Introduction to ARM Assembly Programming

Reference

The Definitive Guide to the ARM Cortex-M0, Joseph Yiu.

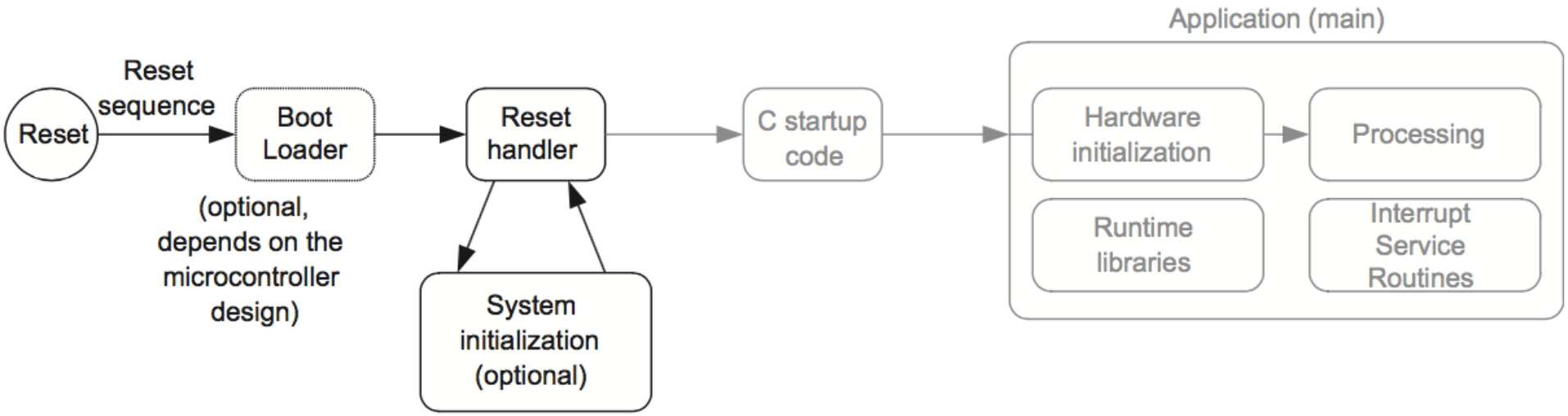
http://www.eecs.umich.edu/courses/eecs373/refs.html

http://www.eecs.umich.edu/courses/eecs373/labs.html

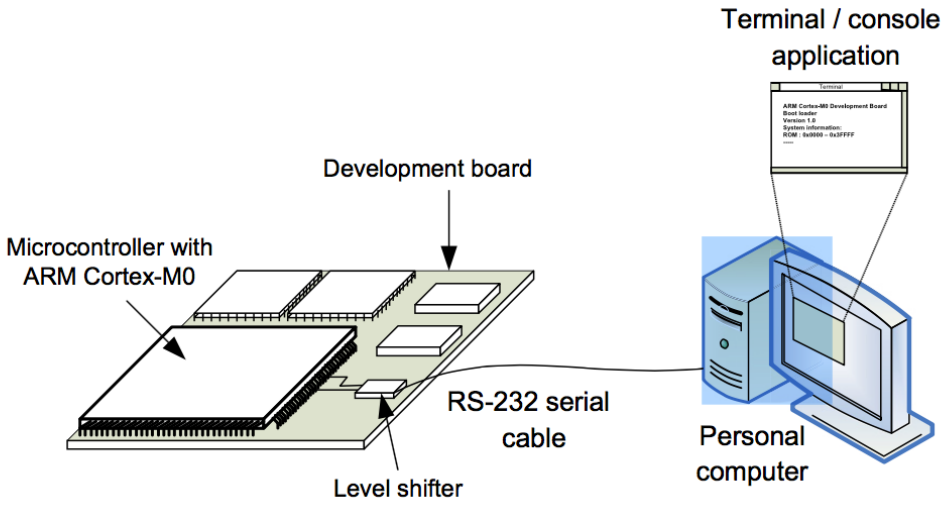
Outline

- Introduction to Cortex-Mx Programming
- Instruction Set
- Instruction Usage Examples

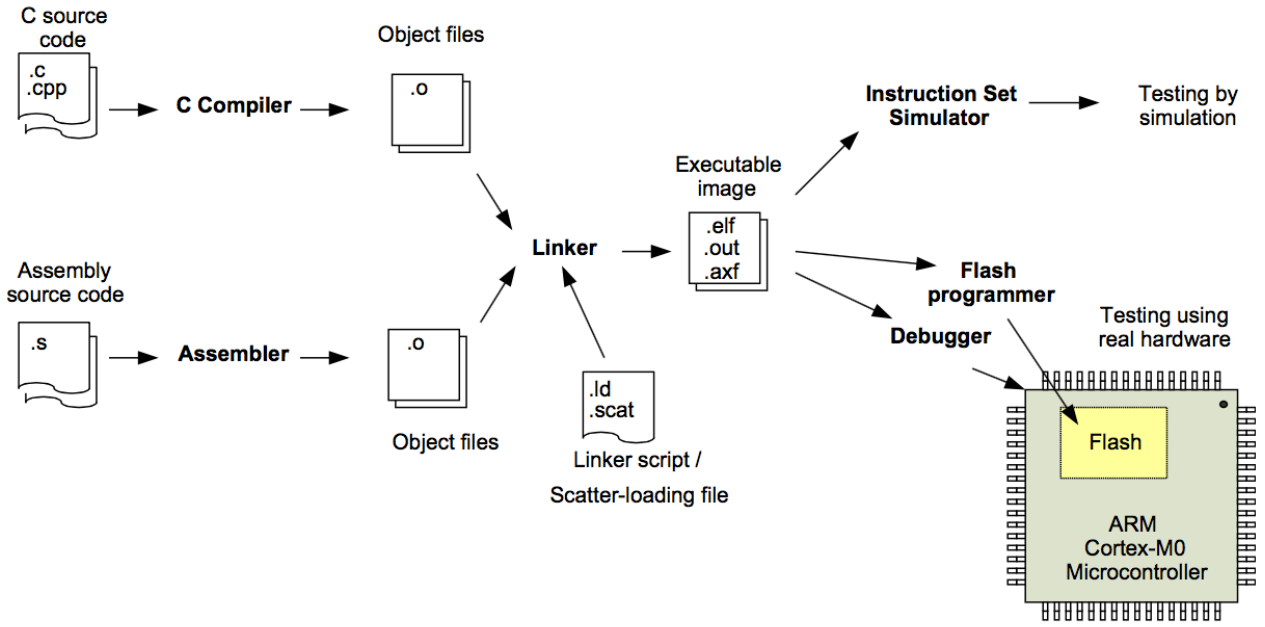
Bootup Code


Reset
Reset sequence
Boot Loader
(optional, depends on the microcontroller design)
Reset handler
System initialization (optional)
C startup code
Application (main)
Hardware initialization
Processing
Runtime libraries
Interrupt Service Routines

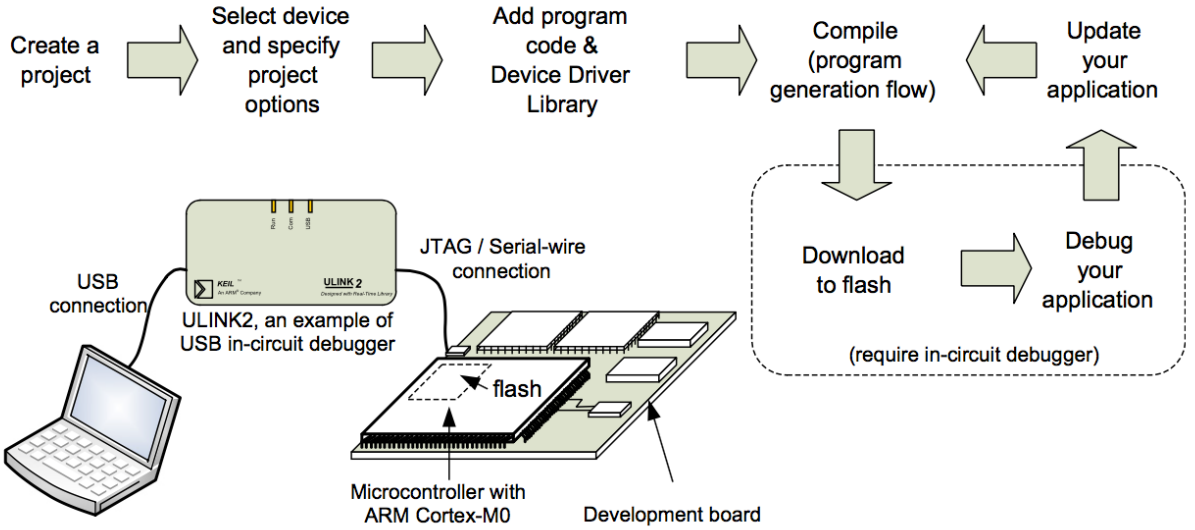
Development Environment



Development Process

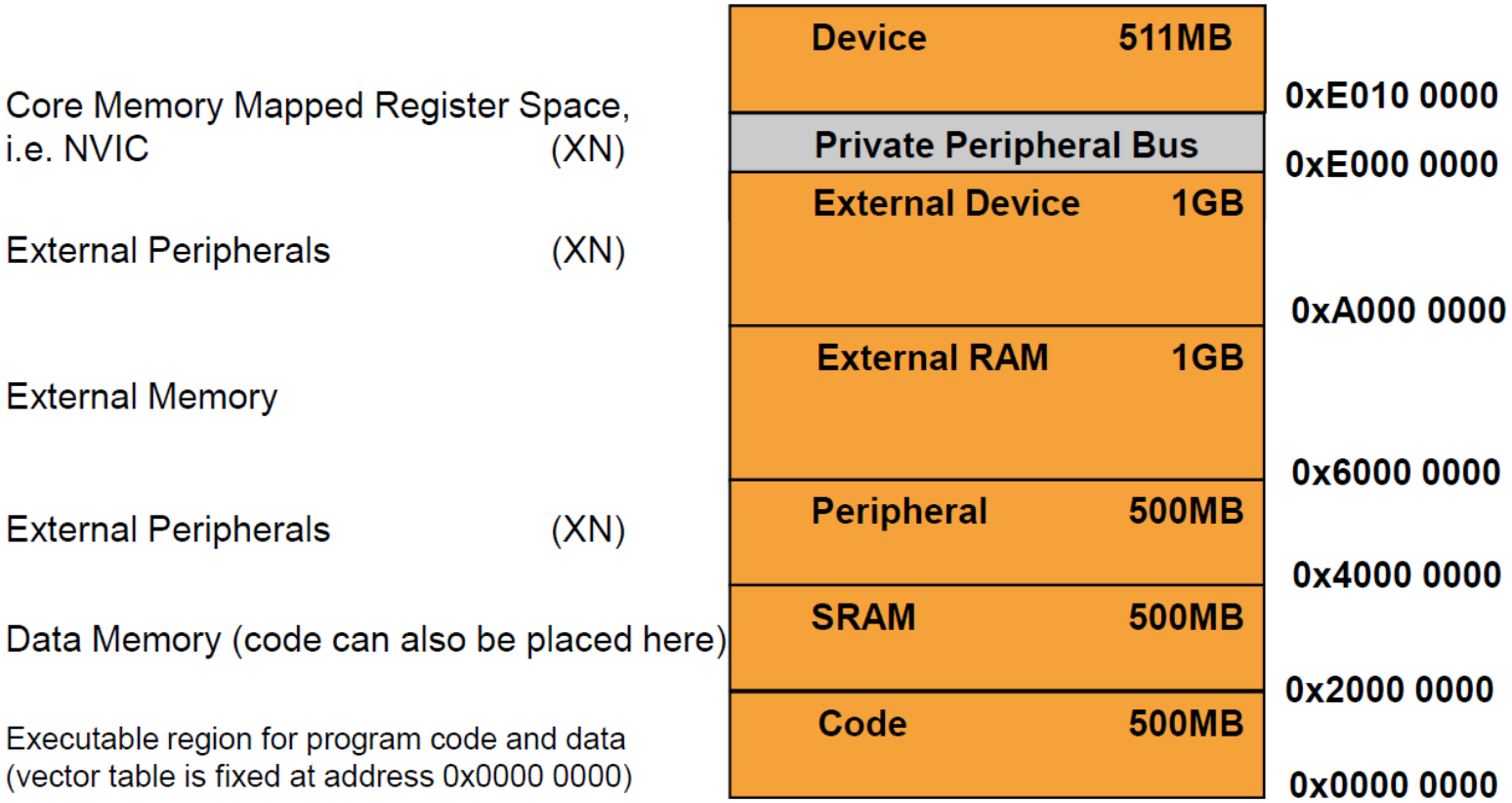


Development Process (Cont.)



ARM Cortex-M0 Memory map

- 32bit address space

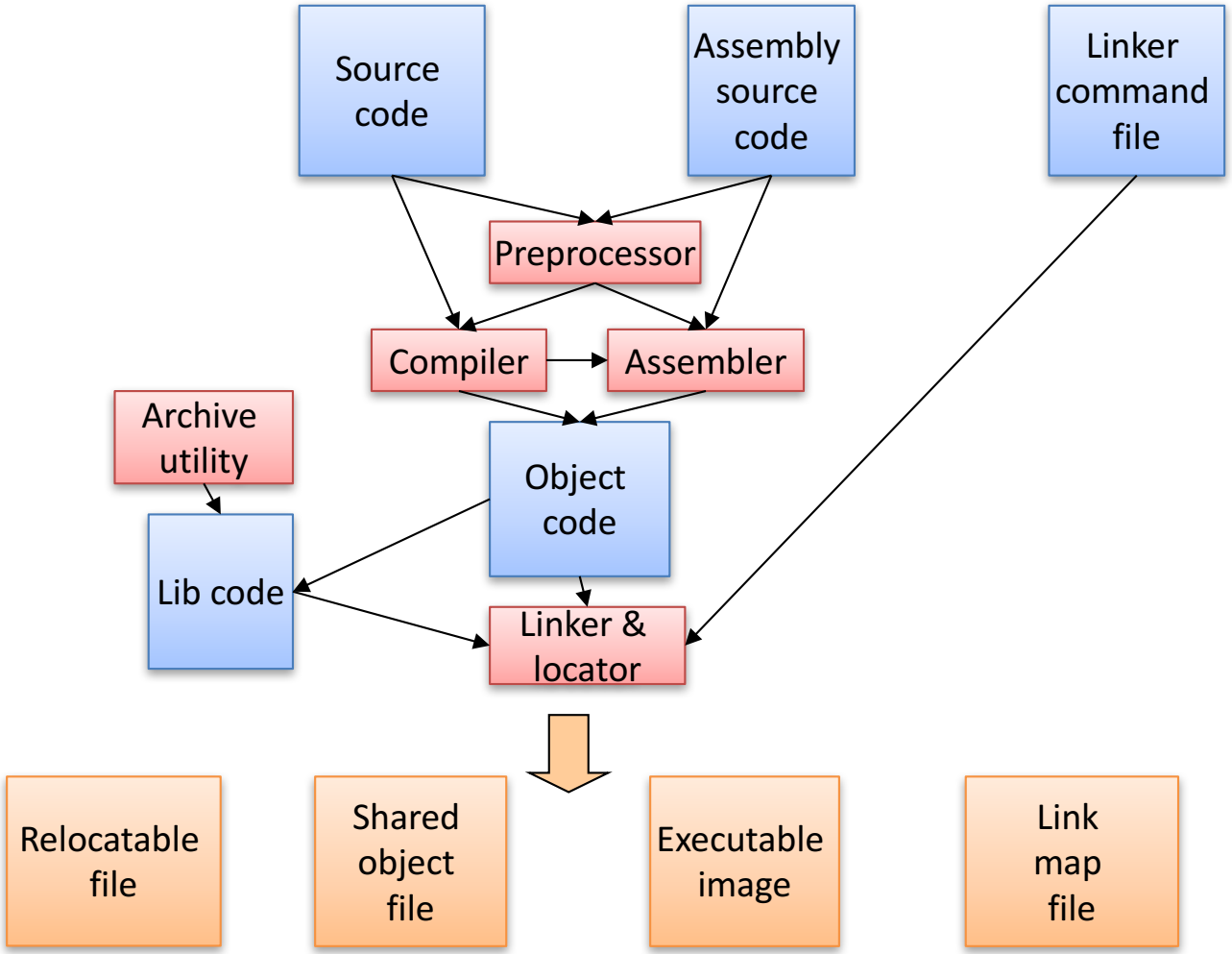


XN – execute never

Please also 003-MCSL-ARM.pdf slides P.11 see the detail map

How to compile kernel codes

- Creating an executable image



Assembly, Machine code, Memory dump

Assembly ->

```
Dump of assembler code for function system_call:
   0x000076d8 <+0>:     cmp    $0x47,%eax
   0x000076db <+3>:     ja     0x76c8 <bad_sys_call>
   0x000076dd <+5>:     push   %ds
   0x000076de <+6>:     push   %es
   0x000076df <+7>:     push   %fs
   0x000076e1 <+9>:     push   %edx
=> 0x000076e2 <+10>:    push   %ecx
   0x000076e3 <+11>:    push   %ebx
   0x000076e4 <+12>:    mov    $0x10,%edx
   0x000076e9 <+17>:    mov    %edx,%ds
   0x000076eb <+19>:    mov    %edx,%es
   0x000076ed <+21>:    mov    $0x17,%edx
   0x000076f2 <+26>:    mov    %edx,%fs
   0x000076f4 <+28>:    call   *0x1a020(,%eax,4)
   0x000076fb <+35>:    push   %eax
   0x000076fc <+36>:    mov    0x1b140,%eax
   0x00007701 <+41>:    cmpl   $0x0,(%eax)
   0x00007704 <+44>:    jne    0x76ce <reschedule>
   0x00007706 <+46>:    cmpl   $0x0,0x4(%eax)
   0x0000770a <+50>:    je     0x76ce <reschedule>
End of assembler dump.
```

Memory dump

|

v

```
0x76d8 <system_call>:   131     248     71      119     235     30      6       15
0x76e0 <system_call+8>: 160     82      81      83      186     16      0       0
0x76e8 <system_call+16>:        0       142     218     142     194     186     23      0
0x76f0 <system_call+24>:        0       0       142     226     255     20      133     32
0x76f8 <system_call+32>:        160     1       0       80      161     64      177     1
0x7700 <system_call+40>:        0       131     56      0       117     200     131     120
0x7708 <system_call+48>:        4       0
```

Sections (Contd.)

```
        .data
arr:    .word 10, 20, 30, 40, 50
len:    .word 5
        .text
start:  mov r1, #10
        mov r2, #20
        .data
result: .skip 4
        .text
        add r3, r2, r1
        sub r3, r2, r1
```

.data section

```
0000_0000 arr:    .word 10, 20, 30, 40, 50
0000_0014 len:    .word 5
0000_0018 result: .skip 4
```

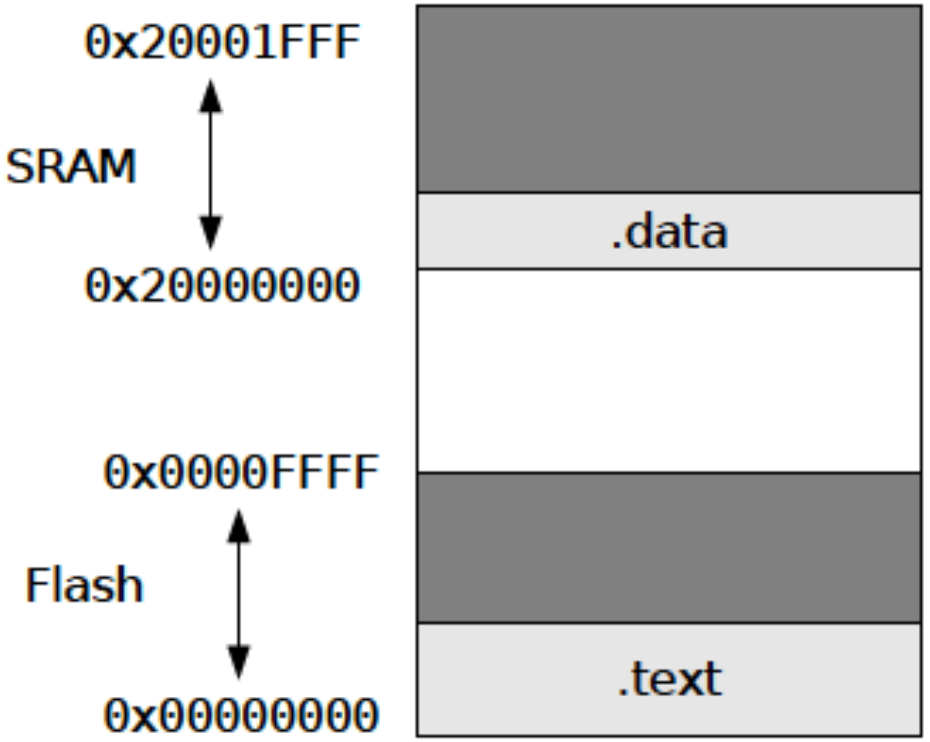
.text section

```
0000_0000 start:  mov r1, #10
0000_0004         mov r2, #20
0000_0008         add r3, r2, r1
0000_000C         sub r3, r2, r1
```

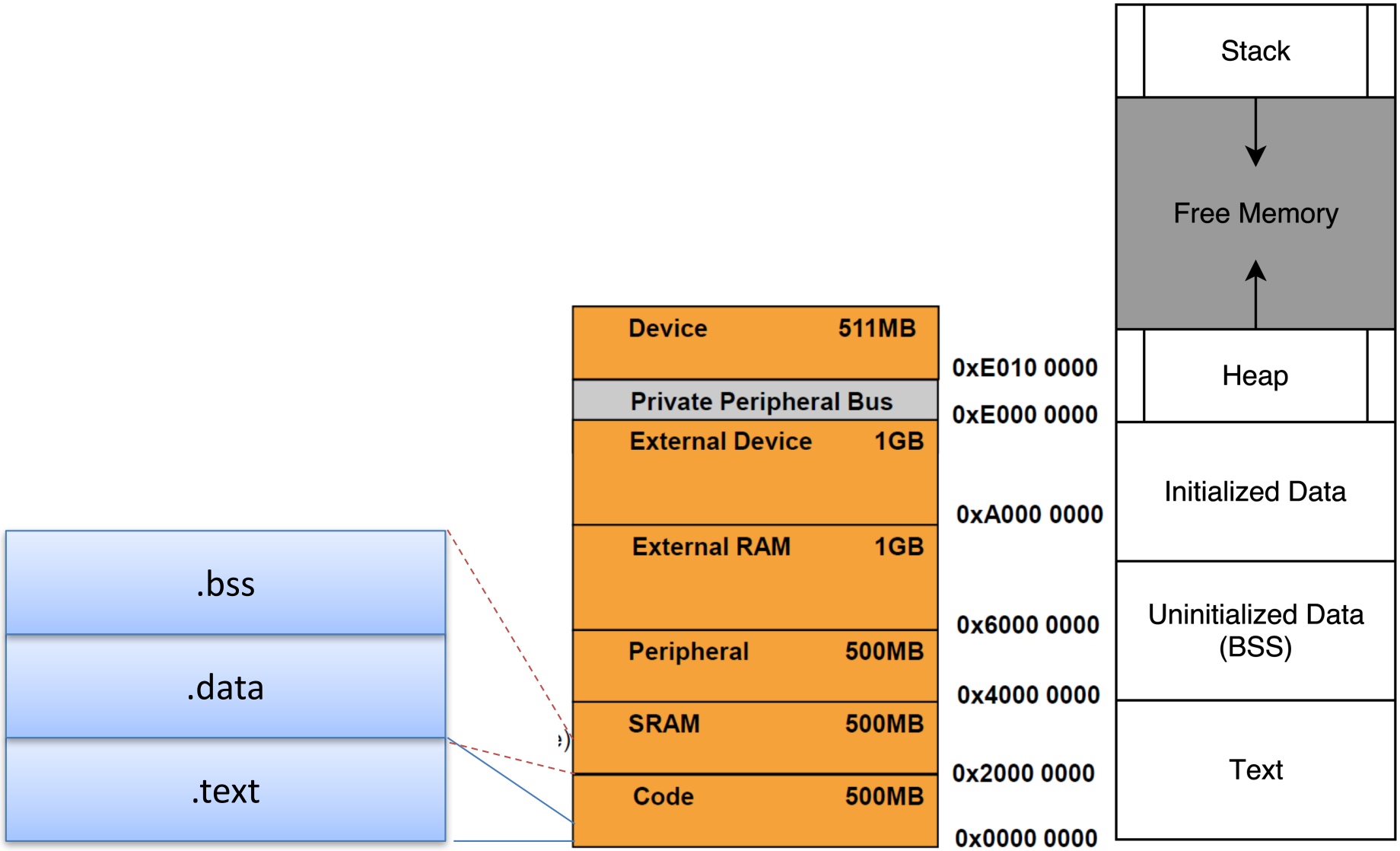
- Source – sections can be interleaved
- Bytes of a section – contiguous addresses

Linker Script Revisited

```
MEMORY {
     FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
     SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
     .text : {
          * (.text);
     } > FLASH

     .data : {
          * (.data);
     } > SRAM
}
```



Executable file, Physical Memory, and Program View



Linker Script

```
32  /* Entry Point */
33  ENTRY(Reset_Handler)
34
35  /* Highest address of the user mode stack */
36  _estack = 0x20020000;    /* end of 128K RAM on AHB bus*/
37
38  /* Generate a link error if heap and stack don't fit into RAM */
39  _Min_Heap_Size = 0;      /* required amount of heap  */
40  _Min_Stack_Size = 0x400; /* required amount of stack */
41
42  /* Specify the memory areas */
43  MEMORY
44  {
45    FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 1024K
46    RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 192K
47    MEMORY_B1 (rx)  : ORIGIN = 0x60000000, LENGTH = 0K
48  }
```

```
61    /* The program code and other data goes into FLASH */
62    .text :
63    {
64      . = ALIGN(4);
65      *(.text)           /* .text sections (code) */
66      *(.text*)          /* .text* sections (code) */
67      *(.rodata)         /* .rodata sections (constants, strings, etc.) */
68      *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
69      *(.glue_7)         /* glue arm to thumb code */
70      *(.glue_7t)        /* glue thumb to arm code */
71      *(.eh_frame)
72
73      KEEP (*(.init))
74      KEEP (*(.fini))
75
76      . = ALIGN(4);
77      _etext = .;        /* define a global symbols at end of code */
78      _exit = .;
79    } >FLASH
```

```
113   /* Initialized data sections goes into RAM, load LMA copy after code */
114   .data : AT ( _sidata )
115   {
116     . = ALIGN(4);
117     _sdata = .;        /* create a global symbol at data start */
118     *(.data)           /* .data sections */
119     *(.data*)          /* .data* sections */
120
121     . = ALIGN(4);
122     _edata = .;        /* define a global symbol at data end */
123   } >RAM
124
125   /* Uninitialized data section */
126   . = ALIGN(4);
127   .bss :
128   {
129     /* This is used by the startup in order to initialize the .bss secion */
130     _sbss = .;         /* define a global symbol at bss start */
131     __bss_start__ = _sbss;
132     *(.bss)
133     *(.bss*)
134     *(COMMON)
135
136     . = ALIGN(4);
137     _ebss = .;         /* define a global symbol at bss end */
138     __bss_end__ = _ebss;
139   } >RAM
```

Make File

```
# put your *.o targets here, make should handle the rest!

SRCS = main.s

# all the files will be generated with this name (main.elf, main.bin, main.hex, etc)

PROJ_NAME=main

# that's it, no need to change anything below this line!

############################################################

CC=arm-none-eabi-gcc
OBJCOPY=arm-none-eabi-objcopy

CFLAGS  = -g -O2 -Wall -Tstm32_flash.ld
CFLAGS += -mlittle-endian -mthumb -mcpu=cortex-m4 -mthumb-interwork
CFLAGS += -mfloat-abi=hard -mfpu=fpv4-sp-d16

############################################################

vpath %.c src
vpath %.a lib

ROOT=$(shell pwd)

CFLAGS += -Iinc -Ilib -Ilib/inc
CFLAGS += -Ilib/inc/core -Ilib/inc/peripherals

SRCS += lib/startup_stm32f4xx.s # add startup file to build

OBJS = $(SRCS:.c=.o)
```

Somethings before main

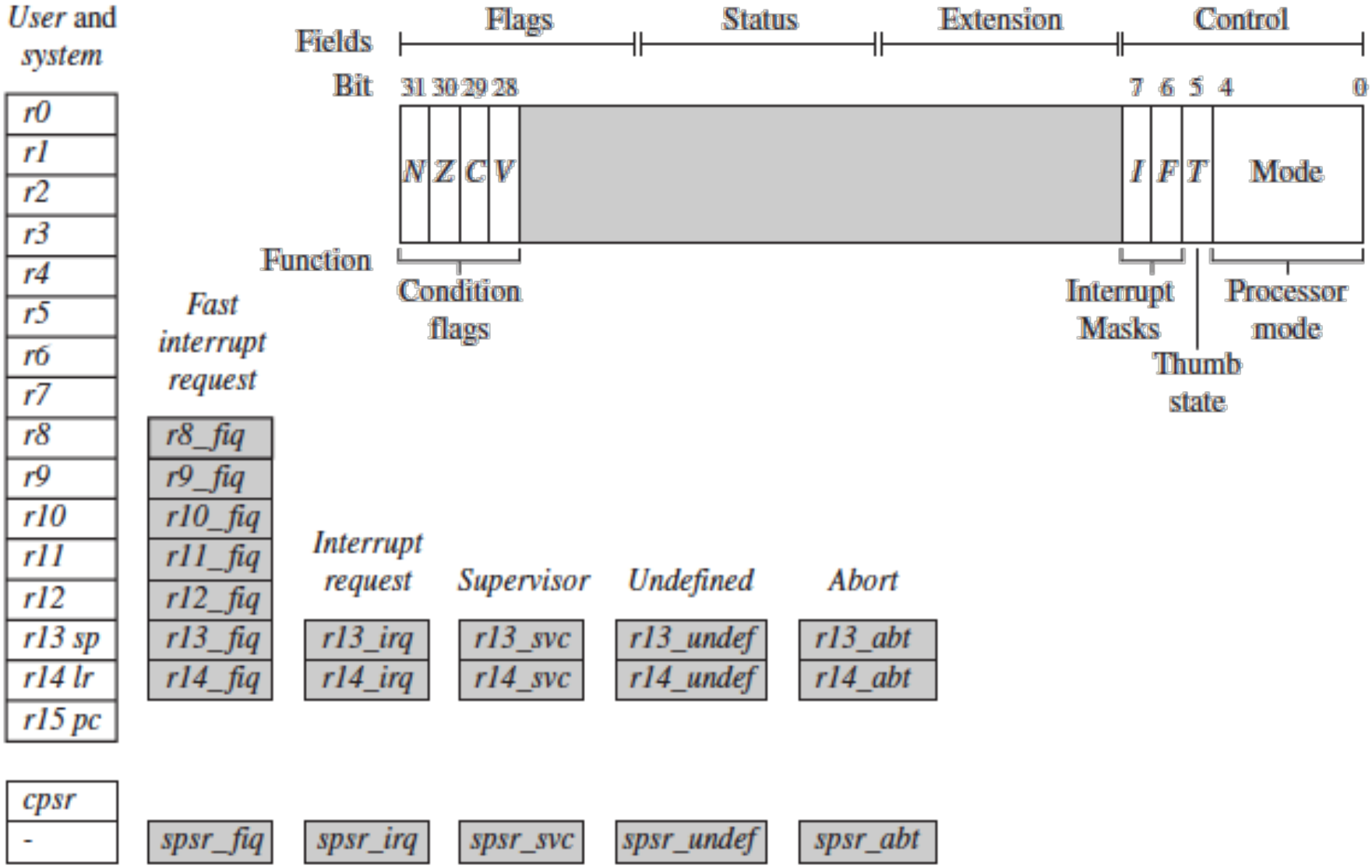
```
118  *
119  * The minimal vector table for a Cortex M3. Note that the proper constructs
120  * must be placed on this to ensure that it ends up at physical address
121  * 0x0000.0000.
122  *
123  *******************************************************************************/
124     .section  .isr_vector,"a",%progbits
125    .type  g_pfnVectors, %object
126    .size  g_pfnVectors, .-g_pfnVectors
127
128
129  g_pfnVectors:
130    .word  _estack
131    .word  Reset_Handler
132    .word  NMI_Handler
133    .word  HardFault_Handler
134    .word  MemManage_Handler
135    .word  BusFault_Handler
136    .word  UsageFault_Handler
137    .word  0
138    .word  0
139    .word  0
140    .word  0
141    .word  SVC_Handler
142    .word  DebugMon_Handler
143    .word  0
144    .word  PendSV_Handler
145    .word  SysTick_Handler
```

```
Reset_Handler:

/* Copy the data segment initializers from flash to SRAM */
  movs  r1, #0
  b  LoopCopyDataInit

CopyDataInit:
  ldr  r3, =_sidata
  ldr  r3, [r3, r1]
  str  r3, [r0, r1]
  adds  r1, r1, #4

LoopCopyDataInit:
  ldr  r0, =_sdata
  ldr  r3, =_edata
  adds  r2, r0, r1
  cmp  r2, r3
  bcc  CopyDataInit
  ldr  r2, =_sbss
  b  LoopFillZerobss
/* Zero fill the bss segment. */
FillZerobss:
  movs  r3, #0
  str  r3, [r2], #4

LoopFillZerobss:
  ldr  r3, = _ebss
  cmp  r2, r3
  bcc  FillZerobss

/* Call the clock system intitialization function.*/
  bl  SystemInit
/* Call static constructors */
    bl __libc_init_array
/* Call the application's entry point.*/
  bl  main
  bx  lr
.size  Reset_Handler, .-Reset_Handler
```

main.s

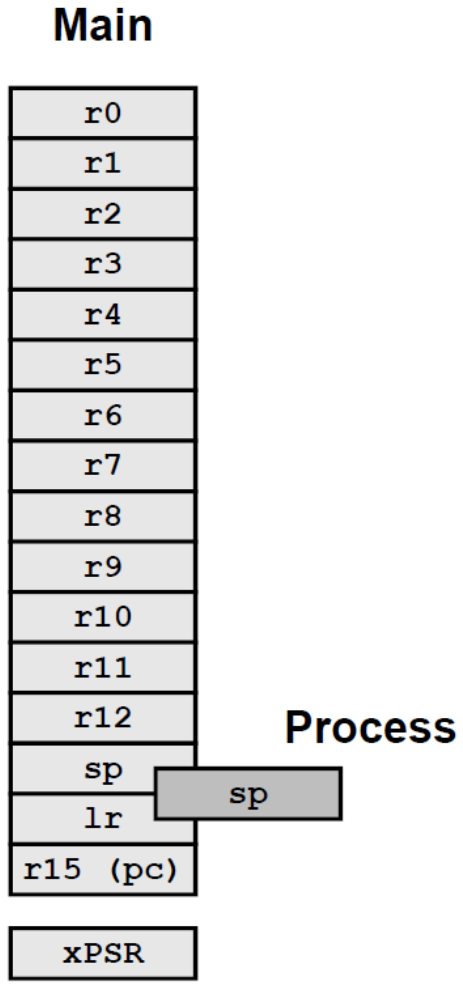
```
1     .syntax unified
2     .cpu cortex-m4
3     .thumb
4
5  .data
6     X: .word 100
7     str: .asciz "Hello World!"
8  .text
9     .global main
10    .equ AA, 0x55
11
12 main:
13    ldr r1, =X
14    ldr r0, [r1]
15    movs r2, #AA
16    adds r2, r2, r0
17    str  r2, [r1]
18
19    ldr  r1, =str
20    ldr  r2, [r1]
21 L: B L
22
```

Complete ARM Register



ARM Cortex-M0 register list

- All registers are 32 bits wide
- 13 general purpose registers
  - Registers r0 – r7 (Low registers)
  - Registers r8 – r12 (High registers)
- 3 registers with special meaning/usage
  - Stack Pointer (SP) – r13
  - Link Register (LR) – r14
  - Program Counter (PC) – r15
- Special-purpose registers
  - xPSR shows a composite of the content of
    - APSR, IPSR, EPSR



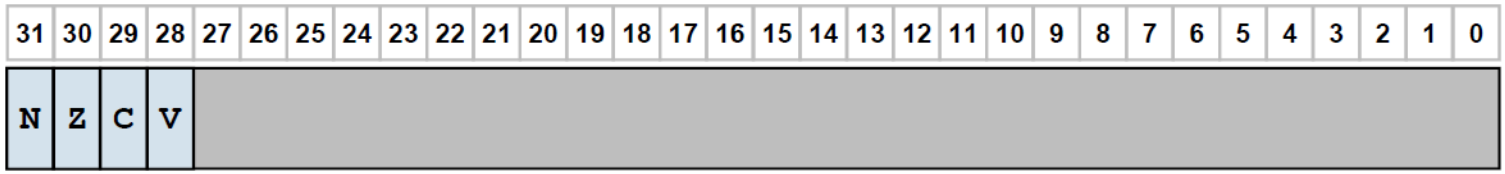
Special Registers

Table 5.5: Special Register Symbols for MRS and MSR Instructions

Symbol	Register	Access Type
APSR	Application Program Status Register (PSR)	Read/Write
EPSR	Execution PSR	Read only
IPSR	Interrupt PSR	Read only
IAPSR	Composition of IPSR and APSR	Read only
EAPSR	Composition of EPSR and APSR	Read only
IEPSR	Composition of IPSR and EPSR	Read only
XPSR	Composition of APSR, EPSR, and IPSR	Read only
MSP	Main stack pointer	Read/Write
PSP	Process stack pointer	Read/Write
PRIMASK	Primary exception mask register	Read/Write
CONTROL	CONTROL register	Read/Write in Thread mode Read only in Handler mode

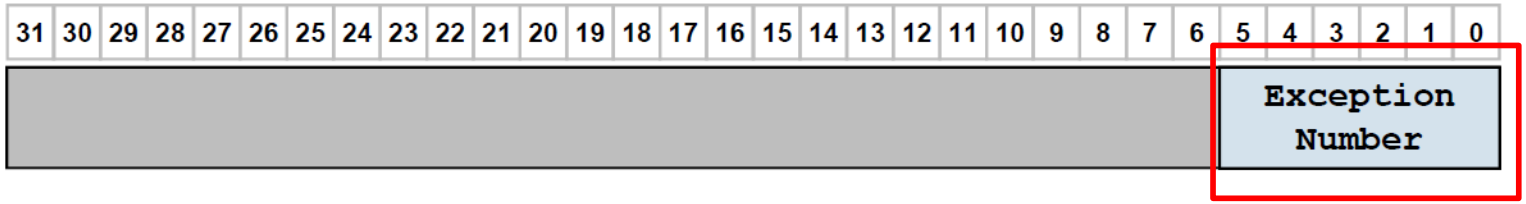
PSR Registers

- APSR - Application Program Status Register

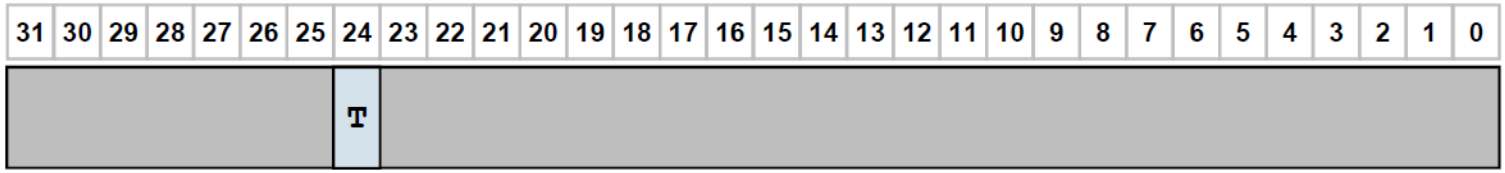


  - Contains the **N**egative, **Z**ero, **C**arry and O**V**erflow flags from the ALU

- IPSR – Interrupt Program Status Register



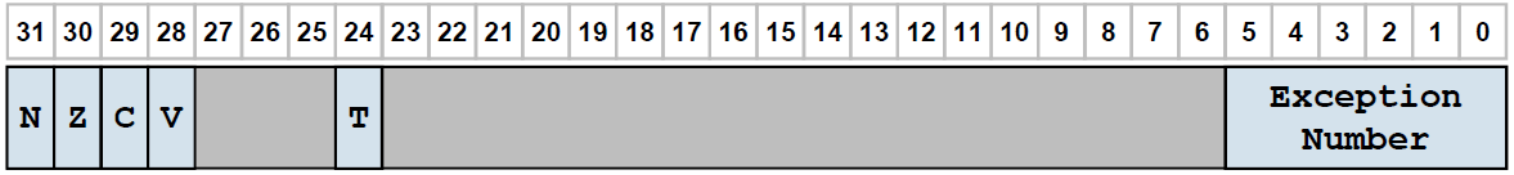
- EPSR – Execution Program Status Register



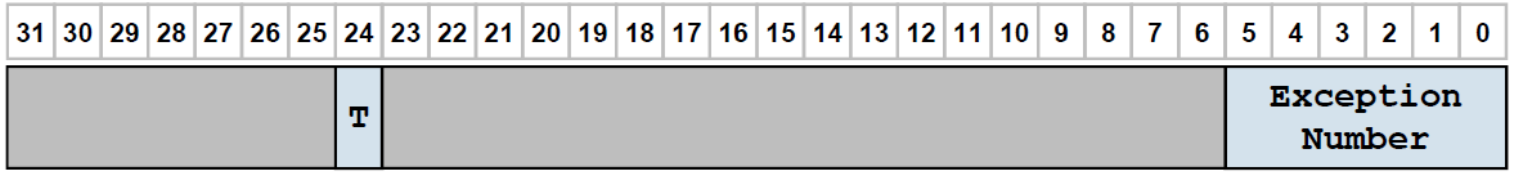
  - **T**humb code is executed

PSR Composite Registers

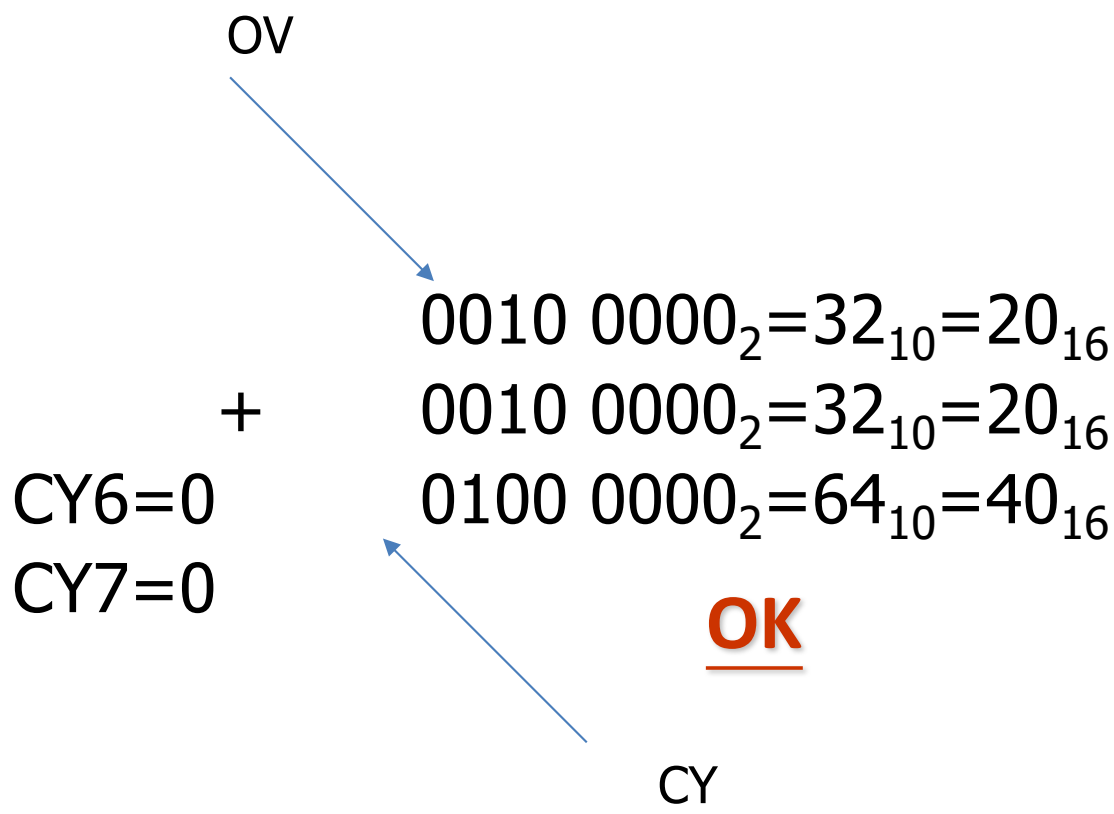
- xPSR: 可以一次性存取APSR, IPSR and EPSR
  - xPSR
    - Composite register of APSR, IPSR and EPSR



  - IEPSR
    - Composite register of IPSR and EPSR



OV


OV
0010 0000$_2$=32$_{10}$=20$_{16}$
+ 0010 0000$_2$=32$_{10}$=20$_{16}$
CY6=0 0100 0000$_2$=64$_{10}$=40$_{16}$
CY7=0
OK
CY

OV

$1111\ 1111_2 = -1_{10} = FF_{16}$

$+ \quad 1111\ 1111_2 = -1_{10} = FF_{16}$

CY6=1 $\quad 1111\ 1110_2 = -2_{10} = FE_{16}$

CY7=1

OK

OV

$0100\ 0000_2 = 64_{10} = 40_{16}$

\+ $0100\ 0000_2 = 64_{10} = 40_{16}$

CY6=1 $0000\ 0000_2 = 00_{10} = 00_{16} = (128_{10})$

CY7=0

OVER

OV

$1000\ 0001_2 = -127_{10} = 81_{16}$

\+ $1000\ 0001_2 = -127_{10} = 81_{16}$

CY6=0 $0000\ 0010_2 = 2_{10} = 2_{16} = (-254_{10})$

CY7=1

OVER

Basic assembly example

```
          PRESERVE8 ; Indicate the code here preserve
                    ; 8 byte stack alignment
          THUMB     ; Indicate THUMB code is used
          AREA    |.text|, CODE, READONLY   ; Start of CODE area
My_Add    FUNCTION
          ADDS    R0, R0, R1  ; Add R0 and R1, result store in R0
          BX      LR          ; Return
          ENDFUNC

          END                 ; End of file
```

Assembly syntax

- UAL: Unified Assembler Language
  - To allow better portability between architectures and to use a single assembly language syntax between different ARM processors with various architectures

For example, a pre-UAL ADD instruction for 16-bit Thumb code is

```
ADD    R0, R1   ; R0 = R0 + R1, update APSR
```

With UAL syntax, this should be written as

```
ADDS   R0, R0, R1   ; R0 = R0 + R1, update APSR
```

Assembly syntax
ARM Keil V.S. GNU Tool

ARM Keil

```
label
     mnemonic     operand1, operand2,...     ; Comments
```

Constant define

```
NVIC_IRQ_SETEN      EQU   0xE000E100
NVIC_IRQ0_ENABLE    EQU   0x1
```

Memory align

```
ALIGN   4
```

GNU

```
label:
     mnemonic     operand1, operand2,...     /* Comments */
```

```
    .equ   NVIC_IRQ_SETEN,      0xE000E100
    .equ   NVIC_IRQ0_ENABLE,    0x1

    .align 4
```

Table 5.3: Commonly Used Directives for Inserting Data into a Program

Type of Data to Insert	ARM Assembler	GNU Assembler
Word	DCD (e.g., DCD 0x12345678)	.word / .4byte (e.g., .word 0x012345678)
Half word	DCW (e.g., DCW 0x1234)	.hword / .2byte (e.g., .hword 0x01234)
Byte	DCB (e.g., DCB 0x12)	.byte (e.g., .byte 0x012)
String	DCB (e.g., TXT DCB “Hello\n”, 0)	.ascii /.asciz (with NULL termination) (e.g., .ascii “Hello\n” .byte 0 /* add NULL character */) (e.g., .asciz “Hello\n”)
Instruction	DCI (e.g., DCI 0xBE00 ; Breakpoint-BKPT 0)	.word /.hword (e.g., .hword 0xBE00 /* Breakpoint (BKPT 0) */)

Assembly Instructions Class

- Branch instructions
  - B, BL, BX,…
- Data-processing instructions
  - MOV, ADD, SUB, MUL,…
- Load and store instructions
  - LDR, STR,…
- Status register access instructions
  - MSR, MRS,…
- Miscellaneous instructions
  - Memory Barrier Instructions
  - Exception-Related Instructions
  - Pseudo instructions

Data Types

Type of Data to Insert	ARM Assembler	GNU Assembler
Word	DCD (e.g., DCD 0x12345678)	.word / .4byte (e.g., .word 0x012345678)
Half word	DCW (e.g., DCW 0x1234)	.hword / .2byte (e.g., .hword 0x01234)
Byte	DCB (e.g., DCB 0x12)	.byte (e.g., .byte 0x012)
String	DCB (e.g., TXT DCB “Hello\n”, 0)	.ascii /.asciz (with NULL termination) (e.g., .ascii “Hello\n” .byte 0 /* add NULL character */) (e.g., .asciz “Hello\n”)
Instruction	DCI (e.g., DCI 0xBE00 ; Breakpoint-BKPT 0)	.word /.hword (e.g., .hword 0xBE00 /* Breakpoint (BKPT 0) */)

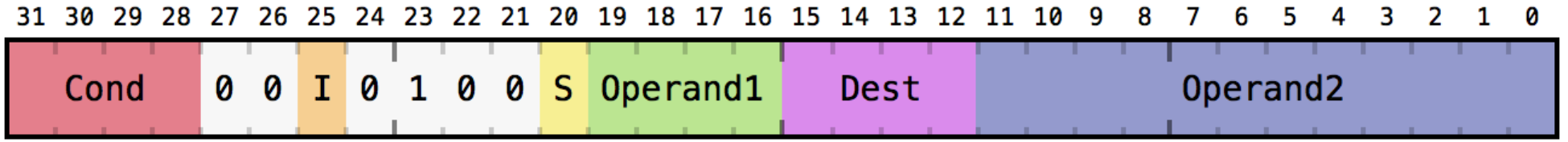
Assembly Syntax

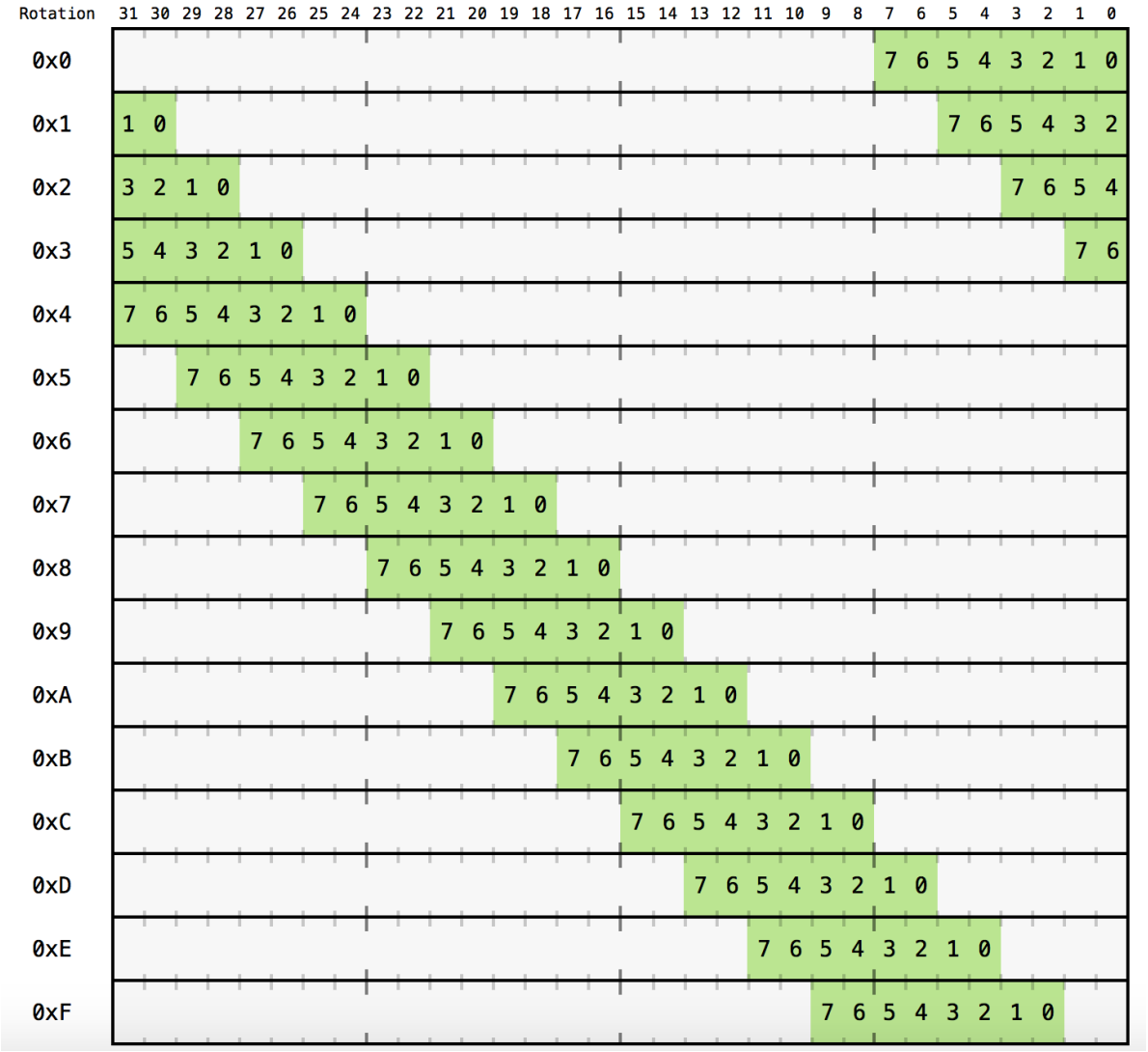
```
        MOVS     R0, #0x12   ; Set R0 = 0x12 (hexadecimal)
        MOVS     R1, #'A'    ; Set R1 = ASCII character A

        NVIC_IRQ_SETEN      EQU   0xE000E100
        NVIC_IRQ0_ENABLE    EQU   0x1
```

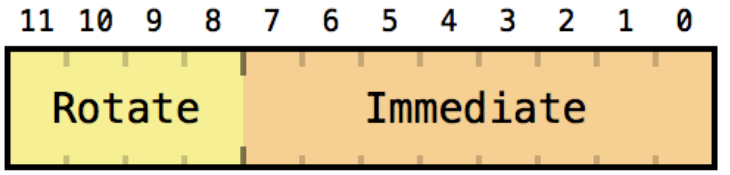
LDR r0,=*label*

```
...
LDR     R0,=NVIC_IRQ_SETEN    ; Put 0xE000E100 into R0
        ; LDR here is a pseudo instruction that will be converted
        ; to a PC relative literal data load by the assembler
MOVS    R1, #NVIC_IRQ0_ENABLE ; Put immediate data (0x1) into
                              ; register R1
STR     R1,[ R0] ; Store 0x1 to 0xE000E100, this enable external
                 ; interrupt IRQ#0
```

LDR rn [pc, #offset to literal pool] ; load register n with one word ; from the address [pc + offset]


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Cond
0 0 I 0 1 0 0 S
Operand1
Dest
Operand2


Rotation 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0x0 7 6 5 4 3 2 1 0
0x1 1 0 7 6 5 4 3 2
0x2 3 2 1 0 7 6 5 4
0x3 5 4 3 2 1 0 7 6
0x4 7 6 5 4 3 2 1 0
0x5 7 6 5 4 3 2 1 0
0x6 7 6 5 4 3 2 1 0
0x7 7 6 5 4 3 2 1 0
0x8 7 6 5 4 3 2 1 0
0x9 7 6 5 4 3 2 1 0
0xA 7 6 5 4 3 2 1 0
0xB 7 6 5 4 3 2 1 0
0xC 7 6 5 4 3 2 1 0
0xD 7 6 5 4 3 2 1 0
0xE 7 6 5 4 3 2 1 0
0xF 7 6 5 4 3 2 1 0


11 10 9 8 7 6 5 4 3 2 1 0
Rotate
Immediate

Second Operand : Immediate Value (1)

* **There is no single instruction which will load a 32 bit immediate constant into a register without performing a data load from memory.**
  - All ARM instructions are 32 bits long
  - ARM instructions do not use the instruction stream as data.
* **The data processing instruction format has 12 bits available for operand2**
  - If used directly this would only give a range of 4096.
* **Instead it is used to store 8 bit constants, giving a range of 0 - 255.**
* **These 8 bits can then be rotated right through an even number of positions (ie RORs by 0, 2, 4,..30).**
  - This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory.

Second Operand : Immediate Value (2)

* **This gives us:**
  - 0 - 255 [0 - 0xff]
  - 256,260,264,..,1020 [0x100-0x3fc, step 4, 0x40-0xff ror 30]
  - 1024,1040,1056,..,4080 [0x400-0xff0, step 16, 0x40-0xff ror 28]
  - 4096,4160, 4224,..,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror 26]
* **These can be loaded using, for example:**
  - MOV r0, #0x40, 26 ; => MOV r0, #0x1000 (ie 4096)
* **To make this easier, the assembler will convert to this form for us if simply given the required constant:**
  - MOV r0, #4096 ; => MOV r0, #0x1000 (ie 0x40 ror 26)
* **The bitwise complements can also be formed using MVN:**
  - MOV r0, #0xFFFFFFFF ; assembles to MVN r0, #0
* **If the required constant cannot be generated, an error will be reported.**

Assembly Syntax (Cont.)

```
      LDR     R3,=MY_NUMBER  ; Get the memory location of MY_NUMBER
      LDR     R4, [ R3]      ; Read the value 0x12345678 into R4
      ...
      LDR     R0,=HELLO_TEXT ; Get the starting address of HELLO_TEXT
      BL      PrintText      ; Call a function called PrintText to
                             ; display string
      ...
      ALIGN   4
MY_NUMBER    DCD  0x12345678
HELLO_TEXT   DCB  "Hello\n", 0 ; Null terminated string
```

Data Processing Instructions

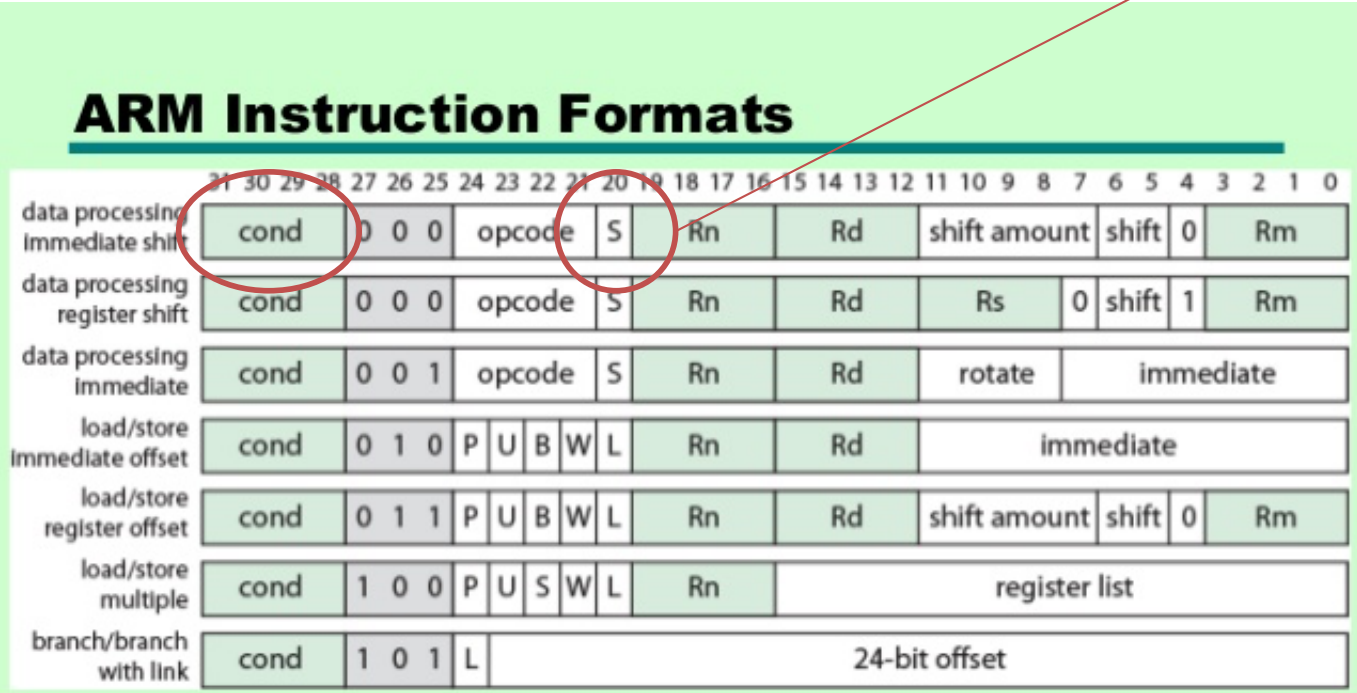
- Standard Data Processing Instructions
  - **ADD, ADC, SUB, SBC, RSB**
  - **AND, ORR, EOR, BIC**
  - **MOV, MVN**
  - **TST, CMP, CMN**
  - **ADR** (Pseudo Instruction)
- Shift and Rotate Instructions
  - **ASR**
  - **LSL, LSR**
  - **ROR**
- Multiply Instruction
  - **MUL**
- Sign/Unsign Extend Instructions
  - **SXTB, SXTH, UXTB, UXTH**
- Miscellaneous Data Processing
  - **REV, REV16, REVSH**

Examples

```
SUBS r0,#1        (r0 ← r0 - 1)
ORRS r0,r1        (r0 ← r0 | r1)
MOVS r0,#1        (r0 ← r0 + 1)
CMP  r0,r1
RSBS r0,r1,#0     (r0 ← -r1)
ADR  r0, Start    (r0 ← [Start])



ASRS r0,r1,#7     (r0 ← r1 >> 7)
LSLS r0,r1,#3     (r0 ← r1 << 3)
RORS r0,r1        (r0 ← r0 >> r1)

MULS r0,r1,r0     (r0 ← r1 * r0)

UXTB r0,r1        (r0 ← r1[7:0])

REV r0,r1         Byte Swap
```

ADD instruction family

ADD and ADDS



- S = For data processing instructions, updates condition codes
- S = For load/store multiple instructions, execution restricted to supervisor mode
- P, U, W = distinguish between different types of addressing_mode
- B = Unsigned byte (B==1) or word (B==0) access
- L = For load/store instructions, Load (L==1) or Store (L==0)
- L = For branch instructions, is return address stored in link register

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

Conditional Execution

```
while (a!=b)
{
  if (a>b) a -= b; else b -= a;
}
```

```
gcd
        CMP     r1, r2
        SUBGT   r1, r1, r2
        SUBLT   r2, r2, r1
        BNE     gcd
```

```
        ; Greatest Common Divisor Algorithm
gcd
        CMP     r1, r2
        BEQ     complete
        BLT     lessthan
        SUB     r1, r1, r2
        B       gcd

lessthan
        SUB     r2, r2, r1
        B       gcd

complete
...
```

Data Processing Instructions

- Arithmetic Operations
- Logic Operations
- Shift and Rotate Operations
  - ASR, LSL, LSR, ROR
- Reverse Ordering Operations
  - REV, REV16, REVSH
- Signed Extended Operations
  - SXTB, SXTH, UXTB, UXTH

Arithmetic Operations

Instruction	ADD
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD <Rd>, SP, <Rd>
Syntax (Thumb)	ADD <Rd>, SP
Note	Rd = Rd + SP. Rd can be high or low register.

Instruction	ADD
Function	Add an immediate constant into a register
Syntax (UAL)	ADDS <Rd>, <Rn>, #immed3 ADDS <Rd>, #immed8
Syntax (Thumb)	ADD <Rd>, <Rn>, #immed3 ADD <Rd>, #immed8
Note	Rd = Rn + ZeroExtend(#immed3), APSR update, or Rd = Rd + ZeroExtend(#immed8), APSR update. Rd, Rn, Rm are low registers.

Instruction	ADD
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD SP, <Rm>
Syntax (Thumb)	ADD SP, <Rm>
Note	SP = SP + Rm. Rm can be high or low register.

Instruction	ADD
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD <Rd>, SP, #immed8
Syntax (Thumb)	ADD <Rd>, SP, #immed8
Note	Rd = SP + ZeroExtend(#immed8 <<2). Rd is a low register.

Instruction	ADD
Function	Add two registers
Syntax (UAL)	ADDS <Rd>, <Rn>, <Rm>
Syntax (Thumb)	ADD <Rd>, <Rn>, <Rm>
Note	Rd = Rn + Rm, APSR update. Rd, Rn, Rm are low registers.

Instruction	ADD
Function	Add an immediate constant to stack pointer
Syntax (UAL)	ADD SP, SP, #immed7
Syntax (Thumb)	ADD SP, #immed7
Note	SP = SP + ZeroExtend(#immed7 <<2). This instruction is useful for C functions to adjust the SP for local variables.

Arithmetic Operations

Instruction	ADR (ADD)
Function	Add an immediate constant with PC to a register without updating APSR
Syntax (UAL)	ADR <Rd>, <label> (normal syntax)
	ADD <Rd>, PC, #immed8 (alternate syntax)
Syntax (Thumb)	ADR <Rd>, (normal syntax)
	ADD <Rd>, PC, #immed8 (alternate syntax)
Note	Rd = (PC[31:2]<<2) + ZeroExtend(#immed8 <<2).
	This instruction is useful for locating a data address within the program memory near to the current instruction. The result address must be word aligned.
	Rd is a low register.

Instruction	ADC
Function	Add with carry and update APSR
Syntax (UAL)	ADCS <Rd>, <Rm>
Syntax (Thumb)	ADC <Rd>, <Rm>
Note	Rd = Rd + Rm + Carry
	Rd and Rm are low registers.

64-bit addition

```
ADDS    R4, R0, R2    ; add the least significant words
ADC     R5, R1, R3    ; add the most significant words with carry
```

Arithmetic Operations

- ADD: Add without updating APSR
- ADR: Add an immediate constant with PC to a register without updating APSR
- ADC: Add with carry and update APSR
- SUB: Subtract
- SBC: Subtract with carry (borrow)
- RSB: Reverse Subtract (negative)
- MUL: Multiply
- CMP: Compare
  - CMP <Rn>, #immed8
- CMN: Compare negative
  - CMN <Rn>, <Rm>

Instruction	CMP
Function	Compare
Syntax (UAL)	CMP <Rn>, <Rm>
Syntax (Thumb)	CMP <Rn>, <Rm>
Note	Calculate Rn – Rm, APSR update but subtract result is not stored.

User Suffix

Suffix	Descriptions
S	Update APSR (flags); for example, `ADDS R0, R1; this ADD operation will update APSR`
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Conditional execution. EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, etc. On the Cortex-M0 processor, these conditions can only be applied to conditional branches. For example, `BEQ label; branch to label if equal`

```
MOVS    R0, R1  ; Move R1 into R0 and update APSR
```

or

```
MOV     R0, R1  ; Move R1 into R0
```

```
ADDNE R0, R0, R2
```

```
ADDNE R0, R0, R2
ADDS R0, R0, R2
ADDNES R0, R0, R2
```

Compare

Instruction	**CMP**
Function	Compare
Syntax (UAL)	CMP <Rn>, #immed8
Syntax (Thumb)	CMP <Rn>, #immed8
Note	Calculate Rn - ZeroExtended(#immed8), APSR update but subtract result is not stored. Rn is a low register.

Instruction	**CMN**
Function	Compare negative
Syntax (UAL)	CMN <Rn>, <Rm>
Syntax (Thumb)	CMN <Rn>, <Rm>
Note	Calculate Rn – NEG(Rm), APSR update but subtract result is not stored. Effectively the operation is an ADD.

Logic Operations

- AND (Logical AND)
  - ANDS <Rd>, <Rd>, <Rm>
    - Rd = AND(Rd, Rm)
- ORR (Logical OR)
- EOR (Logical Exclusive OR)
- BIC (Logical Bitwise Clear)
  - BICS <Rd>, <Rd>, <Rm>
    - Rd = AND(Rd, NOT(Rm))
- MVN (Logical Bitwise NOT)
  - MVNS <Rd>, <Rm>
    - Rd = NOT(Rm)
- TST (Test (bitwise AND))
  - Calculate AND(Rn, Rm), APSR.N, and APSR.Z update, but the AND result is not stored.

Logic operations

Instruction	AND
Function	Logical AND
Syntax (UAL)	ANDS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	AND <Rd>, <Rm>
Note	Rd = AND(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	BIC
Function	Logical Bitwise Clear
Syntax (UAL)	BICS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	BIC <Rd>, <Rm>
Note	Rd = AND(Rd, NOT(Rm)), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	ORR
Function	Logical OR
Syntax (UAL)	ORRS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	ORR <Rd>, <Rm>
Note	Rd = OR(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	MVN
Function	Logical Bitwise NOT
Syntax (UAL)	MVNS <Rd>, <Rm>
Syntax (Thumb)	MVN <Rd>, <Rm>
Note	Rd = NOT(Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	EOR
Function	Logical Exclusive OR
Syntax (UAL)	EORS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	EOR <Rd>, <Rm>
Note	Rd = XOR(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	TST
Function	Test (bitwise AND)
Syntax (UAL)	TST <Rn>, <Rm>
Syntax (Thumb)	TST <Rn>, <Rm>
Note	Calculate AND(Rn, Rm), APSR.N, and APSR.Z update, but the AND result is not stored. Rd and Rm are low registers.

Shift and Rotate Operations

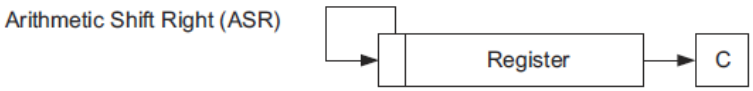


Figure 5.3:
Arithmetic Shift Right.

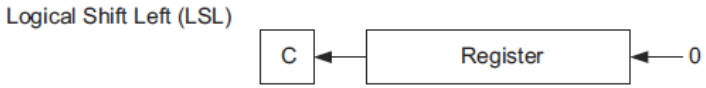


Figure 5.4:
Logical Shift Left.

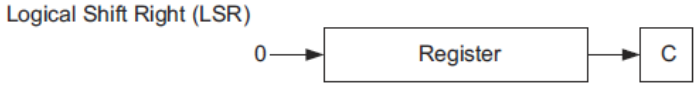


Figure 5.5:
Logical Shift Right.

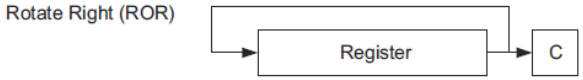


Figure 5.6:
Rotate Right.

Instruction	**ASR**
Function	Arithmetic Shift Right
Syntax (UAL)	ASRS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	ASR <Rd>, <Rm>
Note	Rd = Rd >> Rm, last bit shift out is copy to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

Reverse Ordering Operations

Instruction	REV (Byte-Reverse Word)
Function	Byte Order Reverse
Syntax	REV <Rd>, <Rm>
Note	Rd = {Rm[7:0] , Rm[15:8], Rm[23:16], Rm[31:24]} Rd and Rm are low registers.

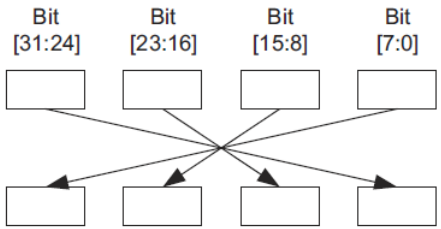


Figure 5.7:
REV operation.

Instruction	REV16 (Byte-Reverse Packed Half Word)
Function	Byte Order Reverse within half word
Syntax	REV16 <Rd>, <Rm>
Note	Rd = {Rm[23:16], Rm[31:24], Rm[7:0] , Rm[15:8]} Rd and Rm are low registers.

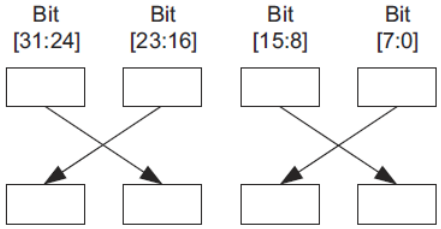


Figure 5.8:
REV16 operation.

Signed Extended Operations

With SXTB or SXTH, the data are extended using bit[7] or bit[15] of the input data, whereas for UXTB and UXTH, the data are extended using zeros. For example, if R0 is 0x55AA8765, the result of these extended instructions is

```
SXTB    R1, R0      ; R1 = 0x00000065
SXTH    R1, R0      ; R1 = 0xFFFF8765
UXTB    R1, R0      ; R1 = 0x00000065
UXTH    R1, R0      ; R1 = 0x00008765
```

MOV

Instruction	MOV
Function	Move register into register
Syntax (UAL)	MOV <Rd>, <Rm>
Syntax (Thumb)	MOV <Rd>, <Rm>
	CPY <Rd>, <Rm>
Note	Rm and Rn can be high or low registers
	CPY is a pre-UAL synonym for MOV (register)

Instruction	MOV
Function	Move immediate data (sign extended) into register
Syntax (UAL)	MOVS <Rd>, #immed8
Syntax (Thumb)	MOV <Rd>, #immed8
Note	Immediate data range 0 to +255
	APSR.Z and APSR.N update

Instruction	MOVS/ADDS
Function	Move register into register
Syntax (UAL)	MOVS <Rd>, <Rm>
	ADDS <Rd>, <Rm>, #0
Syntax (Thumb)	MOVS <Rd>, <Rm>
Note	Rm and Rn are both low registers
	APSR.Z, APSR.N, and APSR.C (for ADDS) update

Instruction	MRS
Function	Move Special Register into register
Syntax	MRS <Rd>, <SpecialReg>
Note	Example:
	MRS R0, CONTROL; Read CONTROL register into R0
	MRS R9, PRIMASK; Read PRIMASK register into R9
	MRS R3, xPSR; Read xPSR register into R3

Instruction	MSR
Function	Move register into Special Register
Syntax	MSR <SpecialReg>, <Rd>
Note	Example:
	MSR CONTROL, R0; Write R0 into CONTROL register
	MSR PRIMASK, R9; Write R9 into PRIMASK register

Load and Store Instructions

In C
ro=*r1

- Unsigned Loads/Stores
  - **LDR/STR**
  - **LDRH/STRH**
  - **LDRB/STRB**
- Signed Loads
  - **LDRSH**
  - **LDRSB**

Examples

```
LDR r0,[r1]        (r0 ← [r1])

STM r0,{r1,r2}  (r1 → [r0])  (r2 → [r0+4])
LDM r0,{r1,r2}  (r1 ← [r0])  (r2 ← [r0+4])

PUSH {r1,r2}       (r1 → [SP], r2 → [SP+4])
POP {r1,r2}        (r1 ← [SP], r2 ← [SP-4])
```

- Load/Stores Multiple
  - **LDM, LDMIA/LDMFD** (same as LDM, but with base register update option)
  - **STM, STMIA/STMEA** (same as STM, but with base register update option)
  - **PUSH, POP**
    - Uses "**Full Descending (FD)**" stack, SP always points to last filled data and SP predecrements for each PUSH)

Memory Accesses

Transfer Size	Unsigned Load	Signed Load	Signed/Unsigned Store
Word	LDR	LDR	STR
Half word	LDRH	LDRSH	STRH
Byte	LDRB	LDRSB	STRB

- It is important to make sure the memory address accessed is aligned
  - a word size access can only be carried out on address locations when address bits[1:0] are set to zero

```
MOVS R0, #0x02
LDR R1, [R0]
```
X

  - a half word size access can only be carried out on address locations when an address bit[0] is set to zero

```
MOVS R0, #0x01
LDRH R1, [R0]
```
X

Data Alignment

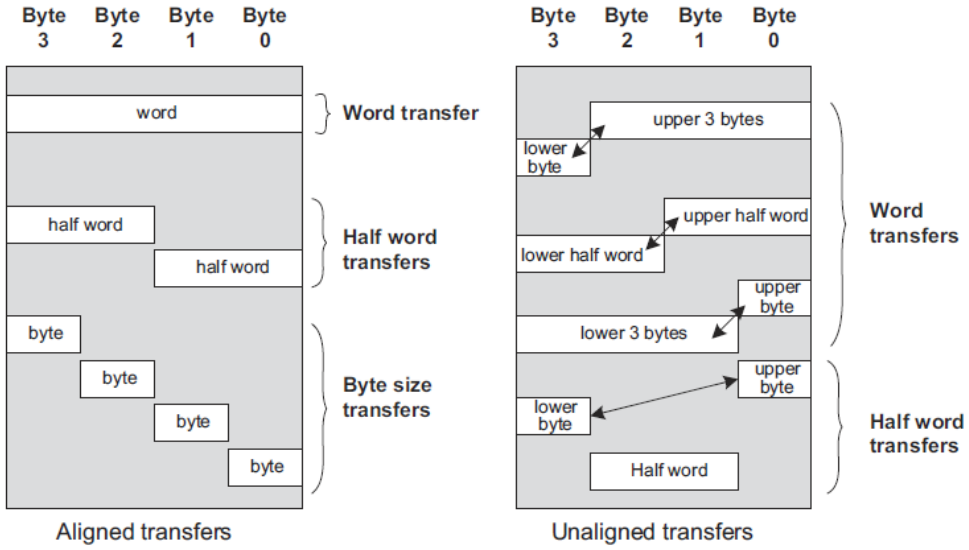


Figure 7.11:
Example of aligned and unaligned transfers (little endian memory).

Memory Accesses

Instruction	LDR/LDRH/LDRB
Function	Read single memory data into register
Syntax	LDR <Rt>, [<Rn>, <Rm>] ; Word read
	LDRH <Rt>, [<Rn>, <Rm>] ; Half Word read
	LDRB <Rt>, [<Rn>, <Rm>] ; Byte read
Note	Rt = memory[Rn + Rm]
	Rt, Rn and Rm are low registers

Instruction	LDR
Function	Read single memory data word into register
Syntax	LDR <Rt>, [PC, #immed8] ; Word read
Note	Rt = memory[WordAligned(PC+4) + ZeroExtend(#immed8 << 2)]
	Rt is a low register, and targeted address must be a word-aligned address, the reason for adding 4.

Instruction	LDR/LDRH/LDRB
Function	Read single memory data into register
Syntax	LDR <Rt>, [<Rn>, #immed5] ; Word read
	LDRH <Rt>, [<Rn>, #immed5] ; Half Word read
	LDRB <Rt>, [<Rn>, #immed5] ; Byte read
Note	Rt = memory[Rn + ZeroExtend(#immed5 << 2)] ; Word
	Rt = memory[Rn + ZeroExtend(#immed5 << 1)] ; Half word
	Rt = memory[Rn + ZeroExtend(#immed5)] ; Byte
	Rt and Rn are low registers

Instruction	LDRSH/LDRSB
Function	Read single signed memory data into register
Syntax	LDRSH <Rt>, [<Rn>, <Rm>] ; Half word read
	LDRSB <Rt>, [<Rn>, <Rm>] ; Byte read
Note	Rt = SignExtend(memory[Rn + Rm])
	Rt, Rn and Rm are low registers

Instruction	LDR
Function	Read single memory data word into register
Syntax	LDR <Rt>, [SP, #immed8] ; Word read
Note	Rt = memory[SP + ZeroExtend(#immed8 << 2)]
	Rt is a low register

```
00 1010 ZeroExtend 0000 0000 0000 1010
11 1111 0001 SignExtend 1111 1111 1111 0001
```

LDR Examples

Instruction	LDR
Example:	
LDR R0,=0x12345678	; A pseudo instruction that uses literal load ; to put an immediate data into a register
LDR R0, [PC, #0x40]	; Load a data in current program address ; with offset of 0x40 into R0
LDR R0, label	; Load a data in current program ; referenced by label into R0

Memory Accesses

Instruction	STR/STRH/STRB
Function	Write single memory data into memory
Syntax	STR <Rt>, [<Rn>, #immed5] ; Word write
	STRH <Rt>, [<Rn>, #immed5] ; Half Word write
	STRB <Rt>, [<Rn>, #immed5] ; Byte write
Note	memory[Rn + ZeroExtend(#immed5 << 2)] = Rt ; Word
	memory[Rn + ZeroExtend(#immed5 << 1)] = Rt ; Half word
	memory[Rn + ZeroExtend(#immed5)] = Rt ; Byte
	Rt and Rn are low registers

Instruction	STR/STRH/STRB
Function	Write single register data into memory
Syntax	STR <Rt>, [<Rn>, <Rm>] ; Word write
	STRH <Rt>, [<Rn>, <Rm>] ; Half Word write
	STRB <Rt>, [<Rn>, <Rm>] ; Byte write
Note	memory[Rn + Rm] = Rt
	Rt, Rn and Rm are low registers

Instruction	STR
Function	Write single memory data word into memory
Syntax	STR <Rt>, [SP, #immed8] ; Word write
Note	memory[SP + ZeroExtend(#immed8 << 2)] = Rt
	Rt is a low register

Memory Accesses

Instruction	LDM (Load Multiple)
Function	Read multiple memory data word into registers, base address register update by memory read
Syntax	LDM <Rn>, {<Ra>, <Rb> ,....} ; Load multiple registers from memory
Note	Ra = memory[Rn], Rb = memory[Rn+4], ... Rn, Ra, Rb .... are low registers. Rn is on the list of registers to be updated by memory read. For example, LDM R2, {R1, R2, R5 – R7} ; Read R1,R2,R5,R6 and R7 from memory.

LDM/STM operation

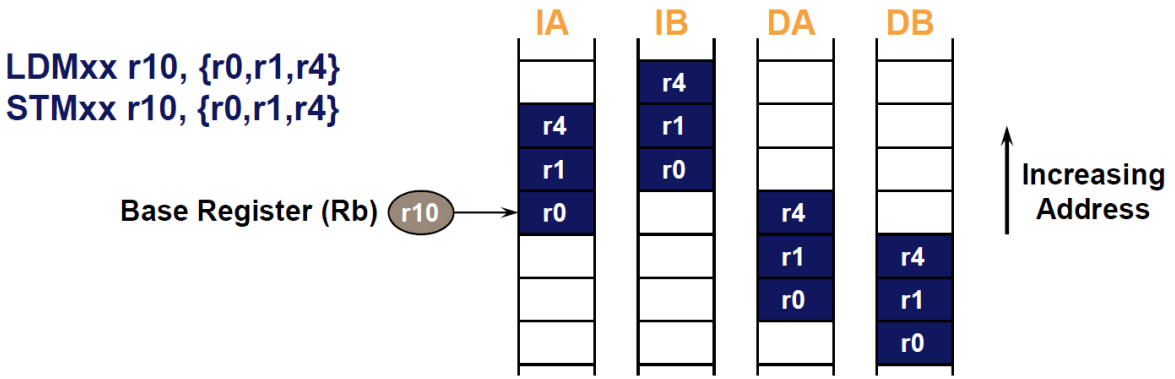
- Load/Store multiple data

  - Syntax:

    `<LDM|STM>`{<cond>}<addressing_mode> Rb{!}, <register list>

  - 4 addressing modes **(NOT ALL SUPPORTED IN v6-M!!!!)**:

`LDMIA / STMIA`	increment after
`LDMIB / STMIB`	increment before
`LDMDA / STMDA`	decrement after
`LDMDB / STMDB`	decrement before



Stack Memory Accesses

Instruction	PUSH
Function	Write single or multiple registers (low register and LR) into memory and update base register (stack pointer)
Syntax	PUSH {<Ra>, <Rb> ,....} ; Store multiple registers to memory and ; decrement SP to the lowest pushed data address PUSH {<Ra>, <Rb>, ..., LR} ; Store multiple registers and LR to ; memory and decrement SP to the lowest pushed data address

Instruction	PUSH
Note	memory[SP-4] = Ra, memory[SP-8] = Rb, ... and then update SP to last store address. For example, PUSH {R1, R2, R5 – R7, LR} ; Store R1, R2, R5, R6, R7, and LR to stack

Instruction	POP
Function	Read single or multiple registers (low register and PC) from memory and update base register (stack pointer)
Syntax	POP {<Ra>, <Rb> ,....} ; Load multiple registers from memory ; and increment SP to the last emptied stack address plus 4 POP {<Ra>, <Rb>, ..., PC} ; Load multiple registers and PC from ; memory and increment SP to the last emptied stack ; address plus 4
Note	Ra = memory[SP], Rb = memory[SP+4], ... and then update SP to last restored address plus 4. For example, POP {R1, R2, R5 – R7} ; Restore R1, R2, R5, R6, R7 from stack

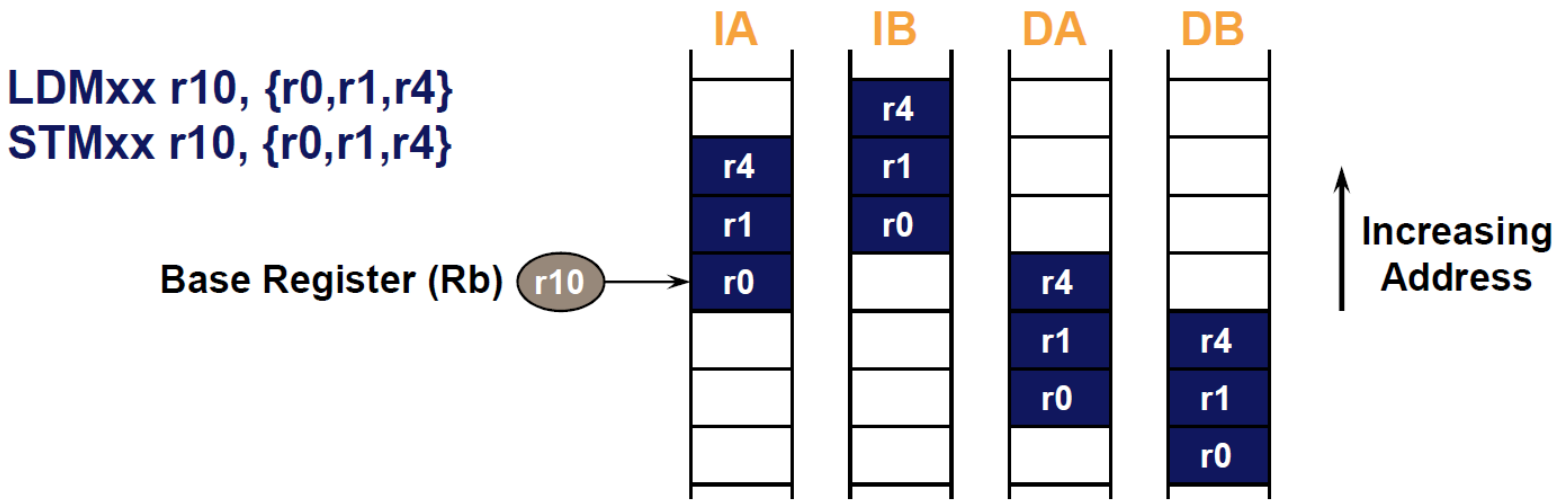
```
my_function
  PUSH { R4, R5, R7, LR} ; Save R4, R5, R7 and LR (return address)
  ... ; function body
  POP  { R4, R5, R7, PC} ; Restore R4, R5, R7 and return
```

LDM/STM operation

- Load/Store multiple data
  - Syntax:

    `<LDM|STM>`{<cond>}<addressing_mode> Rb{!}, <register list>
  - 4 addressing modes **(NOT ALL SUPPORTED IN v6-M!!!!)**:

`LDMIA / STMIA`	increment after
`LDMIB / STMIB`	increment before
`LDMDA / STMDA`	decrement after
`LDMDB / STMDB`	decrement before



Status Register Access Instructions

- MRS/MSR - Move data between a general purpose register and status register
  - **MRS** (Register ← Status Register)
  - **MSR** (Status Register ← Register)

Examples

```
MRS  r0, IPSR      (r0 ← IPSR )
MSR  APSR, r0      (APSR ← r0)
```

- CPS – Change Processor State
  - Allows the enable/disable interrupts

Examples

```
CPSIE      i       (CPS Interrupt Enable)
CPSID      i       (CPS Interrupt Disable (except NMI and Hard Fault))
```

Data access

- In assembly you can access some data via different ways.
  - Moving data within the processor
  - Memory accesses
  - Stack memory accesses

Moving data within the processor (1/2)

- Instruction: MOV, MOVS
  - Copy between registers
  - Example: MOV r1, r2

<Rm> is source register
<Rd> is destination register

Instruction	MOV
Function	Move register into register
Syntax (UAL)	MOV <Rd>, <Rm>
Syntax (Thumb)	MOV <Rd>, <Rm>
	CPY <Rd>, <Rm>
Note	Rm and Rn can be high or low registers
	CPY is a pre-UAL synonym for MOV (register)

Instruction	MOV
Function	Move immediate data (sign extended) into register
Syntax (UAL)	MOVS <Rd>, #immed8
Syntax (Thumb)	MOV <Rd>, #immed8
Note	Immediate data range 0 to +255
	APSR.Z and APSR.N update

Moving data within the processor (2/2)

- Instruction: MRS, MSR
  - Access special registers
  - Example: MRS r1, msp

Instruction	MRS
Function	Move Special Register into register
Syntax	MRS <Rd>, <SpecialReg>
Note	Example: MRS R0, CONTROL; Read CONTROL register into R0 MRS R9, PRIMASK; Read PRIMASK register into R9 MRS R3, xPSR; Read xPSR register into R3

Instruction	MSR
Function	Move register into Special Register
Syntax	MSR <SpecialReg>, <Rd>
Note	Example: MSR CONTROL, R0; Write R0 into CONTROL register MSR PRIMASK, R9; Write R9 into PRIMASK register

Data memory access

- Note: in SAM D21 process use Little-endian as memory format.

Bits	[31:24]	[23:16]	[15:8]	[7:0]
0x00000008	Byte 0xB	Byte 0xA	Byte 9	Byte 8
0x00000004	Byte 7	Byte 6	Byte 5	Byte 4
0x00000000	Byte 3	Byte 2	Byte 1	Byte 0

Figure 7.7:
Little endian 32-bit memory.

Address	Size	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x00000000	Word	Data[31:24]	Data[23:16]	Data[15:8]	Data[7:0]
0x00000000	Half word			Data[15:8]	Data[7:0]
0x00000002	Half word	Data[15:8]	Data[7:0]		
0x00000000	Byte				Data[7:0]
0x00000001	Byte			Data[7:0]	
0x00000002	Byte		Data[7:0]		
0x00000003	Byte	Data[7:0]			

Figure 7.9:
Data access in little endian systems.

Memory access and its corresponding address

Memory access instructions

STR: Store data into memory
LDR: Load data from memory

Table 7.1: Commonly Used Data Types in C Language Development

Type	Number of Bits in ARM	Instructions
"char", "unsigned char"	8	LDRB, LDRSB, STRB
"enum"	8/16/32 (smallest is chosen)	LDRB, LDRH, LDR, STRB, STRH, STR
"short", "unsigned short"	16	LDRH, LDRSH, STRH
"int", "unsigned int"	32	LDR, STR
"long", "unsigned long"	32	LDR, STR

If "stdint.h" in C99 is used, the data types shown in Table 7.2 are available.

Table 7.2: Commonly Used Data Types Provided in "stdint.h" in C99

Type	Number of Bits in ARM	Instructions
"int8_t", "uint8_t"	8	LDRB, LDRSB, STRB
"int16_t", "uint16_t"	16	LDRH, LDRSH, STRH
"int32_t", "uint32_t"	32	LDR, STR

Read single memory data into register

- LDR <Rt>, [<Rn>, <Rm>] ; Word read
- LDR <Rt>, [<Rn>, #immed5] ; Word read
- LDR <Rt>, [PC, #immed8]
  - LDR R0,=0x12345678
    - A pseudo instruction that uses literal load to put an immediate data into a register
  - LDR R0, label
    - Load a data in current program referenced by label into R0
  - LDR R0, [PC, #0x40]
- LDR <Rt>, [SP, #immed8]
  - Rt = memory[SP, ZeroExtend(#immed8 << 2)]

Write single register data into memory

- Similar read data but use STR instruction

Instruction	STR/STRH/STRB
Function	Write single register data into memory
Syntax	STR <Rt>, [<Rn>, <Rm>] ; Word write
	STRH <Rt>, [<Rn>, <Rm>] ; Half Word write
	STRB <Rt>, [<Rn>, <Rm>] ; Byte write
Note	memory[Rn + Rm] = Rt
	Rt, Rn and Rm are low registers

Instruction	STR/STRH/STRB
Function	Write single memory data into memory
Syntax	STR <Rt>, [<Rn>, #immed5] ; Word write
	STRH <Rt>, [<Rn>, #immed5] ; Half Word write
	STRB <Rt>, [<Rn>, #immed5] ; Byte write
Note	memory[Rn + ZeroExtend(#immed5 << 2)] = Rt ; Word
	memory[Rn + ZeroExtend(#immed5 << 1)] = Rt ; Half word
	memory[Rn + ZeroExtend(#immed5)] = Rt ; Byte
	Rt and Rn are low registers

Multiple data read/write

- Instructions: **LDMIA**(Load Multiple Increment After), **STMIA**(Store Multiple Increment After)
- Syntax: LDMIA <Rn>!, {<Ra>, <Rb> ,..}
- Note:
  - Example: LDMIA R0!, {R1,R2}
  - R1 = memory[R0], R2=memory[R0+4]

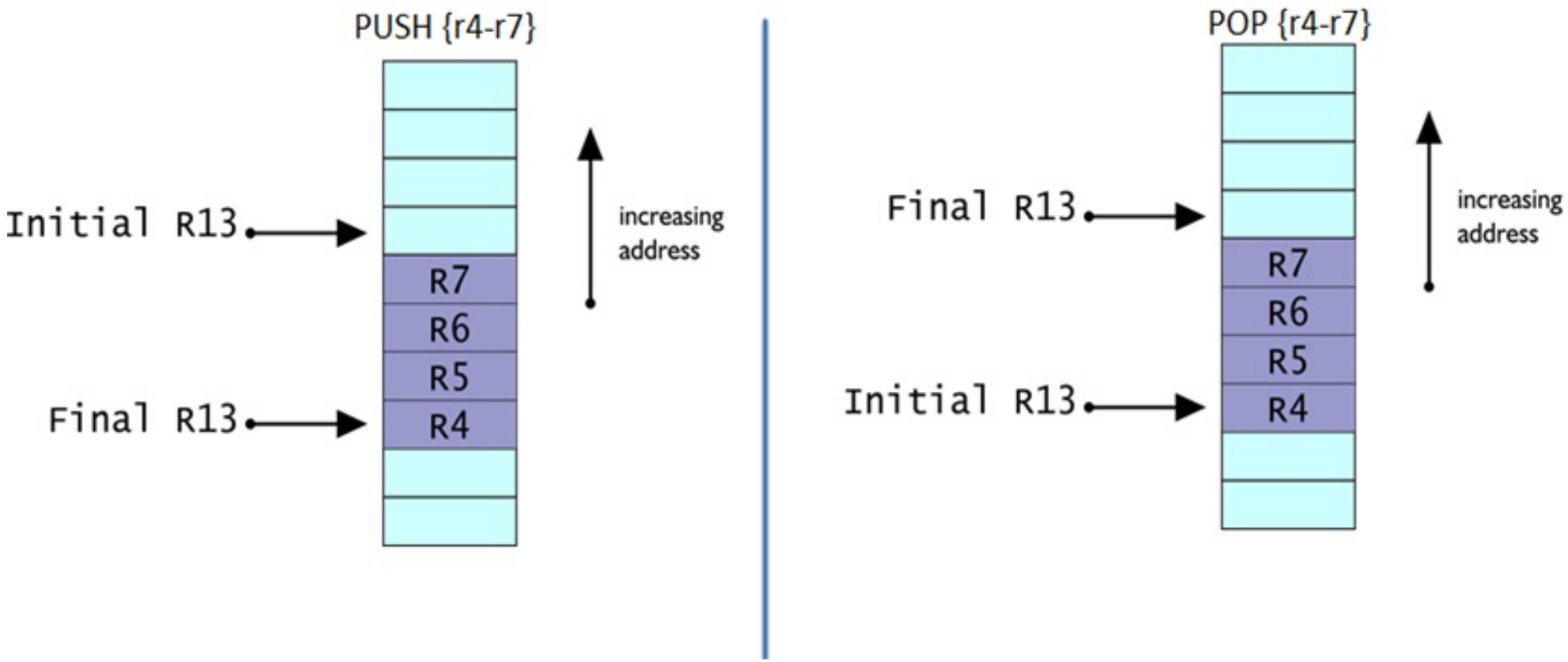
Stack memory access

- Instruction: **PUSH, POP**
- Syntax: PUSH {<Ra>,<Rb>,…}
- Note: After push/pop instruction will update SP register
  - Push -> SP = SP - N*4
  - POP -> SP = SP + N*4

Stack Operations

- The stack can be any combination of Ascending/Descending + Empty/Full
  - Ascending: stack grows towards a higher address
  - Descending: stack grows towards a lower address
  - Empty: SP points to the next empty location
  - Full: SP points to the last item pushed on the stack
- The most common (and default) implementation in ARM is fully descending

Stack Operations (Cont.)



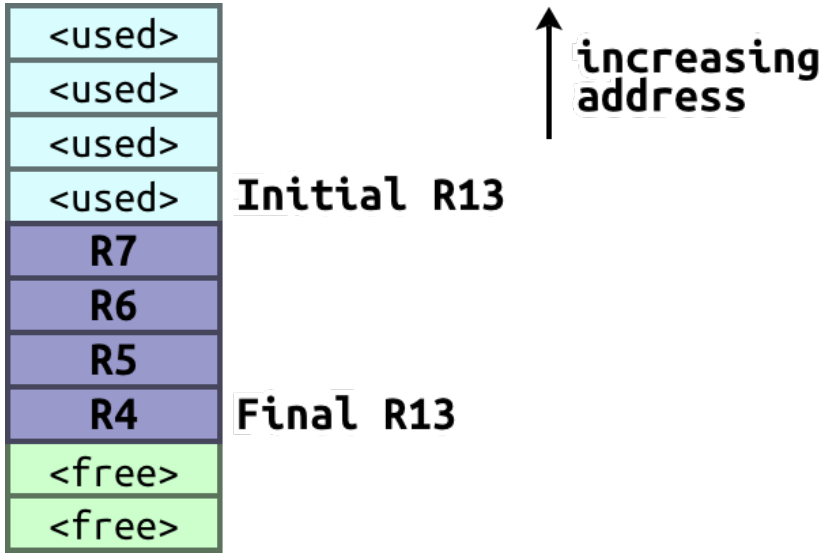
Using STM/LDM Instructions

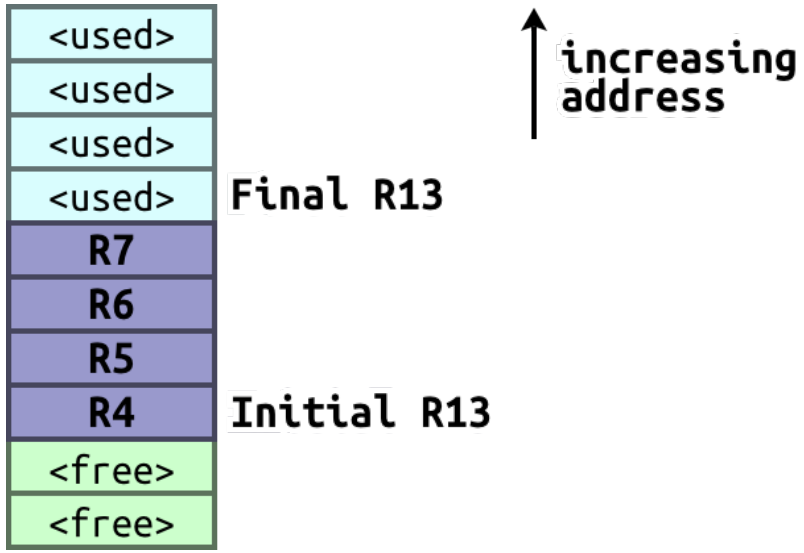
- The STM and LDM instructions' modes have aliases for accessing stacks:
  - FD = Full Descending
    - STMFD/LDMFD = STMDB/LDMIA
  - ED = Empty Descending
    - STMED/LDMED = STMDA/LDMIB
  - FA = Full Ascending
    - STMFA/LDMFA = STMIB/LDMDA
  - EA = Empty Ascending
    - STMEA/LDMEA = STMIA/LDMDB

Using STM/LDM Instructions

`STMFD r13!, {r4-r7}`
- Push R4,R5,R6 and R7 onto the stack.

`LDMFD r13!, {r4-r7}`
- Pop R4,R5,R6 and R7 from the stack.





Memory Barrier Instructions

- Instructions: DMB, SDB, ISB
- Sync memory data before next instruction
- Note: ARM Cortex-M0 system only have single memory(no have cache) so the memory barrier instruction is rarely needed

Instruction	DMB
Function	Data Memory Barrier
Syntax	DMB
Note	Ensures that all memory accesses are completed before new memory access is committed

Instruction	DSB
Function	Data Synchronization Barrier
Syntax	DSB
Note	Ensures that all memory accesses are completed before the next instruction is executed

Instruction	ISB
Function	Instruction Synchronization Barrier
Syntax	ISB
Note	Flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Core A	Core B
STR R0, [R1]	LDR R2, [R3]
STR R2, [R3]	LDR R0, [R1]

Exception-Related Instructions

- Instruction: SVC (supervisor call)
- Note: Use for trigger a software interrupt or exception

Instruction	SVC
Function	Supervisor call
Syntax	SVC #<immed8> SVC <immed8>
Note	Trigger the SVC exception. For example, SVC #3 ; SVC instruction, with parameter, equals 3. Alternative syntax without the "#" is also allowed. For example, SVC 3 ; this is the same as SVC #3.

Other Instructions

- Sleep Mode Feature
  - WFI (Wait-for-Interrupt)
  - WFE (Wait-for-Event)
- No operation(空指令)
  - NOP
- Breakpoint
  - BKPT #<immed8>
- Pseudo Instructions
  - Load a 32-bit immediate data into register Rd
    - LDR <Rd>, =immed32
  - Load a data in specified address (label) into register
    - LDR <Rd>, label

Instruction Cycle Timing

1 cycle	All data-processing operations (without PC as destination - ADD, SUB, MOV, NOP) All 16-bit Thumb branch instructions (when not taken)
2 cycles	All single-element load or store operations (LDR/STR) Wait for interrupt or event (WFI, WFE)
3 cycles	All 16-bit Thumb branch instructions (when taken) Data-processing operations where PC is the destination register
4 cycles	All 32-bit Thumb instructions (BL, DMB, DSB, ISB, MSR, MRS)
1+N	Multiple load and stores containing N elements (without POP with PC in list) LDM, STM, POP and PUSH
4+N	POP with PC in list
32 cycles	Multiplication (MULS)

- Zero wait state memory system assumed

Instruction Usage Examples

- Program Control
  - If-Then-Else
  - Condition branch
  - Loop
  - Function call
- Data access
- Data Processing

If-Then-Else

- Use CMP and conditional branches.
- Example

```
if (counter > 10) then
    counter = 0
else
    counter = counter + 1
```

Assume the R0 is used as a "counter" variable; the preceding operation can be implemented as

```
    CMP  R0, #10        ; compare to 10
    BLE  incr_counter   ; if less or equal, then branch
    MOVS R0, #0         ; counter = 0
    B    counter_done   ; branch to counter_done
incr_counter
    ADDS R0, R0, #1     ; counter = counter +1
counter_done
    ...
```

The program code first carries out a compare and then executes a conditional branch. The program then carries out a required task and finishes at the program address labeled as "counter_done."

Condition branch

- After a compare operation “CMP R0, R1”

Table 6.3: Conditional Branch Instructions for Value Comparison

Required Branch Control	Unsigned Data	Signed Data
If (R0 equal R1) then branch	BEQ label	BEQ label
If (R0 not equal R1) then branch	BNE label	BNE label
If (R0 > R1) then branch	BHI label	BGT label
If (R0 >= R1) then branch	BCS label / BHS label	BGE label
If (R0 < R1) then branch	BCC label / BLO label	BLT label
If (R0 <= R1) then branch	BLS label	BLE label

- Other compare instruction test (TST), compare negative (CMN)

Loop

- Simple for loop example

```
Total = 0;
for (i=0;i<5;i=i+1)
  Total = Total + i;
```

Assume "Total" is R0 and "i" is R1; the program can be implemented as

```
    MOVS R0, #0      ; Total = 0
    MOVS R1, #0      ; i = 0
loop
    ADDS R0, R0, R1  ; Total = Total + i
    ADDS R1, R1, #1  ; i = i + 1
    CMP  R1, #5      ; compare i to 5
    BLT  loop        ; if less than then branch to loop
```

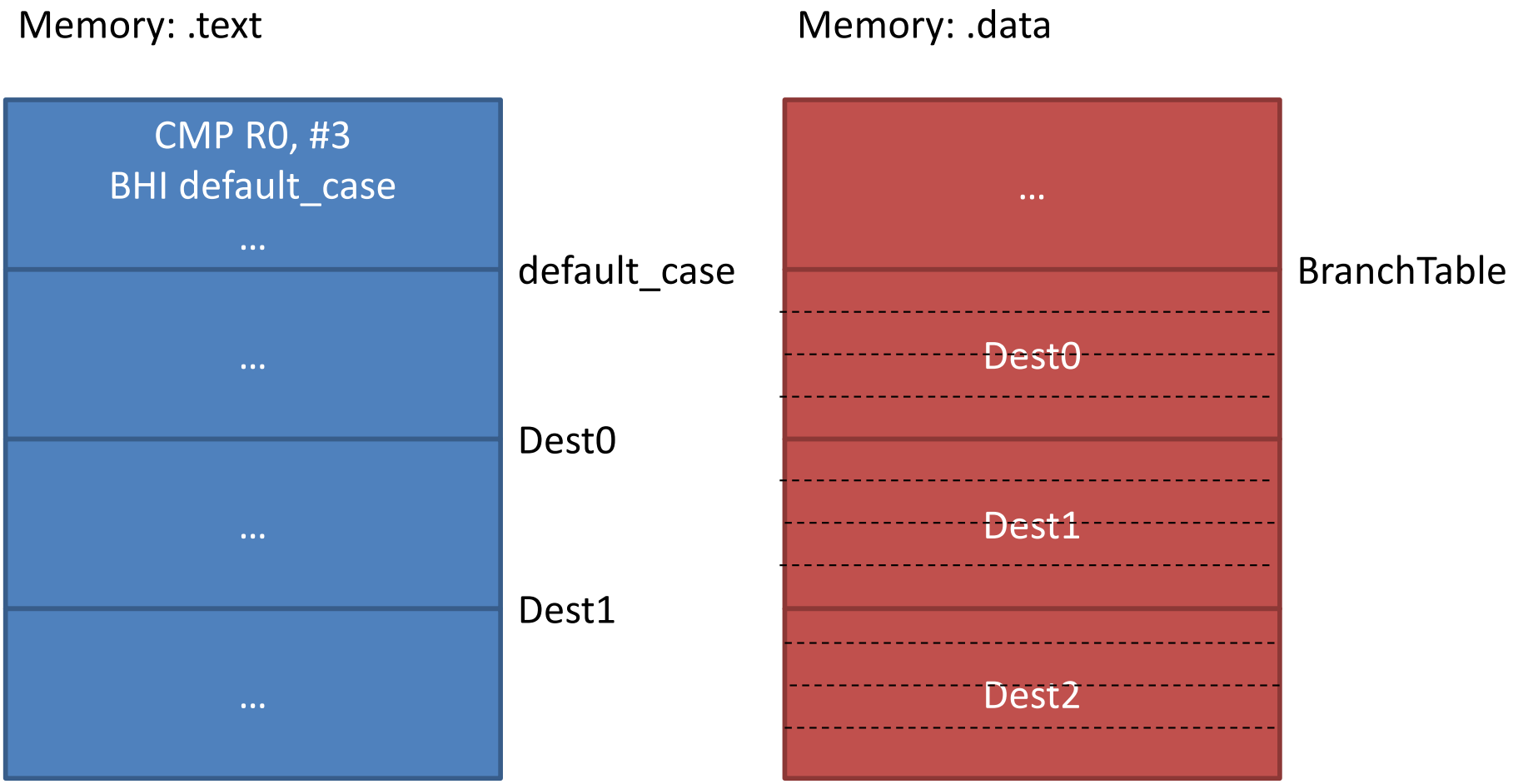
Switch case

- 4 case switch

```
        CMP   R0, #3            ; Compare input to maximum valid choice
        BHI   default_case      ; Branch to default case if higher than 3
        MOVS  R2, #4            ; Multiply branch table offset by 4
        MULS  R0, R2, R0        ; (size of each entry)
        LDR   R1,=BranchTable   ; Get base address of branch table
        LDR   R2,[R1,R0]        ; Get the actual branch destination
        BX    R2                ; Branch to destination
        ALIGN 4                 ; Alignment control. The table has
              ; to be word aligned to prevent unaligned read
BranchTable   ; table of each destination addresses
```

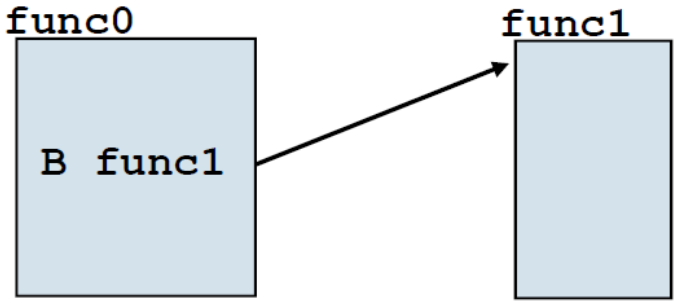
Data define

```
        DCD Dest0
        DCD Dest1
        DCD Dest2
        DCD Dest3
default_case
            ... ; Instructions for default case
Dest0
            ... ; Instructions for case '0'
Dest1
            ... ; Instructions for case '1'
Dest2
            ... ; Instructions for case '2'
Dest3
            ... ; Instructions for case '3'
```

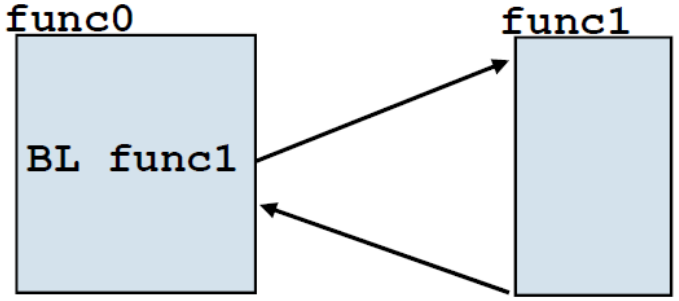

Memory: .text
CMP R0, #3
BHI default_case
...
default_case
...
Dest0
...
Dest1
...
Memory: .data
...
BranchTable
Dest0
Dest1
Dest2

Branch Instructions

- `B` – Branch
  - Absolute branch to a target address, relative to Program Counter (PC)
  - +/- 256 bytes range, conditional execution supported
  - +/- 1MB range, no conditional execution supported



- `BL` – Branch with Link
  - Branch to a subroutine – Link register is updated
  - +/- 16MB range, relative to Program Counter (PC)



Program Flow Control

- Instruction: B, B<c>

Instruction	B (Branch)
Function	Branch to an address (unconditional)
Syntax	B <label>
Note	Branch range is +/− 2046 bytes of current program counter

Instruction	B<cond> (Conditional Branch)
Function	Depending of APSR, branch to an address
Syntax	B<cond> <label>
Note	Branch range is +/− 254 bytes of current program counter. For example, CMP R0, #0x1 ; Compare R0 with 0x1 BEQ process1 ; Branch to process1 if R0 equal 1

Conditional Execution

- APSR condition code flags are used to decide if a branch instruction is executed
- Conditional execution is only supported using 16-bit branch instructions
  - Branch range: PC+ -254~254 bytes
- B<c> addr
  - Example BLE L1; BEQ L1;…

Condition Codes

Condition Code	Interpretation	Status Flag State
EQ	Equal / equals zero	Z set
NE	Not equal	Z clear
CS / HS	Carry set / unsigned higher or same	C set
CC / LO	Carry clear / unsigned lower	C clear
MI	Minus / negative	N set
PL	Plus / positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N equals V
LT	Signed less than	N is not equal to V
GT	Signed greater than	Z clear and N equals V
LE	Signed less than or equal	Z set or N is not equal to V
AL	Always (optional)	Any

Long Branch

- Usually use for function call
  - BL for function call and BX for function return.

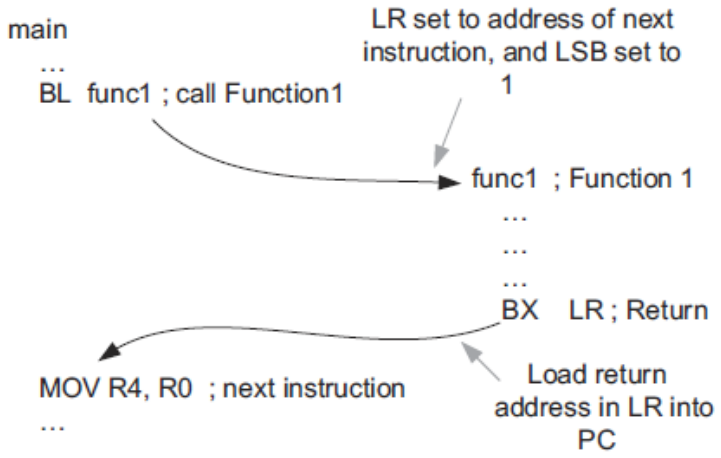


Figure 5.10:
Function call and return using BL and BX instructions.

Instruction	BL (Branch and Link)
Function	Branch to an address and store return address to LR. Usually use for function calls, and can be used for long-range branch that is beyond the branch range of branch instruction (B <label>).
Syntax	BL <label>
Note	Branch range is +/− 16MB of current program counter. For example, BL functionA ; call a function called functionA

Instruction	BX (Branch and Exchange)
Function	Branch to an address specified by a register, and change processor state depending on bit[0] of the register.
Syntax	BX <Rm>
Note	Because the Cortex-M0 processor only supports Thumb code, bit[0] of the register content (Rm) must be set to 1, otherwise it means it is trying to switch to the ARM state and this will generate a fault exception.

Branch Range

Table 6.2: Instruction for Branch Range

Branch Range	Available Instruction
Under +/− 254 bytes	B label
	B<cond> label
Under +/− 2KB	B label
Under +/− 16MB	BL label
Over +/− 16MB	LDR R0,=label; Load the address value of label in R0
	BX R0; Branch to address pointed to by R0, or
	BLX R0; Branch to address pointed to by R0 and update LR

Function Calls and Function Returns (1/3)

- Simple function call

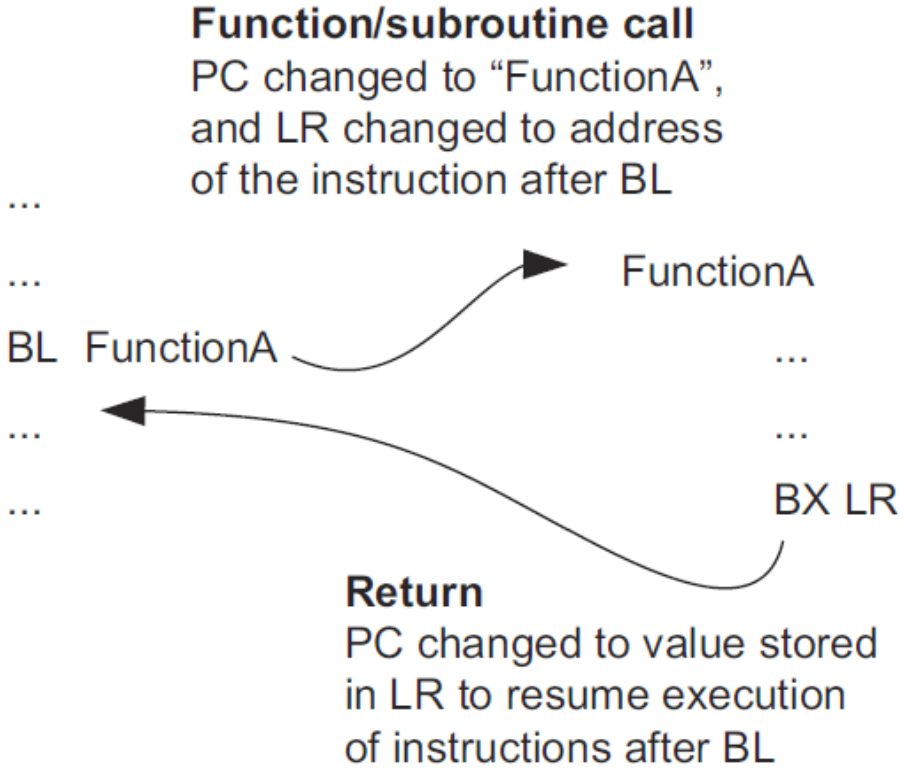


Figure 6.1:
Simple function call and function return.

Function Calls and Function Returns (2/3)

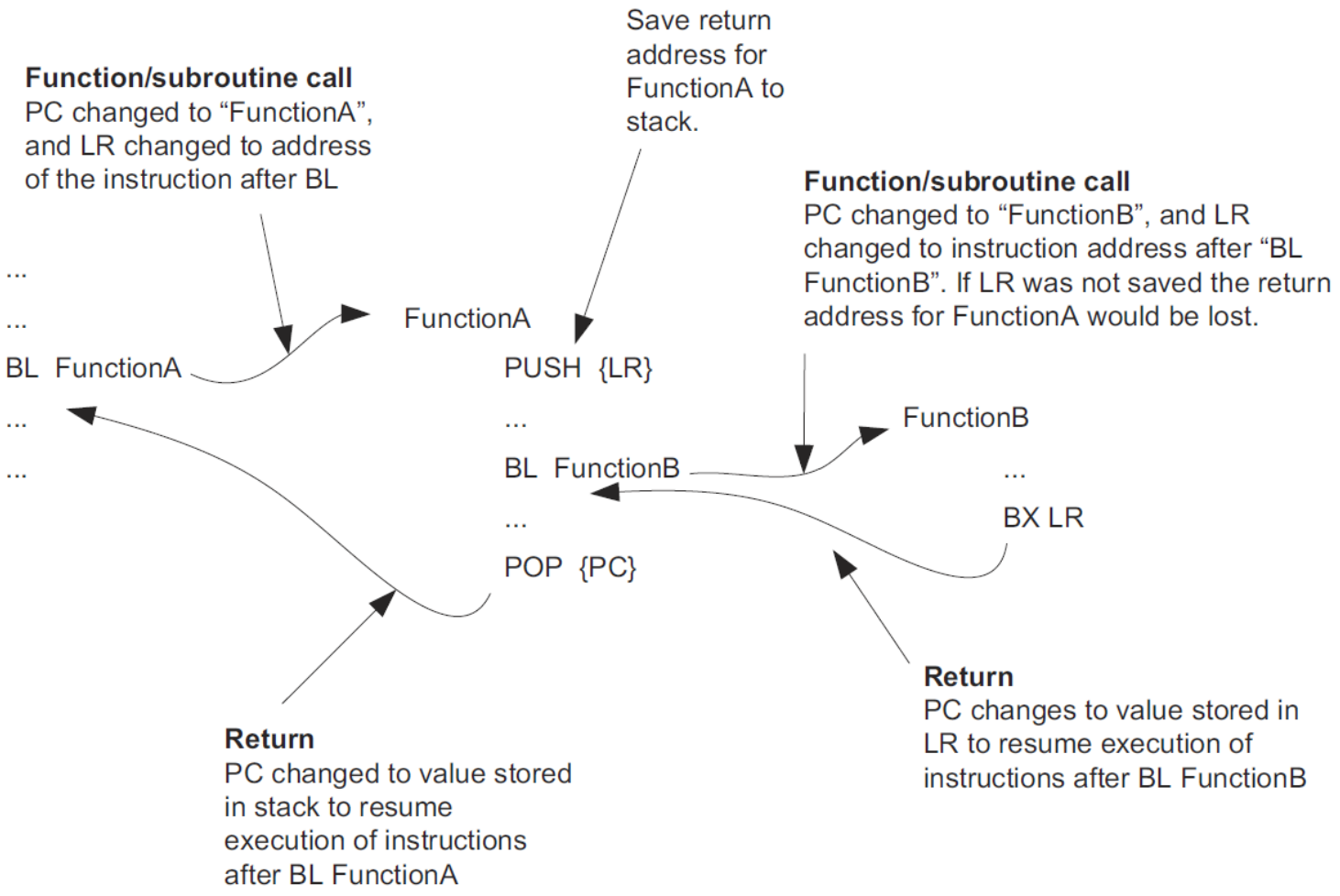


Figure 6.2:
Nested function call and function return.

Function Calls and Function Returns (3/3)

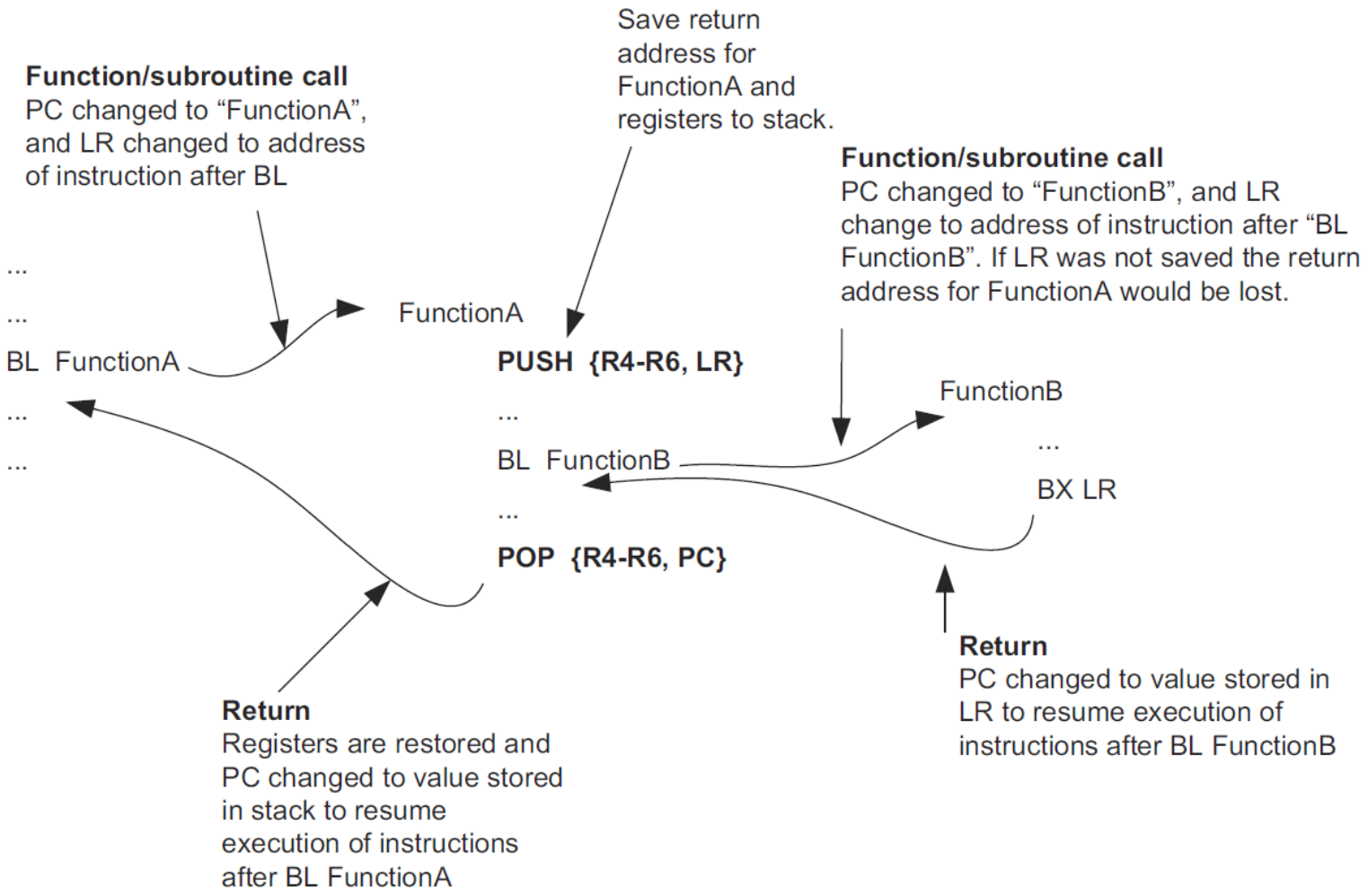


Figure 6.3:
Using push and pop of multiple registers in functions.

Function Call and Parameter Passing

callee

```
void f1(int a, int b)
{
…
}
```

callee

```
int main(void)
{
…
f1(1, 2);
…
}
```

```
main:
…
BL f1
…

f1:
…
BX LR
```

callee

callee

Function Call and Parameter Passing

- How if caller and callee use the same regsiters
- How caller passes parameters to callee
- How callee passes returns values to caller
- Where is the parameters
- Where is the local variables

Registers backup

```
main:
…
Use R1
BL f1
Use R1
…

f1:
Use R1
…
BX LR
```

what are the problems if caller backups and restores all registers

what are the problems if callee backups and restores all registers

ARM Procedure Call Standard

Register	APCS Name	APCS Role
0	a1	argument 1 / result / scratch register for function
1	a2	argument 2 / result / scratch register for function
2	a3	argument 3 / result / scratch register for function
3	a4	argument 4 / result / scratch register for function
4	v1	variable register 1
5	v2	variable register 2
6	v3	variable register 3
7	v4	variable register 4
8	v5	variable register 5
9	sb or v6	static base or variable register 6
10	sl or v7	stack limit or variable register 7
11	fp	frame pointer (pointer to start of current stack frame) or variable register 8
12	ip	the intra-procedure-call scratch register
13	sp	lower end of current stack frame (stack pointer)
14	lr	the link register (return address)
15	pc	the program counter

ARM Procedure Call Standard

Parameters, caller-save if necessary

Register	APCS Name	APCS Role
0	a1	argument 1 / result / scratch register for function
1	a2	argument 2 / result / scratch register for function
2	a3	argument 3 / result / scratch register for function
3	a4	argument 4 / result / scratch register for function
4	v1	variable register 1
5	v2	variable register 2
6	v3	variable register 3
7	v4	variable register 4
8	v5	variable register 5
9	sb or v6	static base or variable register 6
10	sl or v7	stack limit or variable register 7
11	fp	frame pointer (pointer to start of current stack frame) or variable register 8
12	ip	the intra-procedure-call scratch register
13	sp	lower end of current stack frame (stack pointer)
14	lr	the link register (return address)
15	pc	the program counter

ARM Procedure Call Standard

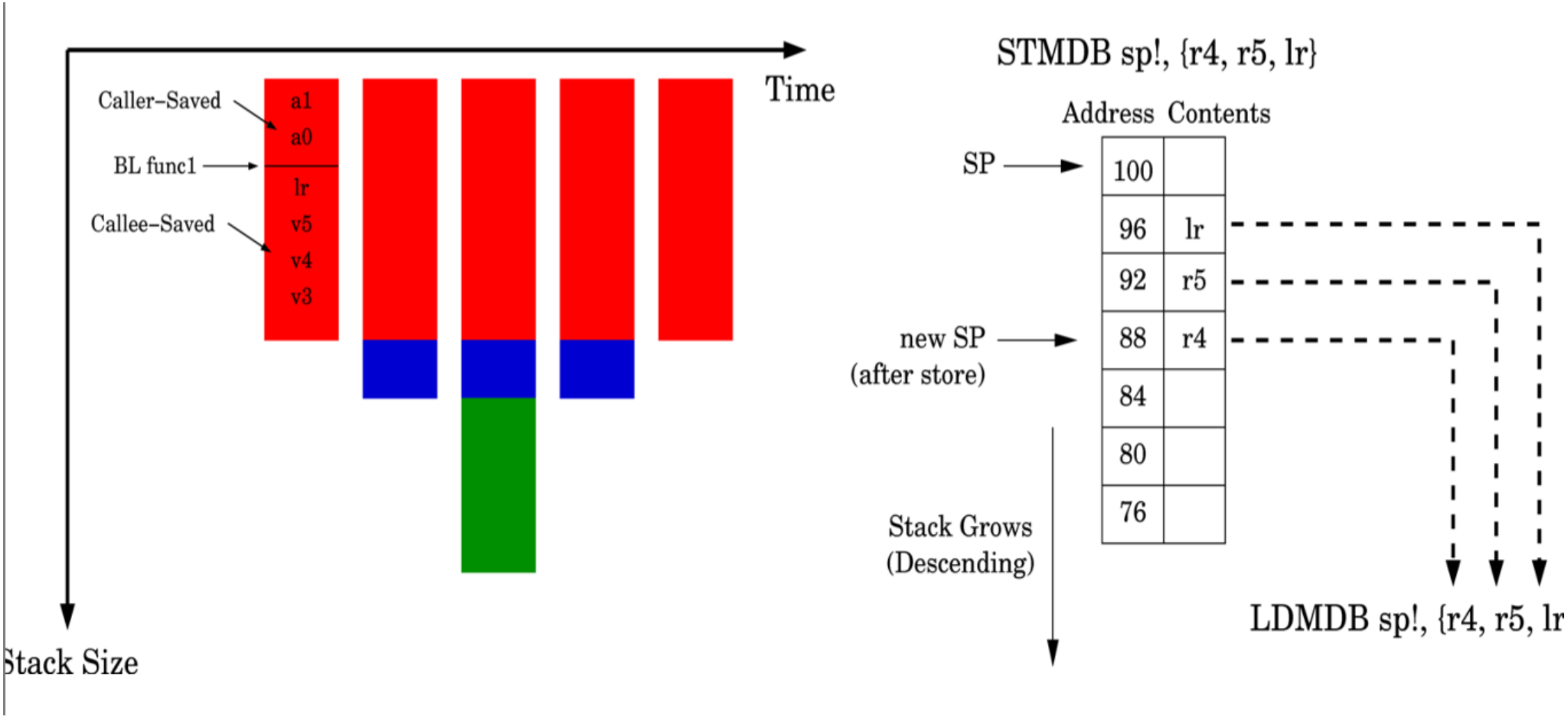
Local variables, callee-save if necessary

Register	APCS Name	APCS Role
0	a1	argument 1 / result / scratch register for function
1	a2	argument 2 / result / scratch register for function
2	a3	argument 3 / result / scratch register for function
3	a4	argument 4 / result / scratch register for function
4	v1	variable register 1
5	v2	variable register 2
6	v3	variable register 3
7	v4	variable register 4
8	v5	variable register 5
9	sb or v6	static base or variable register 6
10	sl or v7	stack limit or variable register 7
11	fp	frame pointer (pointer to start of current stack frame) or variable register 8
12	ip	the intra-procedure-call scratch register
13	sp	lower end of current stack frame (stack pointer)
14	lr	the link register (return address)
15	pc	the program counter

ARM Procedure Call Standard

Special purposes, temporary use if save properly

Register	APCS Name	APCS Role
0	a1	argument 1 / result / scratch register for function
1	a2	argument 2 / result / scratch register for function
2	a3	argument 3 / result / scratch register for function
3	a4	argument 4 / result / scratch register for function
4	v1	variable register 1
5	v2	variable register 2
6	v3	variable register 3
7	v4	variable register 4
8	v5	variable register 5
9	sb or v6	static base or variable register 6
10	sl or v7	stack limit or variable register 7
11	fp	frame pointer (pointer to start of current stack frame) or variable register 8
12	ip	the intra-procedure-call scratch register
13	sp	lower end of current stack frame (stack pointer)
14	lr	the link register (return address)
15	pc	the program counter


Time
Caller–Saved
a1
a0
BL func1
lr
Callee–Saved
v5
v4
v3
Stack Size
STMDB sp!, {r4, r5, lr}
Address Contents
SP
100
96
lr
92
r5
new SP
(after store)
88
r4
84
80
76
Stack Grows
(Descending)
LDMDB sp!, {r4, r5, lr

Instruction Usage Examples

```
if (counter > 10) then
    counter = 0
else
    counter = counter + 1
```

```
    CMP  R0, #10        ; compare to 10
    BLE  incr_counter   ; if less or equal, then branch
    MOVS R0, #0         ; counter = 0
    B    counter_done   ; branch to counter_done
incr_counter
    ADDS R0, R0, #1     ; counter = counter +1
counter_done
    ...
```

Instruction Usage Examples

```
Total = 0;
for (i=0;i<5;i=i+1)
   Total = Total + i;
```

```
    MOVS R0, #0       ; Total = 0
    MOVS R1, #0       ; i = 0
loop
    ADDS R0, R0, R1   ; Total = Total + i
    ADDS R1, R1, #1   ; i = i + 1
    CMP  R1, #5       ; compare i to 5
    BLT  loop         ; if less than then branch to loop
```

Instruction Usage Examples

Branch Range	Available Instruction
Under +/− 254 bytes	B label
	B<cond> label
Under +/− 2KB	B label
Under +/− 16MB	BL label
Over +/− 16MB	LDR R0,=label; Load the address value of label in R0
	BX R0; Branch to address pointed to by R0, or
	BLX R0; Branch to address pointed to by R0 and update LR

Required Branch Control	Unsigned Data	Signed Data
If (R0 equal R1) then branch	BEQ label	BEQ label
If (R0 not equal R1) then branch	BNE label	BNE label
If (R0 > R1) then branch	BHI label	BGT label
If (R0 >= R1) then branch	BCS label / BHS label	BGE label
If (R0 < R1) then branch	BCC label / BLO label	BLT label
If (R0 <= R1) then branch	BLS label	BLE label

Data access (1/2)

- Simple data access
  - Assume “DataIn” is a integer array with 10 elements (32-bit each)
  - Sum of each element in “Sum” variable

```
            LDR  r0,=DataIn; Get the address of variable 'DataIn'
            MOVS r1, #10    ; loop counter
            MOVS r2, #0     ; Result - starting from 0
add_loop
            LDM  r0!,{r3}   ; Load result and increment address
            ADDS r2, r3     ; add to result
            SUBS r1, #1     ; increment loop counter
            BNE  add_loop
            LDR  r0,=Sum    ; Get the address of variable 'Sum'
            STR  r2,[r0]    ; Save result to Sum
```

Data access (2/2)

- Copy data array

```
            LDR  r0,=0x00000000 ; Source address
            LDR  r1,=0x20000000 ; Destination address
            LDR  r2,=100    ; number of bytes to copy
copy_loop
            LDRB r3,  [r0]    ; read 1 byte
            ADDS r0,   r0, #1 ; increment source pointer
            STRB r3,  [r1]    ; write 1 byte
            ADDS r1,   r1, #1 ; increment destination pointer
            SUBS r2,   r2, #1 ; decrement loop counter
            BNE  copy_loop   ; loop until all data copied
```

The program code uses a number of add and subtract instructions in the loop, which reduce the performance. We could modify the code to reduce the program size using a register offset address mode:

```
            LDR  r0,=0x00000000 ; Source address
            LDR  r1,=0x20000000 ; Destination address
            LDR  r2,=100        ; number of bytes to copy, also
copy_loop                               ; acts as loop counter
            SUBS r2, r2, #1     ; decrement offset and loop counter
            LDRB r4,[r0, r2]    ; read 1 byte
            STRB r4,[r1, r2]    ; write 1 byte
            BNE  copy_loop      ; loop until all data copied
```

Data Processing (1/3)

- 64-Bit/128-Bit Add

```
LDR   r0,=0xFFFFFFFF ; X_Low  (X = 0x3333FFFFFFFFFFFF)
LDR   r1,=0x3333FFFF ; X_High
LDR   r2,=0x00000001 ; Y_Low  (Y = 0x3333000000000001)
LDR   r3,=0x33330000 ; Y_High
ADDS  r0,r0,r2 ; lower 32-bit
ADCS  r1,r1,r3 ; upper 32-bit
```

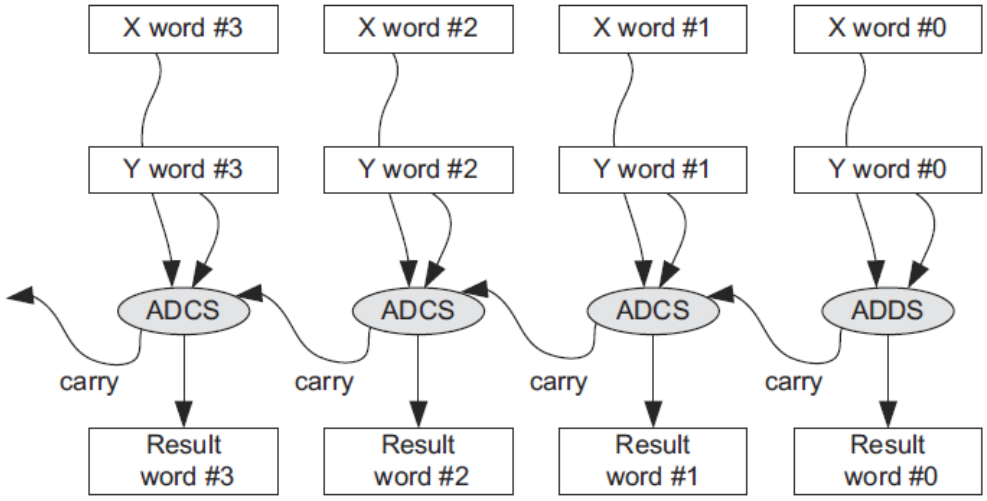


Figure 6.5:
Adding of two 128-bit numbers.

Data Processing (2/3)

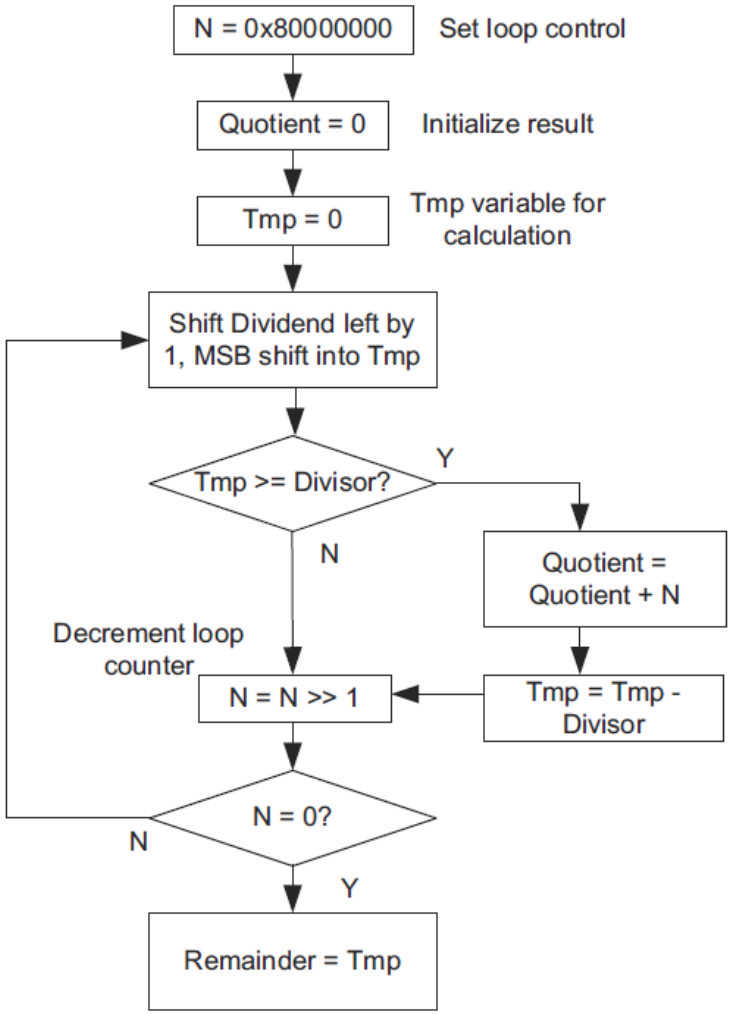


Figure 6.7:
Simple unsigned integer divide function.

- Integer Divide

```
simple_divide
     ; Inputs
     ;   R0 = dividend
     ;   R1 = divider
     ; Outputs
     ;   R0 = quotient
     ;   R1 = remainder
     PUSH {R2-R4}   ; Save registers to stack
     MOV  R2, R0    ; Save dividend to R2 as R0 will be changed
     MOVS R3, #0x1 ; loop control
     LSLS R3, R3, #31 ; N = 0x80000000
     MOVS R0, #0    ; initial Quotient
     MOVS R4, #0    ; initial Tmp
simple_divide_loop
     LSLS R2, R2, #1 ; Shift dividend left by 1 bit, MSB go into carry
     ADCS R4, R4, R4 ; Shift Tmp left by 1 bit, carry move into LSB
     CMP  R4, R1
     BCC  simple_divide_lessthan
     ADDS R0, R0, R3 ; Increment quotient
     SUBS R4, R4, R1
simple_divide_lessthan
     LSRS R3, R3, #1 ; N = N >> 1
     BNE  simple_divide_loop
     MOV  R1, R4  ; Put remainder in R1, Quotient is already in R0
     POP  {R2-R4} ; Restore used register
     BX   LR      ; Return
```

Data Processing (3/3)

- Bit and Bit Field Computations

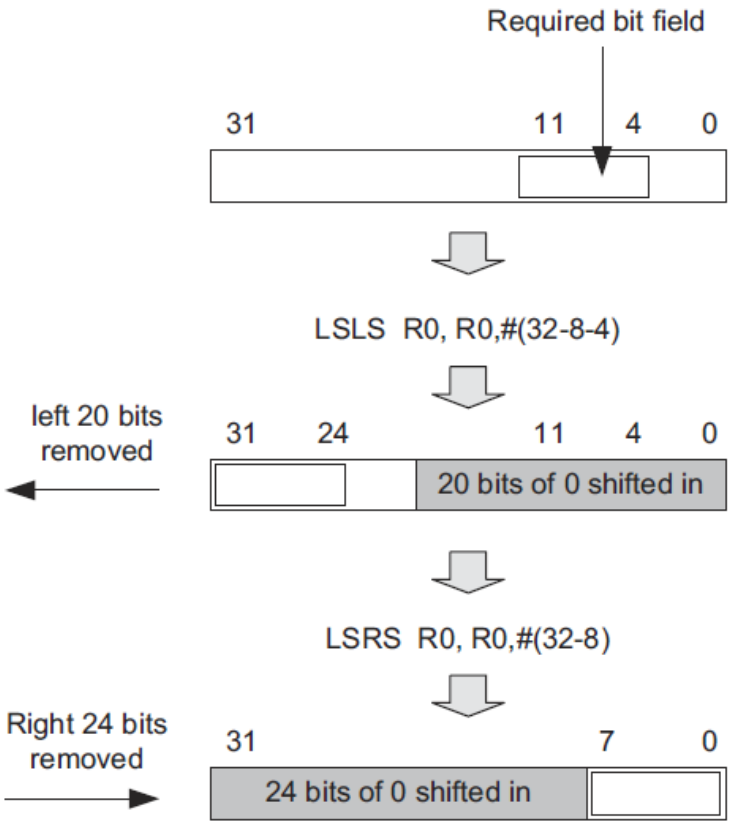


Figure 6.9:
Bit field extract operation.

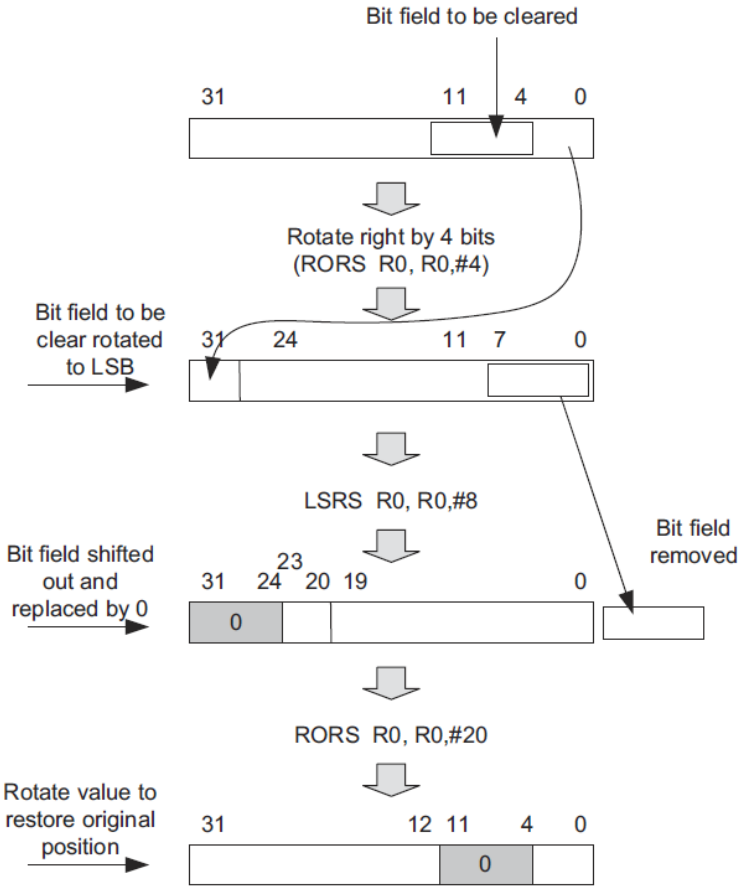


Figure 6.10:
Bit field clear operation.