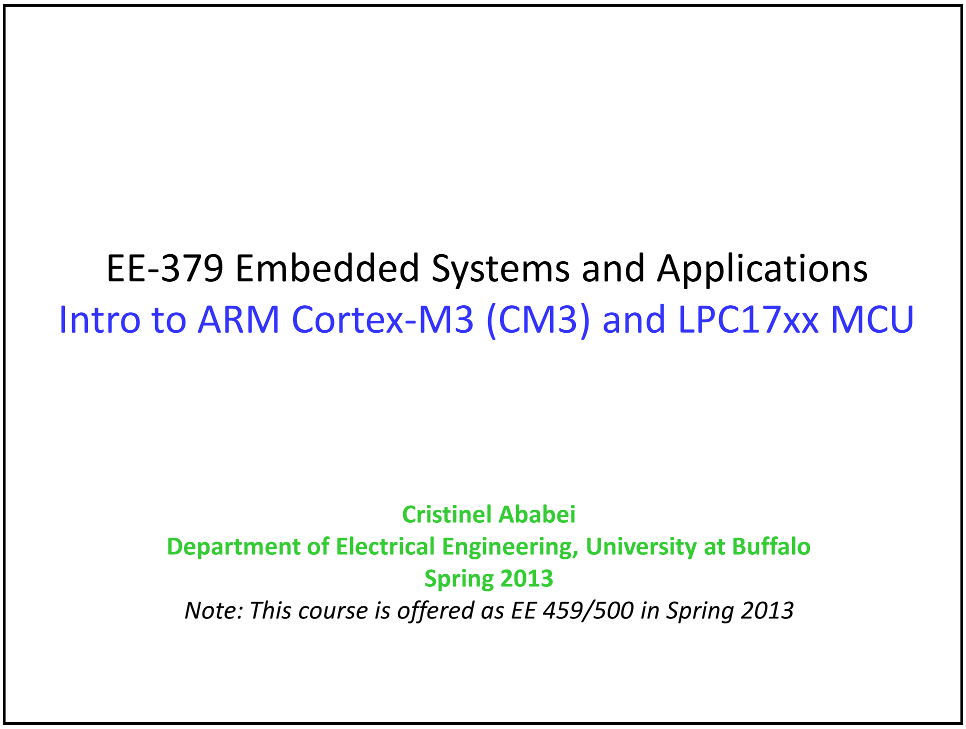
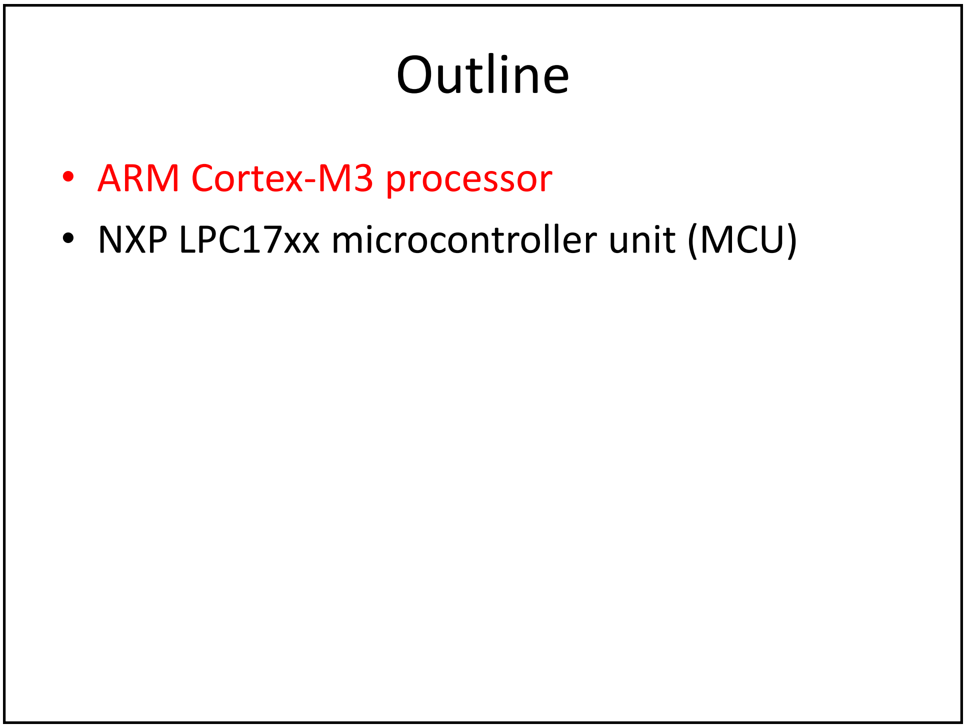


EE-379 Embedded Systems and Applications

Intro to ARM Cortex-M3 (CM3) and LPC17xx MCU

Cristinel Ababei
Department of Electrical Engineering, University at Buffalo
Spring 2013

Note: This course is offered as EE 459/500 in Spring 2013



Outline

- ARM Cortex-M3 processor
- NXP LPC17xx microcontroller unit (MCU)

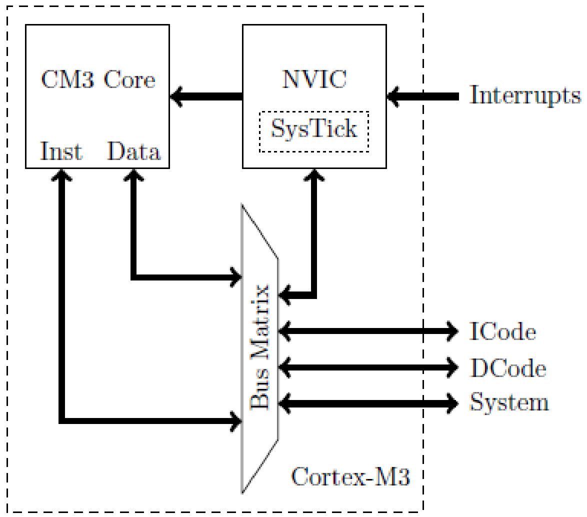
Cortex-M3 Processor

- RISC general purpose 32-bit microprocessor, released 2006
- Cortex-M3 differs from previous generations of ARM processors by defining a number of key peripherals as part of the core:
  - interrupt controller
  - system timer
  - debug and trace hardware (including external interfaces)
- This enables for real-time operating systems and hardware development tools such as debugger interfaces be common across the family of processors
- Various Cortex-M3 based microcontroller families differ significantly in terms of hardware **peripherals and memory**

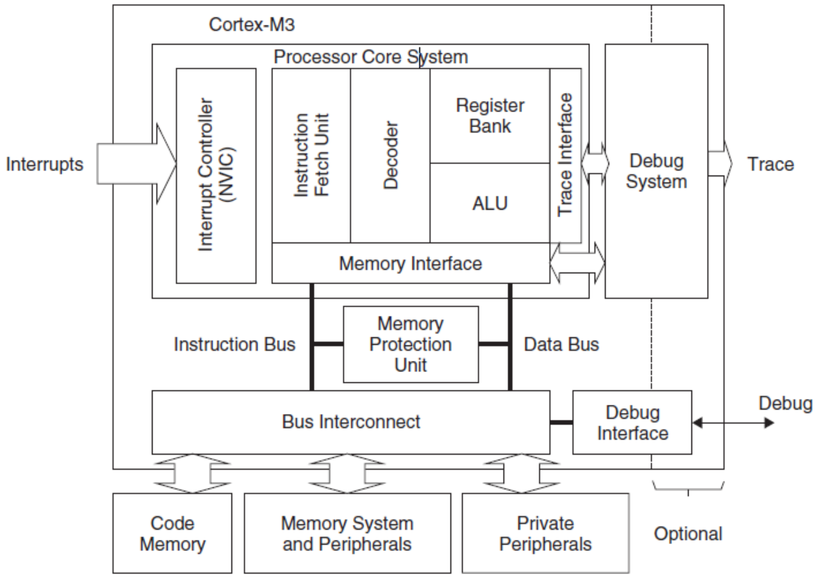
Cortex-M3 Processor

- **Greater performance efficiency**: more work to be done without increasing the frequency or power requirements
  - Implements the new Thumb-2 instruction set architecture
    - 70% more efficient per MHz than an ARM7TDMI-S processor executing Thumb instructions
    - 35% more efficient than the ARM7TDMI-S processor executing ARM instructions for Dhrystone benchmark
- **Low power consumption**: longer battery life, especially critical in portable products including wireless networking applications
- **Improved code density**: code fits in even the smallest memory footprints
- Core pipeline has 3 stages
  - Instruction Fetch
  - Instruction Decode
  - Instruction Execute

Simplified Cortex-M3 Architecture



Simplified Cortex-M3 Architecture



Cortex-M3 Processor Architecture

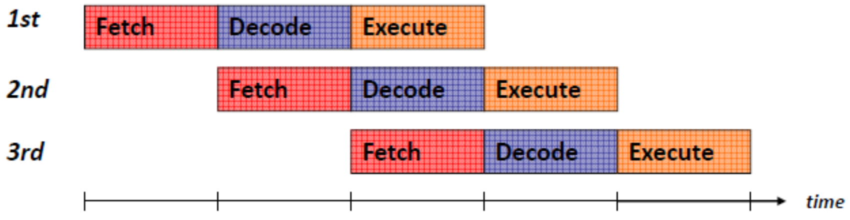
- Harvard architecture: it uses separate interfaces to fetch instructions (Inst) and (Data)
- Processor is not memory starved: it permits accessing data and instruction memories simultaneously
- **From CM3 perspective, everything looks like memory**
  - Only differentiates between instruction fetches and data accesses
- Interface between CM3 and manufacturer specific hardware is through three memory buses:
  - ICode, DCode, and System (for peripherals), which are defined to access different regions of memory

Cortex-M3 Processor

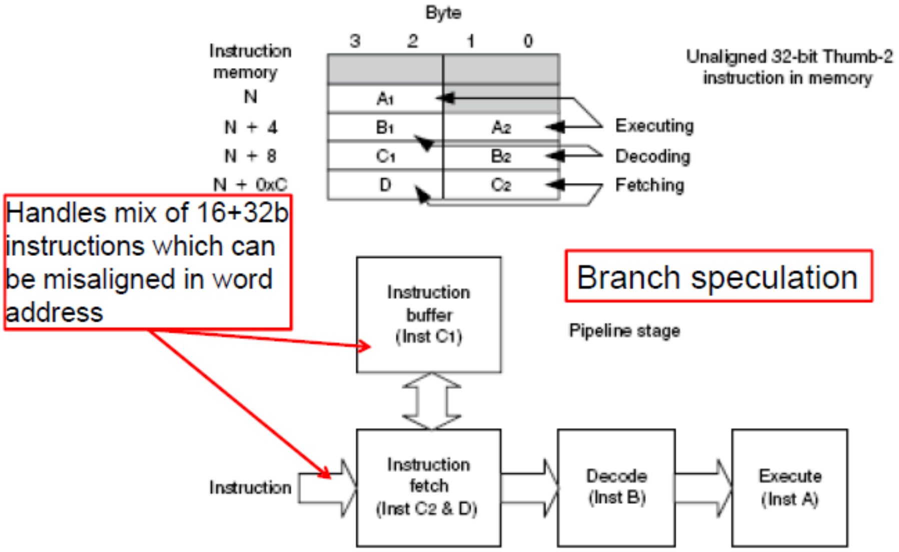
- Cortex-M3 is a load/store architecture with three basic types of instructions
  - **register-to-register** operations for processing data
  - **memory operations** which move data between memory and registers
  - **control flow** operations enabling programming language control flow such as if and while statements and procedure calls

Cortex-M3 Pipeline

- The Cortex-M3 Uses the 3-stage pipeline for instruction executions
  - Fetch ⇒ Decode ⇒ Execute
  - Pipeline design allows effective throughput to increase to one instruction per clock cycle
  - Allows the next instruction to be fetched while still decoding or executing the previous instructions



Instruction Prefetch & Execution



Handles mix of 16+32b instructions which can be misaligned in word address

Branch speculation

Processor Modes

- The ARM has seven basic operating modes:
  - Each mode has access to:
    - Its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

	Mode	Description	
Exception modes	**Supervisor (SVC)**	Entered on reset and when a Software Interrupt instruction (SWI) is executed	**Privileged modes**
	FIQ	Entered when a high priority (fast) interrupt is raised	
	IRQ	Entered when a low priority (normal) interrupt is raised	
	Abort	Used to handle memory access violations	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode	
	User	Mode under which most Applications / OS tasks run	**Unprivileged mode**

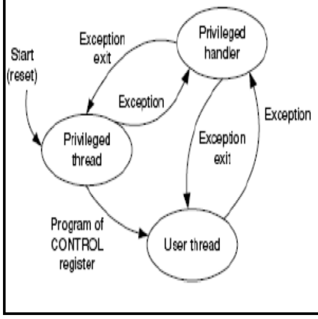
Operating Modes

User mode:

- Normal program execution mode
- System resources unavailable
- Mode changed by exception only

Exception modes:

- Entered upon exception
- Full access to system resources
- Mode changed freely



		Operations (privilege out of reset)	**Stacks** (Main out of reset)
Modes (Thread out of reset)	**Handler** - An exception is being processed	Privileged execution Full control	Main Stack Used by OS and Exceptions
	Thread - No exception is being processed - Normal code is executing	Privileged/Unprivileged	Main/Process

Exceptions

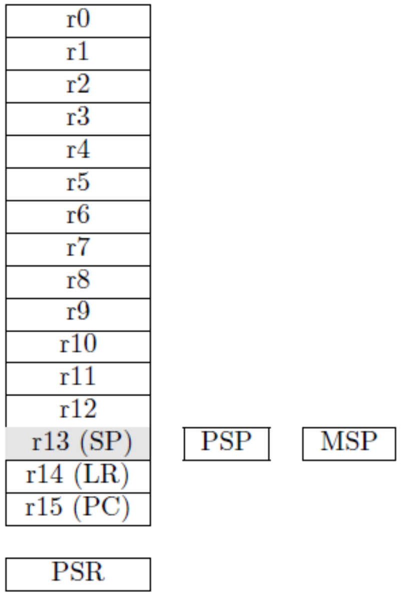
Exception	Mode	Priority	IV Address
Reset	Supervisor	1	0x00000000
Undefined instruction	Undefined	6	0x00000004
Software interrupt	Supervisor	6	0x00000008
Prefetch Abort	Abort	5	0x0000000C
Data Abort	Abort	2	0x00000010
Interrupt	IRQ	4	0x00000018
Fast interrupt	FIQ	3	0x0000001C

Table 1 - Exception types, sorted by Interrupt Vector addresses

Processor Register Set

- Cortex-M3 core has **16 user-visible registers**
  - All processing takes place in these registers
- Three of these registers have dedicated functions
  - **program counter (PC)** - holds the address of the next instruction to execute
  - **link register (LR)** - holds the address from which the current procedure was called
  - **“the” stack pointer (SP)** - holds the address of the current stack top (CM3 supports multiple execution modes, each with their own private stack pointer).
- **Processor status register (PSR)** which is implicitly accessed by many instructions

Processor Register Set



r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (SP)
r14 (LR)
r15 (PC)

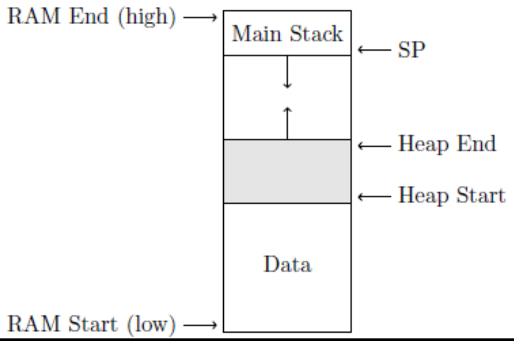
PSP

MSP

PSR

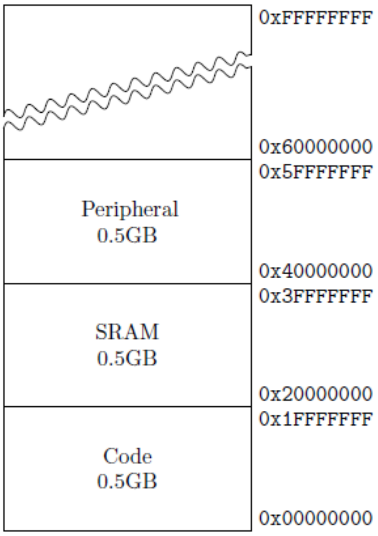
Program Memory Model

- RAM for an executing program is divided into three regions
  - Data in RAM are allocated during the link process and initialized by startup code at reset
  - The (optional) heap is managed at runtime by library code implementing functions such as the malloc and free which are part of the standard C library
  - The stack is managed at runtime by compiler generated code which generates per-procedure-call stack frames containing local variables and saved registers

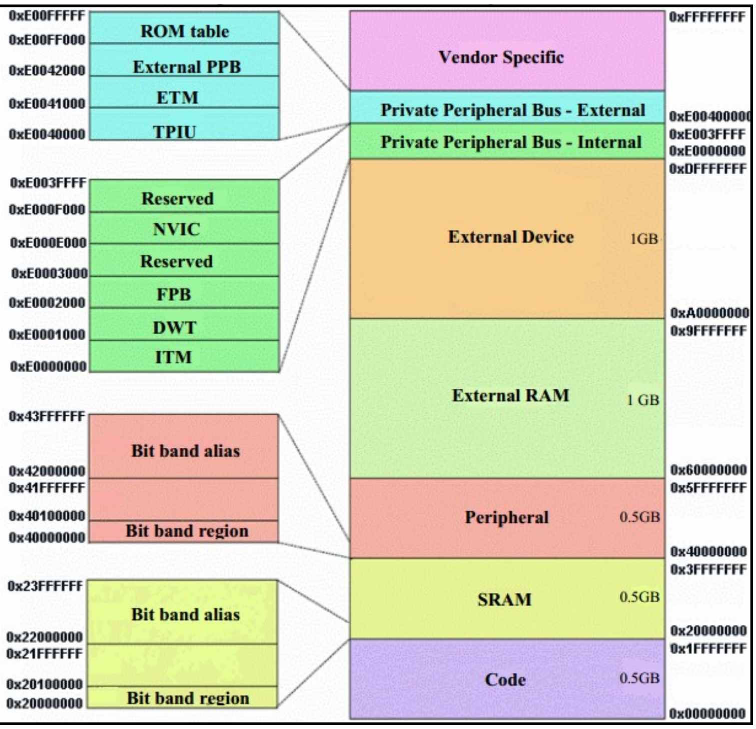


Cortex-M3 Memory Address Space

- ARM Cortex-M3 processor has a single 4 GB address space
- The **SRAM and Peripheral** areas are accessed through the System bus
- The **“Code” region** is accessed through the ICode (instructions) and DCode (constant data) buses



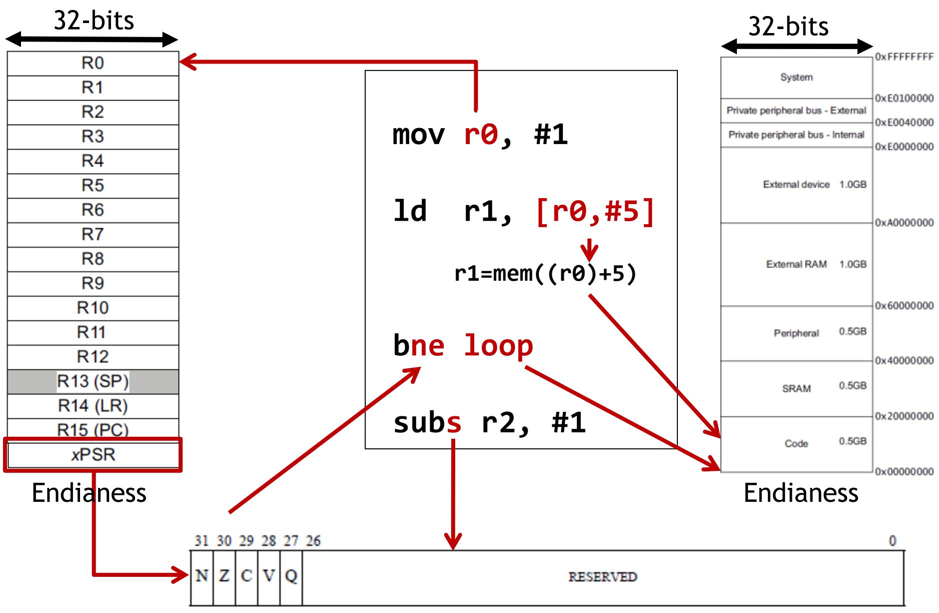
Memory Map



Instruction Set Architecture (ISA)

- Instruction set
  - Addressing modes
  - Word size
  - Data formats
  - Operating modes
  - Condition codes

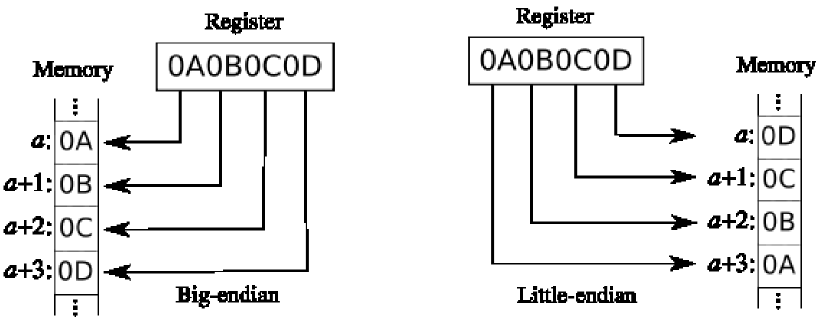
Major Elements of ISA



```
mov r0, #1

ld  r1, [r0,#5]
        r1=mem((r0)+5)

bne loop

subs r2, #1
```

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



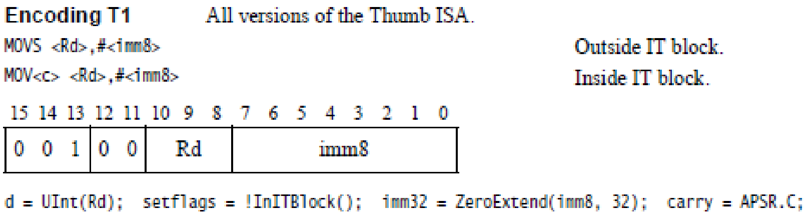
Instruction Encoding

- Instructions are encoded in machine language opcodes

```
Instructions
movs r0, #10


movs r1, #0
```

```
Register Value         Memory Value
001|00|000|00001010    (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.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

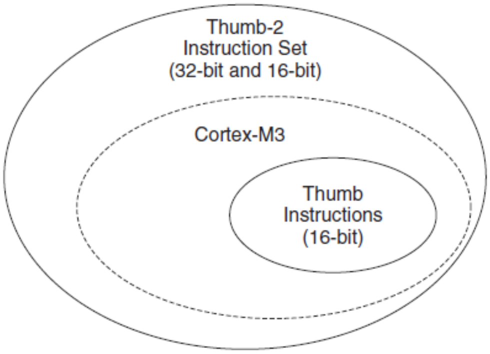
```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Traditional ARM instructions

- Fixed length of 32 bits
- Commonly take two or three operands
- Process data held in registers
- Shift & ALU operation in single clock cycle
- Access memory with load and store instructions only
  - Load/Store multiple register
- Can be extended to execute conditionally by adding the appropriate suffix
- Affect the CPSR status flags by adding the ‘S’ suffix to the instruction

Thumb-2 Instruction Set

- Thumb-2 instruction set is a superset of the previous 16-bit Thumb instruction set
- Provides
  - A large set of 16-bit instructions, enabling 2 instructions per memory fetch
  - A small set of 32-bit instructions to support more complex operations
- **Specific details of this ISA not our focus (we’ll mostly program in C)**

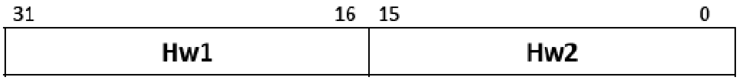


16bit Thumb-2

- Some of the changes used to reduce the length of the instructions from 32 bits to 16 bits
  - reduce the number of bits used to identify the register
    - less number of registers can be used
  - reduce the number of bits used for the immediate value
    - smaller number range
  - remove options such as 'S'
    - make it default for some instructions
  - remove conditional fields (N, Z, V, C)
  - no conditional executions (except branch)
  - remove the optional shift (and no barrel shifter operation
    - introduce dedicated shift instructions
  - remove some of the instructions
    - more restricted coding

Thumb-2 Implementation

- The 32-bit ARM Thumb-2 instructions are added through the space occupied by the Thumb BL and BLX instructions



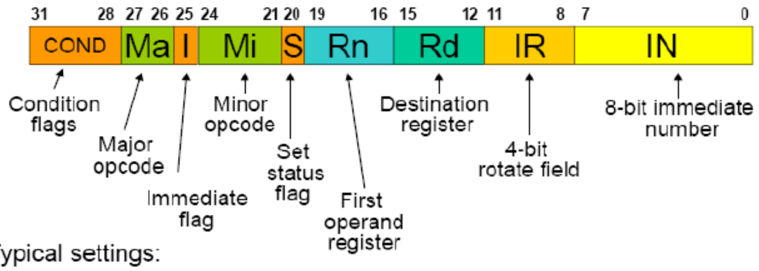
32-bit Thumb-2 Instruction format

- The first Halfword (Hw1)
  - determines the instruction length and functionality
- If the processor decodes the instruction as 32-bit long
  - the processor fetches the second halfword (hw2) of the instruction from the instruction address plus two

32bit Instruction Encoding

Example: ADD instruction format

- ARM 32-bit encoding for ADD with immediate field

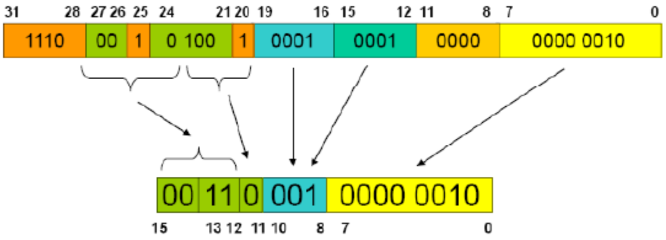


Typical settings:

Major opcode = 00	(this indicates data operation instructions)
Minor opcode = 0100	(specifically, 100 ⇨ ADD instruction)
Immediate flag = 1	(immediate field in operand 2)
Set status flag = 1	(set carry flag after operation)

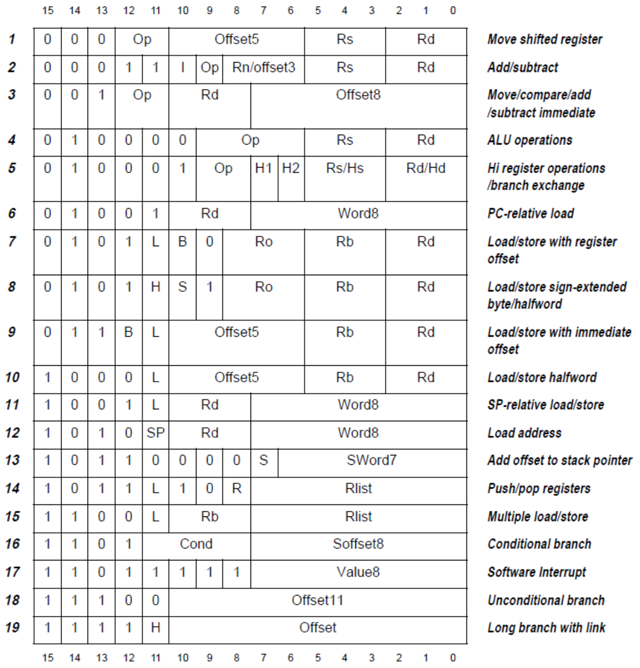
ARM and 16-bit Instruction Encoding

ARM 32-bit encoding: `ADDS r1, r1, #2`



- Equivalent 16-bit Thumb instruction: `ADD r1, #2`
  - No condition flag
  - No rotate field for the immediate number
  - Use 3-bit encoding for the register
  - Shorter opcode with implicit flag settings (e.g. the set status flag is always set)

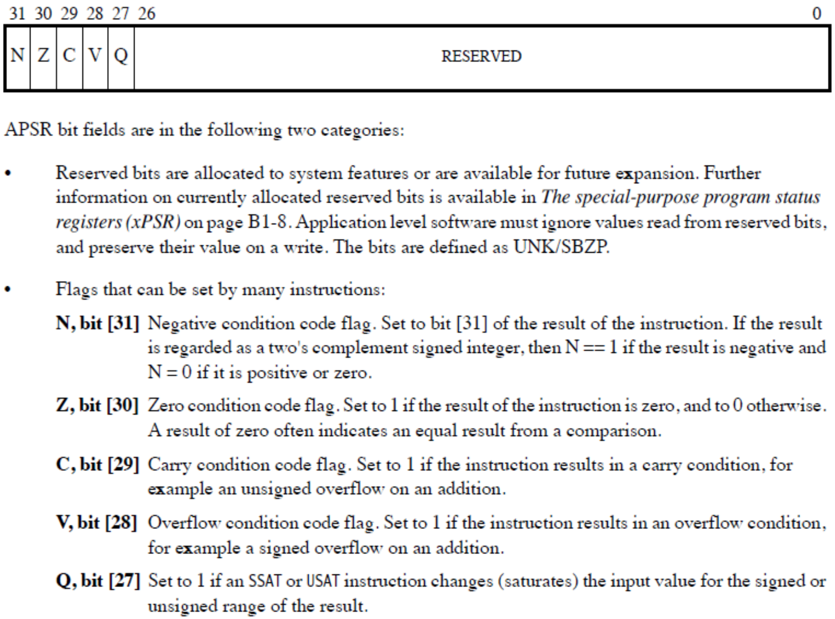
Thumb Instruction Set



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	Op		Offset5					Rs			Rd			*Move shifted register*
2	0	0	0	1	1	I	Op	Rn/offset3			Rs			Rd			*Add/subtract*
3	0	0	1	Op		Rd			Offset8								*Move/compare/add /subtract immediate*
4	0	1	0	0	0	0	Op				Rs			Rd			*ALU operations*
5	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd			*Hi register operations /branch exchange*
6	0	1	0	0	1	Rd			Word8								*PC-relative load*
7	0	1	0	1	L	B	0	Ro			Rb			Rd			*Load/store with register offset*
8	0	1	0	1	H	S	1	Ro			Rb			Rd			*Load/store sign-extended byte/halfword*
9	0	1	1	B	L	Offset5					Rb			Rd			*Load/store with immediate offset*
10	1	0	0	0	L	Offset5					Rb			Rd			*Load/store halfword*
11	1	0	0	1	L	Rd			Word8								*SP-relative load/store*
12	1	0	1	0	SP	Rd			Word8								*Load address*
13	1	0	1	1	0	0	0	0	S	SWord7							*Add offset to stack pointer*
14	1	0	1	1	L	1	0	R	Rlist								*Push/pop registers*
15	1	1	0	0	L	Rb			Rlist								*Multiple load/store*
16	1	1	0	1	Cond				Soffset8								*Conditional branch*
17	1	1	0	1	1	1	1	1	Value8								*Software Interrupt*
18	1	1	1	0	0	Offset11											*Unconditional branch*
19	1	1	1	1	H	Offset											*Long branch with link*
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

- See **4_THUMB_Instr_Set_pt3.pdf** included in **lab1_files.zip**

Application Program Status Register (APSR)



31	30	29	28	27	26 … 0
N	Z	C	V	Q	RESERVED

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 or Z bits might be set
- ADD Rx, Ry
  - Rx = Rx + Ry
  - APSR unchanged
- ADDS
  - Rx = Rx + Ry
  - APSR C or V bits might be set

Overflow and Carry in APSR

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

signed_sum = SInt(x) + SInt(y) + UInt(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

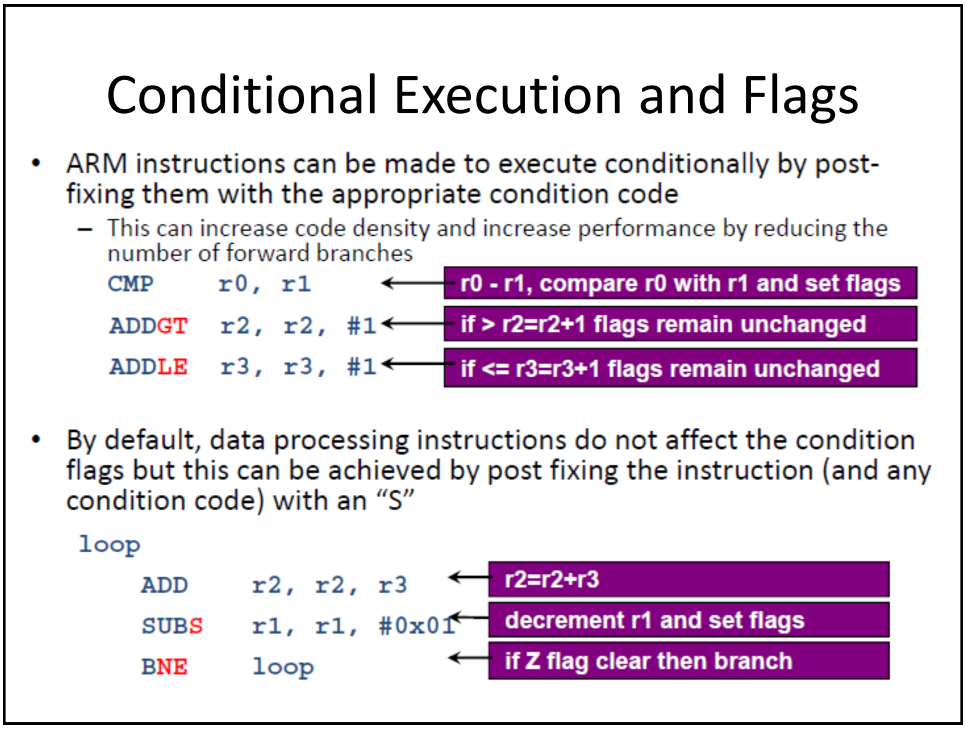
- Each data processing instruction prefixed by condition code
- Result – smooth flow of instructions through pipeline
- 16 condition codes:

EQ	equal	MI	negative	HI	unsigned higher	GT	signed greater than
NE	not equal	PL	positive or zero	LS	unsigned lower or same	LE	signed less than or equal
CS	unsigned higher or same	VS	overflow	GE	signed greater than or equal	AL	always
CC	unsigned lower	VC	no overflow	LT	signed less than	NV	special purpose

Conditional Execution

- Every ARM (32 bit) instruction is conditionally executed.
- The top four bits are ANDed with the CPSR condition codes, If they do not matched the instruction is executed as NOP
- The AL condition is used to execute the instruction irrespective of the value of the condition code flags.
- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S”. Ex: SUBS r1,r1,#1
- Conditional Execution improves code density and performance by reducing the number of forward branch instructions.

```
Normal                Conditional
  CMP   r3,#0               CMP   r3,#0
  BEQ   skip                ADDNE r0,r1,r2
  ADD   r0,r1,r2
skip
```



Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by post-fixing them with the appropriate condition code
  - This can increase code density and increase performance by reducing the number of forward branches

```
CMP     r0, r1
ADDGT   r2, r2, #1
ADDLE   r3, r3, #1
```

r0 - r1, compare r0 with r1 and set flags

if > r2=r2+1 flags remain unchanged

if <= r3=r3+1 flags remain unchanged

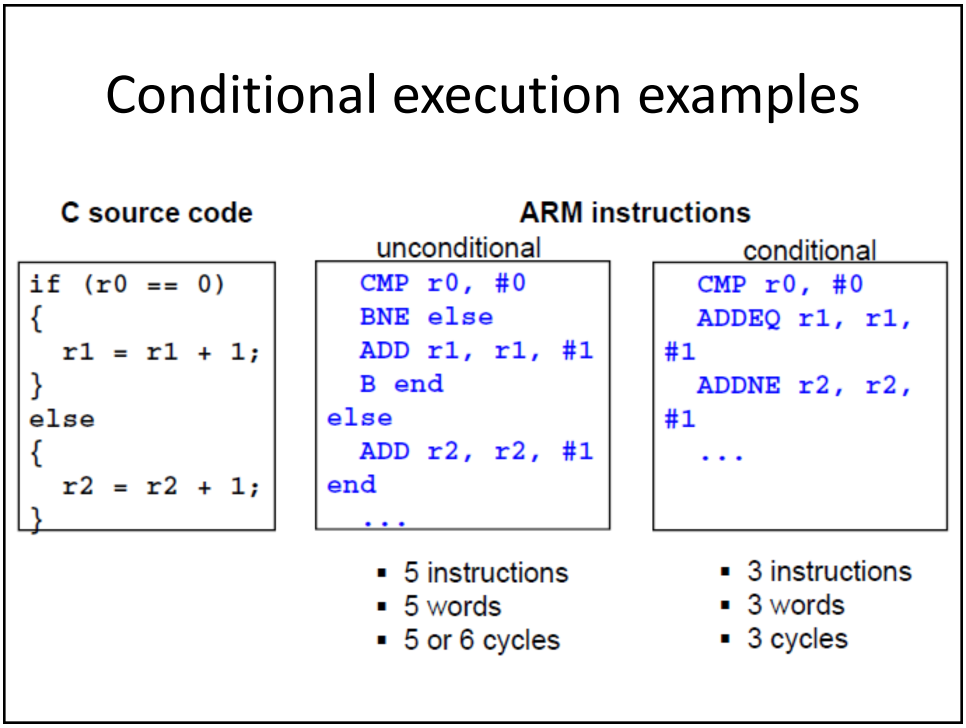
- By default, data processing instructions do not affect the condition flags but this can be achieved by post fixing the instruction (and any condition code) with an "S"

```
loop
    ADD     r2, r2, r3
    SUBS    r1, r1, #0x01
    BNE     loop
```

r2=r2+r3

decrement r1 and set flags

if Z flag clear then branch



Conditional execution examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

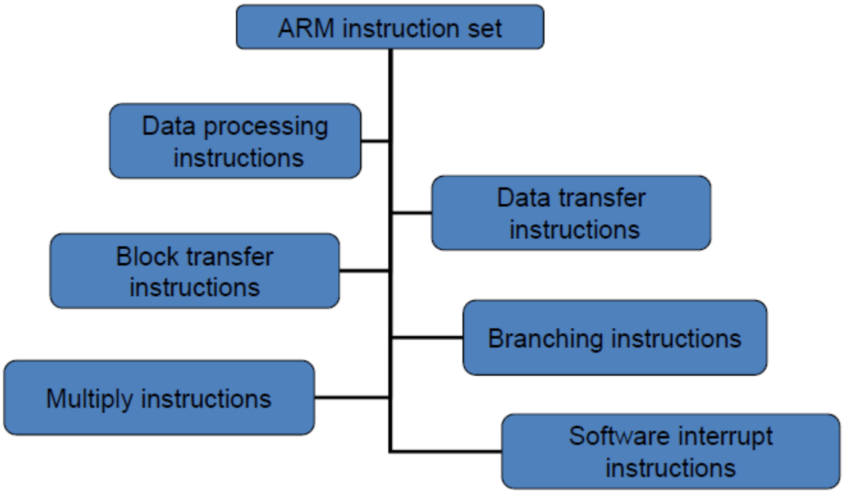
- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1,
#1
  ADDNE r2, r2,
#1
  ...
```

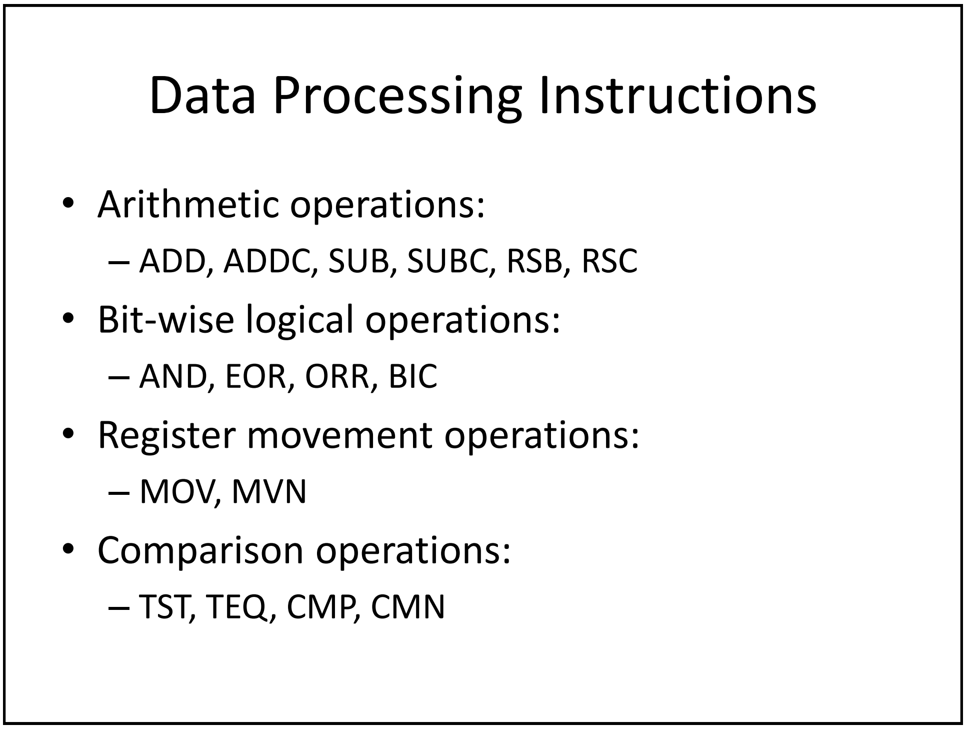
- 3 instructions
- 3 words
- 3 cycles

ARM Instruction Set


ARM instruction set
Data processing instructions
Data transfer instructions
Block transfer instructions
Branching instructions
Multiply instructions
Software interrupt instructions

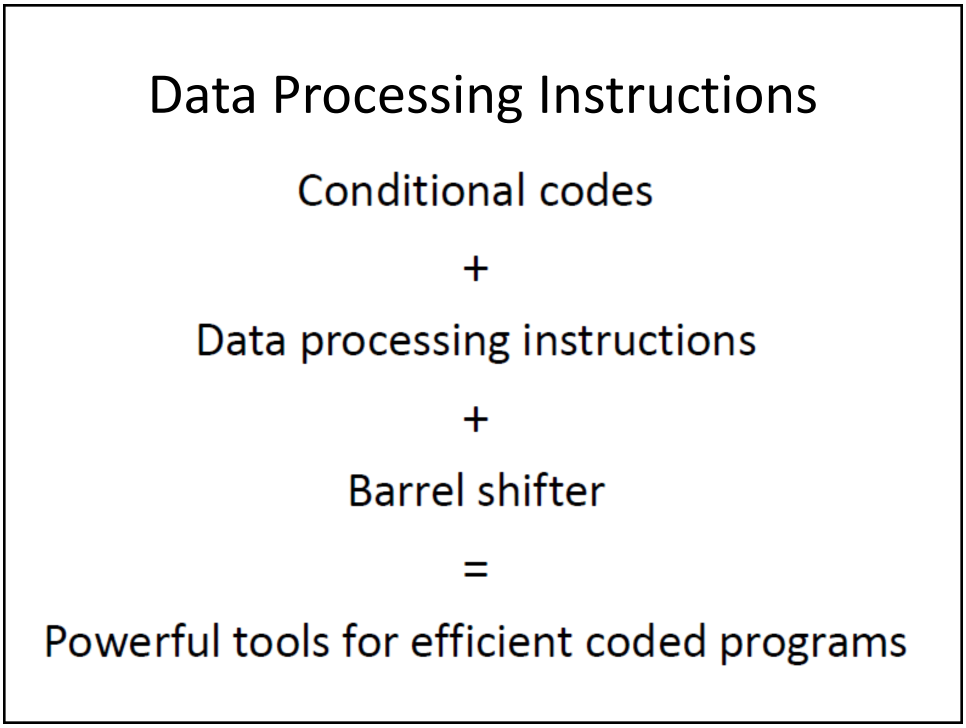
Data Processing Instructions

- Arithmetic and logical operations
- 3-address format:
  - Two 32-bit operands (op1 is register, op2 is register or immediate)
  - 32-bit result placed in a register
- Barrel shifter for op2 allows full 32-bit shift within instruction cycle



Data Processing Instructions

- Arithmetic operations:
  - ADD, ADDC, SUB, SUBC, RSB, RSC
- Bit-wise logical operations:
  - AND, EOR, ORR, BIC
- Register movement operations:
  - MOV, MVN
- Comparison operations:
  - TST, TEQ, CMP, CMN



Data Processing Instructions

Conditional codes

+

Data processing instructions

+

Barrel shifter

=

Powerful tools for efficient coded programs

Data Processing Instructions

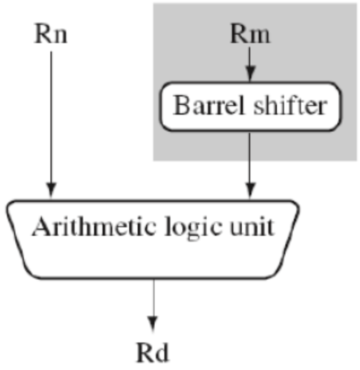
e.g.:

if (z==1) R1=R2+(R3*4)

compiles to

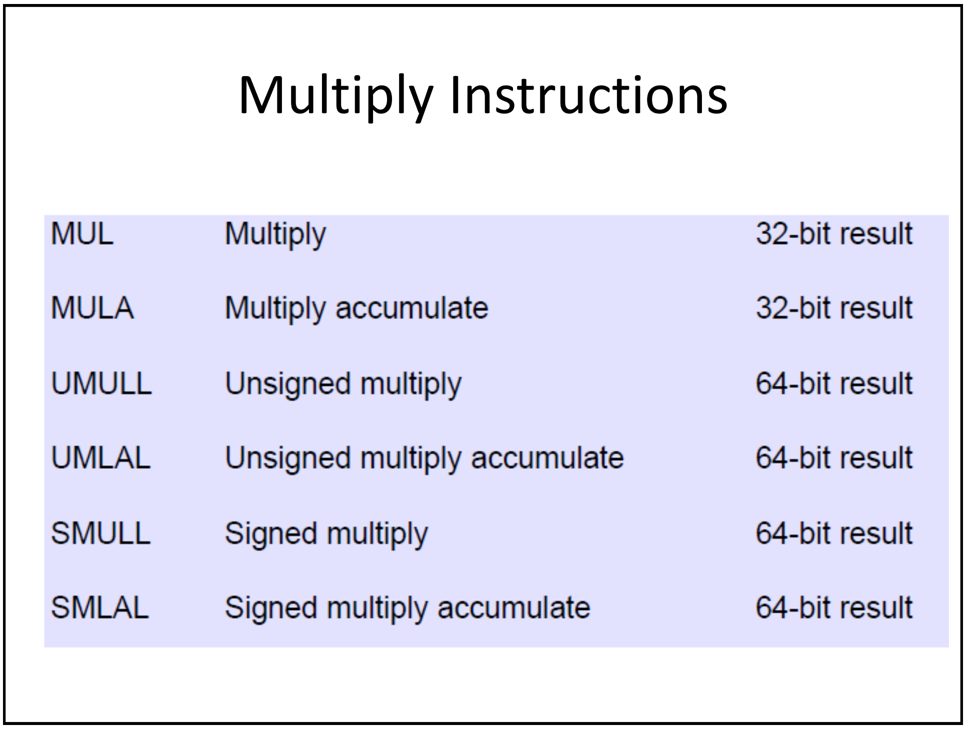
EQADDS R1,R2,R3, LSL #2

(SINGLE INSTRUCTION !)



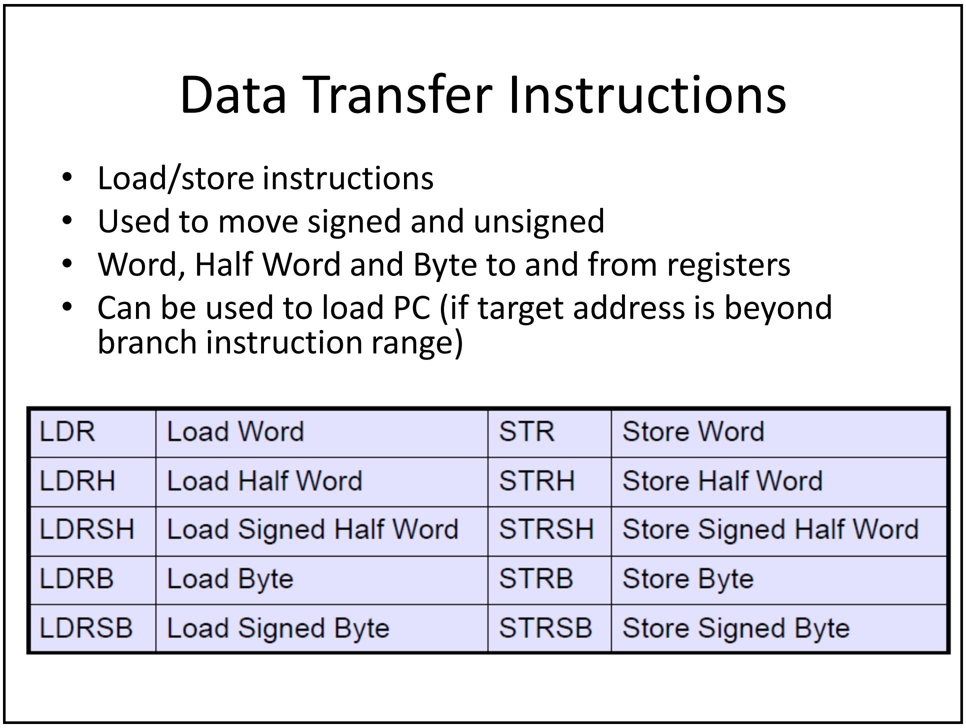
Multiply Instructions

- Integer multiplication (32-bit result)
- Long integer multiplication (64-bit result)
- Built in Multiply Accumulate Unit (MAC)
- Multiply and accumulate instructions add product to running total



Multiply Instructions

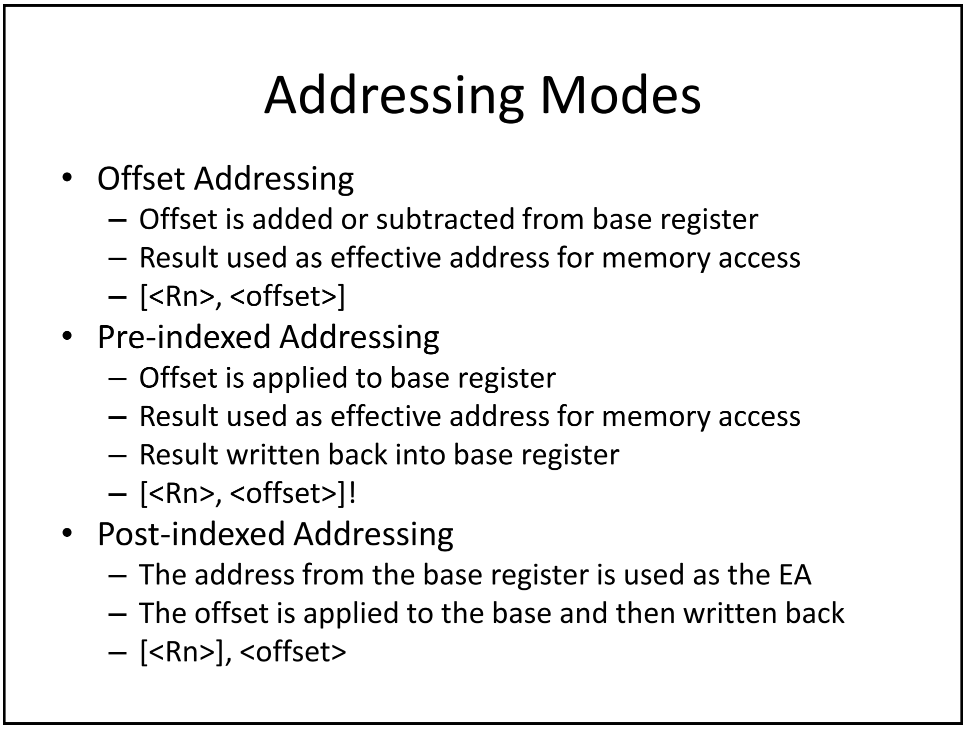
MUL	Multiply	32-bit result
MULA	Multiply accumulate	32-bit result
UMULL	Unsigned multiply	64-bit result
UMLAL	Unsigned multiply accumulate	64-bit result
SMULL	Signed multiply	64-bit result
SMLAL	Signed multiply accumulate	64-bit result



Data Transfer Instructions

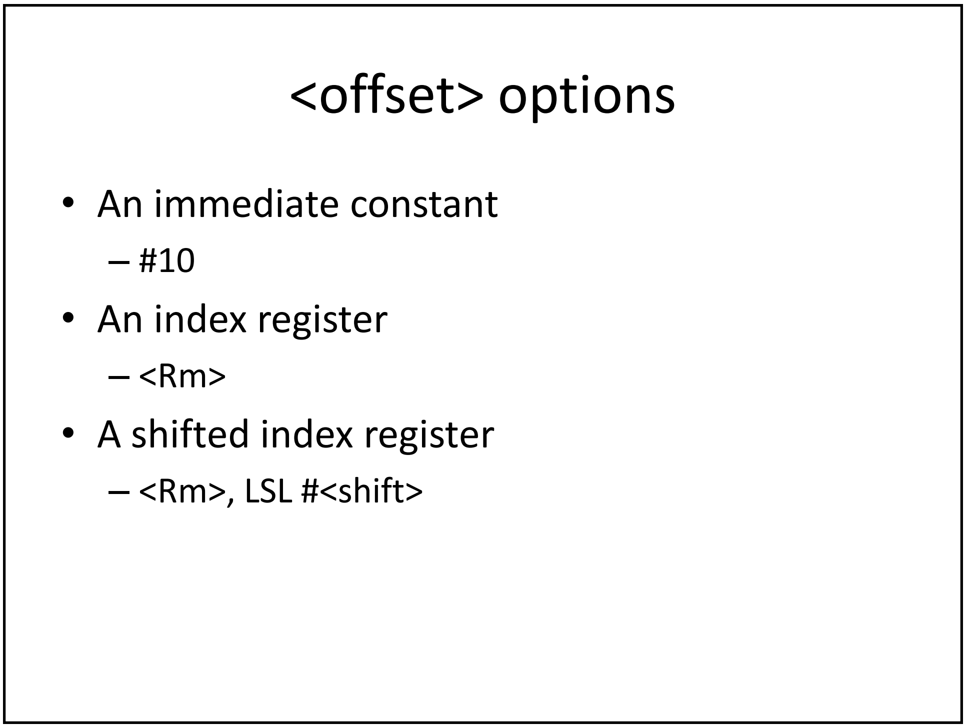
- Load/store instructions
- Used to move signed and unsigned
- Word, Half Word and Byte to and from registers
- Can be used to load PC (if target address is beyond branch instruction range)

LDR	Load Word	STR	Store Word
LDRH	Load Half Word	STRH	Store Half Word
LDRSH	Load Signed Half Word	STRSH	Store Signed Half Word
LDRB	Load Byte	STRB	Store Byte
LDRSB	Load Signed Byte	STRSB	Store Signed Byte



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>

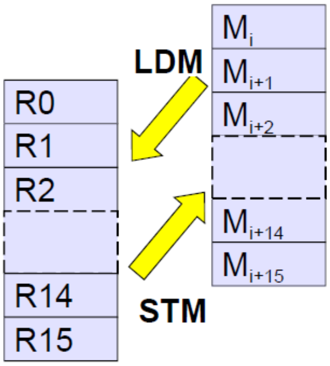


<offset> options

- An immediate constant
  - #10
- An index register
  - <Rm>
- A shifted index register
  - <Rm>, LSL #<shift>

Block Transfer Instructions

- Load/Store Multiple instructions (*LDM*/*STM*)
- Whole register bank or a subset copied to memory or restored with single instruction

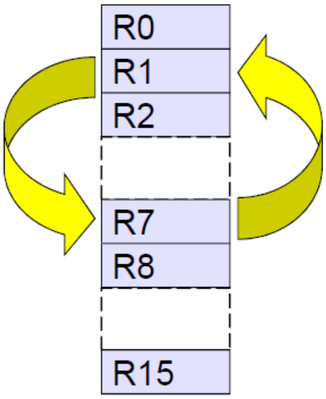


Swap Instruction

- Exchanges a word between registers
  - Two cycles

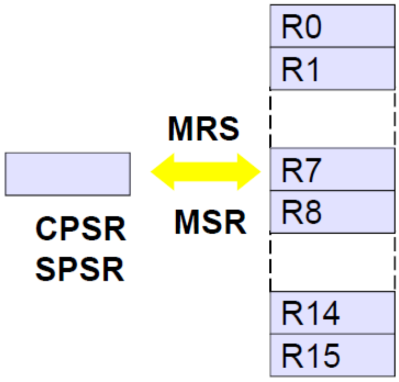
    but

  single atomic action
  - Support for RT semaphores



Modifying the Status Registers

- Only indirectly
- *MSR* moves contents from CPSR/SPSR to selected GPR
- *MRS* moves contents from selected GPR to CPSR/SPSR
- Only in privileged modes



Software Interrupt

- *SWI* instruction
  - Forces CPU into supervisor mode
  - Usage: SWI #n

31 28	27 24	23 0
Cond	Opcode	Ordinal

- Maximum 2^{24} calls
- Suitable for running privileged code and making OS calls

Branching Instructions

- *Branch (B):*
  - jumps forwards/backwards up to 32 MB
- *Branch link (BL):*
  - same + saves (PC+4) in LR
- Suitable for function call/return
- Condition codes for conditional branches

Branching Instructions

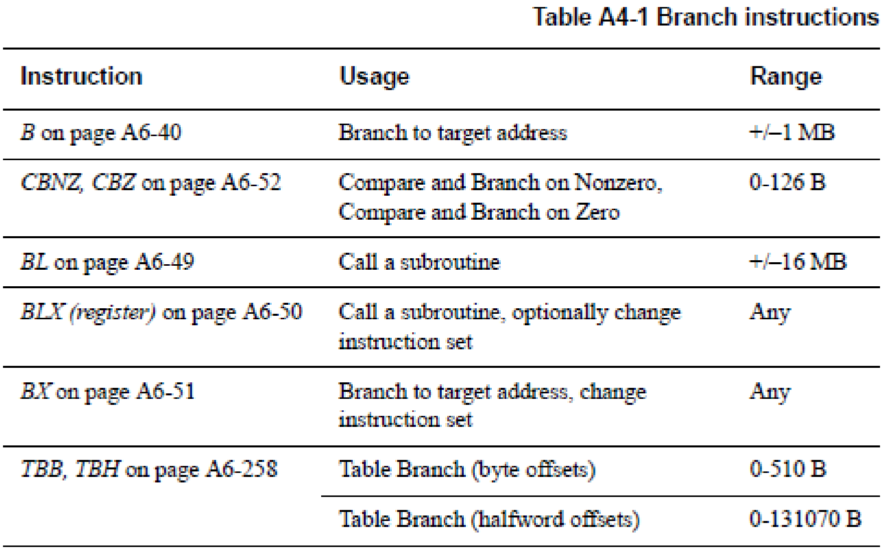


Table A4-1 Branch instructions

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

IF-THEN Instruction

- Another alternative to execute conditional code is the new 16-bit IF-THEN (IT) instruction
  - no change in program flow
  - no branching overhead
- Can use with 32-bit Thumb-2 instructions that do not support the 'S' suffix
- Example:

```
CMP R1, R2          ; If R1 = R2
IT EQ               ; execute next (1st)
                    ; instruction
ADDEQ R2, R1, R0    ; 1st instruction
```

- The conditional codes can be extended up to 4 instructions

Barrier instructions

- Useful for multi-core & Self-modifying code

Instruction	Description
DMB	Data memory barrier; ensures that all memory accesses are completed before new memory access is committed
DSB	Data synchronization barrier; ensures that all memory accesses are completed before next instruction is executed
ISB	Instruction synchronization barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Unified Assembly Language

- UAL supports generation of either Thumb-2 or ARM instructions from the same source code
  - same syntax for both the Thumb code and ARM code
  - enable portability of code for different ARM processor families
- Interpretation of code type is based on the directive listed in the assembly file

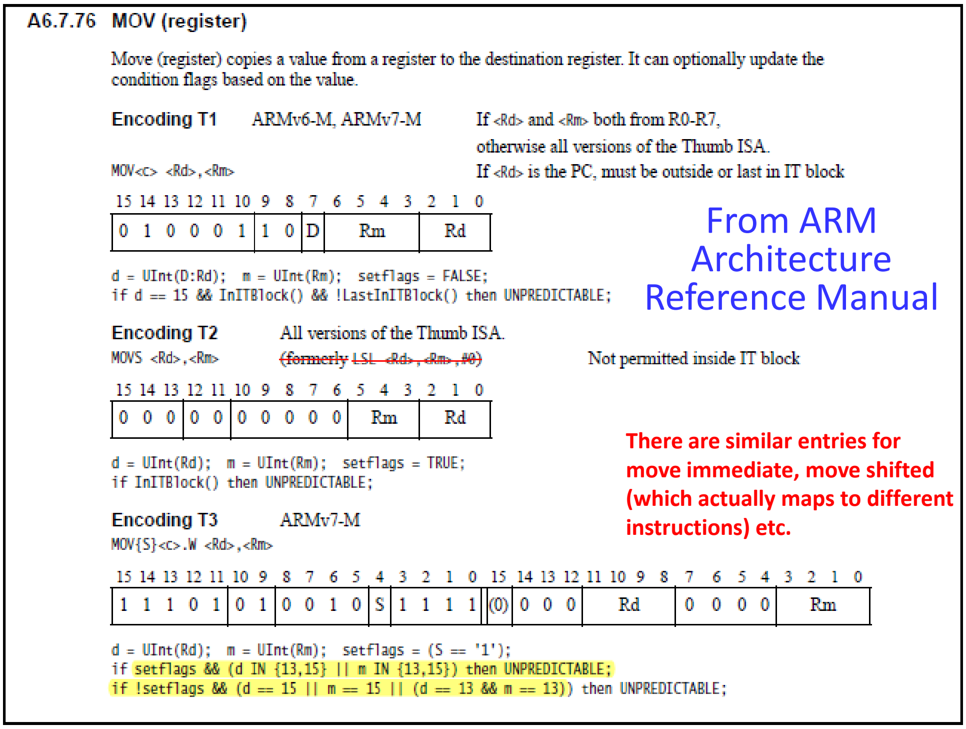
- Example:
  - For GNU Assembler, the directive for UAL is

  .syntax unified
  - For ARM assembler, the directive for UAL is

  THUMB

Example 1

```
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
```



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.
If <Rd> is the PC, must be outside or last in IT block

`MOV<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

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

From ARM Architecture Reference Manual

Encoding T2 All versions of the Thumb ISA.

`MOVS <Rd>,<Rm>` ~~(formerly LSL <Rd>,<Rm>,#0)~~

Not permitted inside IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

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

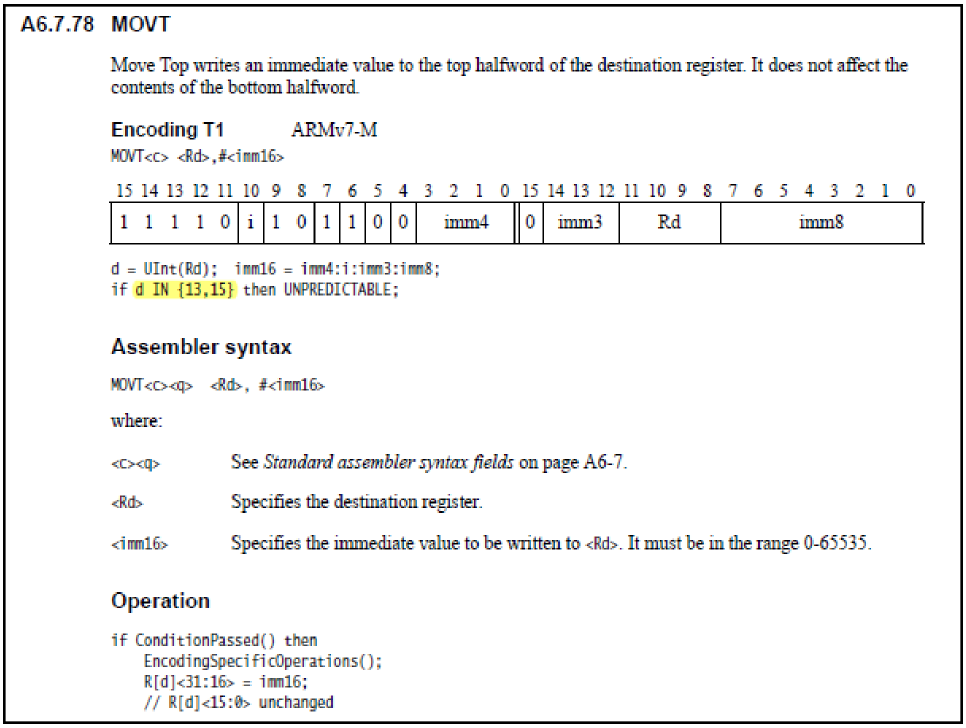
There are similar entries for move immediate, move shifted (which actually maps to different instructions) etc.

Encoding T3 ARMv7-M

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	0	0	0	Rd				0	0	0	0	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;
```



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>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	1	0	0	imm4				0	imm3			Rd				imm8							

```
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
```

Example 2

```
int counter;
int Counter_Inc(void) {
    return counter ++;
}
```

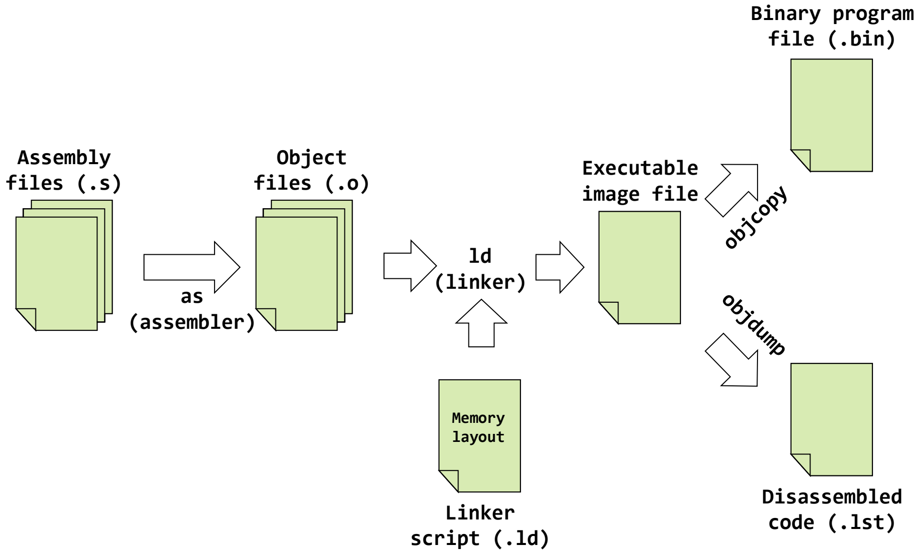
Resulting (annotated) assembly language with corresponding machine code:

```
Counter_Inc:
0: f240 0300   movw r3 , #:lower16:counter    // r3 = &counter
4: f2c0 0300   movt r3 , #:upper16:counter
8: 6818        ldr  r0 , [r3 , #0]            // r0 = *r3
a: 1c42        adds r2 , r0  , #1             // r2 = r0 + 1
c: 601a        str  r2 , [r3 , #0]            // *r3 = r2
e: 4740        bx   lr                        // return r0
```

- Two 32-bit instructions (**movw, movt**) are used to load the lower/upper halves of the address of counter (known at link time, and hence 0 in the code listing)
- Then, three 16-bit instructions load (**ldr**) the value of counter, increment (**adds**) the value, and write back (**str**) the updated value
- Finally, the procedure returns the original counter

- **Key points**:
  - Cortex-M3 utilizes a mixture of 32-bit and 16-bit instructions (mostly the latter) and the core interacts with memory solely through load and store instructions
  - While there are instructions that load/store groups of registers (in multiple cycles) there are no instructions that directly operate on memory locations

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



Binary program file (.bin)

Assembly files (.s)

Object files (.o)

Executable image file

objcopy

as (assembler)

ld (linker)

objdump

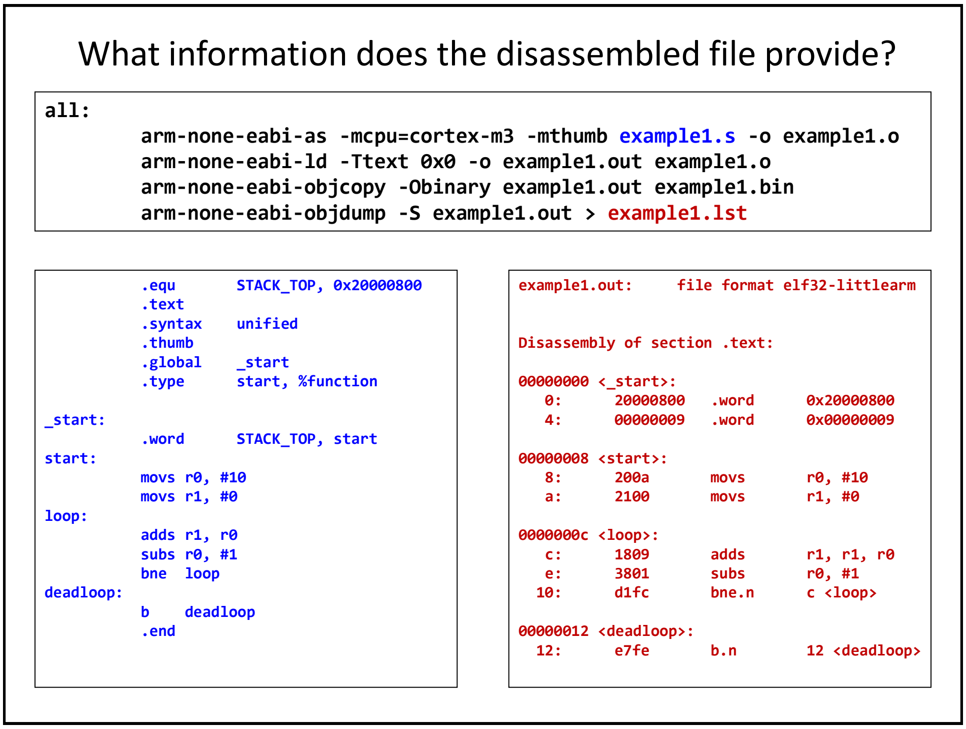
Memory layout

Linker script (.ld)

Disassembled code (.lst)

An ARM assembly language program for GNU

```
        .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
```

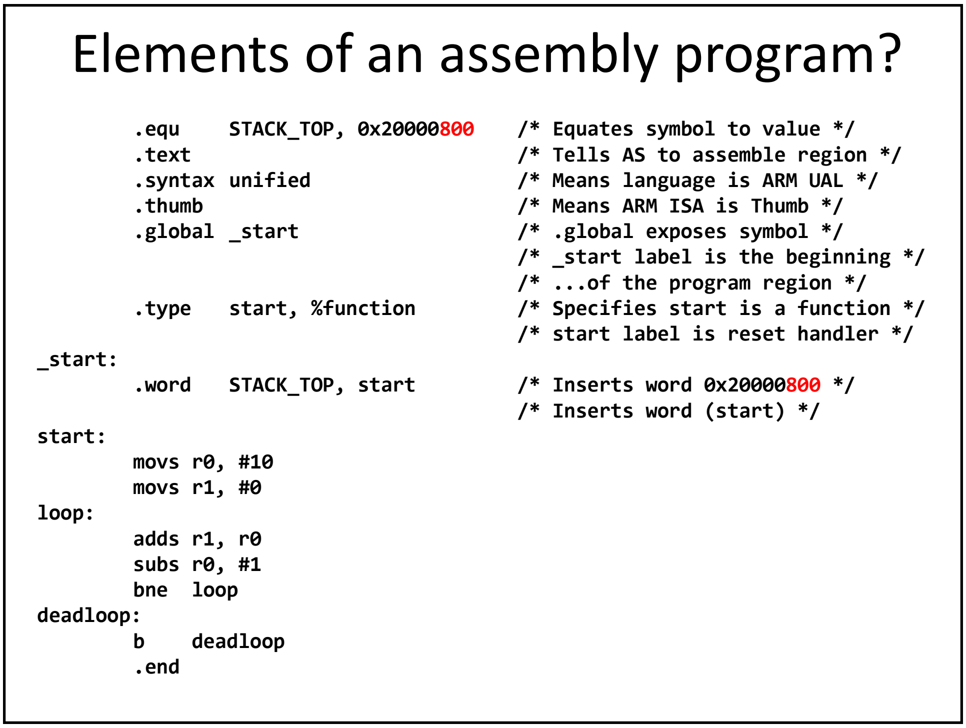


What information does the disassembled file provide?

```
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
```

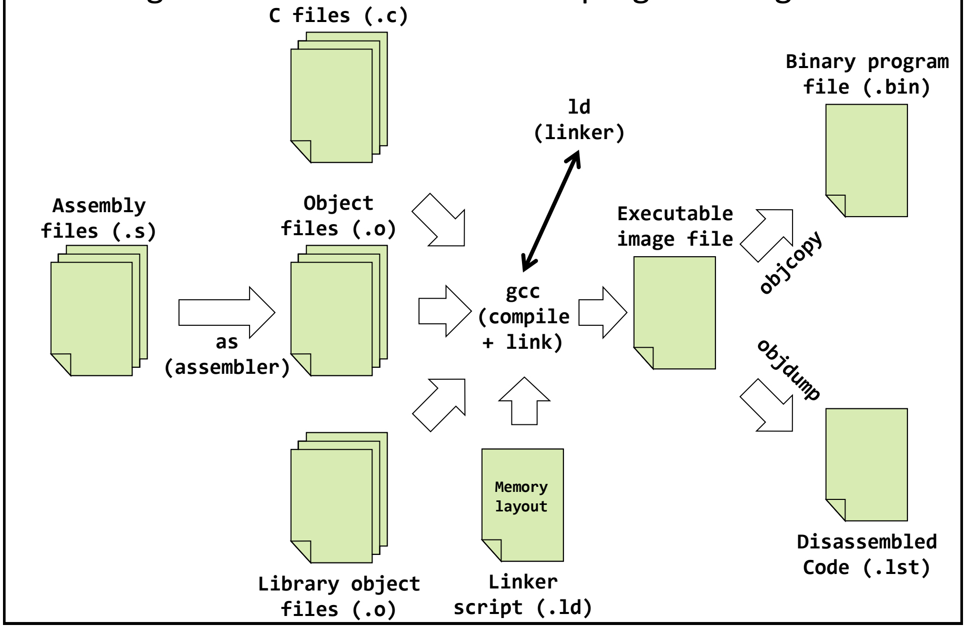
```
example1.out:     file format elf32-littlearm


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>
```



Elements of an assembly program?

```
        .equ    STACK_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
        movs r1, #0
loop:
        adds r1, r0
        subs r0, #1
        bne  loop
deadloop:
        b    deadloop
        .end
```

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


C files (.c)

Assembly files (.s)

as (assembler)

Object files (.o)

Library object files (.o)

ld (linker)

gcc (compile + link)

Memory layout

Linker script (.ld)

Executable image file

objcopy

Binary program file (.bin)

objdump

Disassembled Code (.lst)

Nested Vector Interrupt Controller (NVIC)

- A programmable device that sits between the CM3 core and the microcontroller
- CM3 uses a prioritized vectored interrupt model – the vector table is defined to reside starting at memory location 0
- First 16 entries in this table are defined for all Cortex-M3 implementations while the remainder, up to 240, are implementation specific
- NVIC supports dynamic redefinition of priorities with up to 256 priority levels
- Two entries in the vector table are especially important:
  - address 0 contains the address of the initial stack pointer
  - address 4 contains the address of the “reset handler” to be executed at boot time

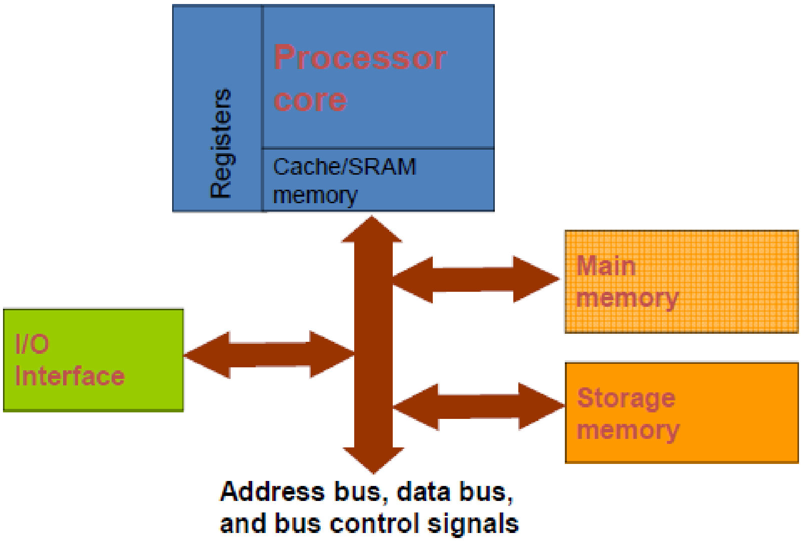
Nested Vector Interrupt Controller (NVIC)

- Provides key system control registers including the System Timer (SysTick) that provides a regular timer interrupt
- Provision for a built-in timer across the Cortex-M3 family has the significant advantage of making operating system code highly portable – all operating systems need at least one core timer for time-slicing
- Registers used to control the NVIC are defined to reside at address 0xE000E000 and are defined by the Cortex-M3 specification
- These registers are accessed with the system bus

Outline

- ARM Cortex-M3 processor
- NXP LPC17xx microcontroller unit (MCU)

Basic Processor Based System



Registers

Processor core

Cache/SRAM memory

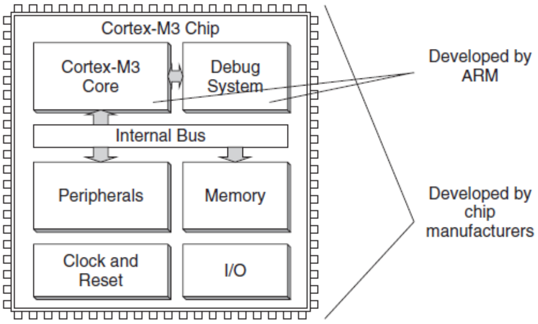
Main memory

I/O Interface

Storage memory

Address bus, data bus,
and bus control signals

Cortex-M3 processor vs.
CM3-based Microcontroller Units



Cortex-M3 Chip

Cortex-M3 Core

Debug System

Internal Bus

Peripherals

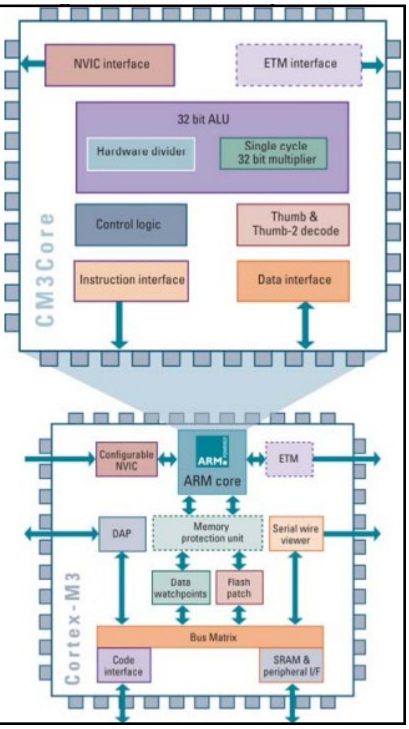
Memory

Clock and Reset

I/O

Developed by ARM

Developed by chip manufacturers



CM3Core

NVIC interface

ETM interface

32 bit ALU

Hardware divider

Single cycle
32 bit multiplier

Control logic

Thumb &
Thumb-2 decode

Instruction interface

Data interface

Cortex-M3

Configurable NVIC

ARM core

ETM

DAP

Memory protection unit

Serial wire viewer

Data watchpoints

Flash patch

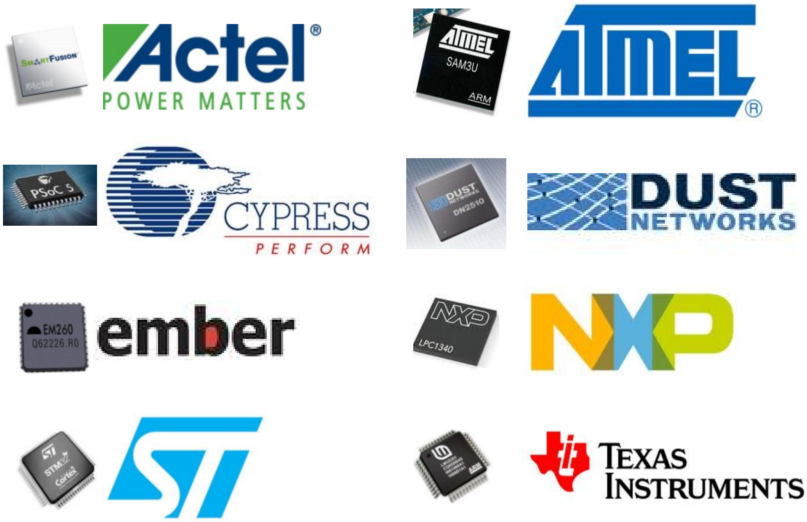
Bus Matrix

Code interface

SRAM & peripheral I/F

While there is significant overlap between the families and their peripherals, there are also important differences

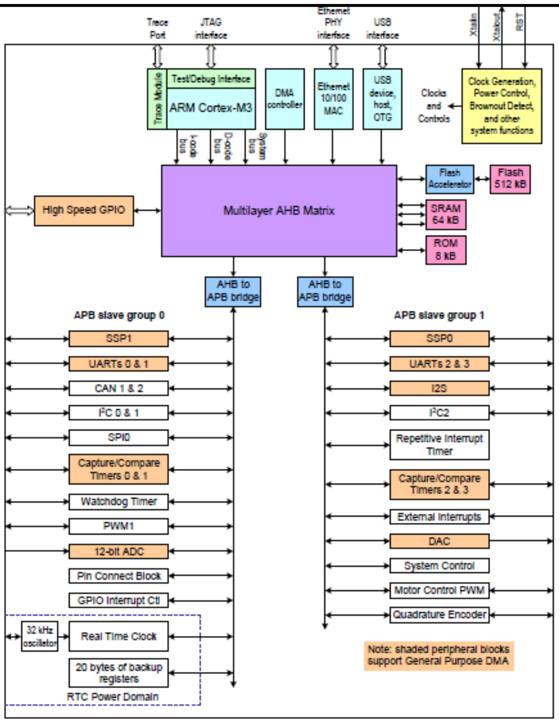
In the lab of this course we focus on the NXP's LPC17xx family



LPC17xx

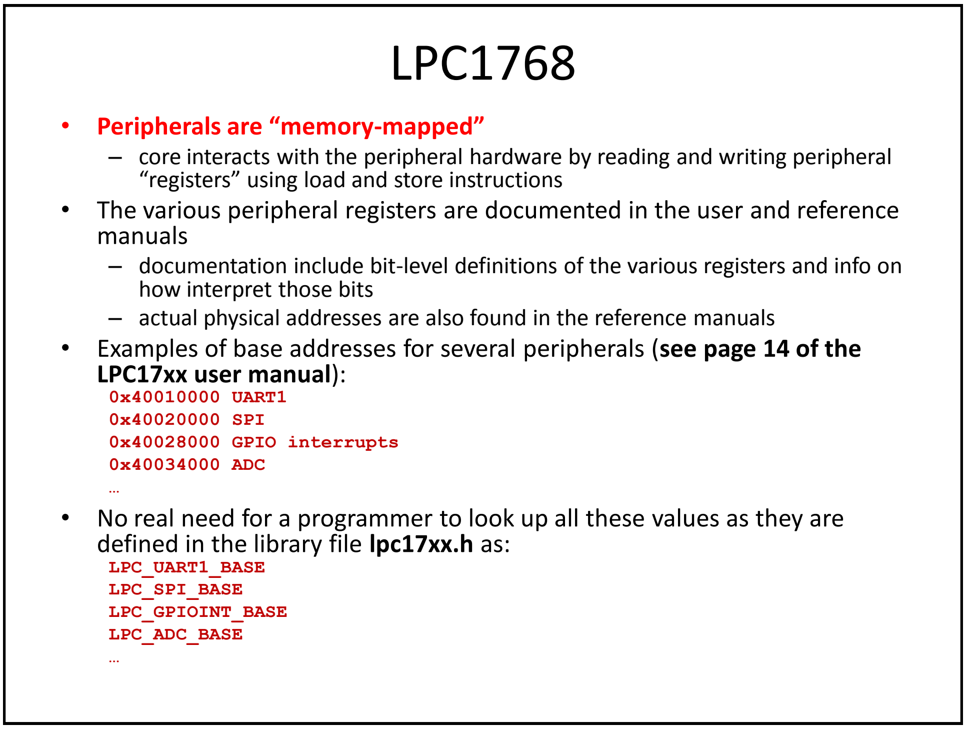
- LPC17xx (of NXP) is an ARM Cortex-M3 based microcontroller
- The Cortex-M3 is also the basis for microcontrollers from other manufacturers including TI, ST, Toshiba, Atmel, etc.
- LPC1768 operates at up to a 100 MHz CPU frequency
- Sophisticated clock system
- Peripherals include:
  - up to 512 kB of flash memory, up to 64 kB of data memory
  - Ethernet MAC
  - a USB interface that can be configured as either Host, Device, or OTG
  - 8 channel general purpose DMA controller
  - 4 UARTs, 2 CAN channels, 2 SSP controllers, SPI interface
  - 3 I2C interfaces, 2-input plus 2-output I2S interface
  - 8 channel 12-bit ADC, 10-bit DAC, motor control PWM
  - Quadrature Encoder interface, 4 general purpose timers,
  - 6-output general purpose PWM
  - ultra-low power RTC with separate battery supply
  - up to 70 general purpose I/O pins

LPC1768



LPC1768

- LPC1768 microcontrollers are based on the Cortex-M3 processor with a set of peripherals distributed across three buses – Advanced High-performance Bus (AHB) and its two Advanced Peripheral Bus (APB) sub-buses APB1 and APB2.
- These peripherals:
  - are controlled by the CM3 core with load and store instructions that access **memory mapped registers**
  - can "interrupt" the core to request attention through peripheral specific interrupt requests routed through the NVIC
- Data transfers between peripherals and memory can be automated using DMA
- Labs will cover among others:
  - basic peripheral configuration (e.g., lab1 illustrates GPIO General Purpose I/O peripherals)
  - how interrupts can be used to build effective software
  - how to use DMA to improve performance and allow processing to proceed in parallel with data transfer



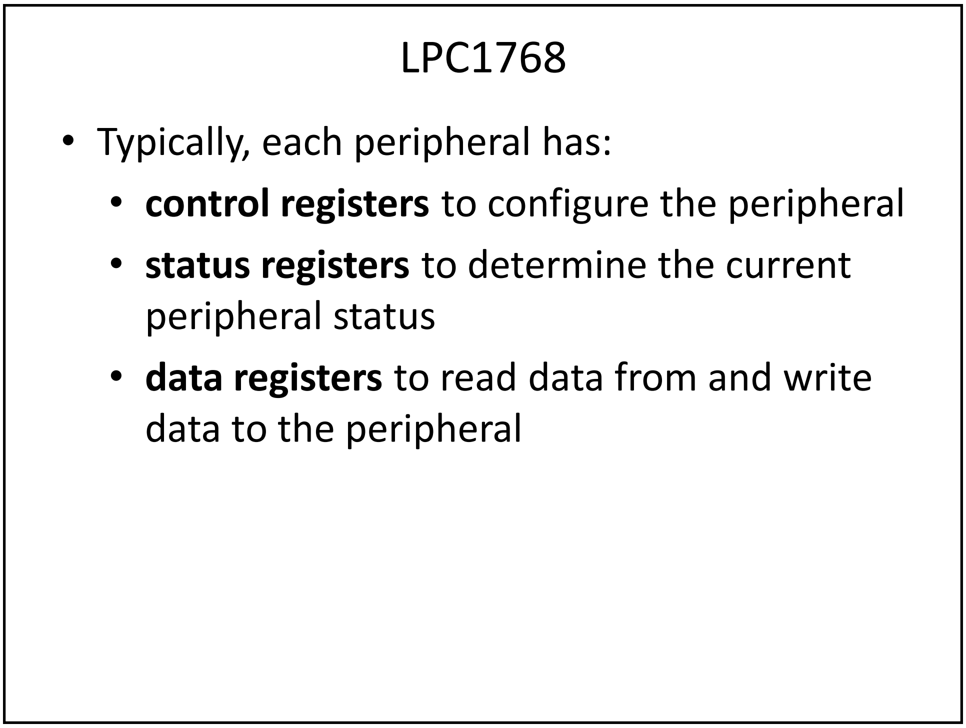
LPC1768

- **Peripherals are “memory-mapped”**
  - core interacts with the peripheral hardware by reading and writing peripheral “registers” using load and store instructions
- The various peripheral registers are documented in the user and reference manuals
  - documentation include bit-level definitions of the various registers and info on how interpret those bits
  - actual physical addresses are also found in the reference manuals
- Examples of base addresses for several peripherals (**see page 14 of the LPC17xx user manual**):

```
0x40010000 UART1
0x40020000 SPI
0x40028000 GPIO interrupts
0x40034000 ADC
…
```

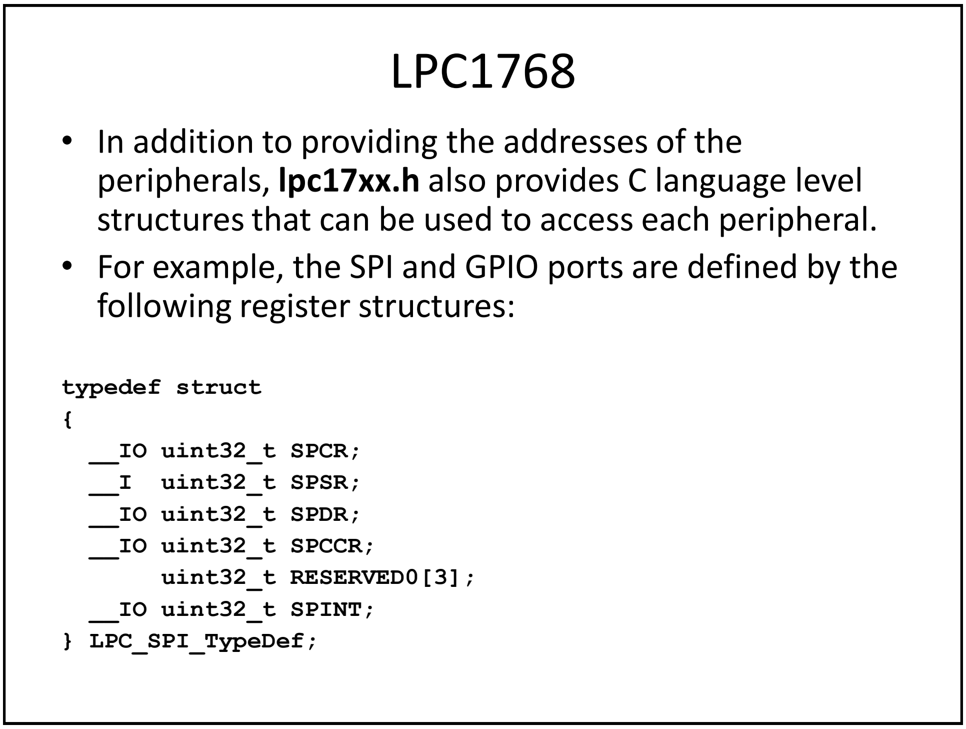
- No real need for a programmer to look up all these values as they are defined in the library file **lpc17xx.h** as:

```
LPC_UART1_BASE
LPC_SPI_BASE
LPC_GPIOINT_BASE
LPC_ADC_BASE
…
```



LPC1768

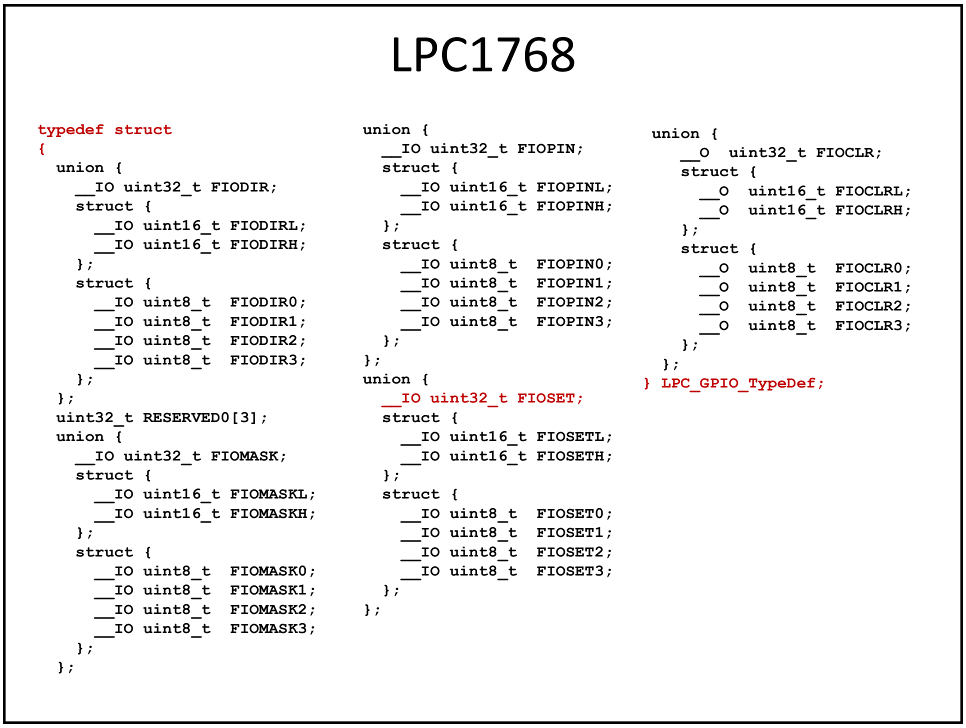
- Typically, each peripheral has:
  - **control registers** to configure the peripheral
  - **status registers** to determine the current peripheral status
  - **data registers** to read data from and write data to the peripheral



LPC1768

- In addition to providing the addresses of the peripherals, **lpc17xx.h** also provides C language level structures that can be used to access each peripheral.
- For example, the SPI and GPIO ports are defined by the following register structures:

```
typedef struct
{
  __IO uint32_t SPCR;
  __I  uint32_t SPSR;
  __IO uint32_t SPDR;
  __IO uint32_t SPCCR;
       uint32_t RESERVED0[3];
  __IO uint32_t SPINT;
} LPC_SPI_TypeDef;
```



LPC1768

```
typedef struct
{
  union {
    __IO uint32_t FIODIR;
    struct {
      __IO uint16_t FIODIRL;
      __IO uint16_t FIODIRH;
    };
    struct {
      __IO uint8_t  FIODIR0;
      __IO uint8_t  FIODIR1;
      __IO uint8_t  FIODIR2;
      __IO uint8_t  FIODIR3;
    };
  };
  uint32_t RESERVED0[3];
  union {
    __IO uint32_t FIOMASK;
    struct {
      __IO uint16_t FIOMASKL;
      __IO uint16_t FIOMASKH;
    };
    struct {
      __IO uint8_t  FIOMASK0;
      __IO uint8_t  FIOMASK1;
      __IO uint8_t  FIOMASK2;
      __IO uint8_t  FIOMASK3;
    };
  };
  union {
    __IO uint32_t FIOPIN;
    struct {
      __IO uint16_t FIOPINL;
      __IO uint16_t FIOPINH;
    };
    struct {
      __IO uint8_t  FIOPIN0;
      __IO uint8_t  FIOPIN1;
      __IO uint8_t  FIOPIN2;
      __IO uint8_t  FIOPIN3;
    };
  };
  union {
    __IO uint32_t FIOSET;
    struct {
      __IO uint16_t FIOSETL;
      __IO uint16_t FIOSETH;
    };
    struct {
      __IO uint8_t  FIOSET0;
      __IO uint8_t  FIOSET1;
      __IO uint8_t  FIOSET2;
      __IO uint8_t  FIOSET3;
    };
  };
  union {
    __O  uint32_t FIOCLR;
    struct {
      __O  uint16_t FIOCLRL;
      __O  uint16_t FIOCLRH;
    };
    struct {
      __O  uint8_t  FIOCLR0;
      __O  uint8_t  FIOCLR1;
      __O  uint8_t  FIOCLR2;
      __O  uint8_t  FIOCLR3;
    };
  };
} LPC_GPIO_TypeDef;
```

LPC1768

- The register addresses of the various ports are defined in the library (see **lpc17xx.h**):

```
#define LPC_APB0_BASE         (0x40000000UL)
…
#define LPC_UART1_BASE        (LPC_APB0_BASE + 0x10000)
#define LPC_SPI_BASE          (LPC_APB0_BASE + 0x20000)
#define LPC_GPIOINT_BASE      (LPC_APB0_BASE + 0x28080)
#define LPC_ADC_BASE          (LPC_APB0_BASE + 0x34000)
…
#define LPC_GPIO1  ((LPC_GPIO_TypeDef *) LPC_GPIO1_BASE)
…
```

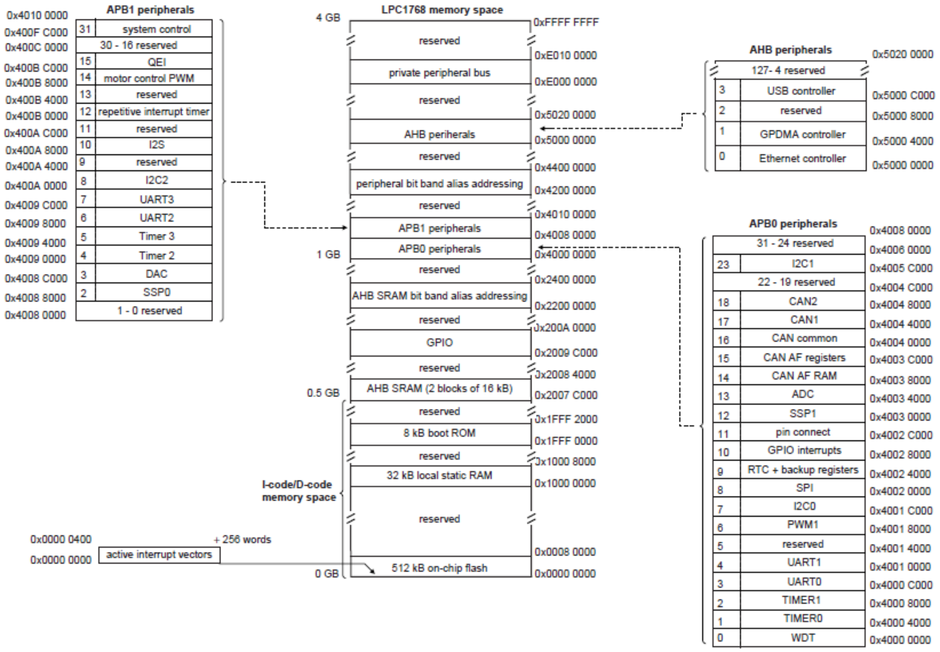
- For example, to turn on LED P1.29 on the development board, the following code can be used:

```
LPC_GPIO1->FIOSET = 1 << 29;
```

Memory

- On-chip flash memory system
  - Up to 512 kB of on-chip flash memory
  - Flash memory accelerator maximizes performance for use with the two fast AHB-Lite buses
  - Can be used for both code and data storage
- On-chip Static RAM
  - Up to 64 kB of on-chip static RAM memory
  - Up to 32 kB of SRAM, accessible by the CPU and all three DMA controllers are on a higher-speed bus
  - Devices with more than 32 kB SRAM have two additional 16 kB SRAM blocks

LPC17xx system memory map



APB1 peripherals

0x4010 0000
0x400F C000 31 system control
0x400C 0000 30 - 16 reserved
0x400B C000 15 QEI
0x400B 8000 14 motor control PWM
0x400B 4000 13 reserved
0x400B 0000 12 repetitive interrupt timer
0x400A C000 11 reserved
0x400A 8000 10 I2S
0x400A 4000 9 reserved
0x400A 0000 8 I2C2
0x4009 C000 7 UART3
0x4009 8000 6 UART2
0x4009 4000 5 Timer 3
0x4009 0000 4 Timer 2
0x4008 C000 3 DAC
0x4008 8000 2 SSP0
0x4008 0000 1 - 0 reserved

LPC1768 memory space

4 GB 0xFFFF FFFF
reserved
0xE010 0000
private peripheral bus
0xE000 0000
reserved
0x5020 0000
AHB periherals
0x5000 0000
reserved
0x4400 0000
peripheral bit band alias addressing
0x4200 0000
reserved
0x4010 0000
APB1 peripherals
0x4008 0000
APB0 peripherals
1 GB 0x4000 0000
reserved
0x2400 0000
AHB SRAM bit band alias addressing
0x2200 0000
reserved
0x200A 0000
GPIO
0x2009 C000
reserved
0x2008 4000
0.5 GB AHB SRAM (2 blocks of 16 kB)
0x2007 C000
reserved
0x1FFF 2000
8 kB boot ROM
0x1FFF 0000
reserved
0x1000 8000
32 kB local static RAM
0x1000 0000
I-code/D-code memory space
reserved
0x0008 0000
512 kB on-chip flash
0 GB 0x0000 0000

0x0000 0400 + 256 words
0x0000 0000 active interrupt vectors

AHB peripherals

0x5020 0000
127- 4 reserved
3 USB controller 0x5000 C000
2 reserved 0x5000 8000
1 GPDMA controller 0x5000 4000
0 Ethernet controller 0x5000 0000

APB0 peripherals

0x4008 0000
31 - 24 reserved 0x4006 0000
23 I2C1 0x4005 C000
22 - 19 reserved 0x4004 C000
18 CAN2 0x4004 8000
17 CAN1 0x4004 4000
16 CAN common 0x4004 0000
15 CAN AF registers 0x4003 C000
14 CAN AF RAM 0x4003 8000
13 ADC 0x4003 4000
12 SSP1 0x4003 0000
11 pin connect 0x4002 C000
10 GPIO interrupts 0x4002 8000
9 RTC + backup registers 0x4002 4000
8 SPI 0x4002 0000
7 I2C0 0x4001 C000
6 PWM1 0x4001 8000
5 reserved 0x4001 4000
4 UART1 0x4001 0000
3 UART0 0x4000 C000
2 TIMER1 0x4000 8000
1 TIMER0 0x4000 4000
0 WDT 0x4000 0000

References & Credits

- Joseph Jiu, The Definitive guide to the ARM Cortext-M3, 2007
- LPC17xx microcontroller user manual
- Cortex-M3 Processor Technical Reference Manual
- Lab manual (G. Brown, Indiana)
- EECS 373, UMich