved in the link register will be overwritten

- need to save and restore return address(es) on a stack

Using the stack to save and restore return addresses

- at the start of every **non leaf** subroutine, push the contents of LR (link register), which contains the return address, onto the system stack

- return from a **non leaf** subroutine by popping return address from stack and assigning to the PC (program counter)

- both steps accomplished easily using PUSH and POP instructions

```
;
; non leaf subroutine
;
SUB1  PUSH   {LR}     ; push link register onto stack
      ...
      ...
      POP    {PC}     ; return by popping saved return address into PC
```

ARM Procedure Calling Convention

- **ARM Architecture Procedure Call Standard (AAPCS)** is a technical document that describes the procedure calling convention that should be followed by high-level language compilers and writers of assembly language subroutines

- simplified version (for CS1021)

- first four subroutine parameters passed in R0, R1, R2 and R3 (respectively)

- result returned in R0

- R0, R1, R2, R3 are considered volatile (subroutines can change/modify these registers)

- R4, R5, R6, R7, R8, R9, R10, R11, R12 are considered non volatile (subroutines **must return** these registers unchanged/unmodified)

- from a caller's perspective

  - R4 - R12 will be unchanged/unmodified by subroutine call

  - MUST ASSUME R0 - R3 will be changed/modified by subroutine call

Subroutine entry and exit

- already seen how stack can be used to save and restore return addresses

- can also use stack to save and restore any of the registers R4 to R12 that the subroutine modifies so that they are returned to the caller unmodified

- again easily accomplished using PUSH and POP instructions at subroutine entry and exit

- assume that the code for the subroutine modifies R5, R6 and R7

```
SUB1  PUSH    {R5, R6, R7, LR}  ; push return address (LR), R7, R6 and R5
      ...                       ; subroutine body…
      ...                       ; modifies R5, R6 and R7
      POP     {R5, R6, R7, PC}  ; pop R5, R6, R7 and return
```
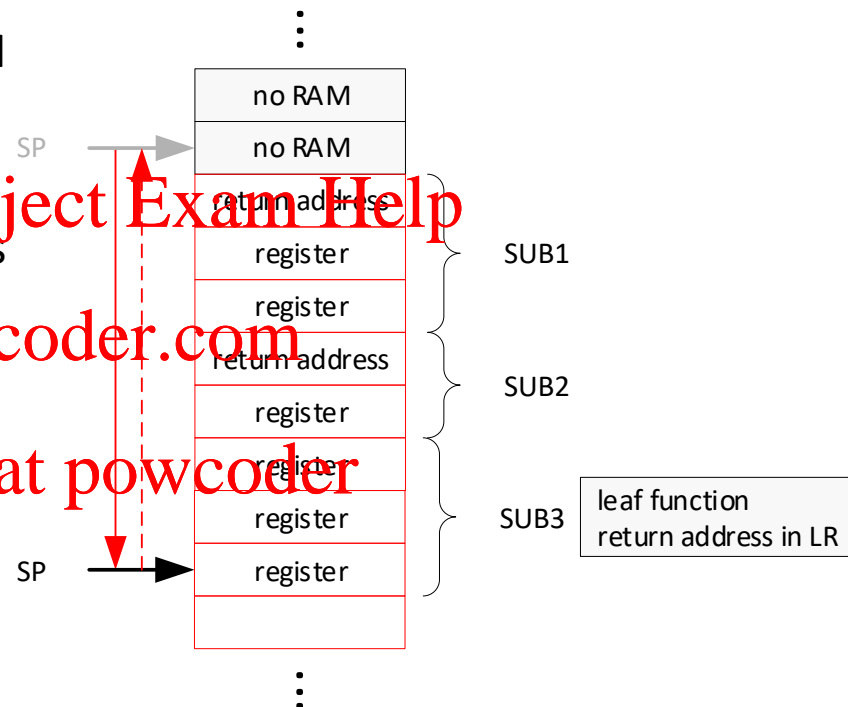
- important that each subroutine pushes and pops the same number of registers at entry and exit otherwise the stack can become corrupted

Subroutine Stack Frames

- subroutine stack frames pushed on and popped from stack

- for non-leaf subroutines return address and saved registers pushed

- for leaf subroutines, no need to push return address (in LR)

⋮

no RAM
no RAM
return address
register
register
return address
register
register
register
register

SP (top)

SP (bottom)

SUB1

SUB2

SUB3

leaf function return address in LR

⋮

subroutine stack frames for
SUB1, SUB2 and SUB3

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Example 1: UPPER (convert ASCII character to UPPER case)

```
;
; at entry:   R0 = ch
; at exit:    R0 = UPPERCASE(ch)
;
; leaf function
;
UPPER      CMP  R0, #'a'          ; ch < 'a' ?
           BLO   UPPER1           ; nothing to do
           CMP  R0, #'z'          ; ch > 'z' ?
           BHI    UPPER1          ; nothing to do
           SUB   R0, R0, #0x20    ; ch = ch - 0x20
UPPER1     BX     LR              ; return
```

Example 2: STRUPR (convert string to upper case using UPPER)

```
;
; at entry:      R0 -> NUL terminated string
; at exit:       R0 -> string converted to UPPER case (in situ)
;
; non leaf function
; MUST ASSUME that calls to UPPER will change R0, R1, R2 and R3
; need to return from STRUPR with R0 unchanged
;
STRUPR     PUSH      {R0, R4, LR}       ; push R0, R4 and return address
           MOV       R4, R0             ; make a copy of R0
STRUPR0    LDRB      R0, [R4]           ; get ch
           BL        UPPER              ; convert ch in R0 to UPPER case
           STRB      R0, [r4], #1       ; store ch AND R4 = R4 + 1
           CMP       R0, #0             ; ch == 0 ?
           BNE       STRUPR0            ; next ch
           POP       {R0, R4, PC}       ; pop R0, R4 and return
```

Example 3: UDIV (unsigned divide)

- convert the "divide code" developped in lab4 into a subroutine

- parameters passed in R0 (Numerator) and R1 (Divisor)

- results returned in R0 (Quotient) and R1 (Remainder)

- code uses R0, R1, R2, R3, R4, R5, R6

- need to save and restore R4, R5 and R6 at entry and exit

- although UDIV is a leaf subroutine, decided to push LR at entry so that only a single PUSH and POP is needed to convert existing code into a subroutine

```
UDIV        PUSH          {R4, R5, R6, LR}   ; push R4, R5, R6 and return address

            ...

            <UDIV body which modifies R0, R1, R2, R3, R4, R5 and R6>

            ...

            POP           {R4, R5, R6, PC}   ; pop R4, R5, R6 and return
```

Example 3: UDIV …

```
;
; at entry   R0 = N (numerator)
;            R1 = D (divisor)
; at exit    R0 = Q (quotient)
;            R1 = R (reminder)
;
UDIV    PUSH    {R4, R5, R6, LR}    ; push R4, R5, R6 and return address
        MOV     R2, R0              ; R2 = N
        MOV     R3, R1              ; R3 = D
        MOV     R0, #0              ; R0 = Q = 0
        MOV     R1, #0              ; R1 = R = 0
        MOV     R4, #31             ; R4 = i = 31
        MOV     R5, #1              ; R5 = 1 (used as a mask)
UDIV0   CMP     R4, #0              ; i == 0 ?
        BLT     UDIV2               ; finished
        MOV     R1, R1, LSL #1      ; R = R << 1
        AND     R6, R5, R2, LSR R4  ; R[0] = N[i]
        ORR     R1, R1, R6          ;
        CMP     R1, R3              ; R >= D?
        BLT     UDIV1               ;
        SUB     R1, R1, R3          ; R = R - D
        ORR     R0, R0, R5, LSL R4  ; Q[i] = 1
UDIV1   SUB     R4, R4, #1          ; i = i - 1
        B       UDIV0               ; next bit
UDIV2   POP     {R4, R5, R6, PC}    ; pop into R4, R5, R6 and return
```

alternative entry/exit code

```
PUSH        {R4, R5, R6}
…
…
POP         {R4, R5, R6}
BX          LR
```
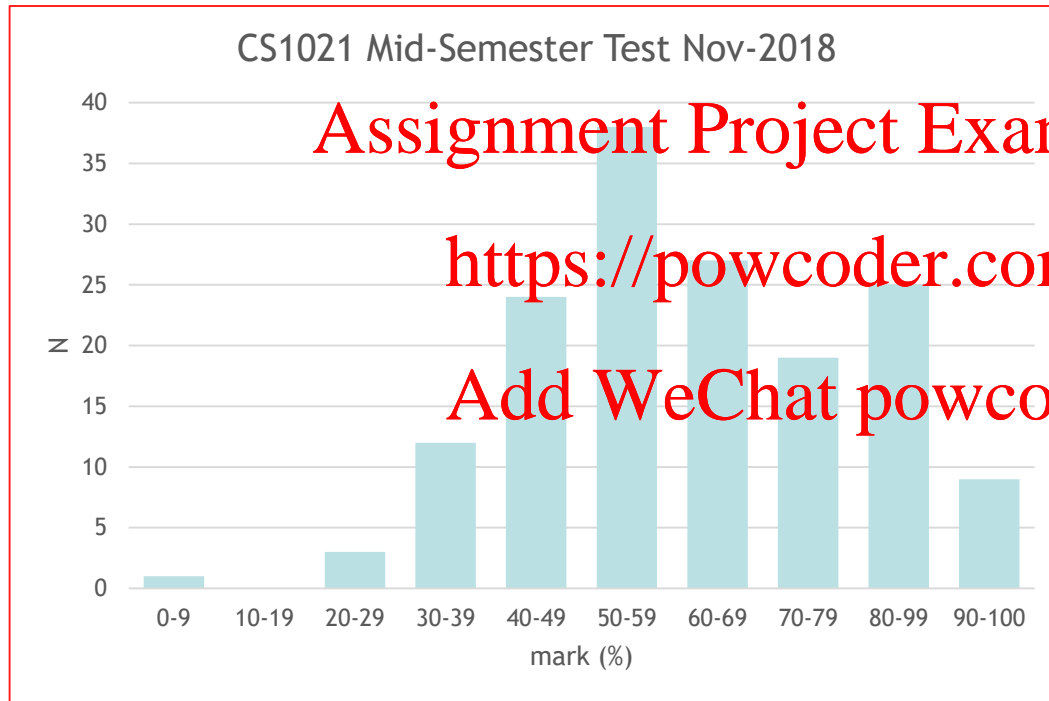
Example 3: calling UDIV

- an array **b** of 8 x 32 bit unsigned integers is stored in memory @ 0x40000000

- write code to divide each integer by 42

- MUST ASSUME that UDIV modifies R2 and R3 as well as R0 and R1, so use R4 and R5 as address registers

```
        LDR     R4, =0x40000000         ; R4 -> b
        ADD     R5, R4, #32             ; R5 -> end of array b
L       LDR     R0, [R4]                ; load integer from b
        LDR     R1, =42                 ; divide by …
        BL      UDIV                    ; 42
        STR     R0, [R4], #4            ; store result AND R4 = R4 + 4
        CMP     R4, R5                  ; finished?
        BNE     L                       ; next integer
```

Mid-Semester Test 2018

CS1021 Mid-Semester Test Nov-2018

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

< 40	F
40-49	III
50-59	II.2
60-69	II.1
70 - 100	I

N = 158 avg = 61.5%

lab6

- "9,589 prime numbers in the first 100,000 integers" is incorrect

- need to compute n / 8 and n % 8 (n mod 8)

- 8 is a 2^3 (a power of 2)

- decimal analogy 1234 / 100 and 1234 % 100

  1234 / 100 = 12 r 34

- binary equivalent

  2^3

- xxxx xxxx xxxx xxxx_2

- n / 8 = n >> 3
- n % 8 = n & 0x07 (where $7 = 2^3 - 1$)

10^2

What has not been covered in module

- ROR (rotate right) and ASR (arithmetic shift right) as per LSL and LSR

- details of LDR instruction (including ROR of immediate operand)

- LDRH (load halfword) and STRH (store halfword)

- LDRSB (load byte with sign extend) and LDRSH (load halfword with sign extend)

- other types of stacks

- subroutines with more than 4 parameters

- subroutines with more local variables than available registers

- recursion

- …

CS1021 Learning Outcomes

at the end of the module you will be able to:

- describe the basic components and operation of a computer system

- represent and interpret information stored in binary form (integers, text, ...)

- design, write, test and document assembly language programs to solve simple problems

- translate high-level programming language constructs into their assembly language equivalents

- evaluate the efficiency of simple algorithms

- make use of appropriate documentation and reference material