EIE/ENE 334 Microprocessors

Lecture 8:

ARM Cortex-M0 Processors

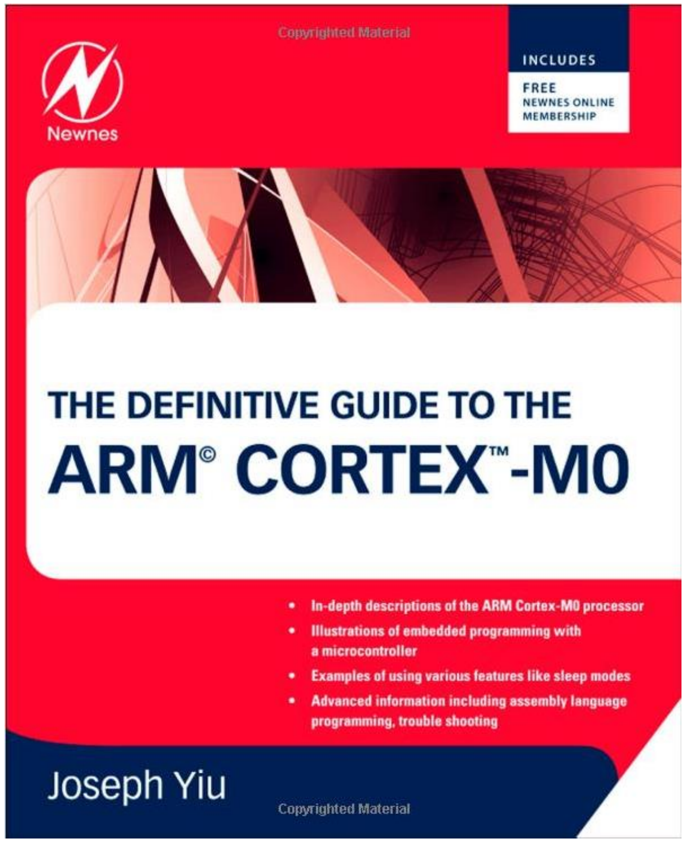
Week #08 : Dejwoot KHAWPARISUTH

Adapted from

http://webstaff.kmutt.ac.th/~dejwoot.kha/

Textbooks:

- Joseph Yiu, "The Definitive Guide to the ARM Cortex-M0", Newnes, 2011


THE DEFINITIVE GUIDE TO THE ARM© CORTEX™-M0

- In-depth descriptions of the ARM Cortex-M0 processor
- Illustrations of embedded programming with a microcontroller
- Examples of using various features like sleep modes
- Advanced information including assembly language programming, trouble shooting

Joseph Yiu

http://www.arm.com/support/university/academic-resources.php

Ref: from ARM

For Final exam:

→ Cortex™-M0 Devices Generic User Guide (DUI0497A)

Ref: from Nuvoton

For Final exam:

→ NuMicro NUC130_140 Technical Reference Manual EN V2.02

ARM-Cortex-M0 Intro

- 32-bit RISC processor
- a 3-stage pipeline von Neumann architecture
- the ARMv6-M architecture
  - Implement
    - the ARMv6-M Thumb® instruction set
    - including Thumb-2 technology

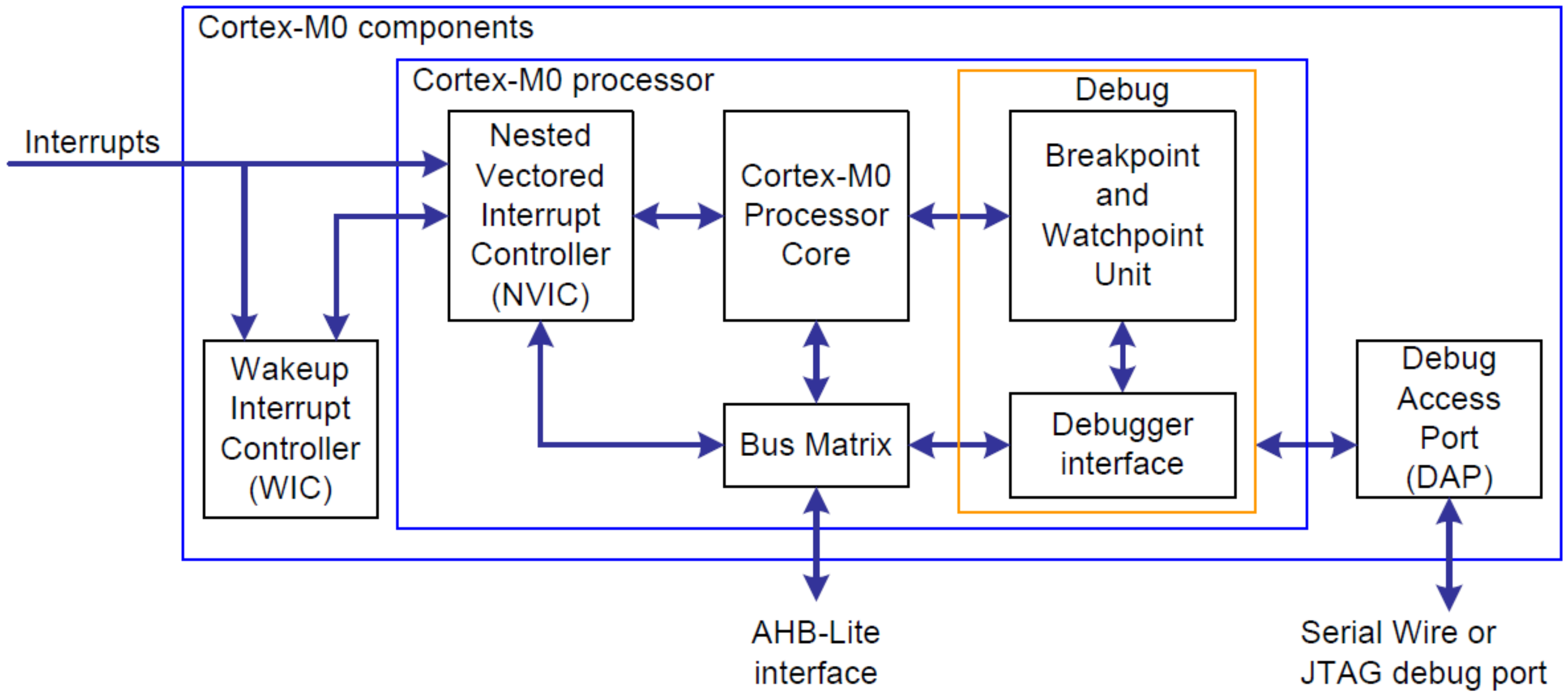
ARM-Cortex-M0 Intro

- 56 instructions (16&32 bits)
- Microcontroller applications
- Zero jitter (the instruction and interrupt timings are fully deterministic)
- Include a nested vector interrupt controller (NVIC)
- Support the CoreSight Debug architecture

ARM-Cortex-M0 Intro

- → very low gate count (12,000 logic gates, minimum configuration)
- → highly energy efficient (0.9 DMIPS/MHz)
- → Code density using Thumb-2-based instruction

ARM-Cortex-M0 Intro



ARM-Cortex-M0 Core

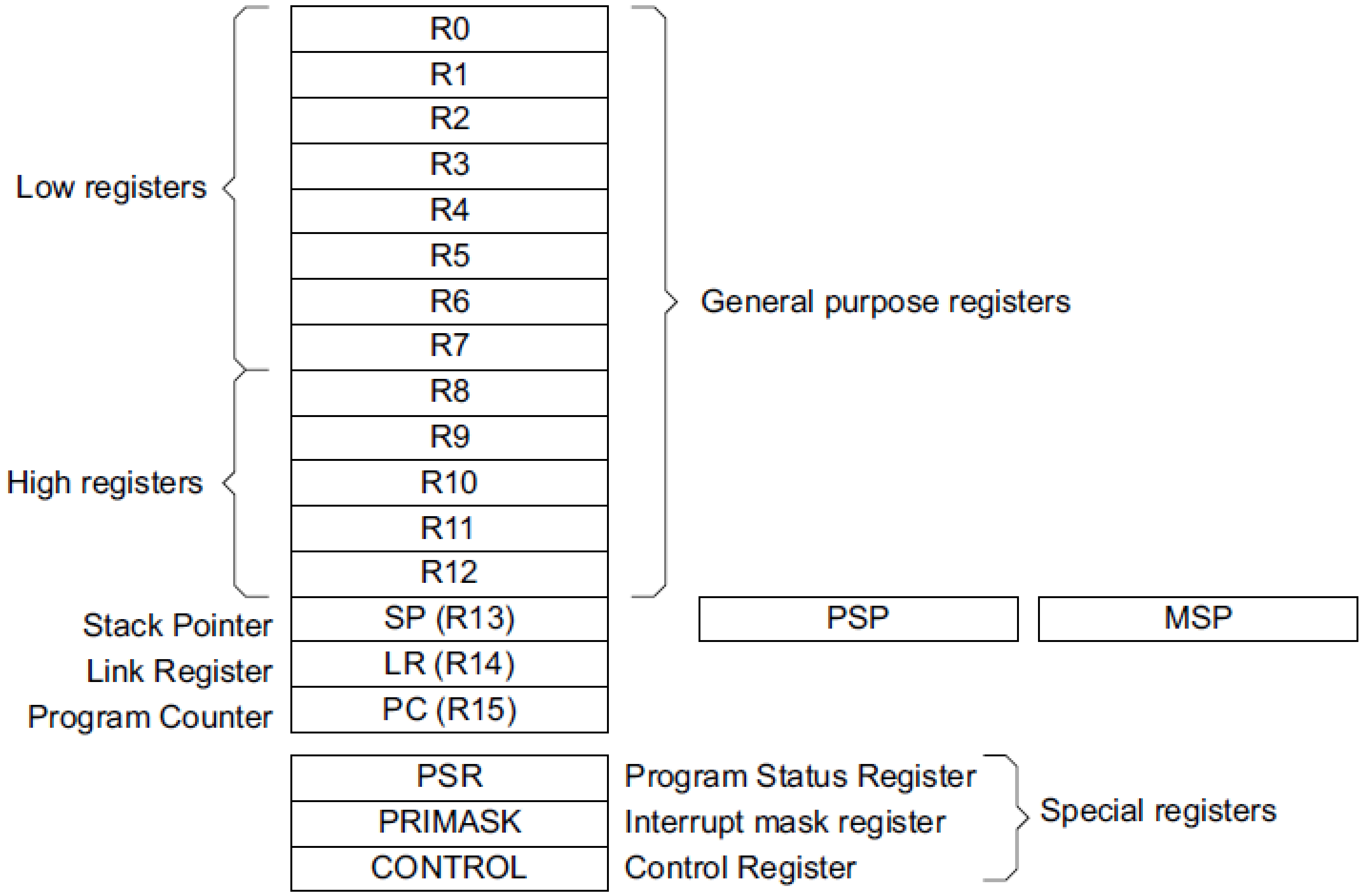
- Programmer's model
- Instruction set
- Exception mechanism
- Memory model
- Debug architecture

Programmer’s model

- Two states
  - Thumb state
    - Two operation modes
      - Thread mode (can use a shadowed stack pointer)
      - Handler mode
  - Debug state

Programmer's model

- Core registers



Programmer's model

- A load-store architecture

- The processor uses a full descending stack: PUSH – decrements the stack pointer and then writes the item onto the stack.

Programmer's model

- On reset, the processor loads the MSP with the value from address 0x00000000.

- In Handler mode, the processor always use the main stack (MSP).

Programmer’s model

- After reset, the main stack pointer (MSP) is used, but can be switched to the process stack pointer (PSP) in Thread mode (when not running an exception handler) by setting bit [1] in the CONTROL register. During running of an exception handler, the CONTROL register reads as zero.

Programmer’s model

- The CONTROL register can only be changed in Thread mode or via the exception entrance and return mechanism.

Programmer's model

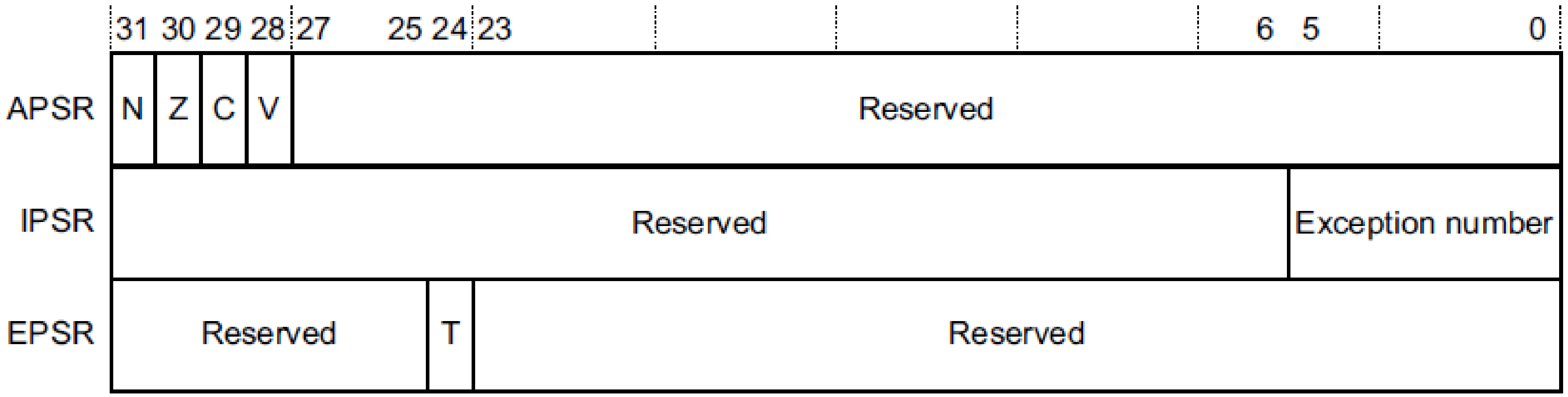
- The Link Register (LR) stores the return information for subroutines, function calls, and exceptions. On reset, the LR value is Unknown.

Programmer's model

- The Program Counter (PC) contains the current program address. On reset, the processor loads the PC with the value of the reset vector, at address 0x00000004. Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

Programmer's model

- The Program Status Register (PSR) combines:



Programmer's model

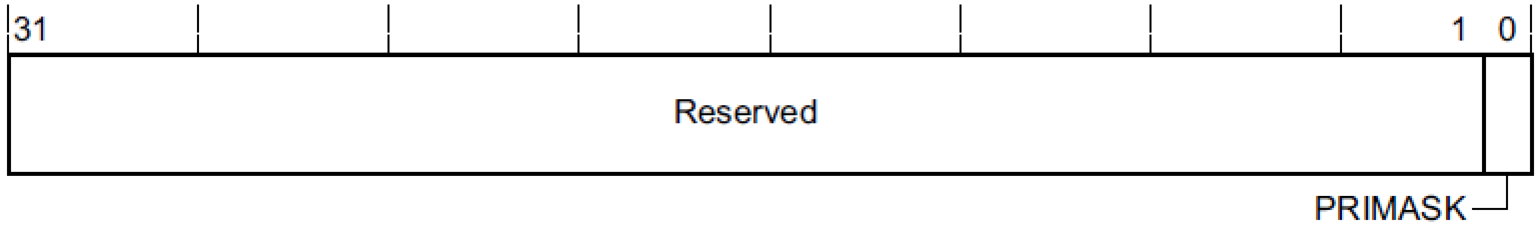
- Application Program Status Register (APSR) contains ALU flags.
- Interrupt Program Status Register (IPSR) contains the current executing interrupt service routine (ISR) number.
- Execution Program Status Register (EPSR).

Programmer's model

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address[b]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	Synchronous
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[e]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[e]	0x00000038	Asynchronous
15	-1	SysTick[c]	Configurable[e]	0x0000003C	Asynchronous
15	-	Reserved	-	-	-
16 and above[d]	0 and above	IRQ	Configurable[e]	0x00000040 and above[f]	Asynchronous

Programmer's model

- The PRIMASK register prevents activation of all exceptions with configurable priority.



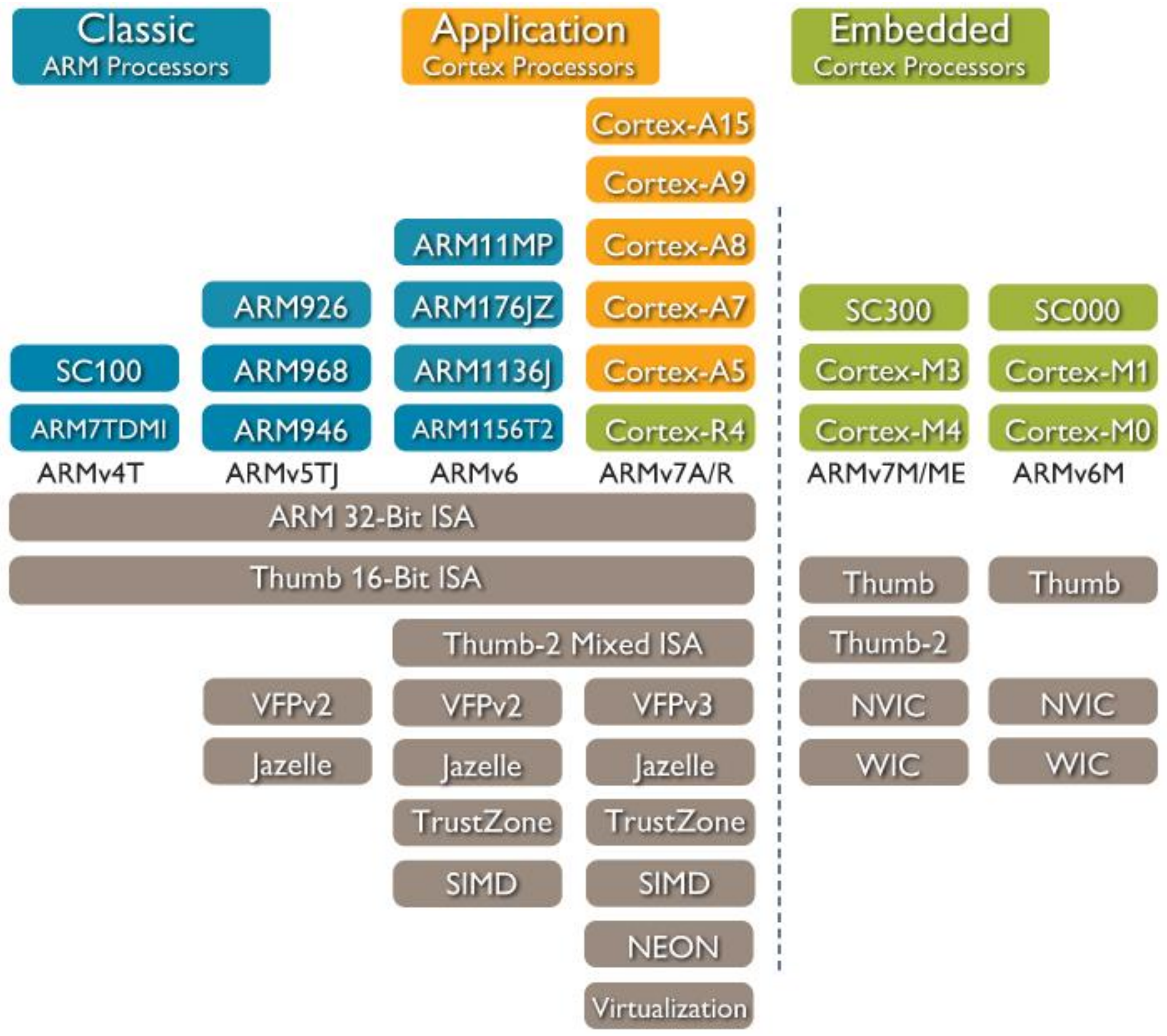
0 = no effect (on RESET)

1 = prevents the activation of all exceptions with configurable priority.

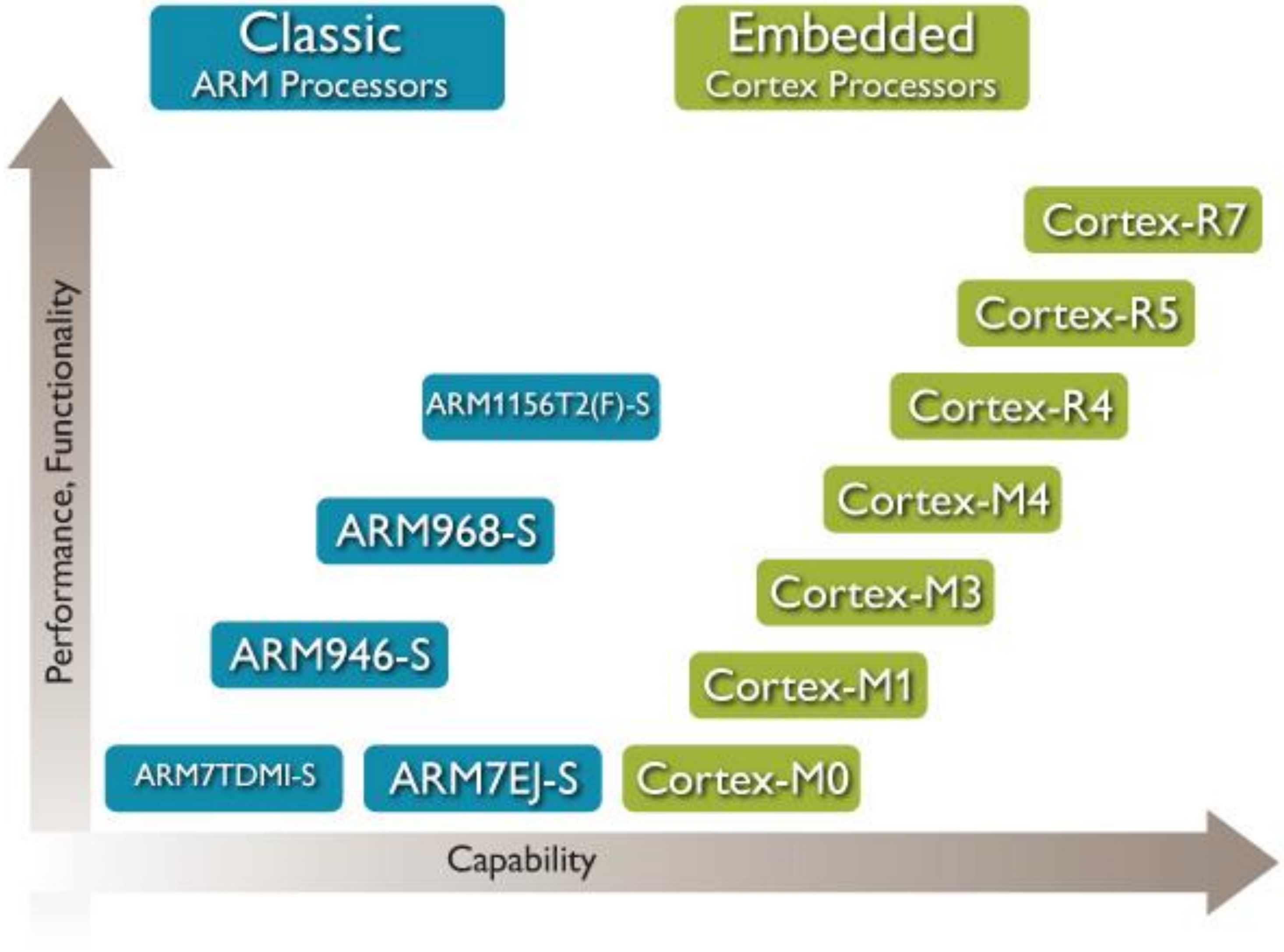
Data Sizes and Instruction Sets

- **ARM is a 32-bit load / store RISC architecture**
  - The only memory accesses allowed are loads and stores
  - Most internal registers are 32 bits wide
  - Most instructions execute in a single cycle
- **When used in relation to ARM cores**
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
  - **Doubleword** means 64 bits (eight bytes)
- **ARM cores implement two basic instruction sets**
  - **ARM** instruction set – instructions are all 32 bits long
  - **Thumb** instruction set – instructions are a mix of 16 and 32 bits
    - Thumb-2 technology added many extra 32- and 16-bit instructions to the original 16-bit Thumb instruction set
- **Depending on the core, may also implement other instruction sets**
  - **VFP** instruction set – 32 bit (vector) floating point instructions
  - **NEON** instruction set – 32 bit SIMD instructions
  - **Jazelle-DBX** - provides acceleration for Java VMs (with additional software support)
  - **Jazelle-RCT** - provides support for interpreted languages

Which architecture is my processor?



Embedded Processors



Development of the ARM Architecture

v4	v5	v6	v7
Halfword and signed halfword / byte support System mode Thumb instruction set (v4T)	Improved interworking CLZ Saturated arithmetic DSP MAC instructions Extensions: Jazelle (5TEJ)	SIMD Instructions Multi-processing v6 Memory architecture Unaligned data support Extensions: Thumb-2 (6T2) TrustZone® (6Z) Multicore (6K) Thumb only (6-M)	Thumb-2 Architecture Profiles 7-A - Applications 7-R - Real-time 7-M - Microcontroller

- **Note that implementations of the same architecture can be different**
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

Instruction Set basics

- **The ARM Architecture is a Load/Store architecture**
  - No direct manipulation of memory contents
  - Memory must be loaded into the CPU to be modified, then written back out

- **Cores are either in ARM *state* or Thumb *state***
  - This determines which instruction set is being executed
  - An instruction must be executed to switch between states

- **The architecture allows programmers and compilation tools to reduce branching through the use of conditional execution**
  - Method differs between ARM and Thumb, but the principle is that most (ARM) or all (Thumb) instructions can be executed conditionally.

Data Processing Instructions

- **These instructions operate on the contents of registers**
  - They DO NOT affect memory

	arithmetic		logical		move
manipulation (has destination register)	ADD ADC	SUB SBC RSB RSC	AND BIC	EOR ORR ORN	MOV MVN
comparison (set flags only)	CMN (ADDS)	CMP (SUBS)	TST (ANDS)	TEQ (EORS)	

- **Syntax:**

```
<Operation>{<cond>}{S} {Rd,} Rn, Operand2
```

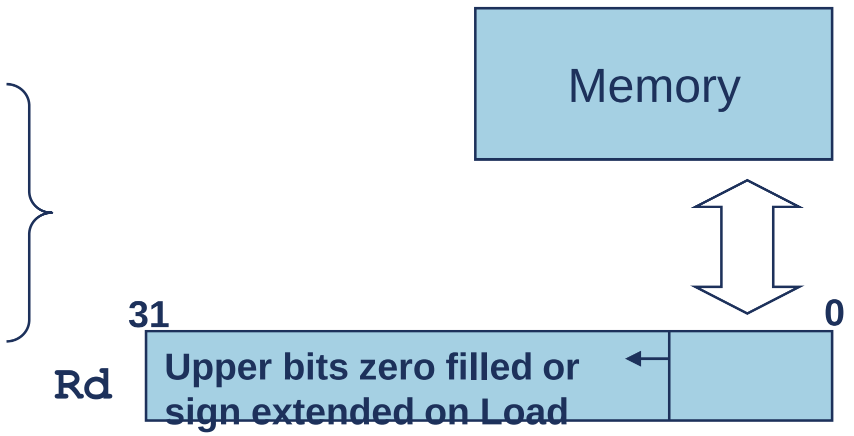
- **Examples:**

```
ADD r0, r1, r2      ; r0 = r1 + r2
TEQ r0, r1          ; if r0 = r1, Z flag will be set
MOV r0, r1          ; copy r1 to r0
```

Single Access Data Transfer

- Use to move data between one or two registers and memory

`LDRD`	`STRD`	Doubleword
`LDR`	`STR`	Word
`LDRB`	`STRB`	Byte
`LDRH`	`STRH`	Halfword
`LDRSB`		Signed byte load
`LDRSH`		Signed halfword load



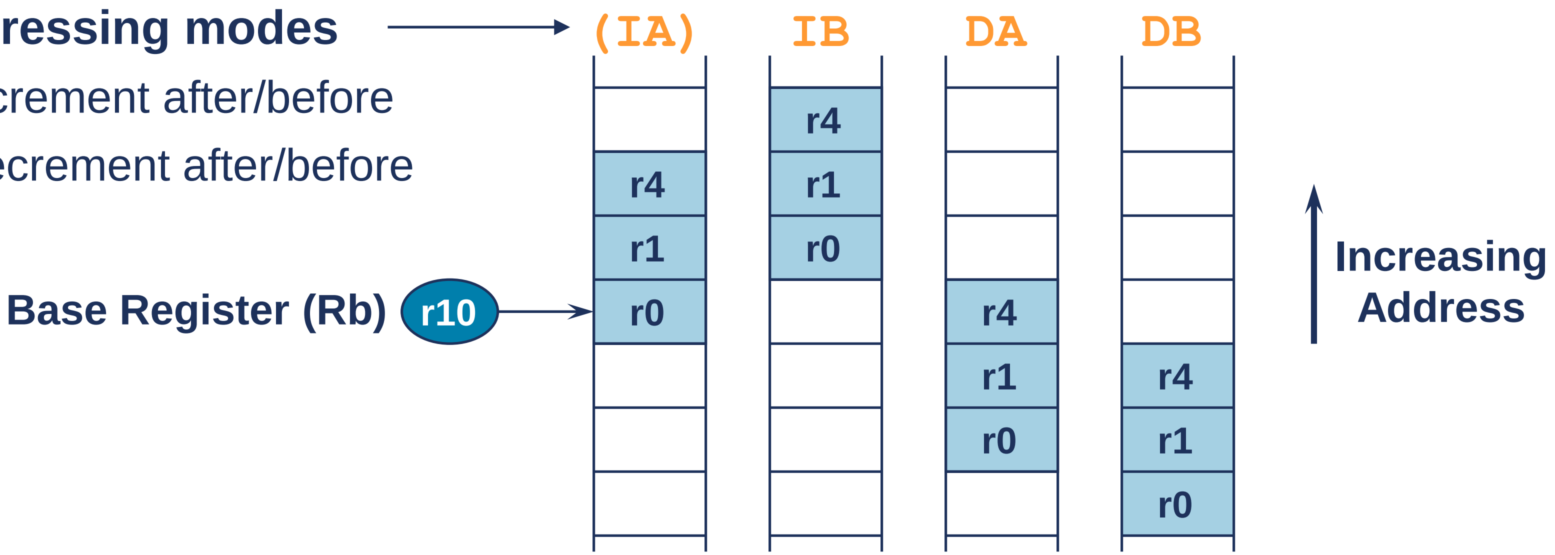
- Syntax:
  - `LDR{<size>}{<cond>} Rd, <address>`
  - `STR{<size>}{<cond>} Rd, <address>`

- Example:

  -
    ```
    LDRB r0, [r1]        ; load bottom byte of r0 from the
                         ; byte of memory at address in r1
    ```

Multiple Register Data Transfer

- **These instructions move data between multiple registers and memory**
- **Syntax**
  - `<LDM|STM>{<addressing_mode>}{<cond>} Rb{!}, <register list>`
- **4 addressing modes**
  - Increment after/before
  - Decrement after/before

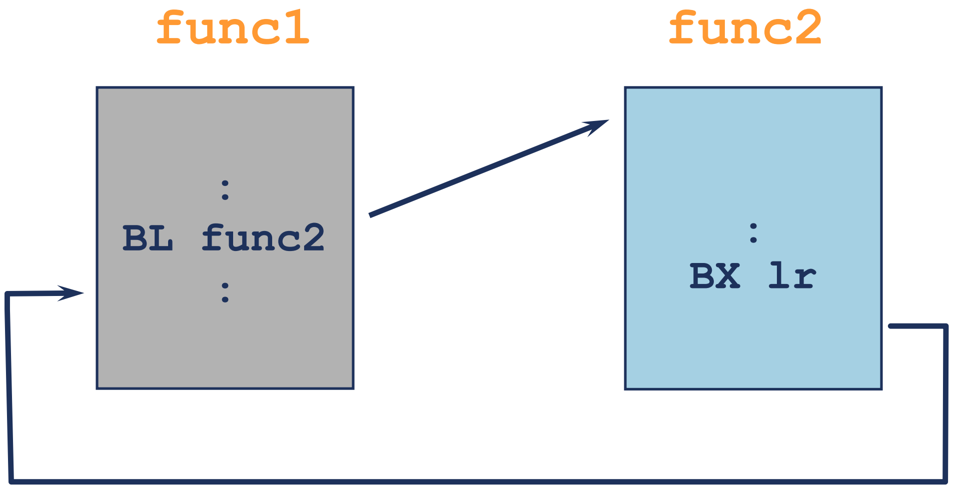


- **Also**
  - `PUSH/POP`, equivalent to `STMDB/LDMIA` with `SP!` as base register
- **Example**
  - `LDM    r10, {r0,r1,r4}     ; load registers, using r10 base`
  - `PUSH   {r4-r6,pc}          ; store registers, using SP base`

Subroutines

- **Implementing a conventional subroutine call requires two steps**
  - Store the return address
  - Branch to the address of the required subroutine
- **These steps are carried out in one instruction, `BL`**
  - The return address is stored in the link register (**`lr/r14`**)
  - Branch to an address (range dependent on instruction set and width)
- **Return is by branching to the address in `lr`**

```
void func1 (void)
{
        :
        func2();
        :
}
```

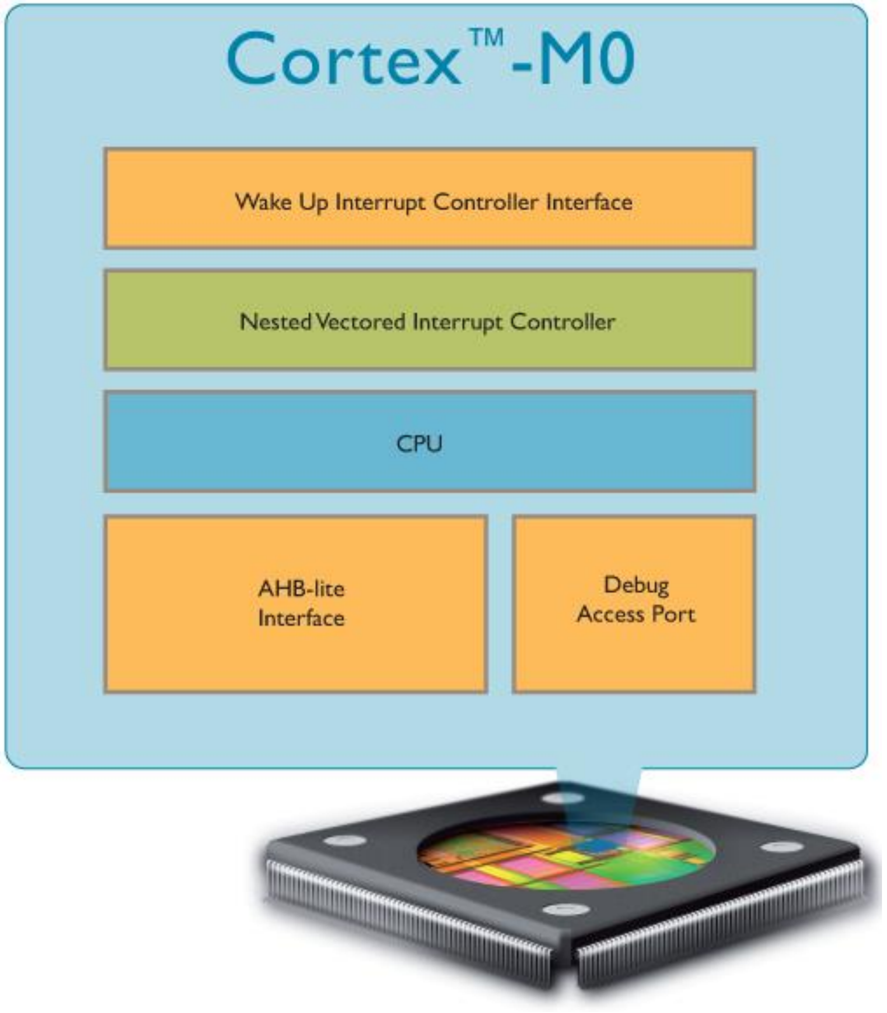


Supervisor Call (SVC)

```
SVC{<cond>} <SVC number>
```

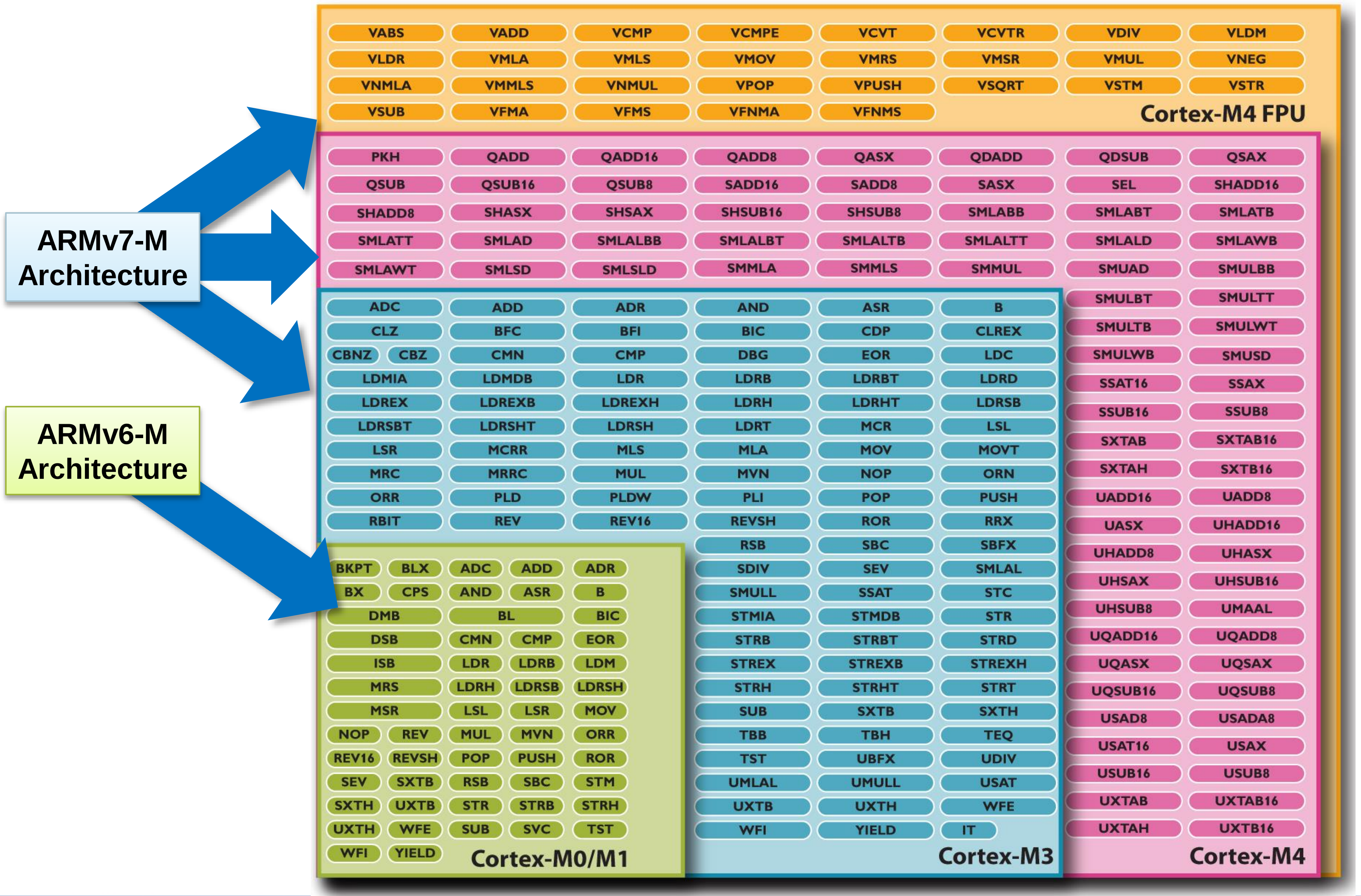
- **Causes an SVC exception**
- **The SVC handler can examine the SVC number to decide what operation has been requested**
  - But the core ignores the SVC number
- **By using the SVC mechanism, an operating system can implement a set of privileged operations (system calls) which applications running in user mode can request**
- **Thumb version is unconditional**

Cortex-M0


Cortex™-M0
Wake Up Interrupt Controller Interface
Nested Vectored Interrupt Controller
CPU
AHB-lite Interface
Debug Access Port

- **ARMv6-M Architecture**
  - 16-bit Thumb-2 with system control instructions
- **Fully programmable in C**
- **3-stage pipeline**
- **von Neuman architecture**
- **AHB-Lite bus interface**
- **Fixed memory map**
- **1-32 interrupts**
  - Configurable priority levels
  - Non-Maskable Interrupt support
- **Low power support**
- **Core configured with or without debug**
  - Variable number of watchpoints and breakpoints

Binary Upwards Compatibility



Instruction set

- only the 16-bit Thumb instructions
- and a minimum subset of 32-bit Thumb instructions

Assembly syntax (ARM assembler)

```
label    mnemonic    operand1, operand2,…    ; Comment
```

label -> used as a reference to an address location (optional) and data address (start at the first column in the line)

Assembly syntax (ARM assembler)

Immediate data: prefixed with "#"

example:

```
label    mnemonic   operand1, operand2,…   ; Comment

         MOVS  R0,#0x1F     ; Set R0 = 0x1F
         MOVS  R0,#'A'      ; Set R0 = 0x41 (ASCII code)
```

Assembly syntax (ARM assembler)

Constant definition:

example:

```
label    mnemonic    operand1, operand2,…   ; Comment

CLK_BA_base    EQU 0x50000200    ; 32-bits
PWRCON         EQU 0x00          ; 8-bits?
```

Assembly syntax (ARM assembler)

Embedded data:

example: (LDR is a pseudo instruction)

```
label    mnemonic   operand1, operand2,…   ; Comment

         LDR    R0,=MyData ; Get the address of MyData
         LDR    R1,[R0]    ; R1 = 0x12345678

         LDR    R0,=MyText ; R0 = the starting addr
         LDR    R1,[R0]    ; R1 = 0x61434241
MyData   DCD    0x12345678
MyText   DCB    "ABCabc0123\n",0 ; Null terminated
                                 ; string
```

Assembly syntax (ARM assembler)

```
    26:           LDR            R0,=MyData     ; Get the address of MyData
0x0000016E 4807       LDR       r0,[pc,#28]   ;
    27:           LDR            R1,[R0]        ; R1 = 0x12345678
    28:
0x00000170 6801       LDR       r1,[r0,#0x00]
    29:           LDR       R0,=MyText          ; R0 = the starting addr
0x00000172 4807       LDR       r0,[pc,#28]   ; @0x00000190
    30:           LDR R1,[R0]
    31:
    32:           ALIGN 4
    33: MyData    DCD       0x12345678
    34: MyText    DCB       "ABCabc0123\n",0          ; Null terminated string
    35: }
0x00000174 6801       LDR       r1,[r0,#0x00]

0x00000176 0000       MOVS      r0,r0
0x00000178 5678       LDRSB     r0,[r7,r1]
0x0000017A 1234       ASRS      r4,r6,#8
0x0000017C 4241       RSBS      r1,r0,#0
0x0000017E 6143       STR       r3,[r0,#0x14]
0x00000180 6362       STR       r2,[r4,#0x34]
0x00000182 3130       ADDS      r1,r1,#0x30
0x00000184 3332       ADDS      r3,r3,#0x32
0x00000186 000A       MOVS      r2,r1
```

Instruction set: Memory access

ADR: Generates a PC-relative address.

example:

```
    20:          ADR R0, MyData
; write address 0x0000016C to R0
; R0 = 0x0000016C
    21:
    22:          ALIGN 4
    23: MyData   DCD         0x12345678

0x00000168 A000      ADR      r0,{pc}+4
                                  ; @0x0000016C
0x0000016A 0000      MOVS     r0,r0
0x0000016C 5678      LDRSB    r0,[r7,r1]
0x0000016E 1234      ASRS     r4,r6,#8
```

Instruction set: Memory access

LDR and STR, immediate offset example:

```
LDR R0, [R5]
; Loads R0 from the address in R5.

STR R1, [R6,#const-struc]
; const-struc is an expression evaluating
; to a constant in the range 0-1020.
```

Instruction set: Memory access

LDR and STR, register offset example:

```
STR R0, [R5, R1]
; Store value of R0 into an address equal to
; sum of R5 and R1

LDRSH R1, [R2, R3]
; Load a halfword from the memory address
; specified by (R2 + R3), sign extend to 32-bits
; and write to R1.
```

Instruction set: Memory access

LDR, PC-relative example:

```
    20:           LDR R0, MyData
; Load R0 with a word of data from an address MyData
; R0 = 0x12345678
    21:
    22:           ALIGN 4
    23: MyData    DCD       0x12345678

0x00000168 4800     LDR      r0,[pc,#0]
                                ; @0x0000016C
0x0000016A 0000     MOVS     r0,r0
0x0000016C 5678     LDRSB    r0,[r7,r1]
0x0000016E 1234     ASRS     r4,r6,#8
```

Instruction set: Memory access

LDM and STM: Load and Store Multiple registers.

example:

```
LDM R0!,{R0,R3,R4}
; LDMIA,LDMFD is a synonym for LDM
; R0=memory[R0], R3=memory[R0+4}, R4=memory[R0+8]

STMIA R1!,{R2-R4,R6}
; memory[R1]=R2, memory[R1+4]=R3, memory[R1+8]=R4,
; memory[R1+12]=R6, and update R1
```

Instruction set: Memory access

PUSH and POP:

example:

```
PUSH {R0,R4-R7}
; Push R0,R4,R5,R6,R7 onto the stack
PUSH {R2,LR}
; Push R2 and the link-register onto the stack

POP {R0,R6,PC}
; Pop r0,r6 and PC from the stack, then branch to
; the new PC.
```

Instruction set: data processing

ADC, ADD, RSB, SBC, and SUB: Add with carry, Add, Reverse Subtract, Subtract with carry, and Subtract.

example: shows two instructions that add a 64-bit integer contained in R0 and R1 to another 64-bit integer contained in R2 and R3, and place the result in R0 and R1.

[R3:R2] = [R1:R0]+[R3:R2]

```
ADDS R0, R0, R2
; add the least significant words

ADCS R1, R1, R3
; add the most significant words with carry
```

Instruction set: data processing

ADC, ADD, RSB, SBC, and SUB: Add with carry, Add, Reverse Subtract, Subtract with carry, and Subtract.

example: shows the RSBS instruction used to perform a 1's complement of a single register.

```
RSBS R7, R7, #0
; subtract R7 from zero
```

S-Suffic indicate an instruction that update APSR(flags: N, Z, C, and V)

Instruction set: data processing

AND, ORR, EOR, and BIC: Logical AND, OR, Exclusive OR, and Bit Clear.

example:

```
ANDS R2, R2, R1
ORRS R2, R2, R5
ANDS R5, R5, R8
EORS R7, R7, R6
BICS R0, R0, R1
```

Instruction set: data processing

ASR, LSL, LSR, and ROR: Arithmetic Shift Right, Logical Shift Left, Logical Shift Right, and Rotate Right.

example:

```
ASRS R7, R5, #9
; Arithmetic shift right by 9 bits
LSLS R1, R2, #3
; Logical shift left by 3 bits with flag update
LSRS R4, R5, #6
; Logical shift right by 6 bits
RORS R4, R4, R6
; Rotate right by the value in
; the bottom byte of R6.
```

Instruction set: data processing

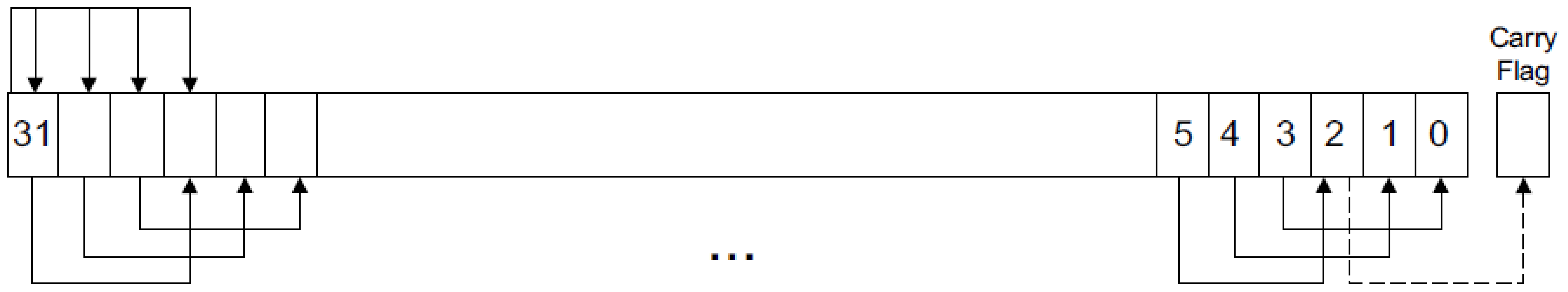


Figure 3-1 ASR #3

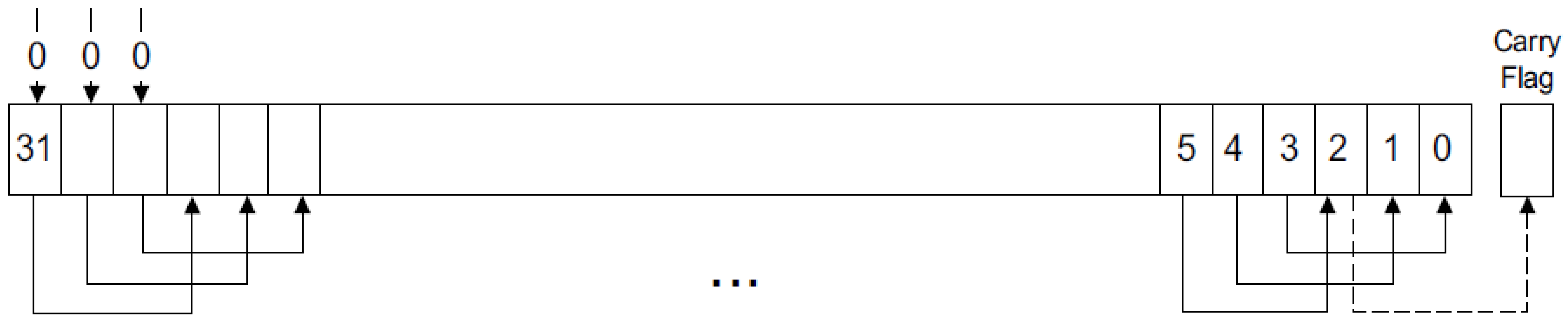


Figure 3-2 LSR #3

Instruction set: data processing

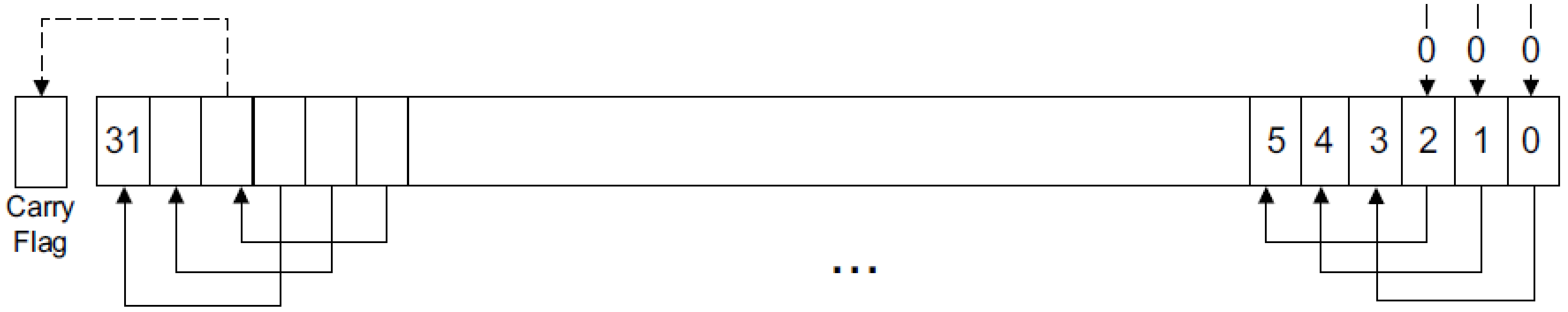


Figure 3-3 LSL #3

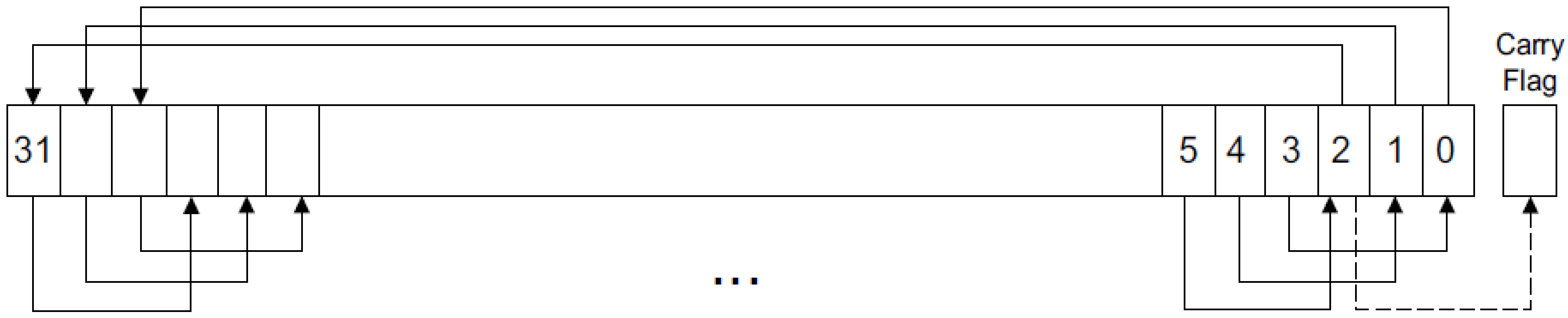


Figure 3-4 ROR #3

Instruction set: data processing

CMP and CMN: Compare and Compare Negative.

example:

```
    CMP R2, R9
    CMN R0, R2
```

Instruction set: data processing

MOV and MVN: Move and Move NOT.

example:

```
MOVS R0, #0x000B
; Write value of 0x000B to R0, flags get updated
MOVS R1, #0x0
; Write value of zero to R1, flags are updated
MOV R10, R12
; Write value in R12 to R10, flags are not updated
MOVS R3, #23
; Write value of 23 to R3
MOV R8, SP
; Write value of stack pointer to R8
MVNS R2, R0
; Write inverse of R0 to the R2 and update flags
```

Instruction set: data processing

MULS: Multiply using 32-bit operands, and producing a 32-bit result.

example:

```
MULS R0, R2, R0
; Multiply with flag update, R0 = R0 x R2
```

Instruction set: data processing

REV, REV16, and REVSH: Reverse bytes.

example:

```
REV R3, R7
; Reverse byte order of value in R7 and write it
; to R3

REV16 R0, R0
; Reverse byte order of each 16-bit halfword in R0

REVSH R0, R5
; Reverse signed halfword
```

Instruction set: data processing

SXT and UXT: Sign extend and Zero extend.

example:

```
SXTH R4, R6
; Obtain the lower halfword of the
; value in R6 and then sign extend to
; 32 bits and write the result to R4.

UXTB R3, R1
; Extract lowest byte of the value in R10 and zero
; extend it, and write the result to R3
```

Instruction set: data processing

TST: Test bits.

example:

```
    TST R0, R1
    ; Perform bitwise AND of R0 value and R1 value,
    ; condition code flags are updated but result is
    ; discarded
```

Instruction set: Branch and control

B, BL, BX, and BLX: Branch instructions.

example:

```
B loopA
; Branch to loopA
BL funC
; Branch with link (Call) to function funC, return
; address stored in LR
BX LR
; Return from function call
BLX R0
; Branch with link and exchange (Call) to a
; address stored in R0
BEQ labelD
; Conditionally branch to labelD if last flag
; setting instruction set the Z flag, else do not branch.
```

Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal, last flag setting result was zero
NE	Z = 0	Not equal, last flag setting result was non-zero
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL	Can have any value	Always. This is the default when no suffix is specified.

Instruction set: Miscellaneous

BKPT: Breakpoint.

example:

```
BKPT #0
; Breakpoint with immediate value set to 0x0.
```

Instruction set: Miscellaneous

CPS: Change Processor State.

example:

```
    CPSID i
    ; Disable all interrupts except NMI (set PRIMASK)

    CPSIE i
    ; Enable interrupts (clear PRIMASK)
```

Instruction set: Miscellaneous

DMB: Data Memory Barrier.

example:

```
DMB
; Data Memory Barrier
```

Instruction set: Miscellaneous

DSB: Data Synchronization Barrier.

example:

```
DSB
; Data Synchronization Barrier
```

Instruction set: Miscellaneous

ISB: Instruction Synchronization Barrier.

example:

```
ISB
; Instruction Synchronization Barrier
```

Instruction set: Miscellaneous

MRS: Move the contents of a special register to a general-purpose register.

example:

```
MRS R0, PRIMASK
; Read PRIMASK value and write it to R0
```

Instruction set: Miscellaneous

MSR: Move the contents of a general-purpose register into the specified special register.

example:

```
MSR CONTROL, R1
; Read R1 value and write it to the CONTROL
; register
```

Instruction set: Miscellaneous

NOP: No operation.

example:

```
NOP
; No operation
```

Instruction set: Miscellaneous

SEV: Send Event.

example:

```
SEV
; Send Event
```

Instruction set: Miscellaneous

SVC: Supervisor Call.

example:

```
SVC #0x32
; Supervisor Call (SVC handler can extract the
; immediate value by locating it using the
; stacked PC)
```

Instruction set: Miscellaneous

WFE: Wait For Event.

example:

```
WFE    ; Wait For Event
```

Instruction set: Miscellaneous

WFI: Wait for Interrupt.

example:

```
WFI    ; Wait for Interrupt
```