Support for high-level languages

- Outline:
  - memory organization
  - ARM data types
  - conditional statements & loop structures
  - the ARM Procedure Call Standard

☞ hands-on: writing & debugging C programs

Support for high-level languages

- ARM has a ‘vanilla’ instruction set
  - it has no language specific support
  - the basic instruction set supports...
    - various data types
    - expressions
    - conditional statements
    - loops

  ...in straightforward ways
    - see book Chapter 6

Support for high-level languages

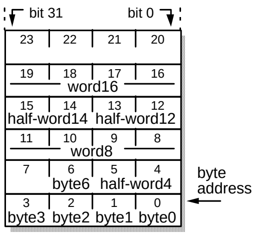
- Outline:
  - ➔memory organization
  - ARM data types
  - conditional statements & loop structures
  - the ARM Procedure Call Standard

☞ hands-on: writing & debugging C programs

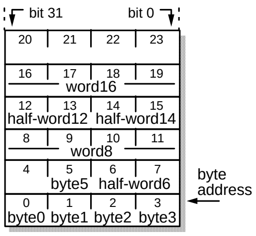
Memory organization

- Little-endian memory
  - least significant byte stored at lowest memory address
- Big-endian memory
  - least signifiant byte stored at highest memory address
- ARM can be configured either way
  - we will stick to the little-endian organization, as nature intended!

Little- and big-endian memory



(a) Little-endian memory organization

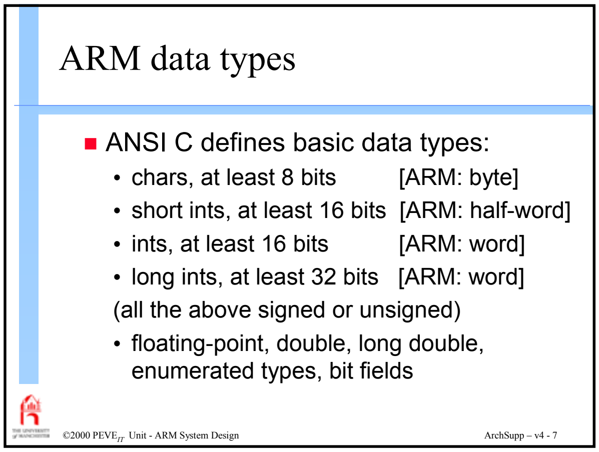


(b) Big-endian memory organization

Support for high-level languages

- Outline:
  - memory organization
  - ➔ARM data types
  - conditional statements & loop structures
  - the ARM Procedure Call Standard

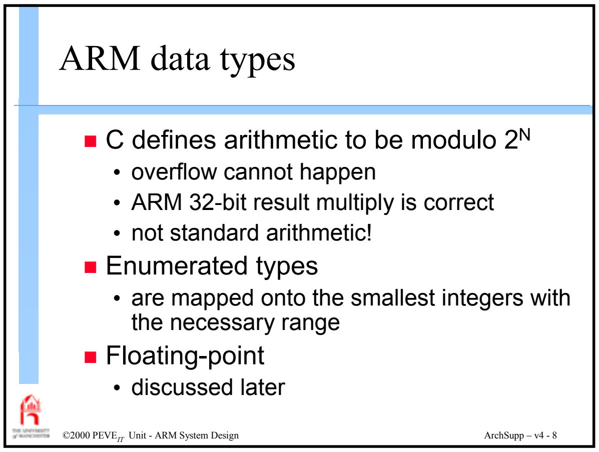
☞ hands-on: writing & debugging C programs



ARM data types

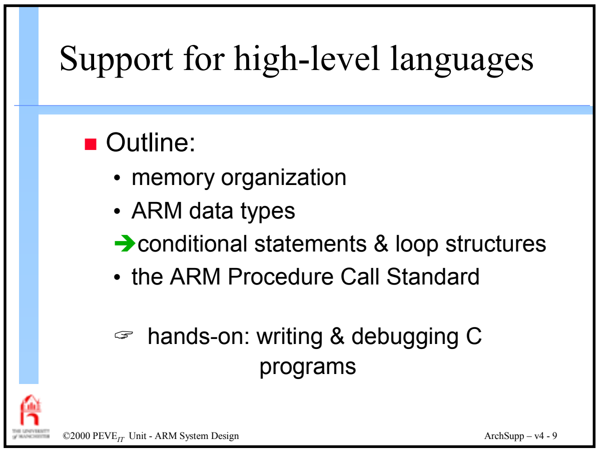
- ANSI C defines basic data types:
  - chars, at least 8 bits [ARM: byte]
  - short ints, at least 16 bits [ARM: half-word]
  - ints, at least 16 bits [ARM: word]
  - long ints, at least 32 bits [ARM: word]

  (all the above signed or unsigned)
  - floating-point, double, long double, enumerated types, bit fields



ARM data types

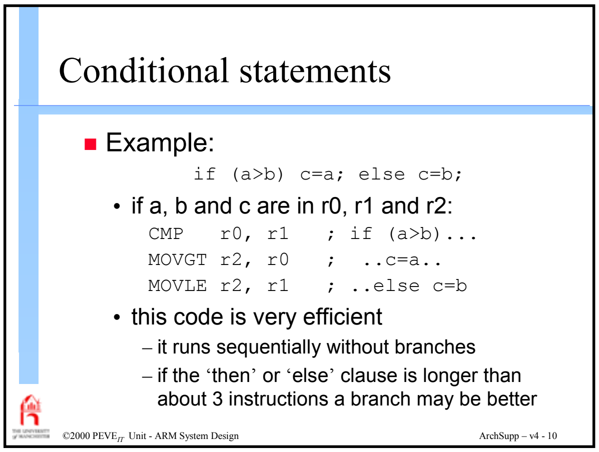
- C defines arithmetic to be modulo 2^N
  - overflow cannot happen
  - ARM 32-bit result multiply is correct
  - not standard arithmetic!
- Enumerated types
  - are mapped onto the smallest integers with the necessary range
- Floating-point
  - discussed later



Support for high-level languages

- Outline:
  - memory organization
  - ARM data types
  - ➔conditional statements & loop structures
  - the ARM Procedure Call Standard

  ☞ hands-on: writing & debugging C programs



Conditional statements

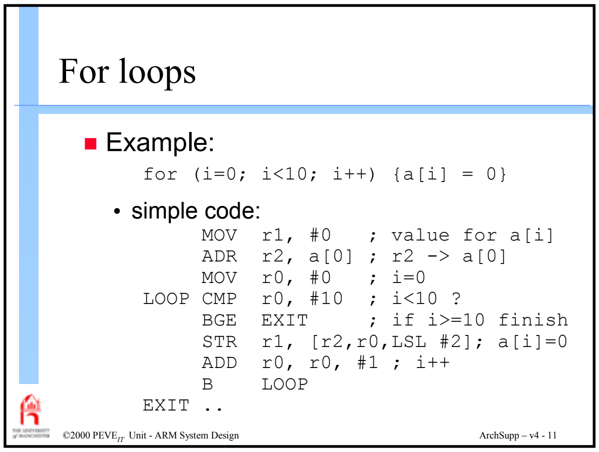
- Example:

```
      if (a>b) c=a; else c=b;
```

  - if a, b and c are in r0, r1 and r2:

```
CMP   r0, r1   ; if (a>b)...
MOVGT r2, r0   ;  ..c=a..
MOVLE r2, r1   ; ..else c=b
```

  - this code is very efficient
    - it runs sequentially without branches
    - if the ‘then’ or ‘else’ clause is longer than about 3 instructions a branch may be better



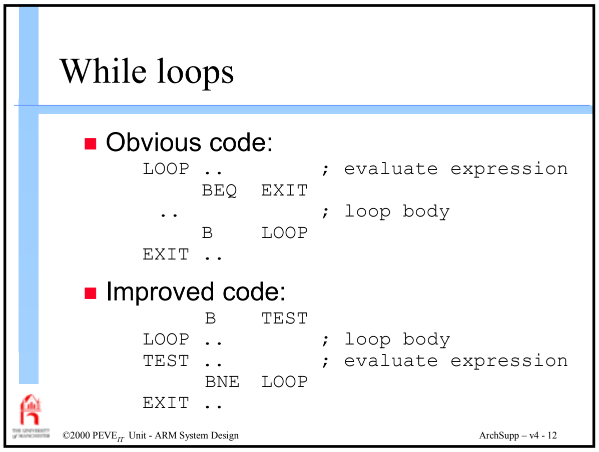
For loops

- Example:

```
for (i=0; i<10; i++) {a[i] = 0}
```

  - simple code:

```
     MOV  r1, #0    ; value for a[i]
     ADR  r2, a[0]  ; r2 -> a[0]
     MOV  r0, #0    ; i=0
LOOP CMP  r0, #10   ; i<10 ?
     BGE  EXIT      ; if i>=10 finish
     STR  r1, [r2,r0,LSL #2]; a[i]=0
     ADD  r0, r0, #1 ; i++
     B    LOOP
EXIT ..
```



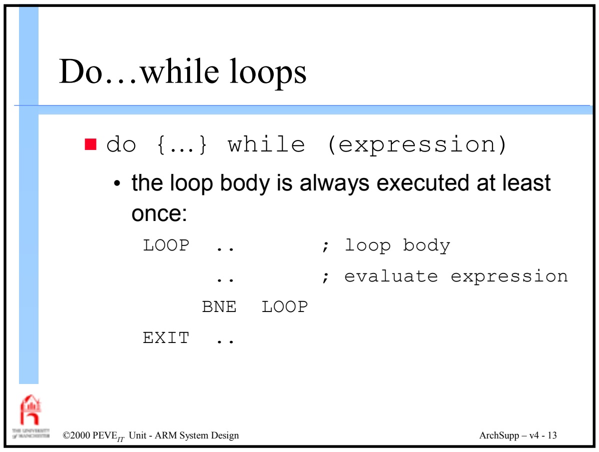
While loops

- Obvious code:

```
LOOP ..           ; evaluate expression
     BEQ  EXIT
 ..               ; loop body
     B    LOOP
EXIT ..
```

- Improved code:

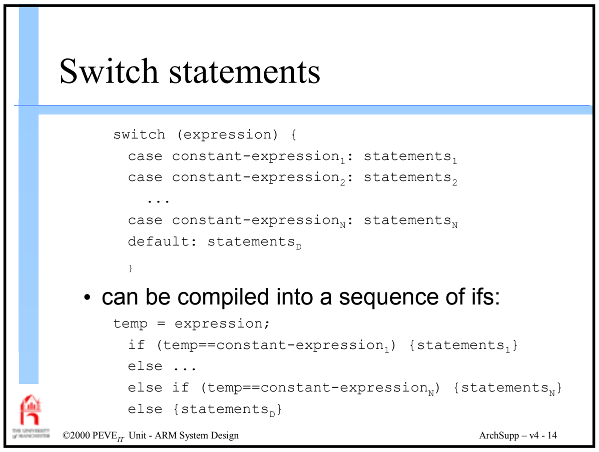
```
     B    TEST
LOOP ..           ; loop body
TEST ..           ; evaluate expression
     BNE  LOOP
EXIT ..
```



Do…while loops

- `do {…} while (expression)`
  - the loop body is always executed at least once:

```
LOOP  ..        ; loop body
      ..        ; evaluate expression
         BNE  LOOP
EXIT  ..
```

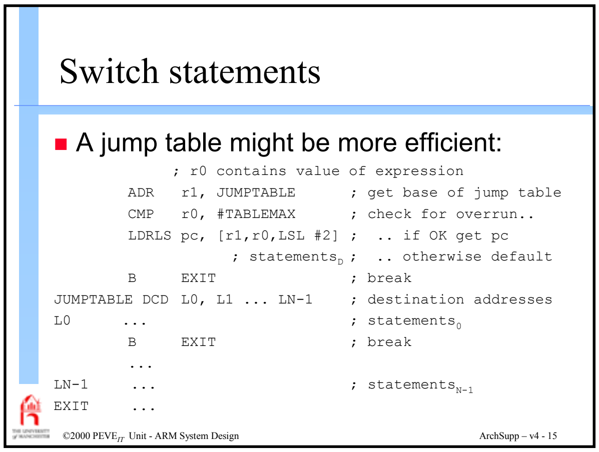


Switch statements

```
switch (expression) {
  case constant-expression1: statements1
  case constant-expression2: statements2
    ...
  case constant-expressionN: statementsN
  default: statementsD
  }
```

- can be compiled into a sequence of ifs:

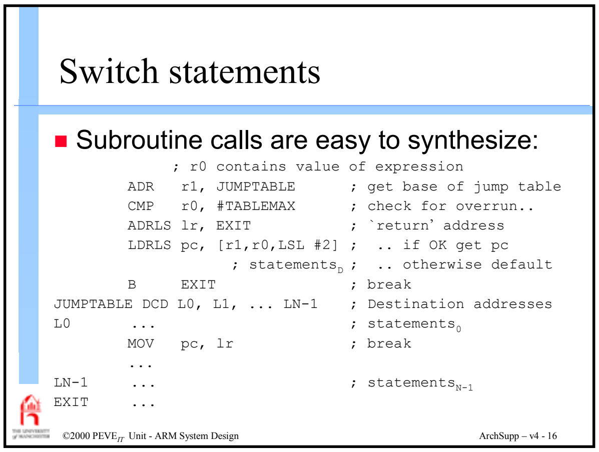
```
temp = expression;
  if (temp==constant-expression1) {statements1}
  else ...
  else if (temp==constant-expressionN) {statementsN}
  else {statementsD}
```



Switch statements

- A jump table might be more efficient:

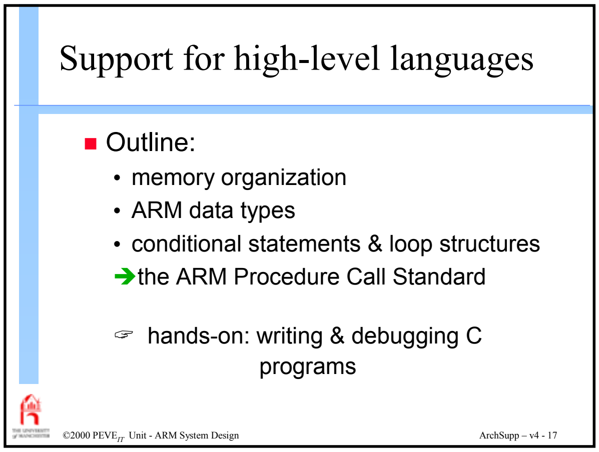
```
            ; r0 contains value of expression
        ADR   r1, JUMPTABLE       ; get base of jump table
        CMP   r0, #TABLEMAX       ; check for overrun..
        LDRLS pc, [r1,r0,LSL #2]  ;   .. if OK get pc
                  ; statementsD   ;   .. otherwise default
        B     EXIT                ; break
JUMPTABLE DCD L0, L1 ... LN-1     ; destination addresses
L0      ...                       ; statements0
        B     EXIT                ; break
        ...
LN-1     ...                      ; statementsN-1
EXIT     ...
```



Switch statements

- Subroutine calls are easy to synthesize:

```
            ; r0 contains value of expression
        ADR   r1, JUMPTABLE       ; get base of jump table
        CMP   r0, #TABLEMAX       ; check for overrun..
        ADRLS lr, EXIT            ; `return' address
        LDRLS pc, [r1,r0,LSL #2]  ;   .. if OK get pc
                  ; statementsD   ;   .. otherwise default
        B     EXIT                ; break
JUMPTABLE DCD L0, L1, ... LN-1    ; Destination addresses
L0       ...                      ; statements0
        MOV   pc, lr              ; break
        ...
LN-1     ...                      ; statementsN-1
EXIT     ...
```

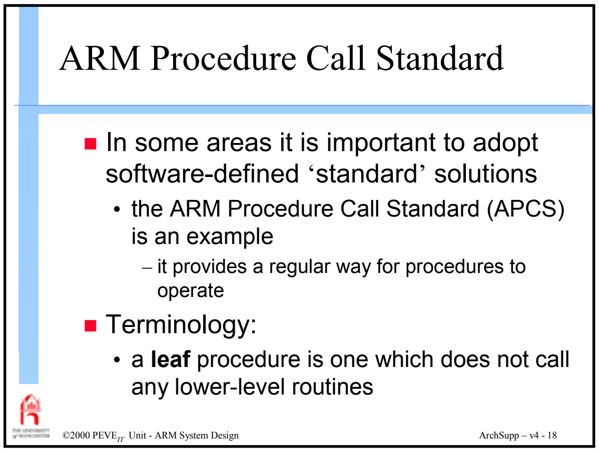


Support for high-level languages

- Outline:
  - memory organization
  - ARM data types
  - conditional statements & loop structures

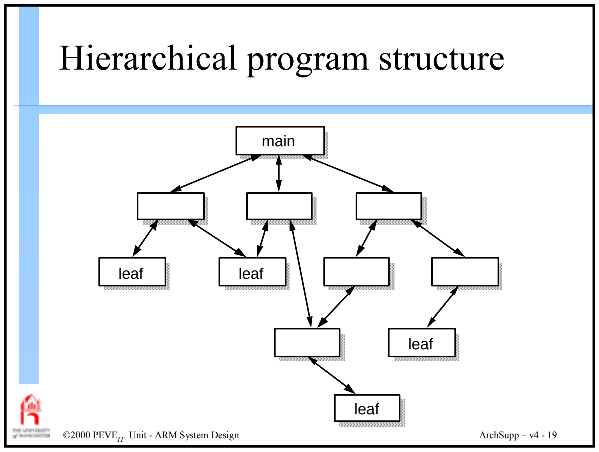
  ➔the ARM Procedure Call Standard

  ☞ hands-on: writing & debugging C programs

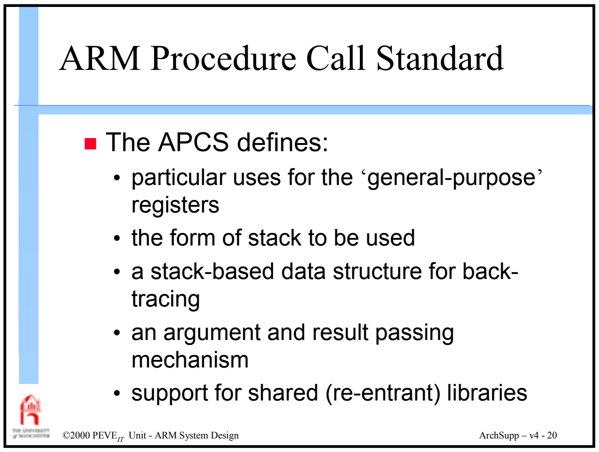


ARM Procedure Call Standard

- In some areas it is important to adopt software-defined ‘standard’ solutions
  - the ARM Procedure Call Standard (APCS) is an example
    - it provides a regular way for procedures to operate
- Terminology:
  - a **leaf** procedure is one which does not call any lower-level routines

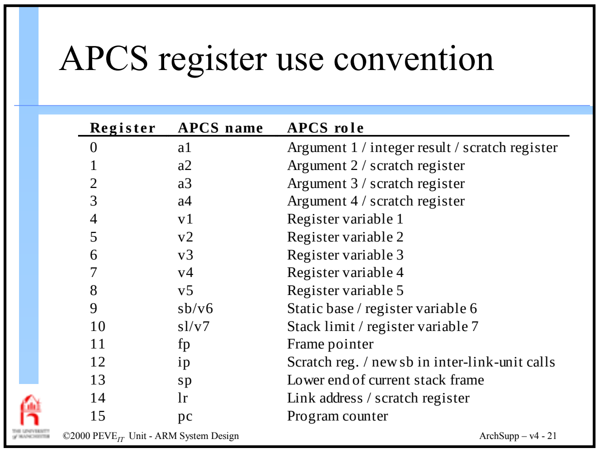


Hierarchical program structure



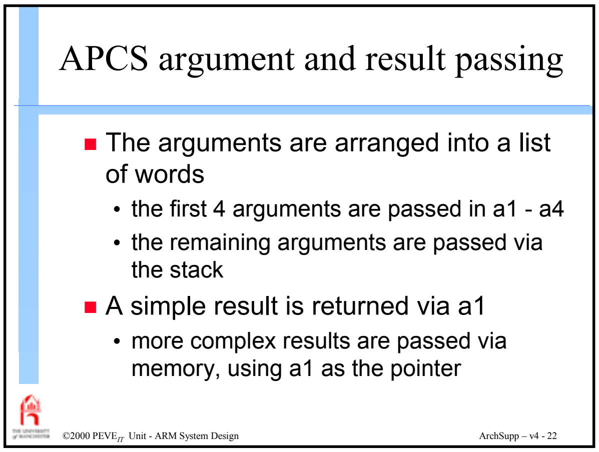
ARM Procedure Call Standard

- The APCS defines:
  - particular uses for the ‘general-purpose’ registers
  - the form of stack to be used
  - a stack-based data structure for back-tracing
  - an argument and result passing mechanism
  - support for shared (re-entrant) libraries



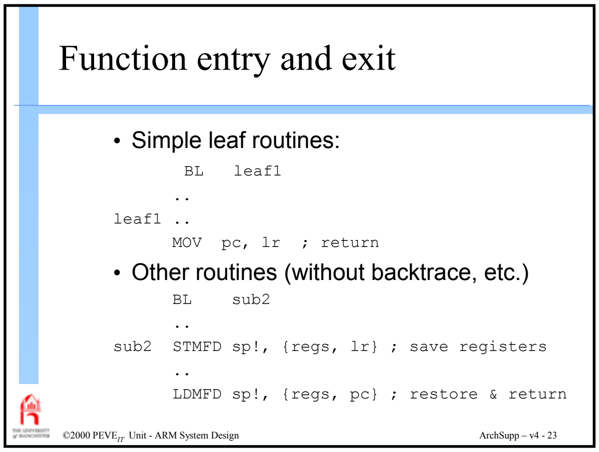
APCS register use convention

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter



APCS argument and result passing

- The arguments are arranged into a list of words
  - the first 4 arguments are passed in a1 - a4
  - the remaining arguments are passed via the stack
- A simple result is returned via a1
  - more complex results are passed via memory, using a1 as the pointer



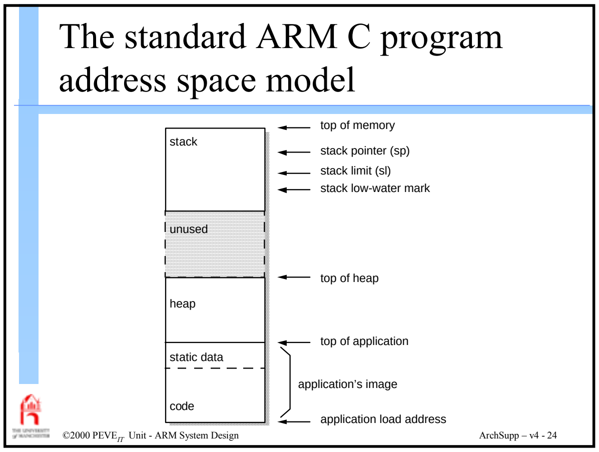
Function entry and exit

- Simple leaf routines:

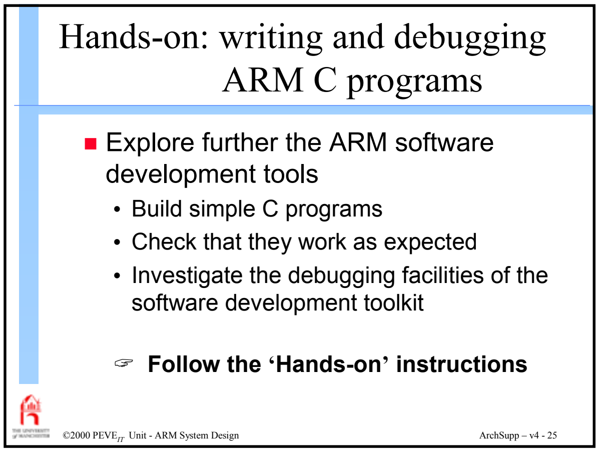
```
        BL    leaf1
        ..
leaf1   ..
        MOV   pc, lr   ; return
```

- Other routines (without backtrace, etc.)

```
        BL     sub2
        ..
sub2    STMFD sp!, {regs, lr} ; save registers
        ..
        LDMFD sp!, {regs, pc} ; restore & return
```



The standard ARM C program address space model



Hands-on: writing and debugging ARM C programs

- Explore further the ARM software development tools
  - Build simple C programs
  - Check that they work as expected
  - Investigate the debugging facilities of the software development toolkit

☞ **Follow the ‘Hands-on’ instructions**

©2000 PEVE$_{IT}$ Unit - ARM System Design ArchSupp – v4 - 25