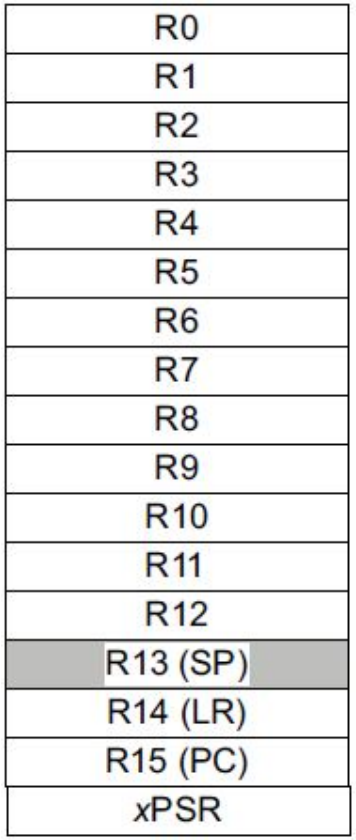
Microprocessor-Based Systems

Dr. Sadr
Yazd University

Instruction Set Architecture
Assembly Language Programming
Software Development Toolchain
Application Binary Interface (ABI)



This week…

Finish ARM assembly example from last time

Walk though of the ARM ISA

Software Development Tool Flow

Application Binary Interface (ABI)

Assembly example

```
data:
     .byte 0x12, 20, 0x20, -1
func:
          mov r0, #0
          mov r4, #0
          movw   r1, #:lower16:data
          movt   r1, #:upper16:data
top:      ldrb   r2, [r1],#1
          add r4, r4, r2
          add r0, r0, #1
          cmp r0, #4
          bne top
```

Instructions used

- mov
  - Moves data from register or immediate.
  - Or also from shifted register or immediate!
    - the mov assembly instruction maps to a bunch of different encodings!
  - If immediate it might be a 16-bit or 32-bit instruction
    - Not all values possible
    - why? (not greater than 2^{32}-1)
- movw
  - Actually an alias to mov
    - “w” is “wide”
    - hints at 16-bit immediate

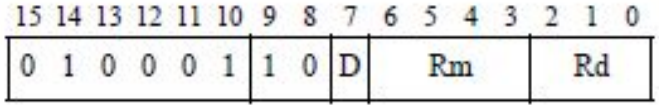
From the ARMv7-M Architecture Reference Manual

A6.7.76 MOV (register)

Move (register) copies a value from a register to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 ARMv6-M, ARMv7-M — If <Rd> and <Rm> both from R0-R7, otherwise all versions of the Thumb ISA.

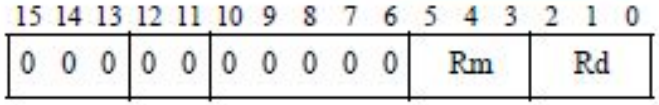
`MOV<c> <Rd>,<Rm>` — If <Rd> is the PC, must be outside or last in IT block




```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb ISA.

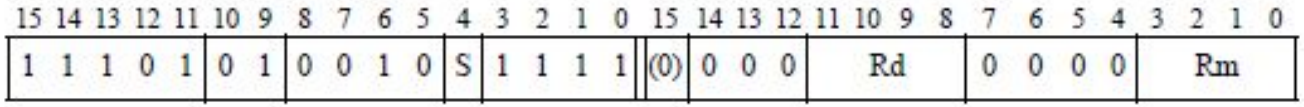
`MOVS <Rd>,<Rm>` (formerly ~~`LSL <Rd>,<Rm>,#0`~~) — Not permitted inside IT block




```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
if InITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

`MOV{S}<c>.W <Rd>,<Rm>`




```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if setflags && (d IN {13,15} || m IN {13,15}) then UNPREDICTABLE;
if !setflags && (d == 15 || m == 15 || (d == 13 && m == 13)) then UNPREDICTABLE;
```

There are similar entries for move immediate, move shifted (which actually maps to different instructions) etc.

ARM and Thumb Encodings:

- Encoding T ▫ Thumb encoding.
- **Different processors** have **different encodings** for a single instruction leading to several encodings, A1, A2, T1, T2, ...

- <u>The Thumb instruction set:</u>
- Each Thumb instruction is either a s**ingle 16-bit halfword**, or a **32-bit instruction consisting of two consecutive halfwords**, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model.
- Thumb instructions operate with the standard ARM register configuration, allowing **excellent interoperability** between ARM and Thumb states.
- On execution, 16-bit Thumb instructions are **decompressed** to full 32-bit ARM instructions in real time, **without performance loss**.
- Thumb code is typically **65% of the size** of ARM code, and provides **160% of the performance** of ARM code when running from a 16-bit memory system. Thumb, therefore, makes the corresponding core ideally suited to embedded applications with **restricted memory bandwidth**, where code density and footprint is important.
- The **different encoding of the same instructions** of which one is ARM and another is Thumb would come from the **encoding policy**.

Directives

- `#:lower16:data`
  - What does that do?
  - Why?

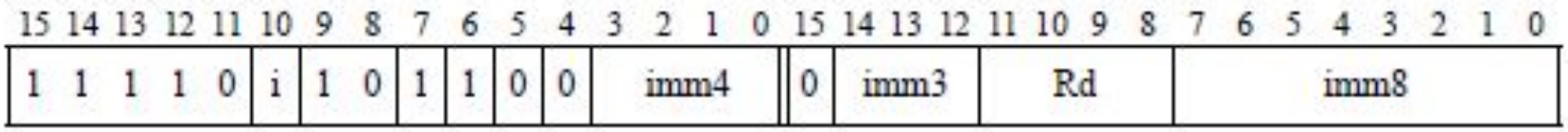
- Note:
  - “data” is a label for a memory address!

A6.7.78 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

Encoding T1 ARMv7-M

```
MOVT<c> <Rd>,#<imm16>
```




```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
MOVT<c><q>  <Rd>, #<imm16>
```

where:

<c><q>	See *Standard assembler syntax fields* on page A6-7.
<Rd>	Specifies the destination register.
<imm16>	Specifies the immediate value to be written to <Rd>. It must be in the range 0-65535.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

Loads!

- **ldrb** -- Load register byte
  - Note this takes an 8-bit value and moves it into a 32-bit location!
    - Zeros out the top 24 bits

- **ldrsb** -- Load register signed byte
  - Note this also takes an 8-bit value and moves it into a 32-bit location!
    - Uses sign extension for the top 24 bits

Addressing Modes

- Offset Addressing
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
- Pre-indexed Addressing
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
- Post-indexed Addressing
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>

Instruction	Effective Address (Final R1 value)	
LDR R2, [R0]		→ Load R2 with the word pointed by R0
LDR R0, [R1, #20]	R1 + 20	→ loads R0 with the word pointed at by R1+20
LDR R0, [R1, #4]!	R1 + 4	→ loads R0 with the word pointed at by R1+4 ; then update the pointer by adding 4 to R1
LDR R0, [R1], #4	R1	→ loads R0 with the word pointed at by R1 ; then update the pointer by adding 4 to R1

So what does the program _do_?

```
data:
    .byte 0x12, 20, 0x20, -1
func:
        mov r0, #0
        mov r4, #0
        movw   r1, #:lower16:data
        movt   r1, #:upper16:data
top:    ldrb   r2, [r1],#1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne top
```

- **CMP Rn, Operand2** :
- Compare the value in a register with Operand2 and **update the condition flags** on the result, but **do not place** the result in any register.
- Condition flags These instructions update the N, Z, C and V flags according to the result.
- The CMP instruction **subtracts** the value of Operand2 from the value in Rn. This is the same as a **SUBS instruction**, except that the **result is discarded**.
- BNE (branch if not equal) cmp comes before branch operations

This Week…

Finish ARM assembly example from last time

Walk though of the ARM ISA

Software Development Tool Flow

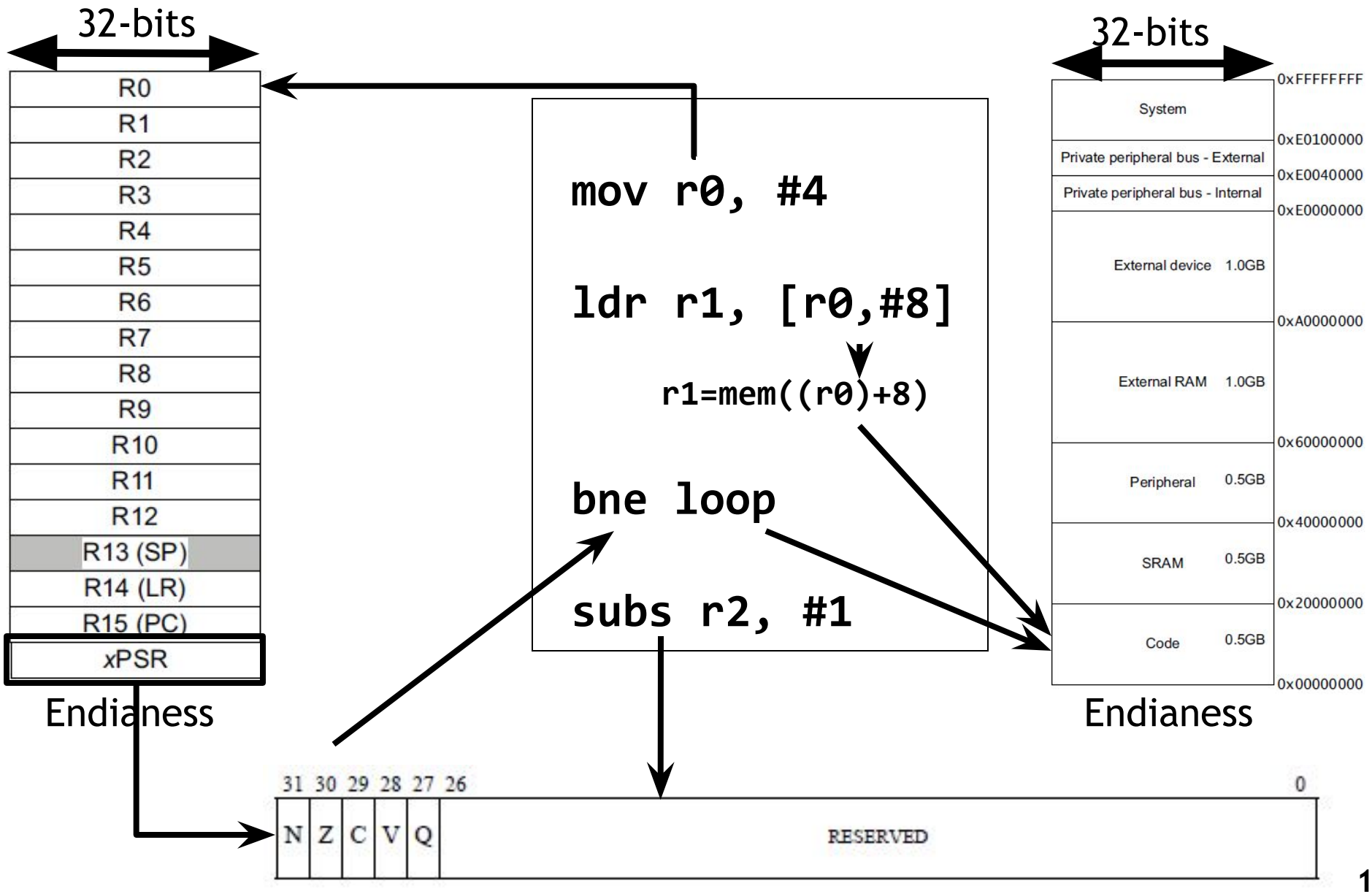
Application Binary Interface (ABI)

An ISA defines the hardware/software interface

- A “contract” between architects and programmers
- Register set
- Instruction set
  - Addressing modes
  - Word size
  - Data formats
  - Operating modes
  - Condition codes

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



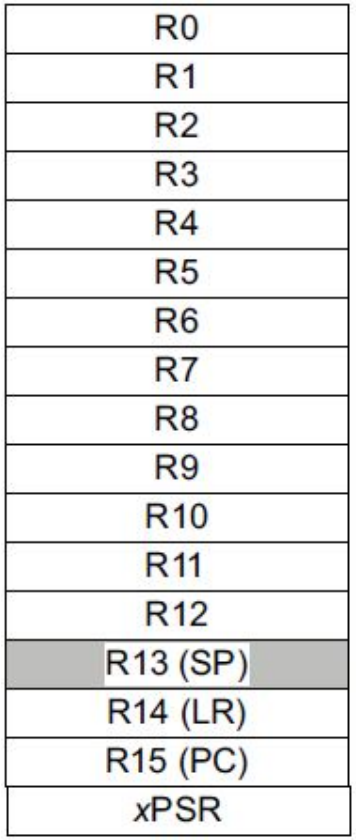
ARM Cortex-M3 ISA

Instruction Set

```
ADD Rd, Rn, <op2>
```

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

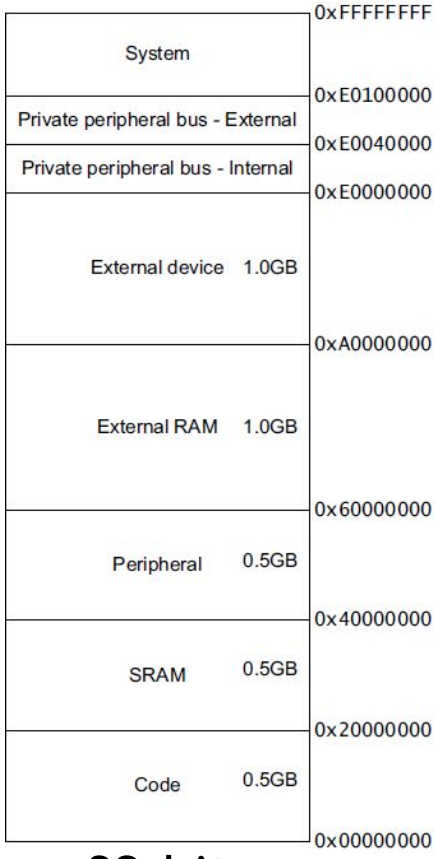
Register Set



32-bits

Endianess

Address Space

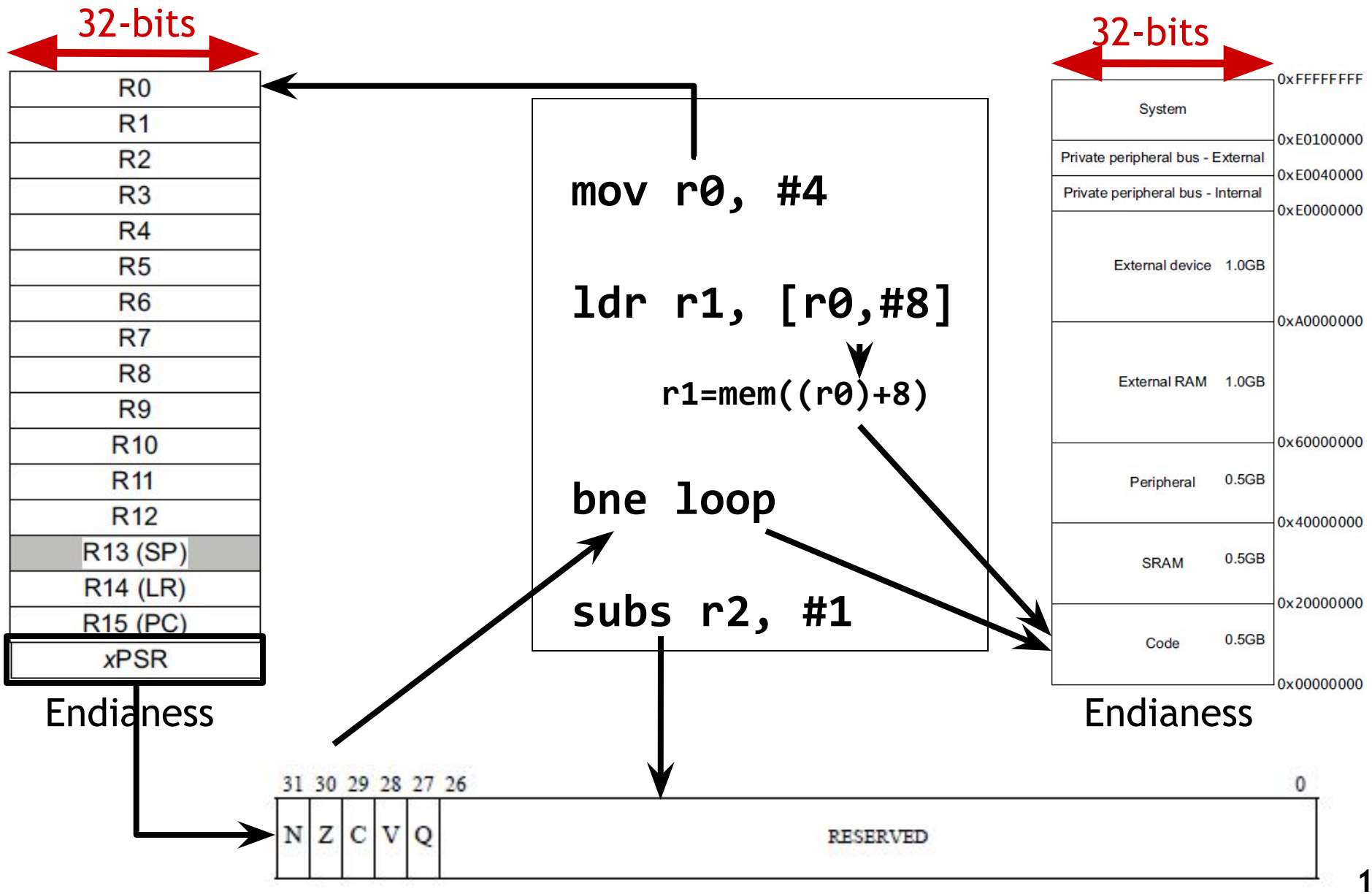


32-bits

Endianess

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)

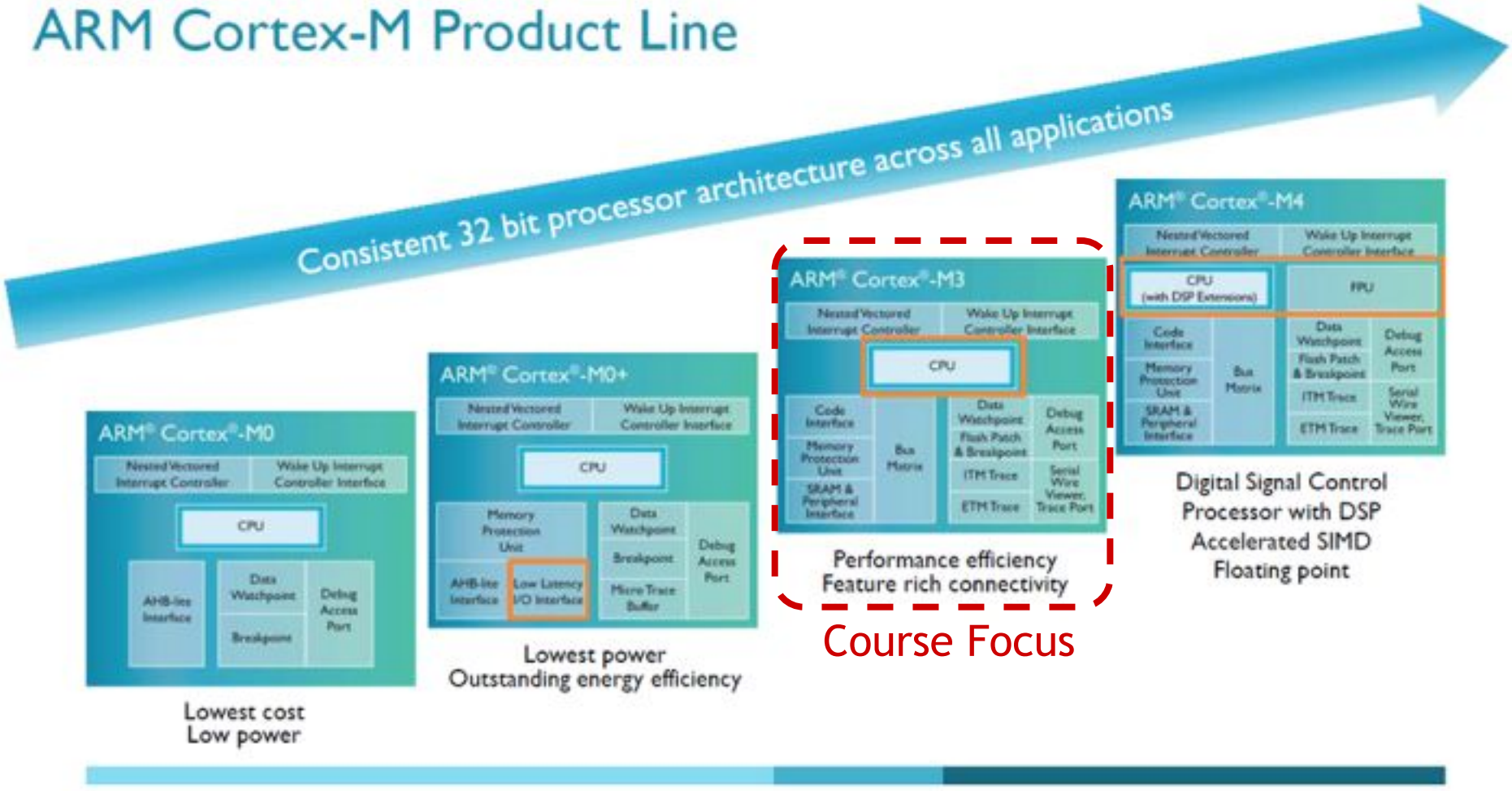


Word Size

- Perhaps most defining feature of an architecture
  - IA-32 (Intel Architecture, 32-bit)
- Word size is what we're referring to when we say
  - 8-bit, 16-bit, 32-bit, or 64-bit machine, microcontroller, microprocessor, or computer
- Determines the size of the addressable memory
  - A 32-bit machine can address 2^32 bytes
  - 2^32 bytes = 4,294,967,296 bytes = 4GB
  - Note: just because you can address it doesn't mean that there's actually something there!
- In embedded systems, tension between 8/16/32 bits
  - Code density/size/expressiveness
  - CPU performance/addressable memory

Word Size ▫ 32-bit ARM Architecture

- ARM's Thumb-2 adds 32-bit instructions to 16-bit ISA
- Balance between 16-bit density and 32-bit performance



A quick comment on the ISA: From: ARMv7-M Architecture Reference Manual

A4.1 About the instruction set

ARMv7-M supports a large number of 32-bit instructions that were introduced as Thumb-2 technology into the Thumb instruction set. Much of the functionality available is identical to the ARM instruction set supported alongside the Thumb instruction set in ARMv6T2 and other ARMv7 profiles. This chapter describes the functionality available in the ARMv7-M Thumb instruction set, and the *Unified Assembler Language* (UAL) that can be assembled to either the Thumb or ARM instruction sets.

Thumb instructions are either 16-bit or 32-bit, and are aligned on a two-byte boundary. 16-bit and 32-bit instructions can be intermixed freely. Many common operations are most efficiently executed using 16-bit instructions. However:

- Most 16-bit instructions can only access eight of the general purpose registers, R0-R7. These are known as the low registers. A small number of 16-bit instructions can access the high registers, R8-R15.
- Many operations that would require two or more 16-bit instructions can be more efficiently executed with a single 32-bit instruction.

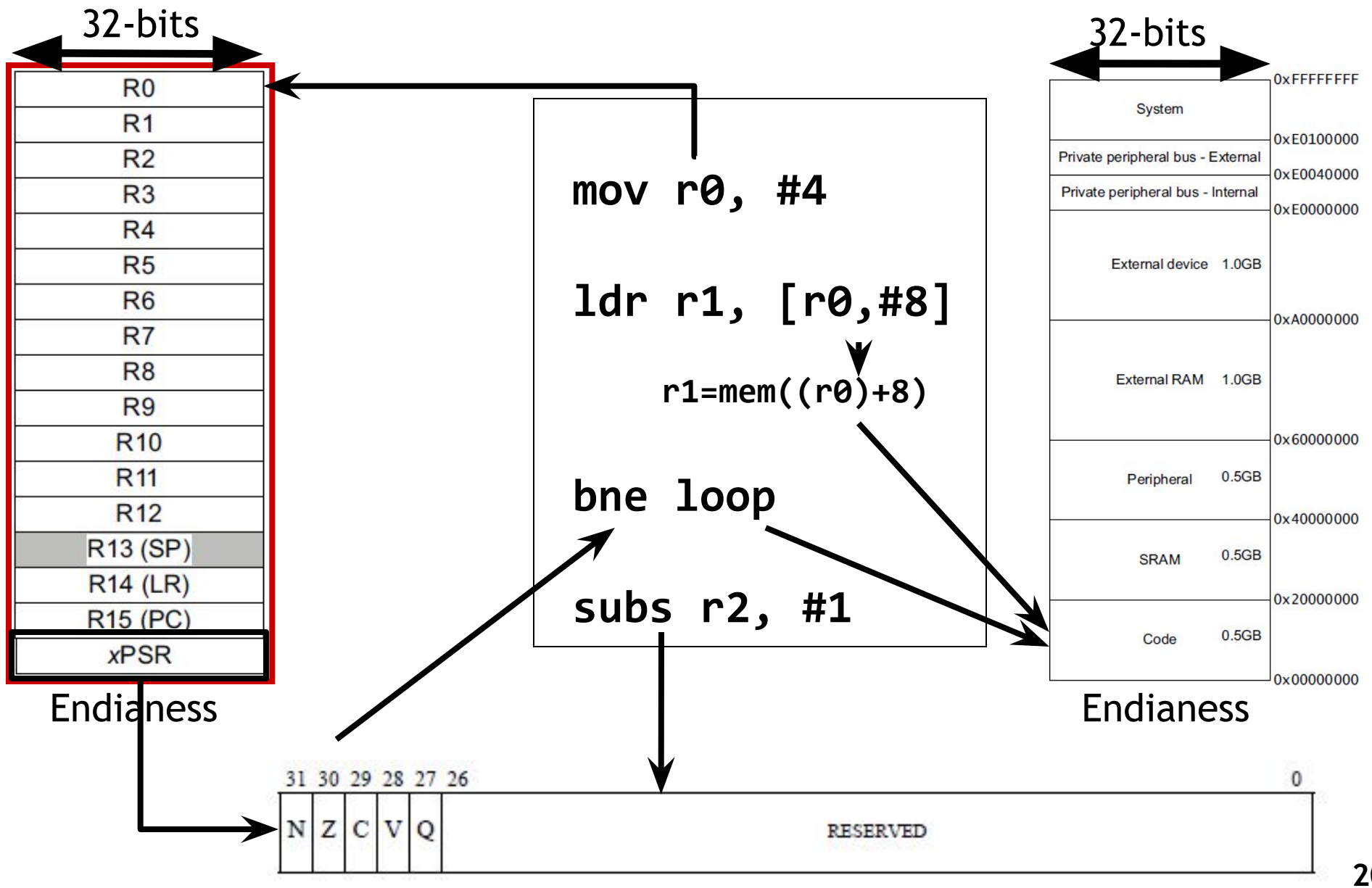
The ARM and Thumb instruction sets are designed to *interwork* freely. Because ARMv7-M only supports Thumb instructions, interworking instructions in ARMv7-M must only reference Thumb state execution, see *ARMv7-M and interworking support* for more details.

In addition, see:

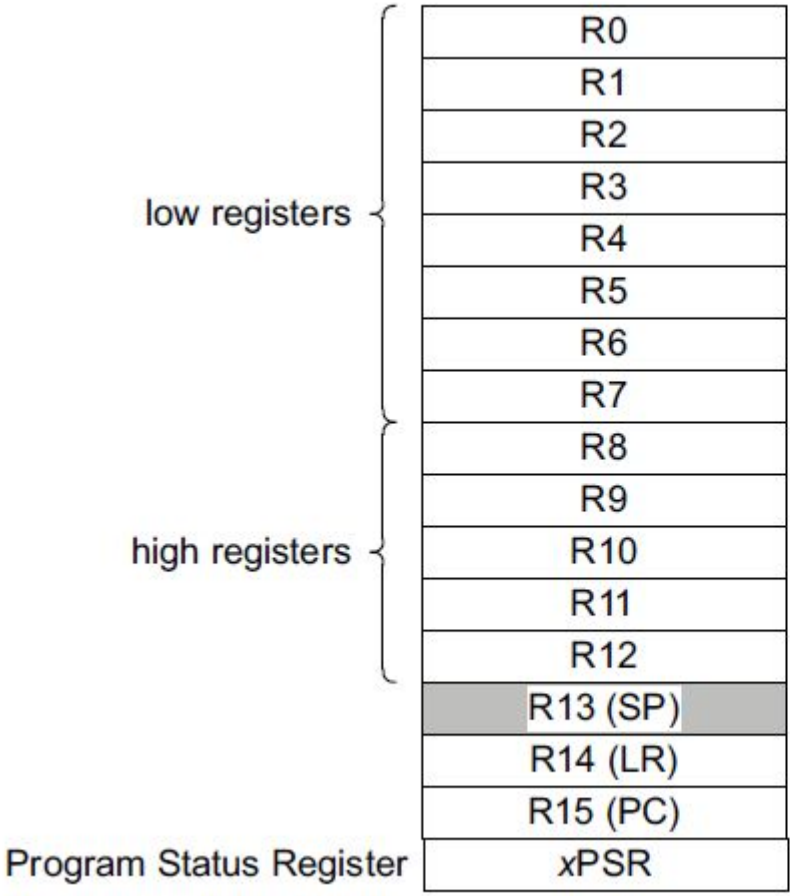
- Chapter A5 *Thumb Instruction Set Encoding* for encoding details of the Thumb instruction set
- Chapter A6 *Thumb Instruction Details* for detailed descriptions of the instructions.

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



ARM Cortex-M3 Registers



- R0-R12
  - General-purpose registers
  - Some 16-bit (Thumb) instruction only access R0-R7
- R13 (SP, PSP, MSP)
  - Stack pointer(s)
  - More details on next slide
- R14 (LR)
  - Link Register
  - When a subroutine is called, return address kept in LR
- R15 (PC)
  - Holds the currently executing program address
  - Can be written to control program flow

ARM Cortex-M3 Registers

- The **Stack** is a **memory region** within the program/process. This part of the memory gets allocated **when a process is created**. We use Stack for storing **temporary data** (local variables/environment variables)
- When the processor pushes a new item onto the stack, it **decrements the stack pointer** and then writes the item to the new memory location.
- The processor implements two stacks, the **main stack** and the **process stack**, with a pointer for each held in independent registers
- When an application is started on an operating system and a process is created, **MSP** mostly used by **OS kernel** itself but **PSP** is mostly by **application itself**.

Note: there are two stack pointers!

Process SP (PSP) used by:
- Base app code (when not running an exception handler)

Main SP (MSP) used by:
- OS kernel
- Exception handlers
- App code w/ priviledged access

Mode dependent

ARM Cortex-M3 Registers

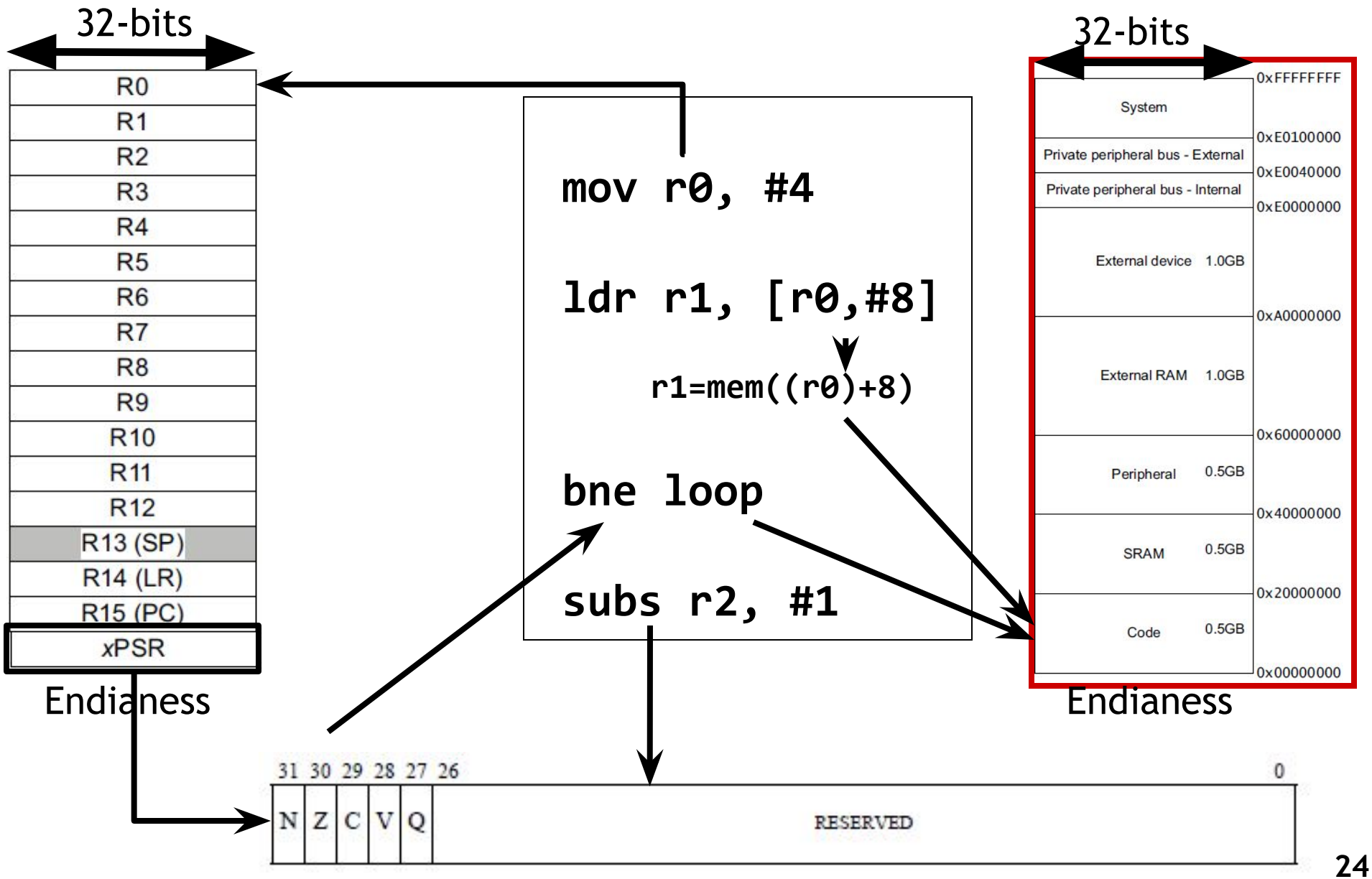
- xPSR
  - Program Status Register
  - Provides arithmetic and logic processing flags
  - We'll return to these later

- PRIMASK, FAULTMASK, BASEPRI
  - Interrupt mask registers
  - PRIMASK: disable all interrupts except NMI and hard fault
  - FAULTMASK: disable all interrupts except NMI
  - BASEPRI: Disable all interrupts of specific priority level or lower
  - We'll return to these during the **interrupt lectures**

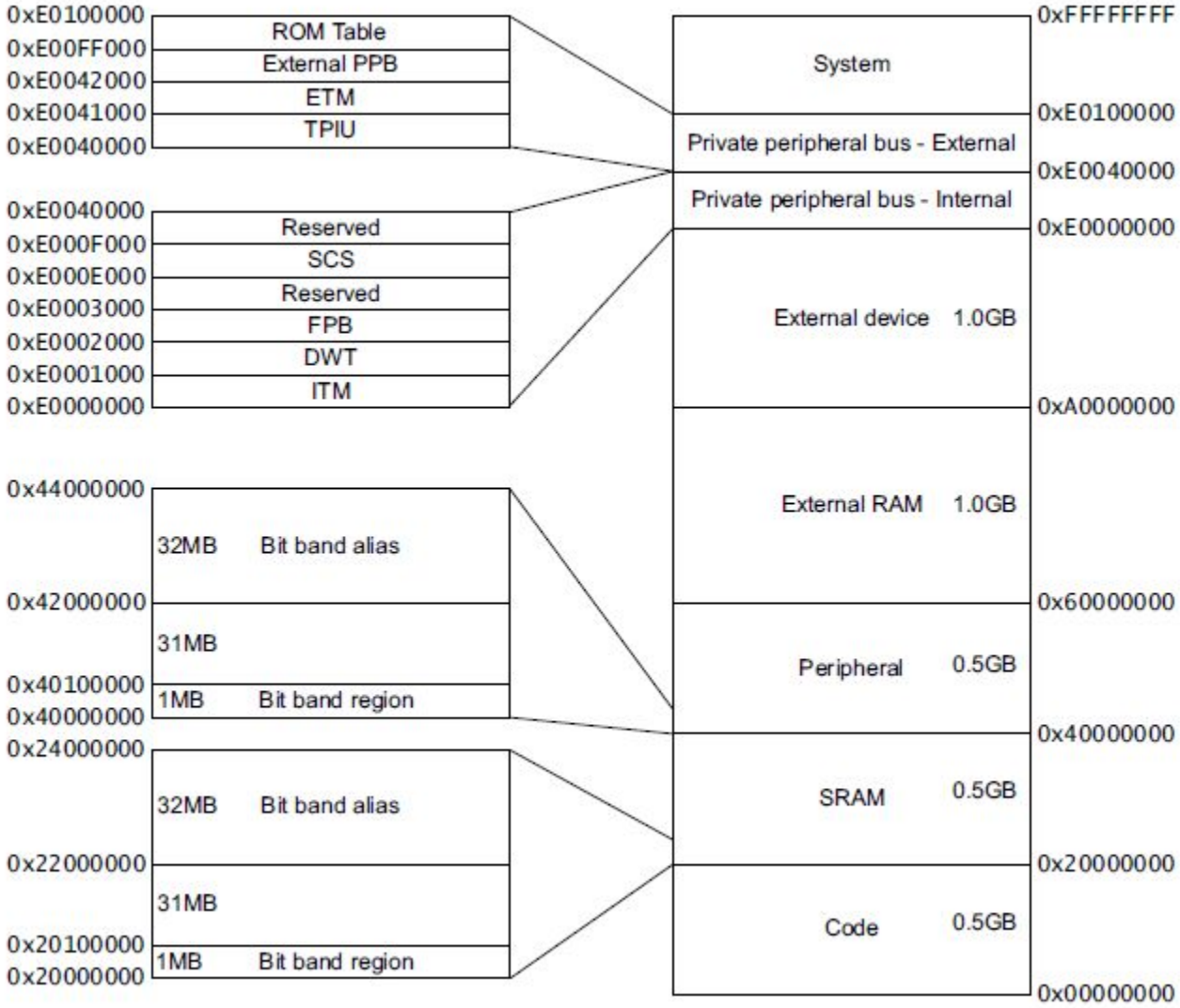
- CONTROL (control register)
  - Define **priviledged status** and **stack pointer selection (PSP, MSP)**
  - The CONTROL register is one of the special registers implemented in the Cortex-M processors. This can be accessed using **MSR** and **MRS** instructions.

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



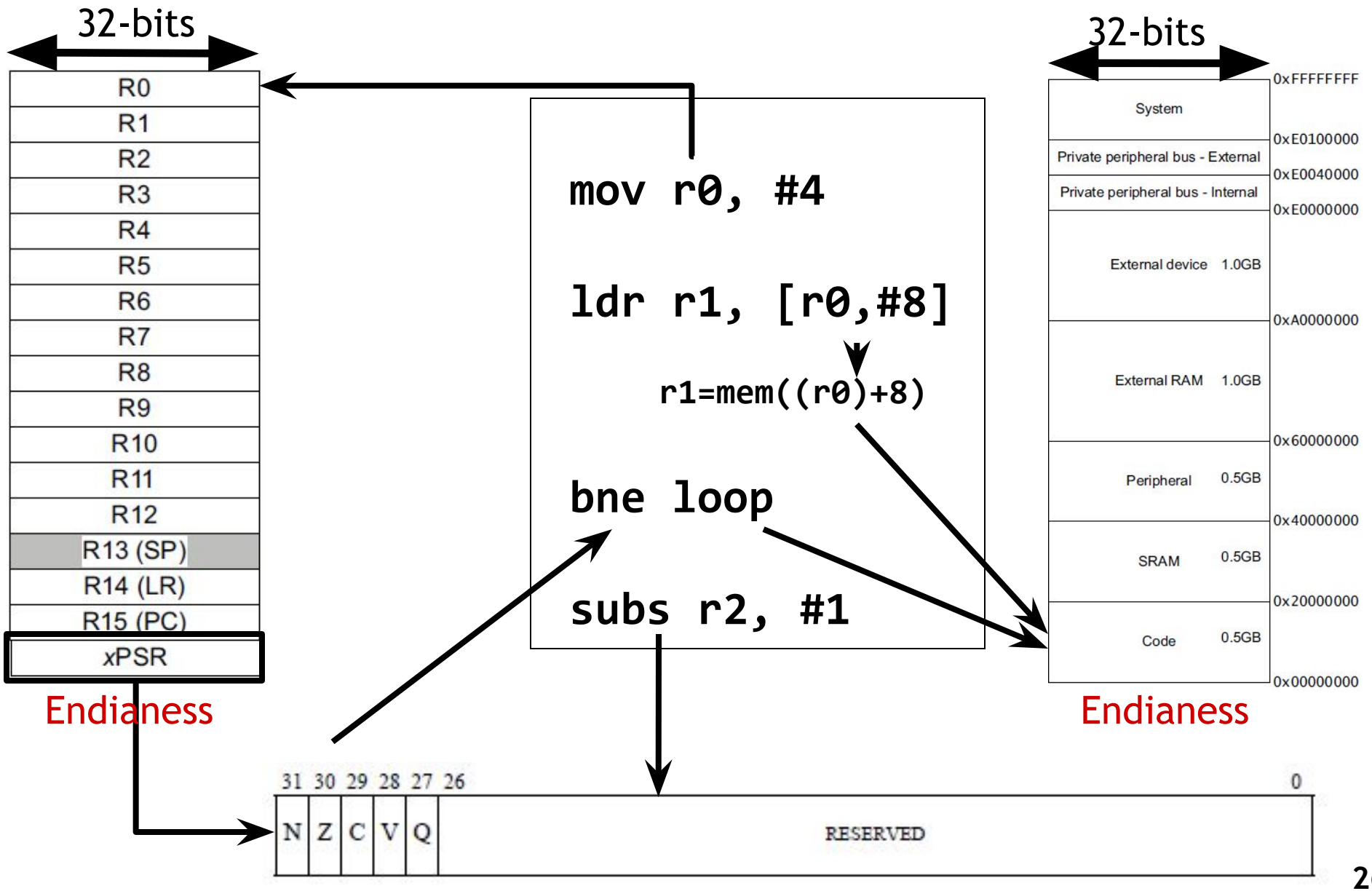
ARM Cortex-M3 Address Space / Memory Map



Unlike most previous ARM cores, the overall **layout of the memory map** of a device based around the Cortex-M3 is **fixed**. This allows easy porting of software between **different systems** based on the Cortex-M3. The address space is split into a number of different sections.

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



The endianess religious war: 289 years and counting!

- Modern version
  - Danny Cohen
  - IEEE Computer, v14, #10
  - Published in 1981
  - Satire on CS religious war

- Historical Inspiration
  - Jonathan Swift
  - *Gulliver's Travels*
  - Published in 1726
  - Satire on Henry-VIII's split with the Church
    - Now a major motion picture!

- Little-Endian
  - LSB is at lower address

```
                                Memory     Value
                                Offset   (LSB) (MSB)
                                ======   ===========
uint8_t a  = 1;                 0x0000   01 02 FF 00
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;        0x0004   78 56 34 12
```

- Big-Endian
  - MSB is at lower address

```
                                Memory     Value
                                Offset   (LSB) (MSB)
                                ======   ===========
uint8_t a  = 1;                 0x0000   01 02 00 FF
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;        0x0004   12 34 56 78
```

Endian-ness

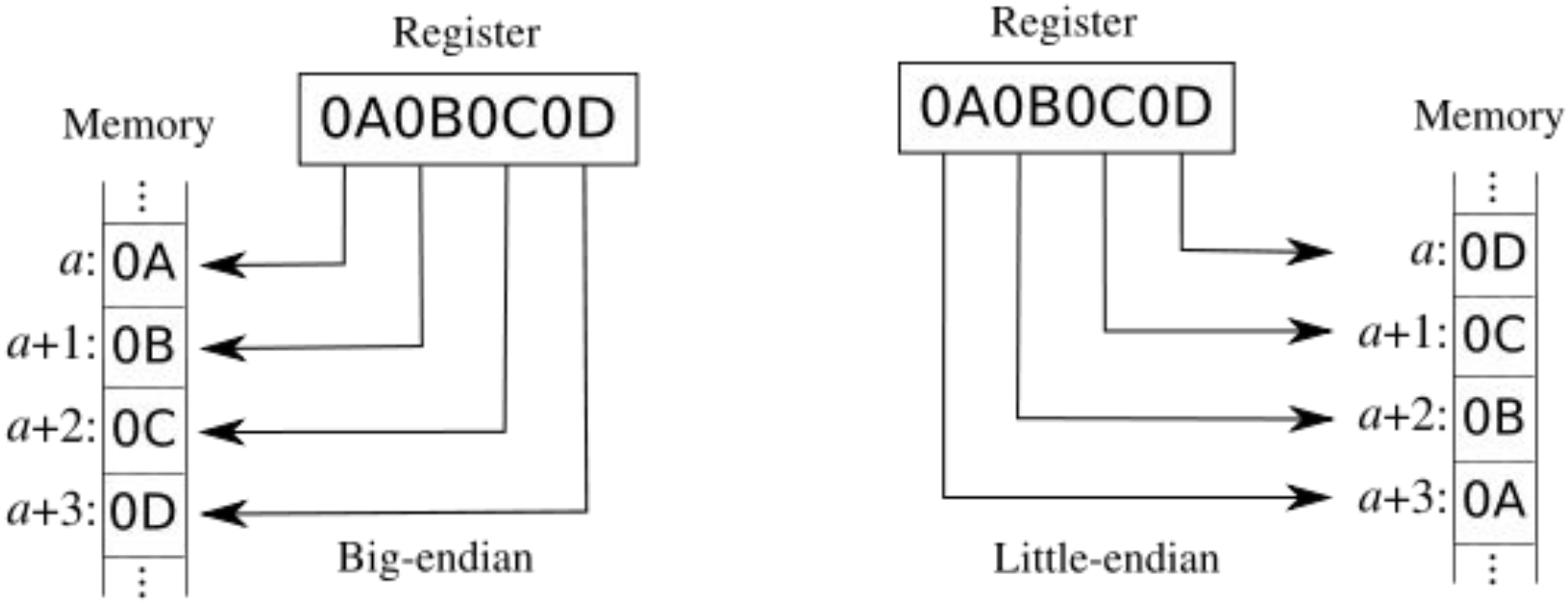
- Endian-ness includes 2 types ▫
  - Little endian : Little endian processors order bytes in memory with the least significant byte of a multi-byte value in the lowest-numbered memory location.
  - Big endian : Big endian architectures instead order them with the most significant byte at the lowest-numbered address.

- The **x86** architecture as well as several **8-bit architectures** are **little endian**.
- Most **RISC architectures** (SPARC, Power, PowerPC, MIPS) were originally **big endian** (ARM was little endian), but many (including **ARM**) are now **configurable**.
- Endianness **only** applies to processors that **allow individual addressing** of units of data (such as bytes) that are smaller than the basic addressable machine word.

- RISC ▫ Reduced Instruction Set Computer (exp. ARM)
- CISC ▫ Complex Instruction Set Computer (x86 processors in most PCs)
- Processors that have a **RISC** architecture typically require **fewer transistors** than those with a CISC architecture which improves **cost, power** consumption, and **heat** dissipation.

Addressing: Big Endian vs Little Endian

- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big – The opposite, most significant byte first
  - MIPS is big endian, x86 is little endian



ARM Cortex-M3 Memory Formats (Endian)

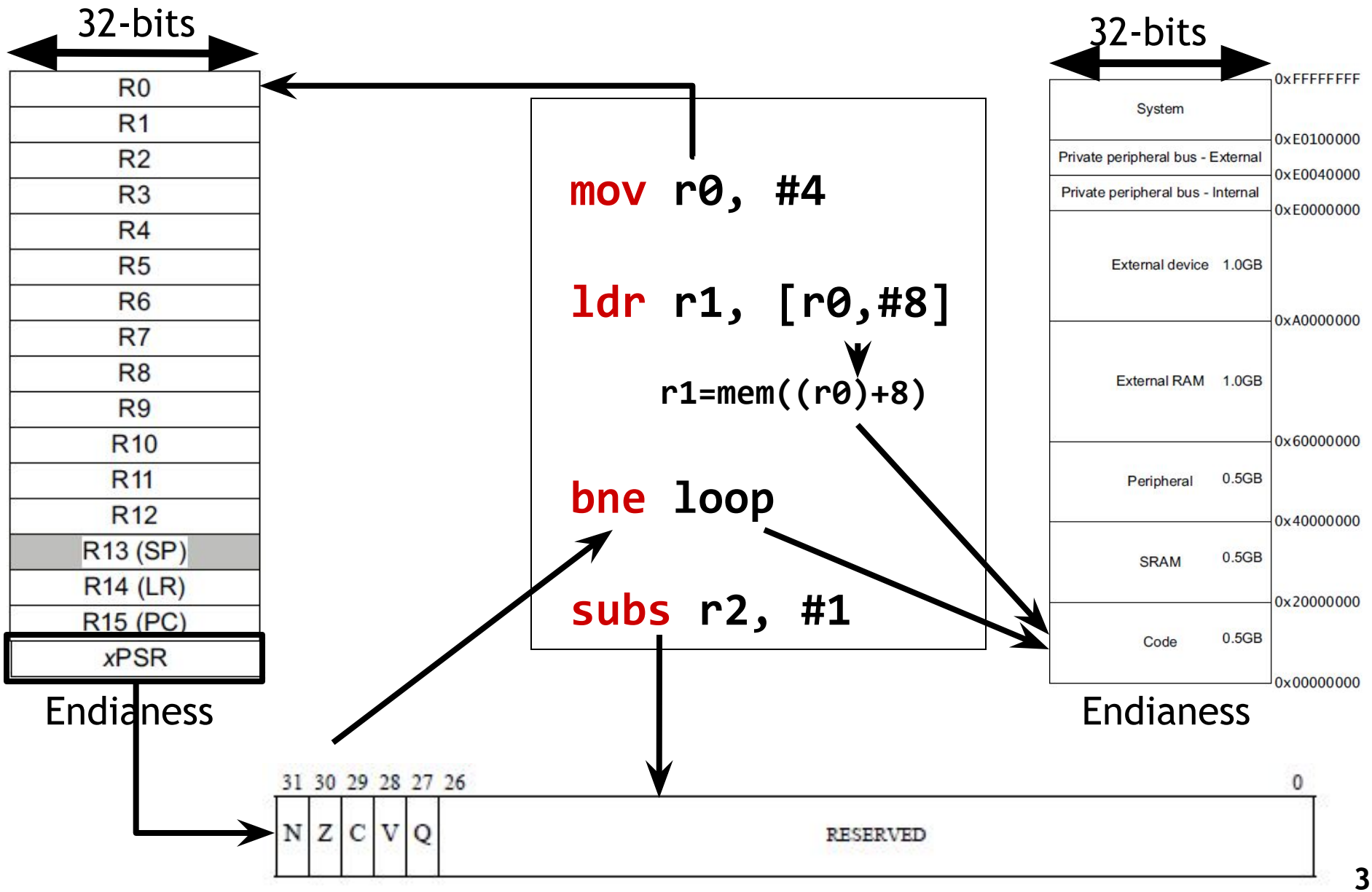
- Default memory format for ARM CPUs: LITTLE ENDIAN

- Processor contains a configuration pin BIGEND
  - Enables hardware system developer to select format:
    - Little Endian
    - Big Endian (BE-8)
  - Pin is sampled on reset
  - Cannot change endianness when out of reset

- Source: [ARM TRM] ARM DDI 0337E, "Cortex-M3 Technical Reference Manual," Revision r1p1, pg 67 (2-11).

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



Instruction encoding

- Instructions are encoded in machine language opcodes
- Sometimes
  - Necessary to hand generate opcodes
  - Necessary to verify if assembled code is correct
- How? Refer to the "ARM ARM"

Instructions
`movs r0, #10`
`movs r1, #0`

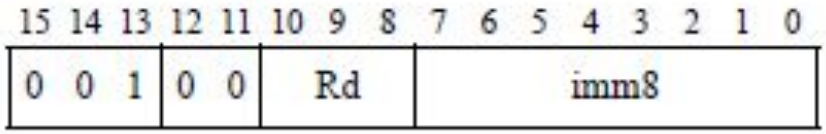
Register Value	Memory Value
001|00|000|00000010	(LSB) (MSB)
(msb) (lsb)	0a 20 00 21
001|00|001|00000000	

ARMv7 ARM

Encoding T1 All versions of the Thumb ISA.

MOVS <Rd>,#<imm8> Outside IT block.

MOV<c> <Rd>,#<imm8> Inside IT block.

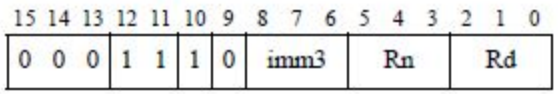



```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Instruction Encoding
ADD immediate

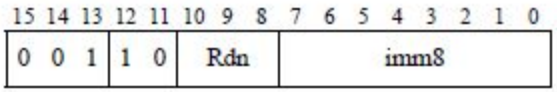
Encoding T1 All versions of the Thumb ISA.

```
ADDS <Rd>,<Rn>,#<imm3>
ADD<c> <Rd>,<Rn>,#<imm3>
```



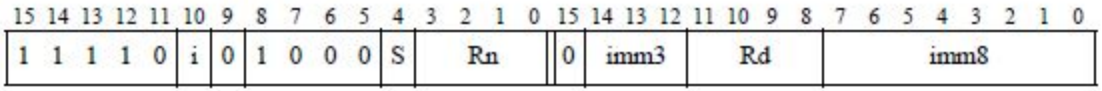
Encoding T2 All versions of the Thumb ISA.

```
ADDS <Rdn>,#<imm8>
ADD<c> <Rdn>,#<imm8>
```



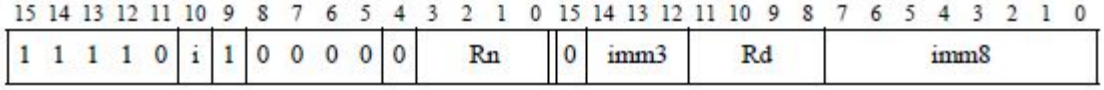
Encoding T3 ARMv7-M

```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```



Encoding T4 ARMv7-M

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

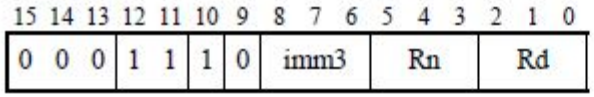


A6.7.3 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

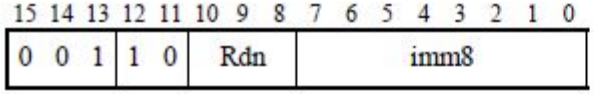
```
ADDS <Rd>,<Rn>,#<imm3>                    Outside IT block.
ADD<c> <Rd>,<Rn>,#<imm3>                  Inside IT block.
```




```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb ISA.

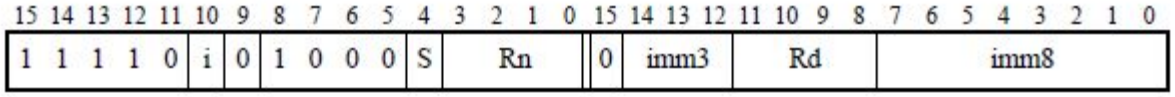
```
ADDS <Rdn>,#<imm8>                        Outside IT block.
ADD<c> <Rdn>,#<imm8>                      Inside IT block.
```




```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv7-M

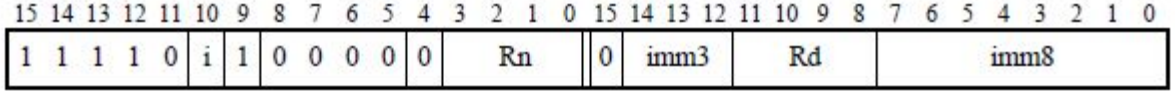
```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```




```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

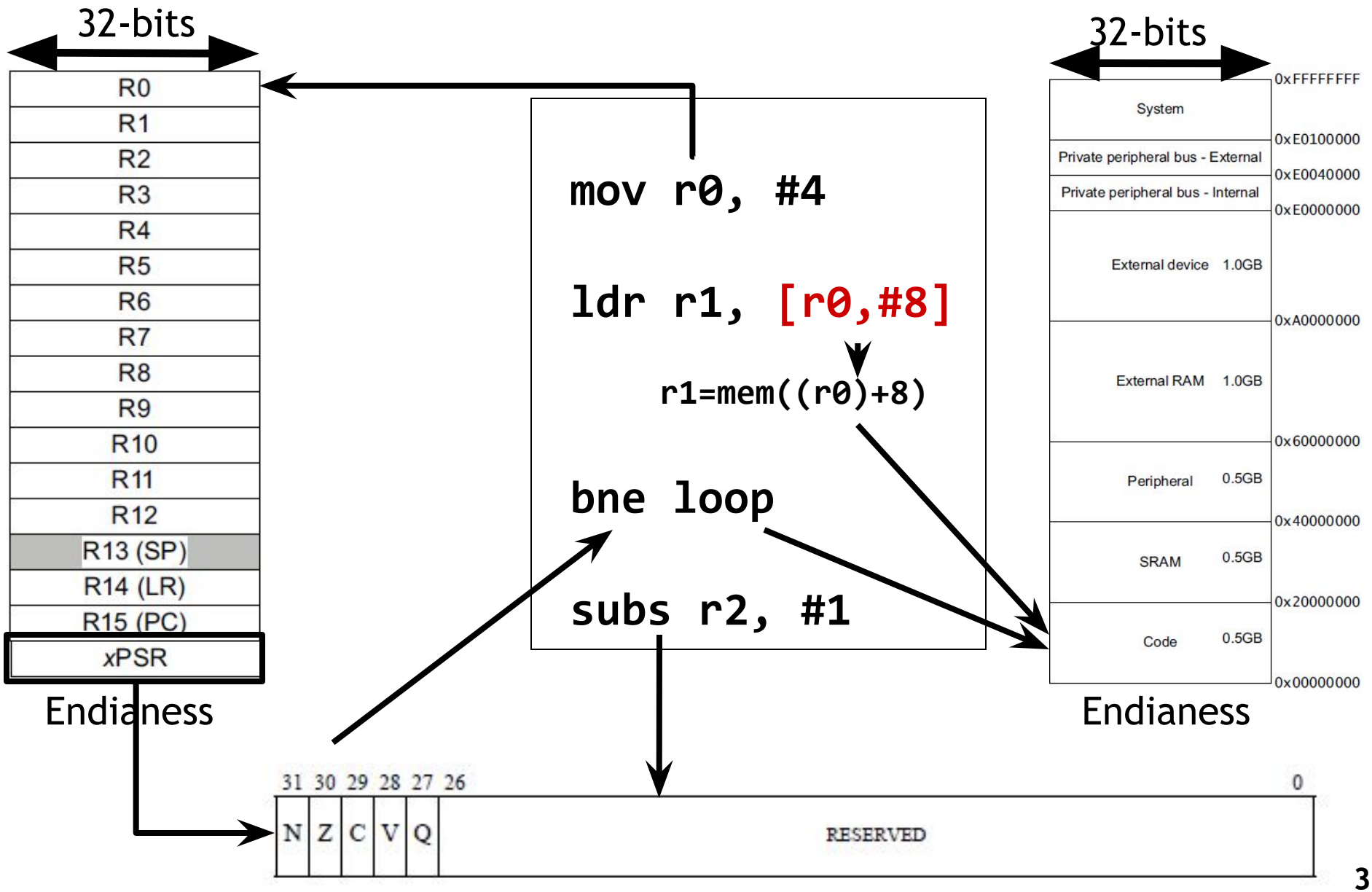
```
ADDW<c> <Rd>,<Rn>,#<imm12>
```




```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



Addressing Modes

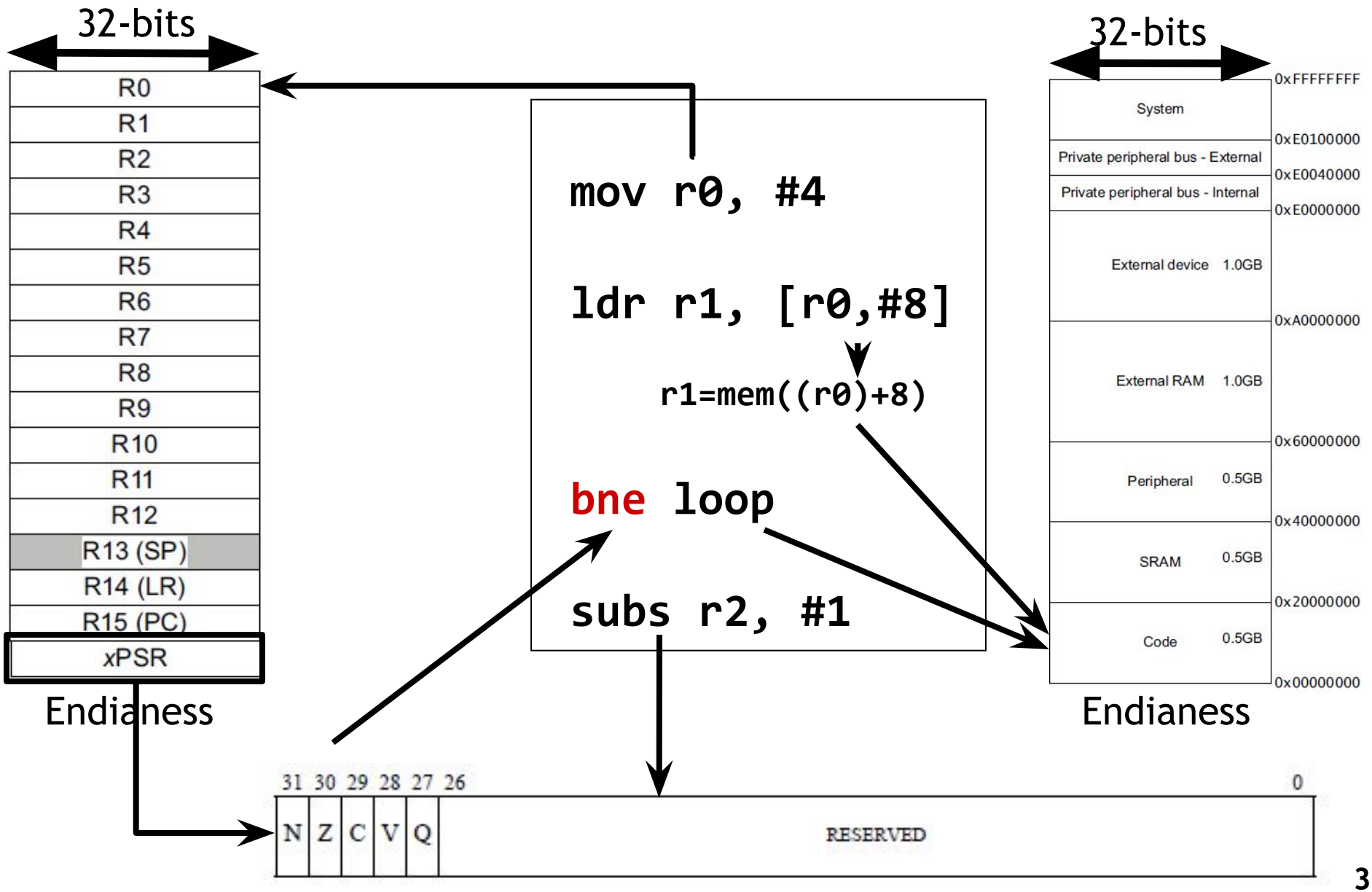
- Offset Addressing
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
- Pre-indexed Addressing
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
- Post-indexed Addressing
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>

<offset> options

- An immediate constant
  - #10
- An index register
  - <Rm>
- A shifted index register
  - <Rm>, LSL #<shift>
- Lots of weird options…

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



Branch

Table A4-1 Branch instructions

Instruction	Usage	Range
B on page A6-40	Branch to target address	+/–1 MB
CBNZ, CBZ on page A6-52	Compare and Branch on Nonzero, Compare and Branch on Zero	0-126 B
BL on page A6-49	Call a subroutine	+/–16 MB
BLX (register) on page A6-50	Call a subroutine, optionally change instruction set	Any
BX on page A6-51	Branch to target address, change instruction set	Any
TBB, TBH on page A6-258	Table Branch (byte offsets)	0-510 B
	Table Branch (halfword offsets)	0-131070 B

Range □ offset range

BL □ Branch with link (copy the address of the next instruction into lr)

BLX □ Branch with link, and exchange instruction set (X for exchange to Thumb/ARM)

TBB [R0, R1] ; R1 is the index, R0 is the base address of the branch table □ branch to the $R1^{th}$ element of the table starting at R0 address

Branch examples

- `b target`
  - Branch without link (i.e. no possibility of return) to `target`
  - The PC is not saved!
- `bl func`
  - Branch with link (**call**) to function `func`
  - Store the return address in the link register (`lr`)
- `bx lr (Branch and exchange)`
  - Use to **return** from a function
  - Moves the `lr` value into the `pc`
  - Could be a different register than lr as well
- `blx reg (Branch with Link and exchange)`
  - Branch to address specified by `reg`
  - Save return address in `lr`
  - When using `blx`, makre sure lsb of `reg` is 1 (otherwise, the CPU will fault b/c it's an attempt to go into the ARM state)

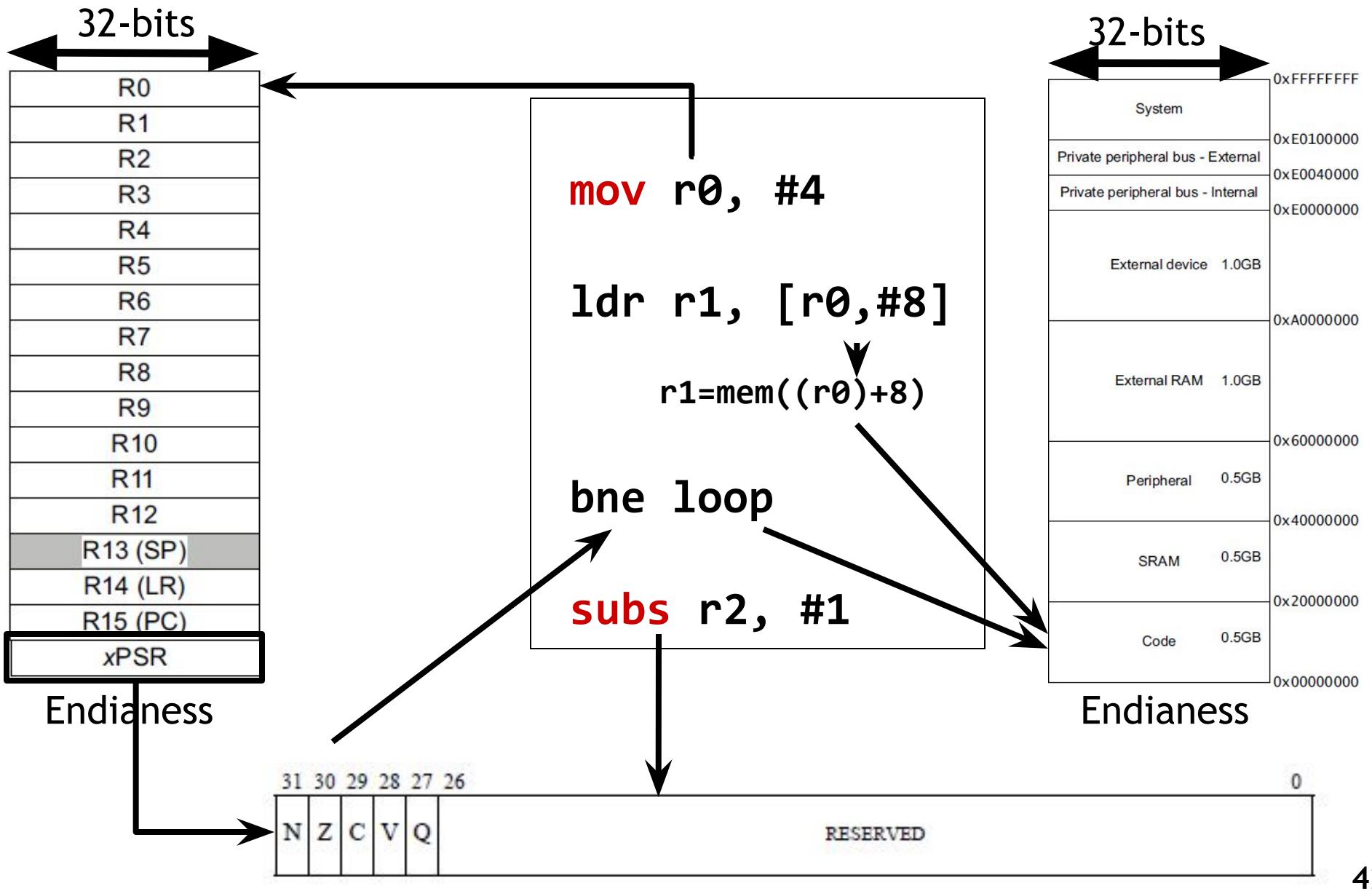
Branch examples (2)

- `blx label`
  - Branch with link and exchange state. With immediate data, `blx` changes to ARM state. But since CM-3 does not support ARM state, this instruction causes a fault!
- `mov r15, r0`
  - Branch to the address contained in r0
- `ldr r15, [r0]`
  - Branch to the to address in memory specified by r0

- Calling `bl` overwrites contents of `lr`!
  - So, save `lr` if your function needs to call a function!

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



Data processing instructions

Table A4-2 Standard data-processing instructions

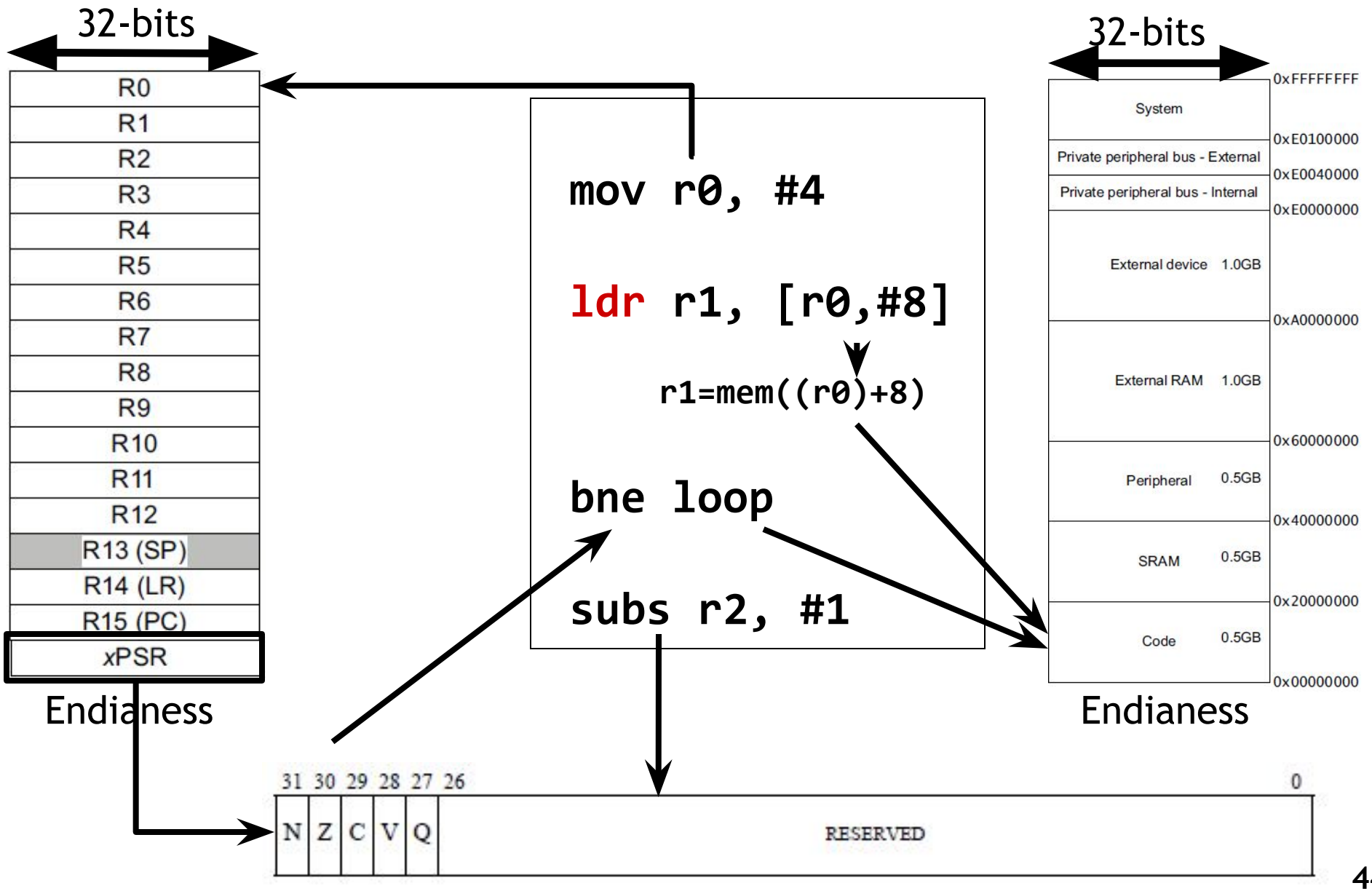
Mnemonic	Instruction	Notes
ADC	Add with Carry	-
ADD	Add	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
ADR	Form PC-relative Address	First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
AND	Bitwise AND	-
BIC	Bitwise Bit Clear	-
CMN	Compare Negative	Sets flags. Like ADD but with no destination register.
CMP	Compare	Sets flags. Like SUB but with no destination register.
EOR	Bitwise Exclusive OR	-
MOV	Copies operand to destination	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See *Shift instructions* on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.

- ADR PC, imm ▫ The assembler generates an instruction that adds or subtracts a value to the PC.
- CMP{cond} Rn, Operand2 (Rn-Operand2)
- CMN{cond} Rn, Operand2 (Rn+Operand2)
- The CMP instruction subtracts the value of Operand2 from the value in Rn. This is the same as a **SUBS** instruction, except that the result is discarded.
- The CMN instruction adds the value of Operand2 to the value in Rn. This is the same as an **ADDS** instruction, except that the result is discarded.

Many, Many More!

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



Load/Store instructions

Table A4-10 Load and store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load exclusive	Store exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
two 32-bit words	LDRD	STRD	-	-	-	-

Exclusive access is for when a memory is shared between some processors. When making access as exclusive, it means only letting 1 processor to access that.

An application running **unprivileged**:
• means only OS can allocate **system resources** to the application, as either private or shared resources
• provides a degree of **protection** from other processes and tasks, and so helps protect the operating system from malfunctioning applications.

Miscellaneous instructions

Table A4-12 Miscellaneous instructions

Instruction	See
Clear Exclusive	*CLREX* on page A6-56
Debug hint	*DBG* on page A6-67
Data Memory Barrier	*DMB* on page A6-68
Data Synchronization Barrier	*DSB* on page A6-70
Instruction Synchronization Barrier	*ISB* on page A6-76
If Then (makes following instructions conditional)	*IT* on page A6-78
No Operation	*NOP* on page A6-167
Preload Data	*PLD, PLDW (immediate)* on page A6-176
	PLD (register) on page A6-180

For example:

CLREX clear the **local record** of the executing processor that an address has had a request for an **exclusive access**.

DMB Data Memory Barrier acts as a memory barrier. It ensures that all **explicit memory accesses** that appear in program order **before** the DMB instruction are observed before any explicit memory accesses that appear in program order after the DMB instruction. It does not affect the ordering of any other instructions executing on the processor.

.

.

A5.3.2 Modified immediate constants in Thumb instructions

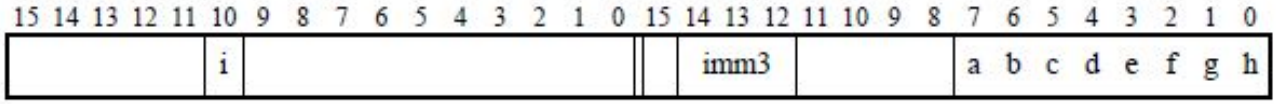


Table A5-11 shows the range of modified immediate constants available in Thumb data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

Table A5-11 Encoding of modified immediates in Thumb data-processing instructions

i:imm3:a	<const> [a]
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh [b]
0010x	abcdefgh 00000000 abcdefgh 00000000 [b]
0011x	abcdefgh abcdefgh abcdefgh abcdefgh [b]
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000
.	.
.	. 8-bit values shifted to other positions
.	.
11101	00000000 00000000 000001bc defgh000
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0

a. In this table, the immediate constant value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

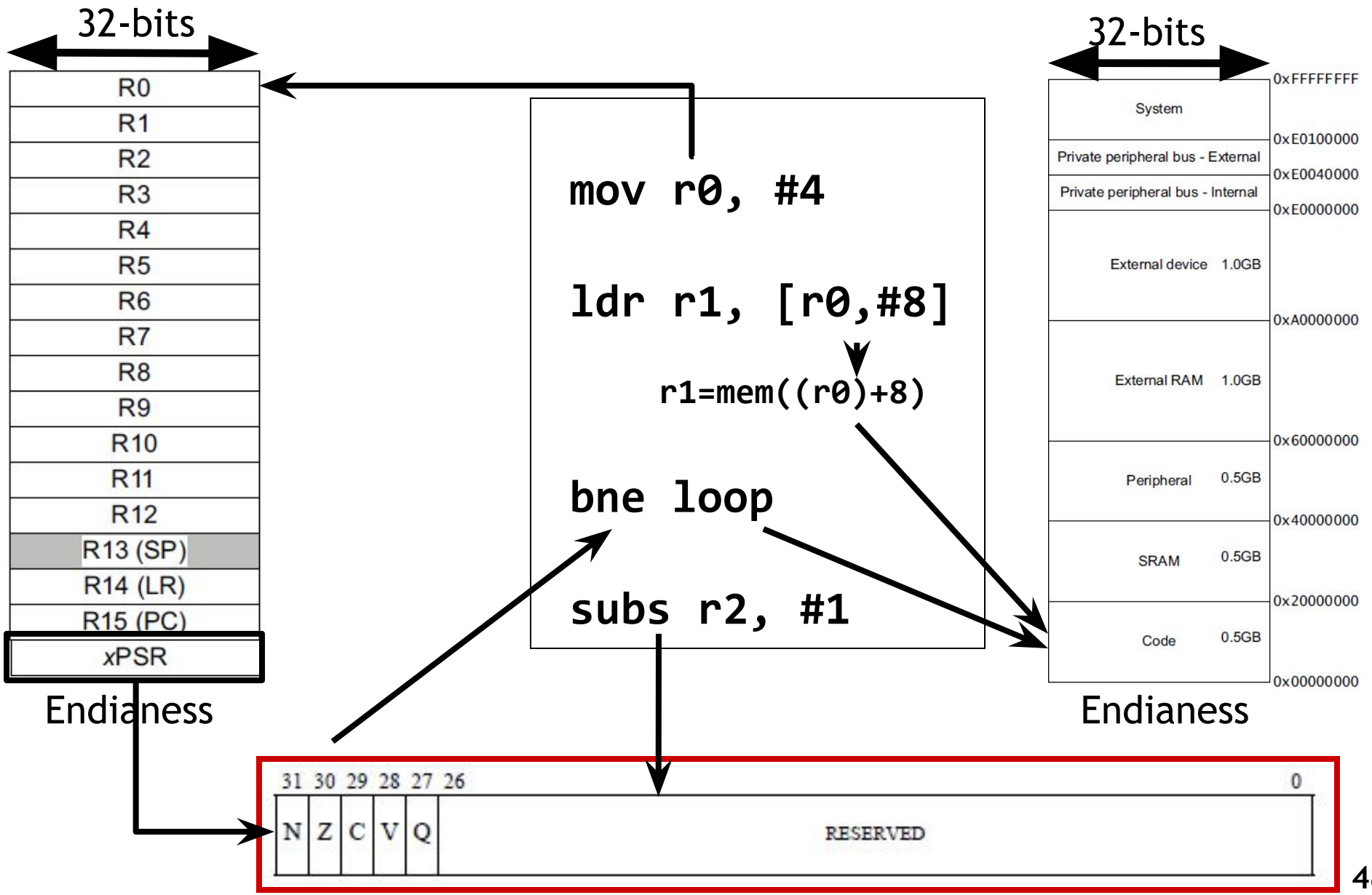
b. UNPREDICTABLE if abcdefgh == 00000000.

ARMv7-M Architecture Reference Manual

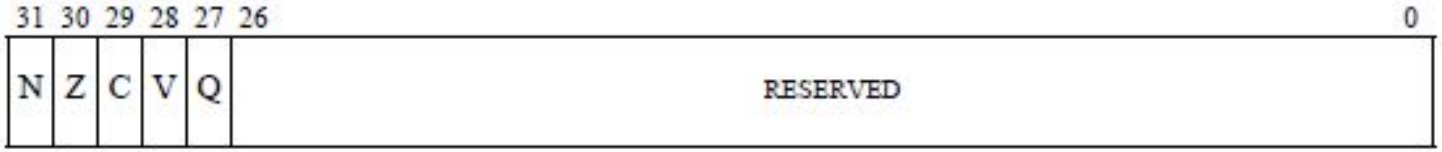
ARMv7-M_ARM.pdf

Major elements of an Instruction Set Architecture

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



Application Program Status Register (APSR)



APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.

- Flags that can be set by many instructions:

  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

  Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  Q, bit [27] Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

Updating the APSR

- SUB Rx, Ry
  - Rx = Rx - Ry
  - APSR unchanged
- SUBS
  - Rx = Rx - Ry
  - APSR N, Z, C, V updated
- ADD Rx, Ry
  - Rx = Rx + Ry
  - APSR unchanged
- ADDS
  - Rx = Rx + Ry
  - APSR N, Z, C, V updated

Overflow and carry in APSR

unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);

signed_sum = **S**Int(x) + **S**Int(y) + **U**Int(carry_in);

result = unsigned_sum<N-1:0>; // == signed_sum<N-1:0>

carry_out = if UInt(result) == unsigned_sum then ’0’ else ’1’;

overflow = if SInt(result) == signed_sum then ’0’ else ’1’;

Conditional execution:

Table A6-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) [ab]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS [c]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC [d]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) [e]	Always (unconditional)	Always (unconditional)	Any

- EQ, NE, … are **suffixes** that add to other instructions (B+NE=BNE, ADDS+CS=ADDCS, …) and check their corresponding condition flags which make the original instruction conditional.
- Most ARM /Thumb instructions can be executed conditionally, based on the values of the APSR condition flags.
- The type of instruction that **last** updated the flags in the APSR determines the meaning of condition codes.

IT blocks

- Conditional execution in C-M3 done in “IT” block
- IT [T|E]*3
- More on this later…

Conditional Execution on the ARM

- ARM instruction can include conditional suffixes, e.g.
  - EQ, NE, GE, LT, GT, LE, …
- Normally, such suffixes are used for branching (BNE)
- However, other instructions can be conditionally executed
  - They **must** be inside of an **IF-THEN** block
  - When placed inside an IF-THEN block
    - Conditional execution (EQ, NE, GE,… suffix) and
    - Status register update (S suffix) can be used together
  - Conditional instructions use a special “IF-THEN” or “IT” block
- IT (IF-THEN) blocks
  - Support conditional execution (e.g. ADDNE)
  - Without a branch penalty (e.g. BNE)
  - For no more than a few instructions (i.e. 1-4)

Conditional Execution using IT Instructions

- In an IT block
  - Typically an instruction that updates the status register is exec'ed
  - Then, the first line of an IT instruction follows (of the form ITxyz)
    - Where each x, y, and z is replaced with "T", "E", or nothing ("")
    - T's must be first, then E's (between 1 and 4 T's and E's in total)
    - Ex: IT, ITT, ITE, ITTT, ITTE, ITEE, ITTTT, ITTTE, ITTEE, ITEEE
  - Followed by 1-4 conditional instructions
    - where # of conditional instruction equals total # of T's & E's

```
IT<x><y><z> <cond>           ; IT instruction (<x>, <y>,
                        ; <z> can be "T", "E" or "")
instr1<cond> <operands>          ; 1st instruction (<cond>
                        ; must be same as IT)
instr2<cond or not cond> <operands>  ; 2nd instruction (can be
                        ; <cond> or <!cond>
instr3<cond or not cond> <operands>  ; 3rd instruction (can be
                        ; <cond> or <!cond>
instr4<cond or not cond> <operands>  ; 4th instruction (can be
                        ; <cond> or <!cond>
```

Source: Joseph Yiu, "The Definitive Guide to the ARM Cortex-M3", 2nd Ed., Newnes/Elsevier, © 2010.

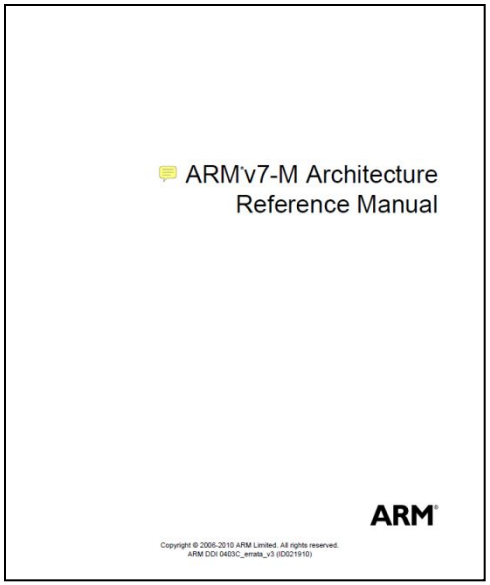
Example of Conditional Execution using IT Instructions

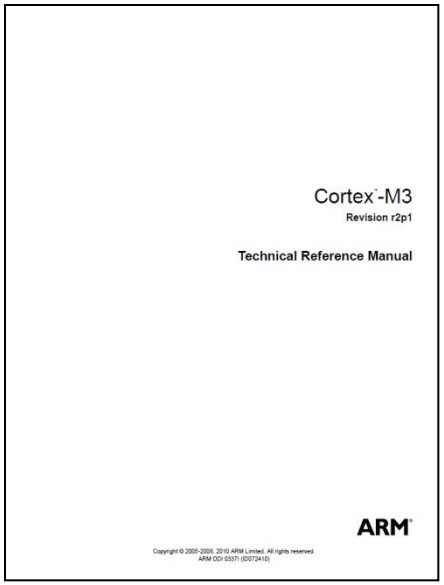
```
    CMP r1, r2        ; if r1 < r2 (less than, or LT)
    ITTEE   LT        ; then execute 1st & 2nd instruction
                  ; (indicated by 2 T's)
                  ; else execute 3rd and 4th instruction
                  ; (indicated by 2 E's)
    SUBLT   r2, r1        ; 1st instruction
    LSRLT   r2, #1        ; 2nd instruction
    SUBGE   r1, r2        ; 3rd instruction (GE opposite of LT)
    LSRGE   r1, #1        ; 4th instruction (GE opposite of LT)
```

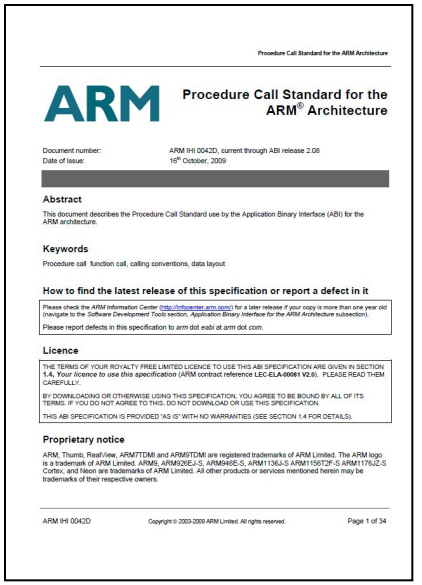
```
IT<x><y><z> <cond>              ; IT instruction (<x>, <y>,
                        ; <z> can be "T", "E" or "")
instr1<cond> <operands>             ; 1st instruction (<cond>
                        ; must be same as IT)
instr2<cond or not cond> <operands>  ; 2nd instruction (can be
                        ; <cond> or <!cond>
instr3<cond or not cond> <operands>  ; 3rd instruction (can be
                        ; <cond> or <!cond>
instr4<cond or not cond> <operands>  ; 4th instruction (can be
                        ; <cond> or <!cond>
```

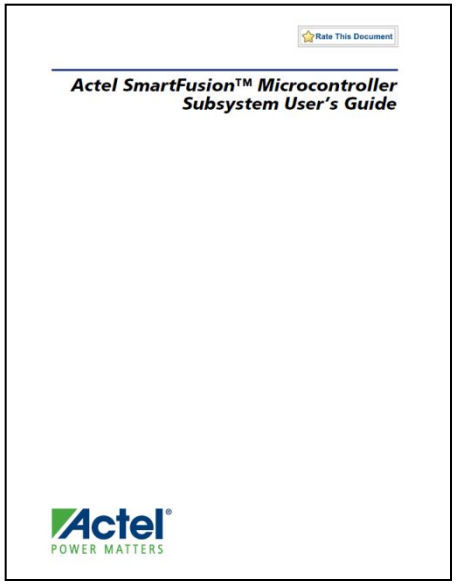
Source: Joseph Yiu, "The Definitive Guide to the ARM Cortex-M3", 2nd Ed., Newnes/Elsevier, © 2010.

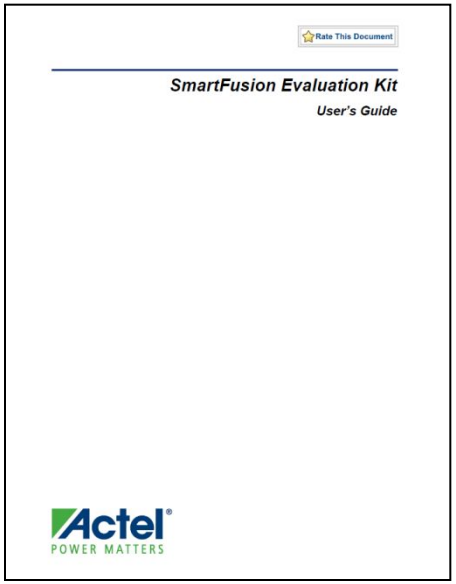
The ARM architecture “books” for this class

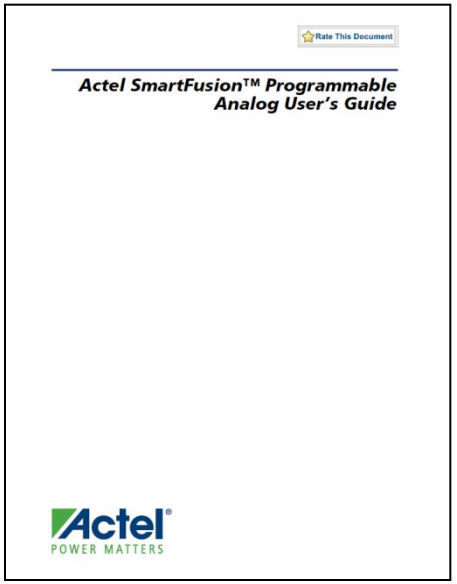












Exercise:
What is the value of r2 at done?

```
...
start:
   movs r0, #1
   movs r1, #1
   movs r2, #1
   sub  r0, r1
   bne  done
   movs r2, #2
done:
   b    done
...
```

Solution:
What is the value of r2 at done?

```
...
start:
   movs r0, #1   // r0 ← 1, Z=0
   movs r1, #1   // r1 ← 1, Z=0
   movs r2, #1   // r2 ← 1, Z=0
   sub  r0, r1   // r0 ← r0-r1
            // but Z flag untouched
            // since sub vs subs
   bne  done     // NE true when Z==0
            // So, take the branch
   movs r2, #2   // not executed
done:
   b    done     // r2 is still 1
...
```

Today…

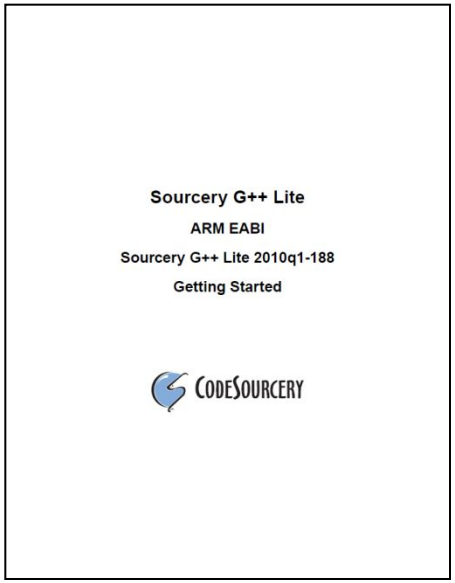
Finish ARM assembly example from last time

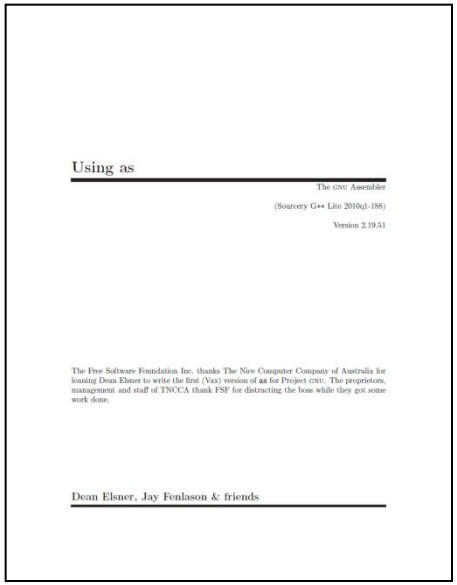
Walk though of the ARM ISA

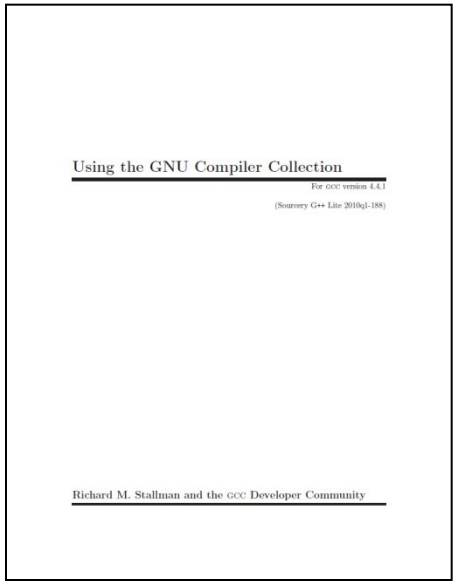
Software Development Tool Flow

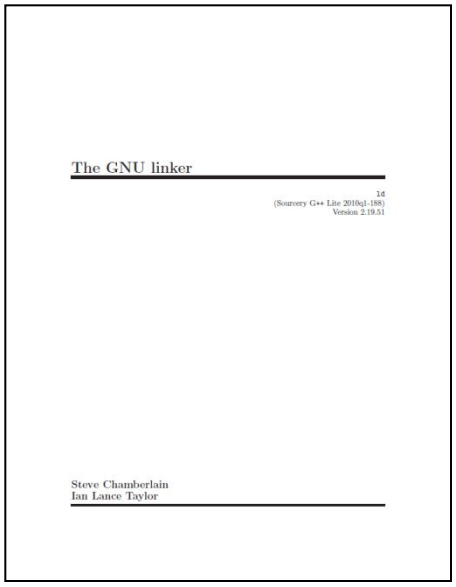
Application Binary Interface (ABI)

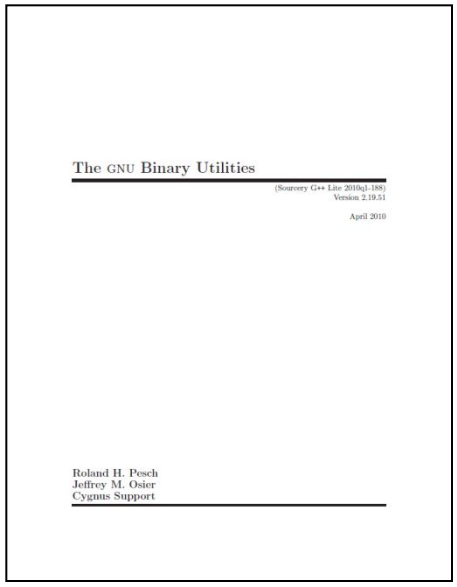
The ARM software tools "books" for this class

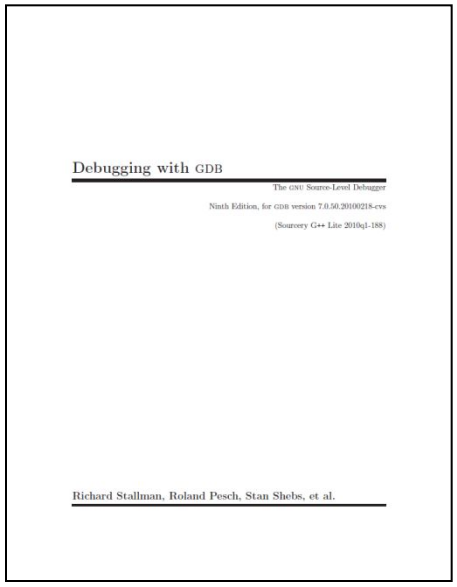




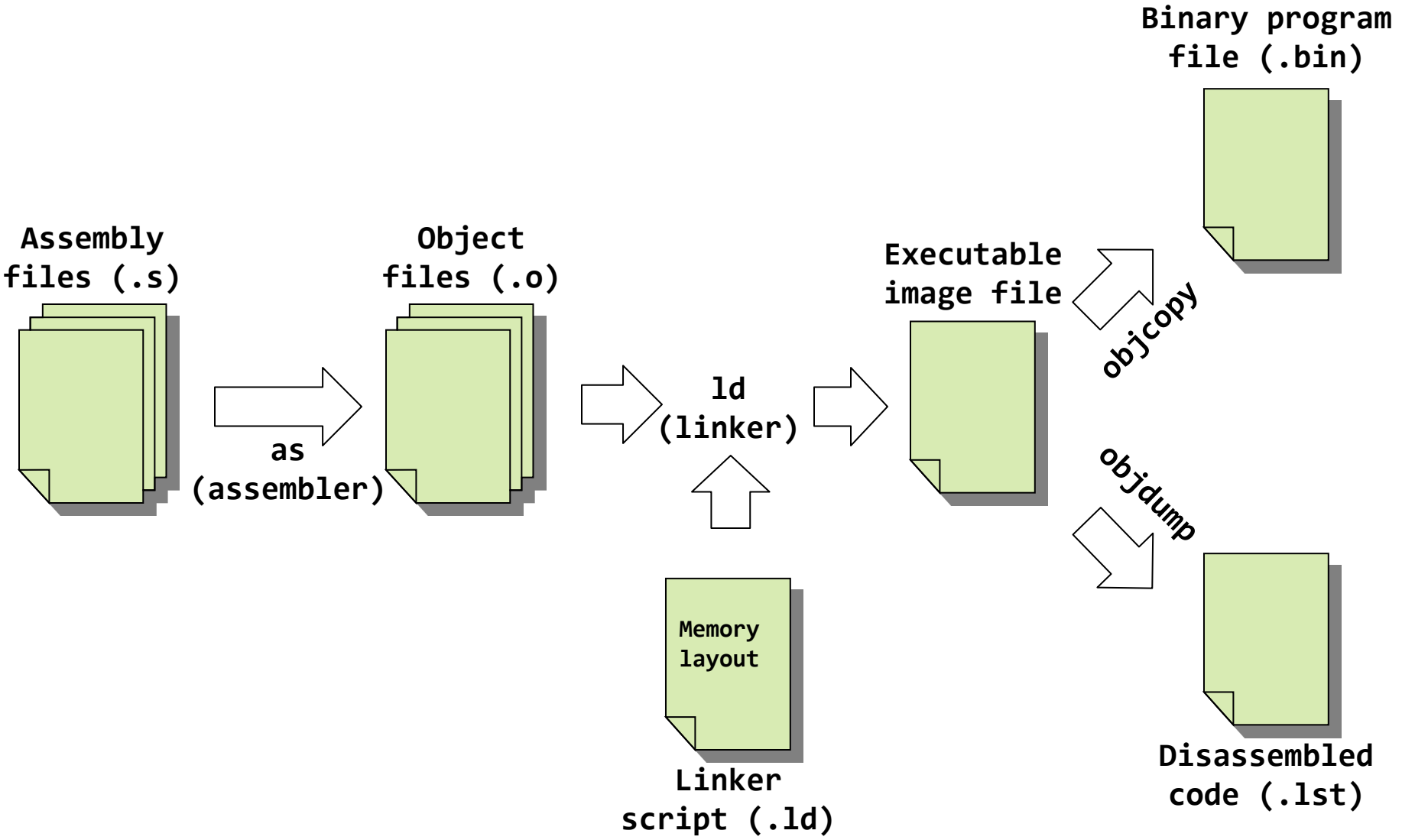






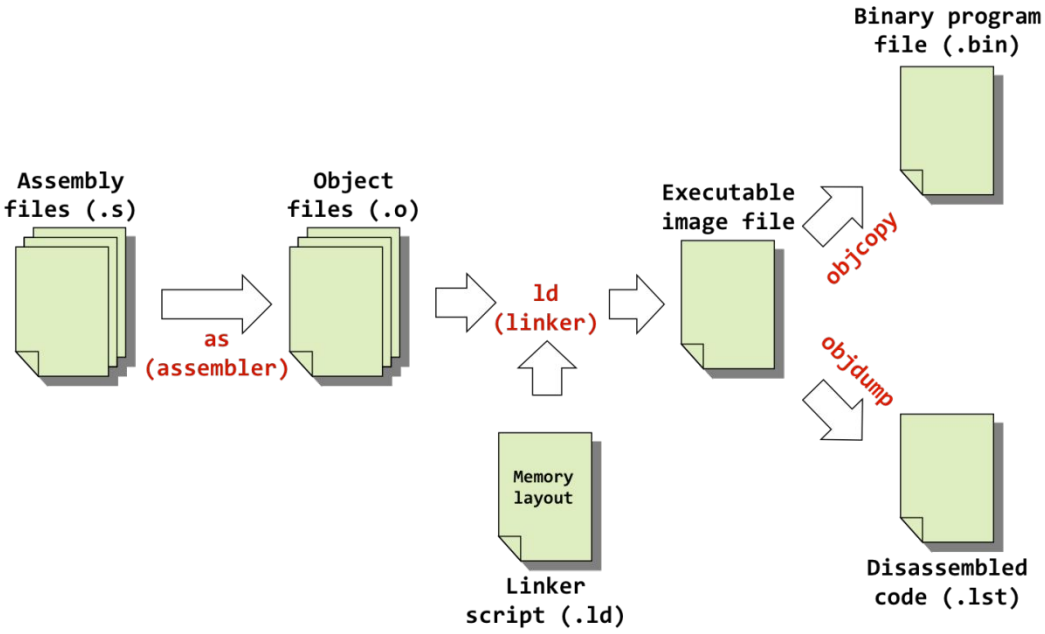


How does an assembly language program get turned into an executable program image?



What are the real GNU executable names for the ARM?

- Just add the prefix “arm-none-eabi-” prefix
- Assembler (as)
  - arm-none-eabi-as
- Linker (ld)
  - arm-none-eabi-ld
- Object copy (objcopy)
  - arm-none-eabi-objcopy
- Object dump (objdump)
  - arm-none-eabi-objdump
- C Compiler (gcc)
  - arm-none-eabi-gcc
- C++ Compiler (g++)
  - arm-none-eabi-g++



Tools Tutorial: The GNU Linker
http://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/Linker.pdf

Real-world example

- To the terminal! Example code online:

https://github.com/brghena/eecs373_toolchain_examples)

- First, get the code. Open a shell and type:

```
$ git clone https://github.com/brghena/eecs373_toolchain_examples
```

- Next, find the example and look at the Makefile and .s

```
$ cd eecs373_toolchain_examples/example
$ cat Makefile
$ cat example.s
```

- Assemble the code* and look at the .lst, .o, .out files

```
$ make
$ cat example.lst
$ hexdump example.o
$ hexdump example.out
```

* You'll need to have the tools installed. Two useful links:

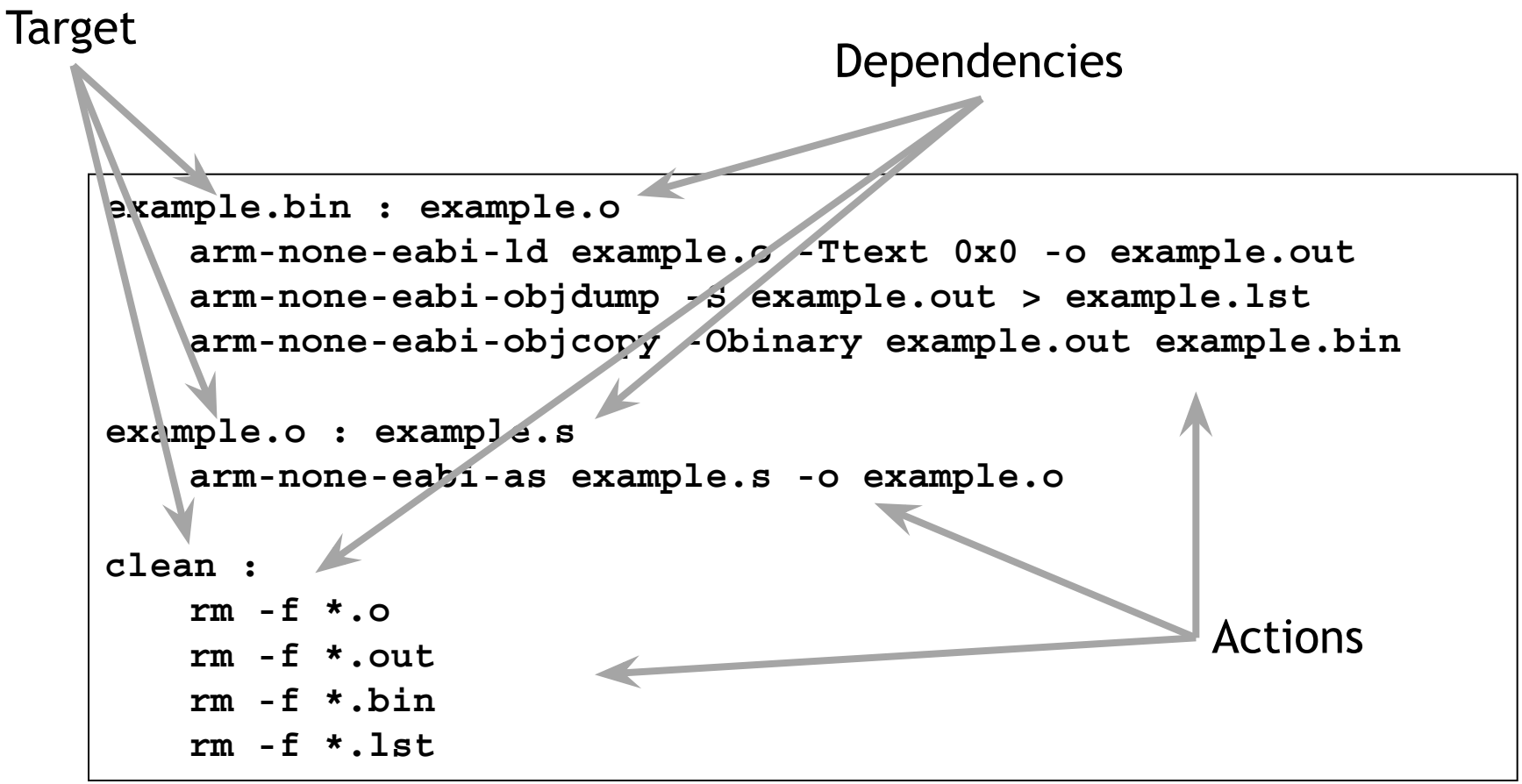
https://launchpad.net/gcc-arm-embedded/+download
http://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/Linker.pdf

How are assembly files assembled?

- $ arm-none-eabi-as
  - Useful options
    - -mcpu
    - -mthumb
    - -o

```
$ arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
```

A simple Makefile example



A “real” ARM assembly language program for GNU

```
        .equSTACK_TOP, 0x20000800
        .text
        .syntax unified
        .thumb
        .global _start
        .type   start, %function

_start:
        .word   STACK_TOP, start
start:
        movs r0, #10
        movs r1, #0
loop:
        adds r1, r0
        subs r0, #1
        bne  loop
deadloop:
        b    deadloop
        .end
```

What's it all mean?

```
    .equSTACK_TOP, 0x20000800    /* Equates symbol to value */
    .text                /* Tells AS to assemble region */
    .syntax unified          /* Means language is ARM UAL */
    .thumb               /* Means ARM ISA is Thumb */
    .global _start           /* .global exposes symbol */
                    /* _start label is the beginning
                       ...of the program region */
    .type   start, %function     /* Specifies start is a function */
                    /* start label is reset handler */
_start:
    .word   STACK_TOP, start     /* Inserts word 0x20000800 */
                    /* Inserts word (start) */
start:
    movs r0, #10         /* We've seen the rest ... */
    movs r1, #0
loop:
    adds r1, r0
    subs r0, #1
    bne  loop
deadloop:
    b    deadloop
    .end
```

What information does the disassembled file provide?

```
all:
	arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
	arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
	arm-none-eabi-objcopy -Obinary example1.out example1.bin
	arm-none-eabi-objdump -S example1.out > example1.lst
```

```
	.equ  STACK_TOP, 0x20000800
	.text
	.syntax    unified
	.thumb
	.global    _start
	.type start, %function

_start:
	.word STACK_TOP, start
start:
	movs r0, #10
	movs r1, #0
loop:
	adds r1, r0
	subs r0, #1
	bne  loop
deadloop:
	b    deadloop
	.end
```

```
example1.out:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0: 20000800    .word 0x20000800
   4: 00000009    .word 0x00000009

00000008 <start>:
   8: 200a        movs  r0, #10
   a: 2100        movs  r1, #0

0000000c <loop>:
   c: 1809        adds  r1, r1, r0
   e: 3801        subs  r0, #1
  10: d1fc        bne.n c <loop>

00000012 <deadloop>:
  12: e7fe        b.n   12 <deadloop>
```

Linker script that puts all the code in the right places

```
OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(main)

MEMORY
{
 /* SmartFusion internal eSRAM */
 ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

SECTIONS
{
 .text :
 {
    . = ALIGN(4);
     *(.text*)
    . = ALIGN(4);
     _etext = .;
 } >ram
}
end = .;
```

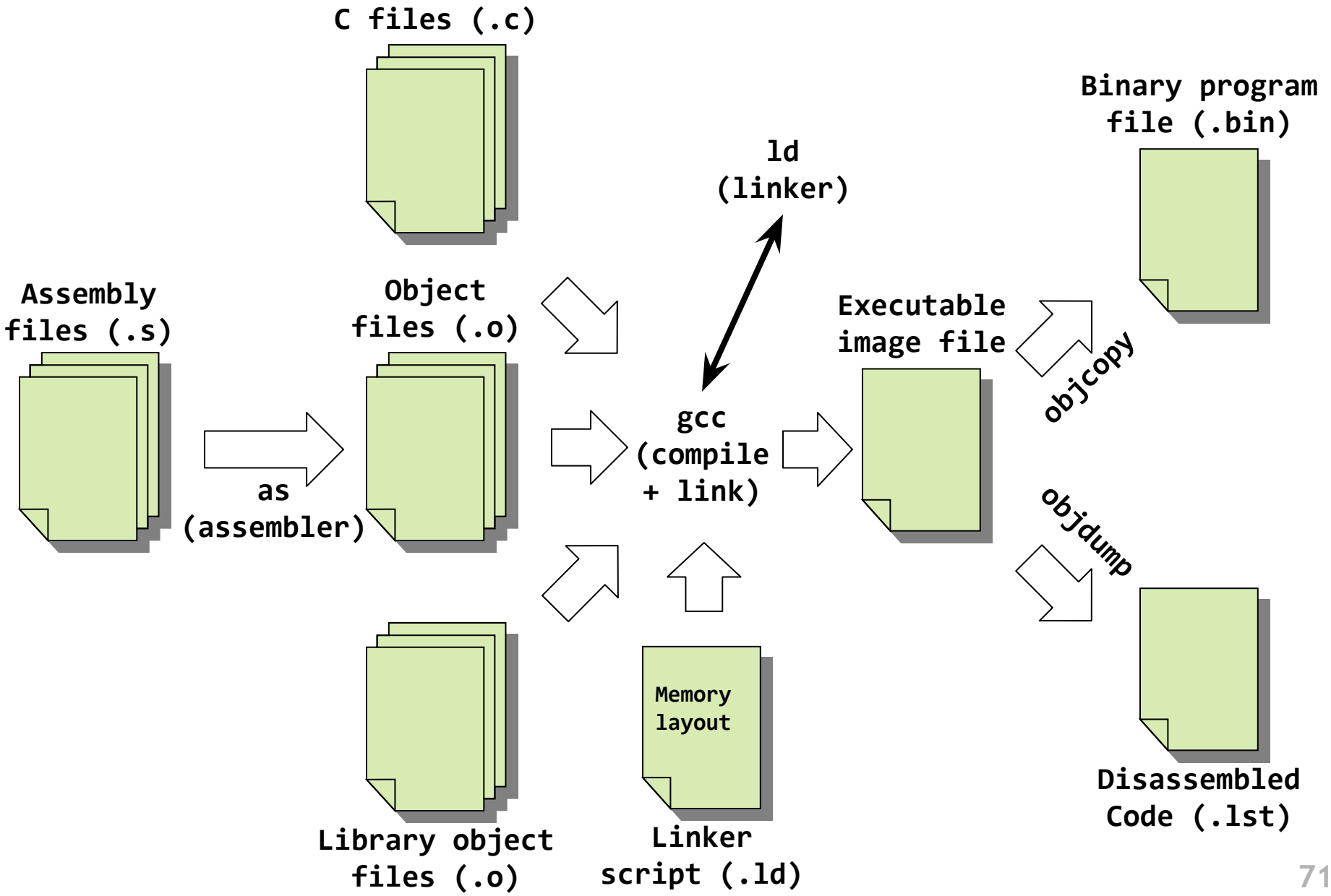
- Specifies little-endian arm in ELF format
- Specifies ARM CPU
- Start executing at label named “main”

- We have 64k of memory starting at 0x20000000. You can read, write and execute out of it. We named it “ram”

- “.” is a reference to the current memory location
- First align to a word (4 byte) boundary
- Place all sections that include .text at the start (* here is a wildcard)
- Define a label named _etext to be the current address.
- Put it all in the memory location defined by the ram symbol’s location.

More info: The GNU Linker
http://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/Linker.pdf

How does a mixed C/Assembly program get turned into a executable program image?



Real-world example: Mixing C and assembly code

- To the terminal again! Example code online:

https://github.com/brghena/eecs373_toolchain_examples

$ git clone https://github.com/brghena/eecs373_toolchain_examples

- Inline assembly

$ cd eecs373_toolchain_examples/inline_asm

$ cat cfile.c

- Separate C and assembly

$ cd eecs373_toolchain_examples/inline_asm

$ cat asmfile.s

$ cat cfile.c

* You'll need to have the tools installed. Two useful links:

https://launchpad.net/gcc-arm-embedded/+download

http://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/Linker.pdf

Today…

Finish ARM assembly example from last time

Walk though of the ARM ISA

Software Development Tool Flow

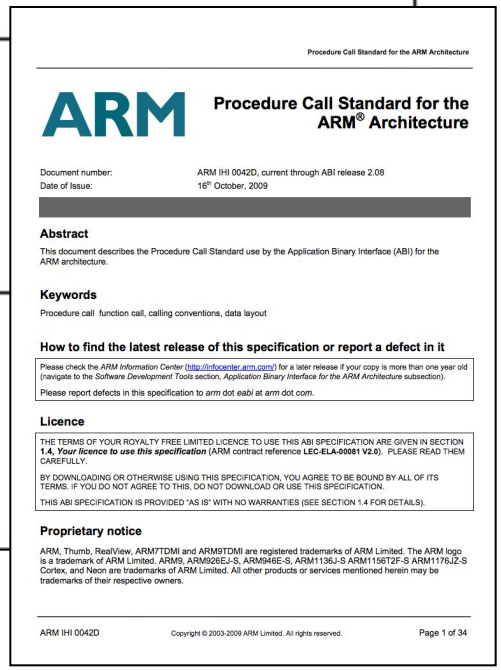
Application Binary Interface (ABI)

When is this relevant?

- The ABI establishes caller/callee responsibilities
  - Who saves which registers
  - How function parameters are passed
  - How return values are passed back
- The ABI is a contract with the compiler
  - All assembled C code will follow this standard
- You need to follow it if you want C and Assembly to work together correctly

From the Procedure Call Standard

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.



Procedure Call Standard for the ARM Architecture

ARM

Procedure Call Standard for the ARM® Architecture

Document number: ARM IHI 0042D, current through ABI release 2.08
Date of Issue: 16th October, 2009

Abstract

This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the ARM architecture.

Keywords

Procedure call function call, calling conventions, data layout

How to find the latest release of this specification or report a defect in it

Please check the *ARM Information Center* (http://infocenter.arm.com/) for a later release if your copy is more than one year old (navigate to the *Software Development Tools* section, *Application Binary Interface for the ARM Architecture* subsection).

Please report defects in this specification to *arm* dot *eabi* at *arm* dot *com*.

Licence

THE TERMS OF YOUR ROYALTY FREE LIMITED LICENCE TO USE THIS ABI SPECIFICATION ARE GIVEN IN SECTION **1.4, *Your licence to use this specification*** (ARM contract reference **LEC-ELA-00081 V2.0**). PLEASE READ THEM CAREFULLY.

BY DOWNLOADING OR OTHERWISE USING THIS SPECIFICATION, YOU AGREE TO BE BOUND BY ALL OF ITS TERMS. IF YOU DO NOT AGREE TO THIS, DO NOT DOWNLOAD OR USE THIS SPECIFICATION.

THIS ABI SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES (SEE SECTION 1.4 FOR DETAILS).

Proprietary notice

ARM, Thumb, RealView, ARM7TDMI and ARM9TDMI are registered trademarks of ARM Limited. The ARM logo is a trademark of ARM Limited. ARM9, ARM926EJ-S, ARM946E-S, ARM1136J-S ARM1156T2F-S ARM1176JZ-S Cortex, and Neon are trademarks of ARM Limited. All other products or services mentioned herein may be trademarks of their respective owners.

ARM IHI 0042D Copyright © 2003-2009 ARM Limited. All rights reserved. Page 1 of 34

Source: Procedure Call Standard for the ARM Architecture
http://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/ARM-AAPCS-EABI-v2.08.pdf

ABI Basic Rules

1. A subroutine must preserve the contents of the registers r4-11 and SP
   - These are the callee save registers
   - Let's be careful with r9 though
2. Arguments are passed though r0 to r3
   - These are the caller save registers
   - If we need more arguments, we put a pointer into memory in one of the registers
     - We'll worry about that later
3. Return value is placed in r0
   - r0 and r1 if 64-bits
4. Allocate space on stack as needed. Use it as needed. Put it back when done…
   - Keep things word aligned*

Let's write a simple ABI routine

- int bob(int a, int b)
  - returns $a^2 + b^2$
- Instructions you might need
  - add adds two values
  - mul multiplies two values
  - bx branch to register

Other useful facts

- Stack grows down.
  - And pointed to by "sp"
- Return address is held in "lr"

Register	Synonym
r15	
r14	
r13	
r12	
r11	v8
r10	v7
r9	
r8	v5
r7	v4
r6	v3
r5	v2
r4	v1
r3	a4
r2	a3
r1	a2
r0	a1

Questions?

Comments?

Discussion?