

Embedded Systems

The ARM Architecture and ISA

Lesson 04

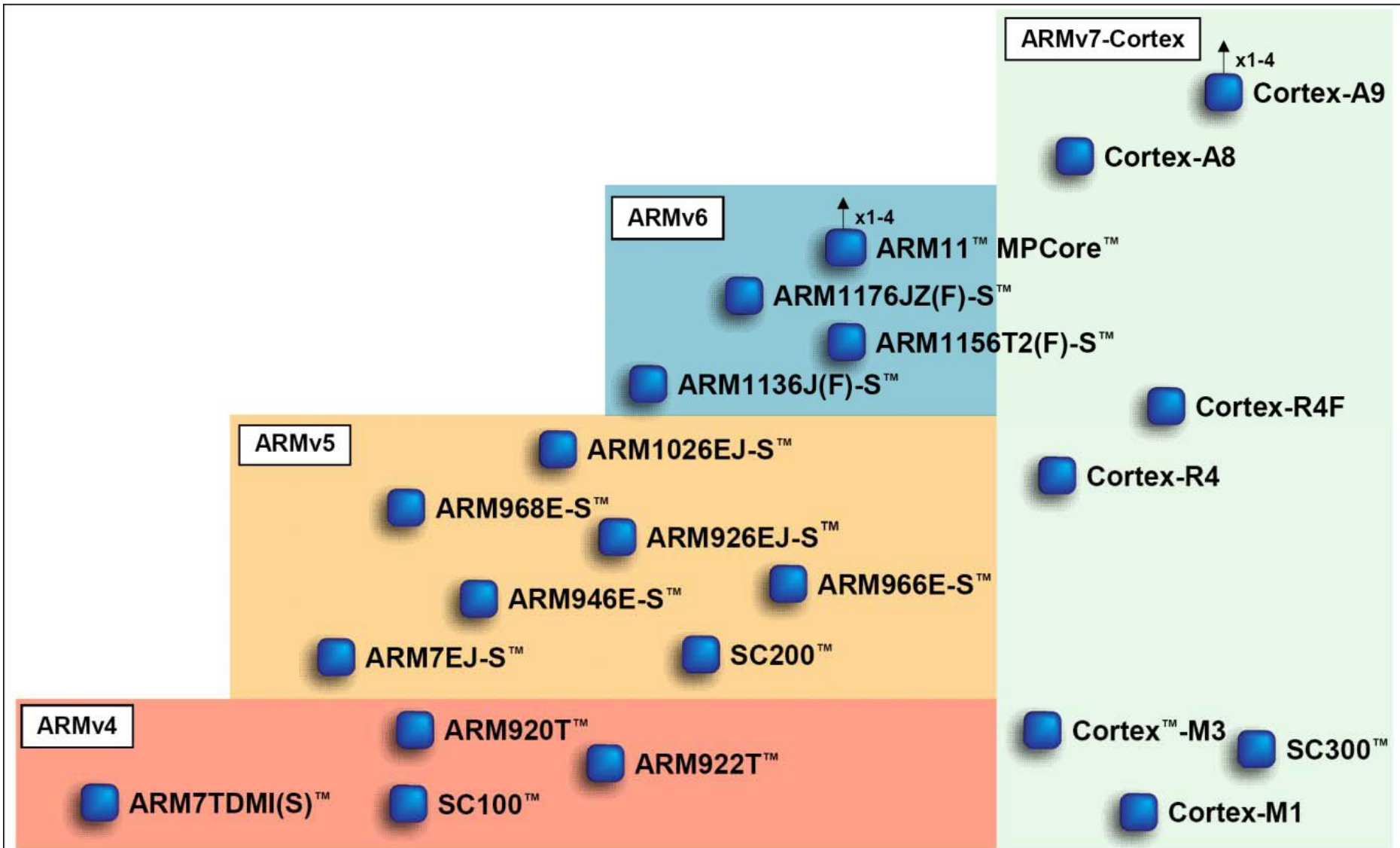
Francesco Menichelli

francesco.menichelli@uniroma1.it

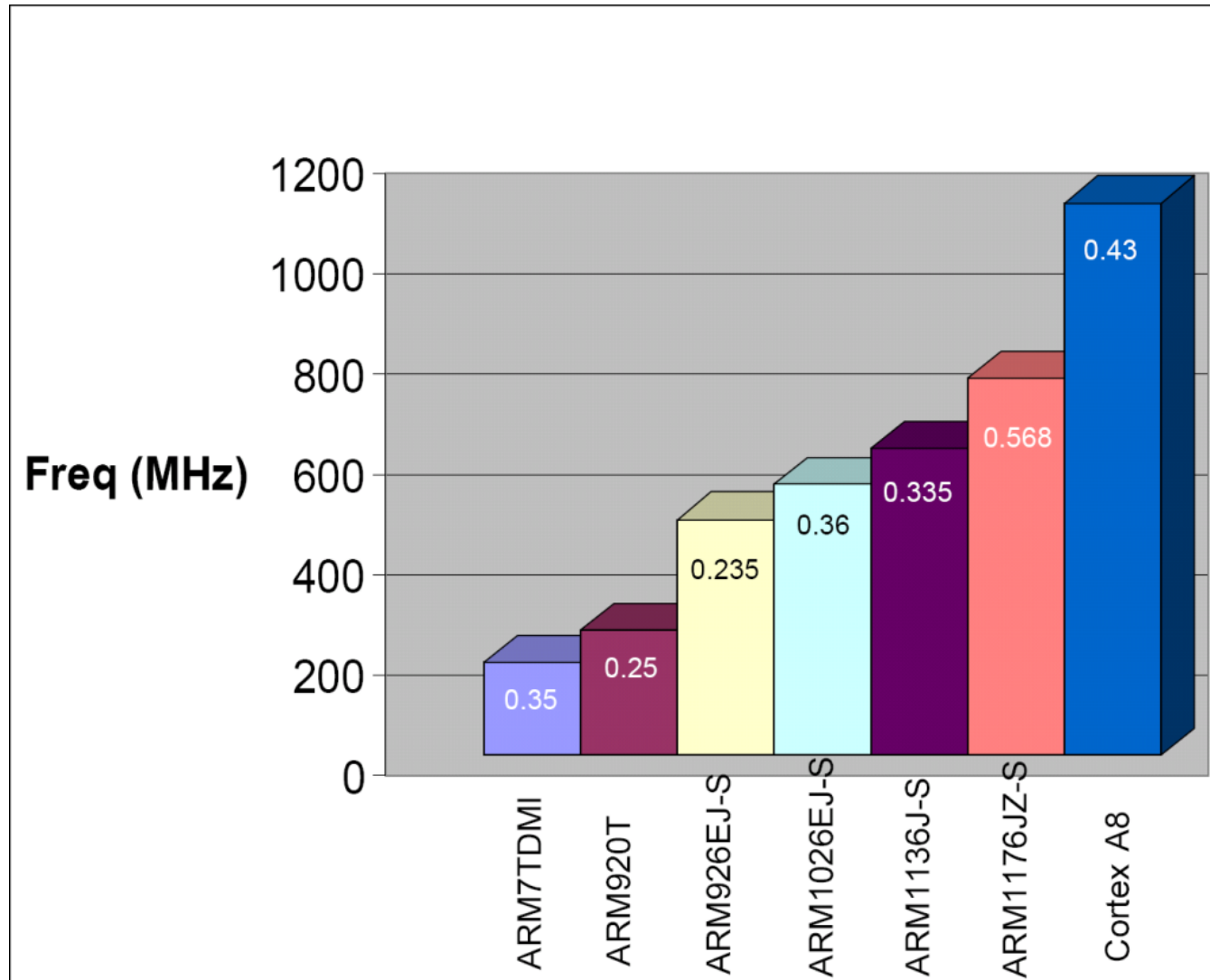
Outline

- **Architecture Overview**
 - Family of cores
 - Pipeline
 - Datapath
 - AMBA Bus
 - Intelligent Energy Manager
- **Instruction Set Architecture**

ARM Architecture Family

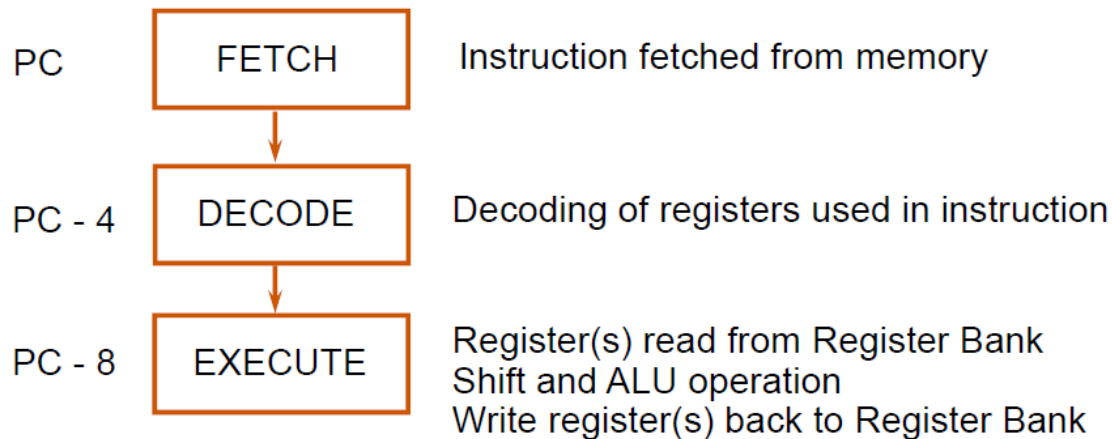


ARM Processor Frequencies (max)



The Original Instruction Pipeline

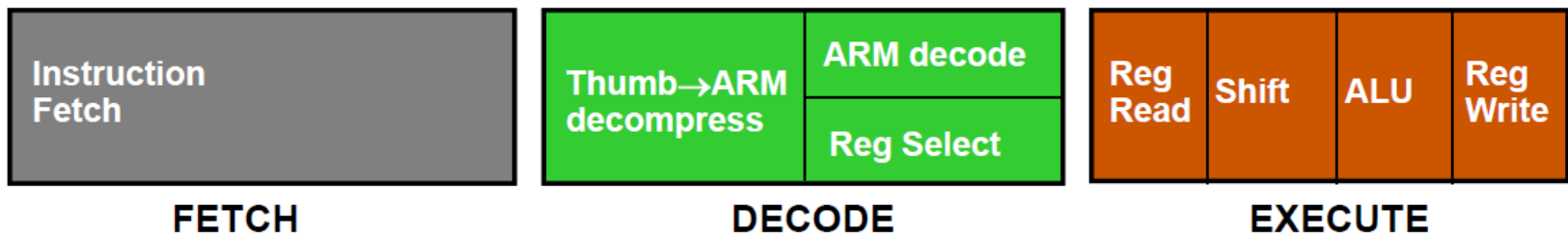
- The ARM uses a pipeline in order to increase the speed of the flow of instructions to the processor.
 - Allows several operations to be undertaken simultaneously, rather than serially.



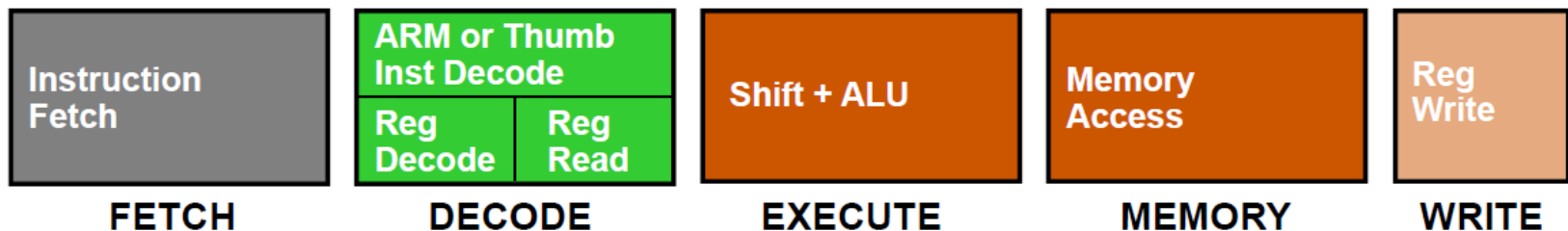
- Rather than pointing to the instruction being executed, the PC points to the instruction being fetched.

Pipeline changes for ARM9TDMI

ARM7TDMI

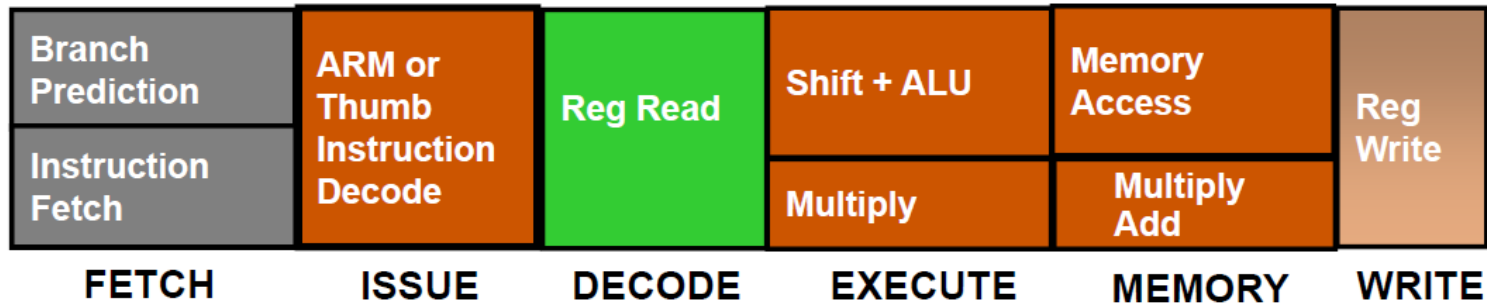


ARM9TDMI

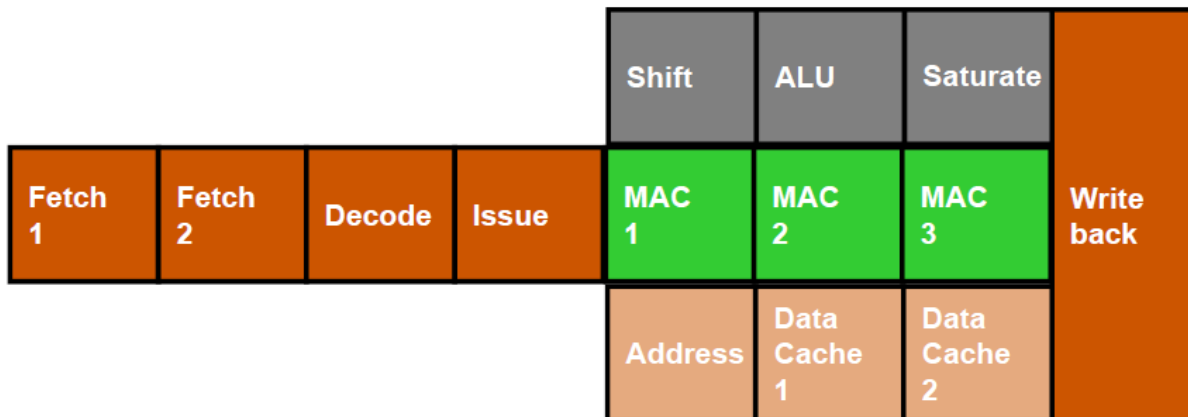


Pipeline changes for ARM10 vs. ARM11 Pipelines

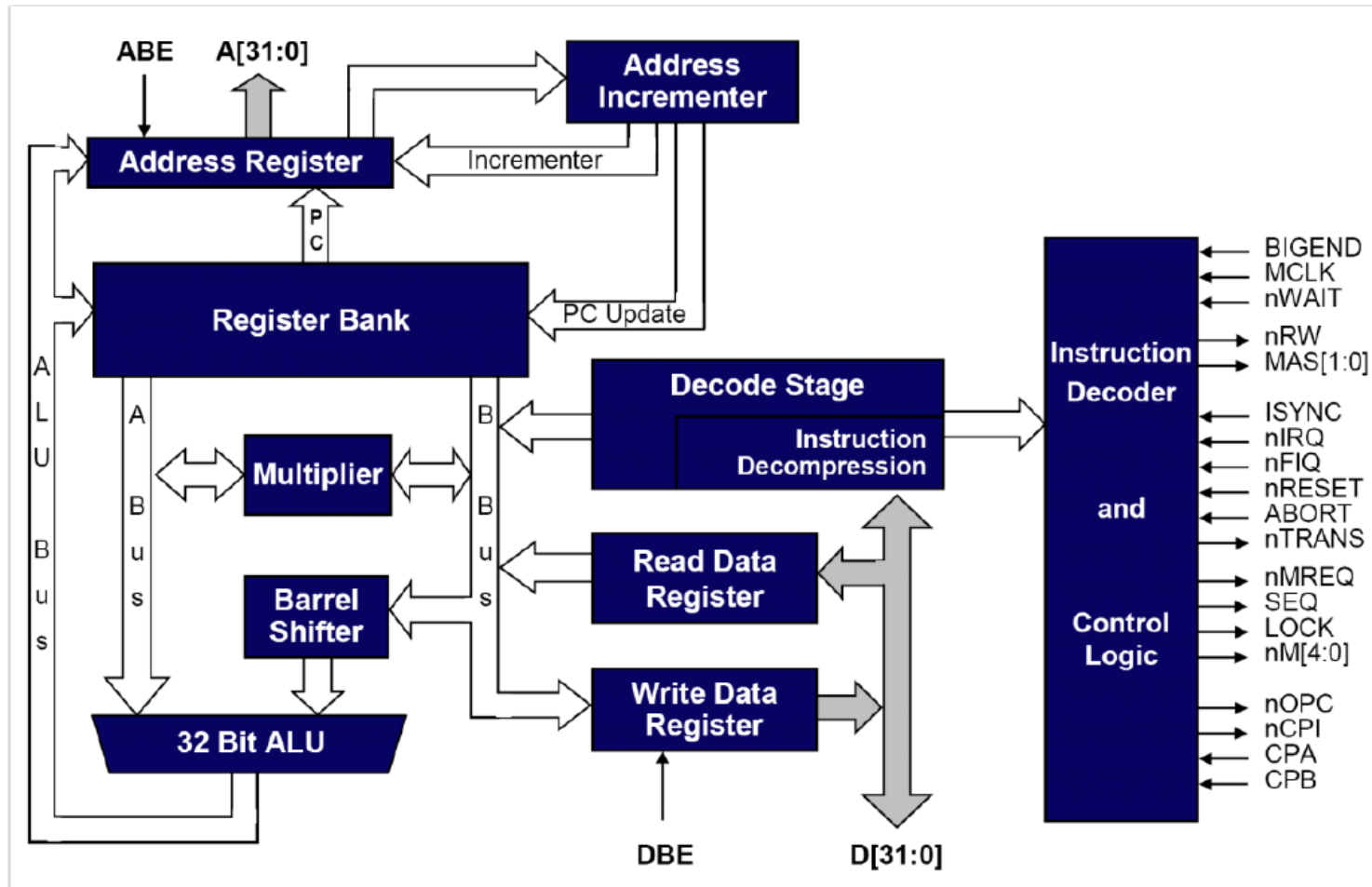
ARM10



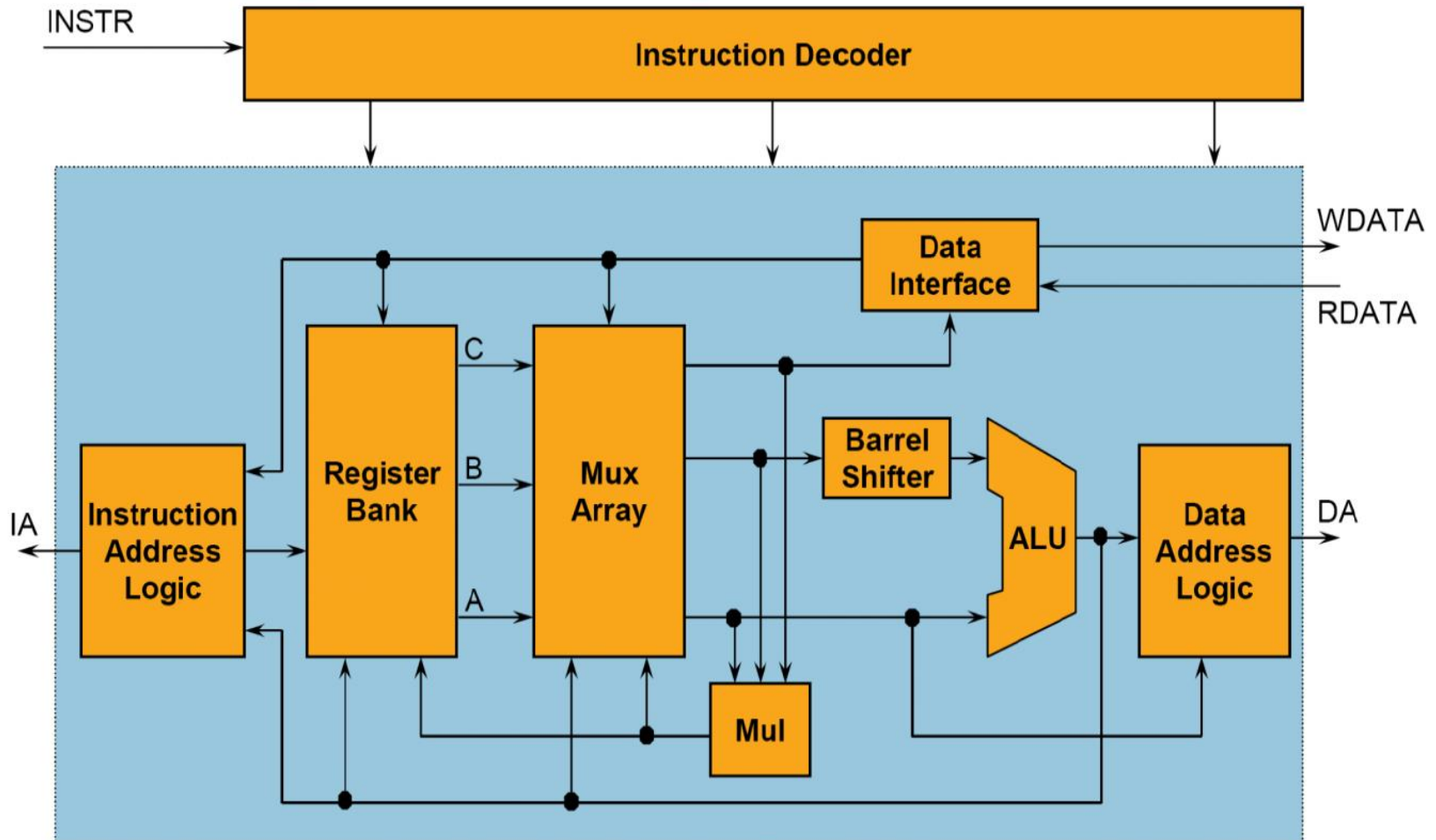
ARM11



ARM 7TDM Core

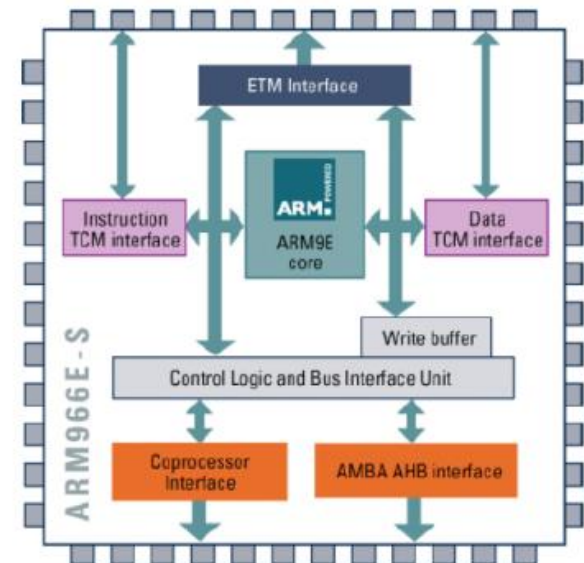
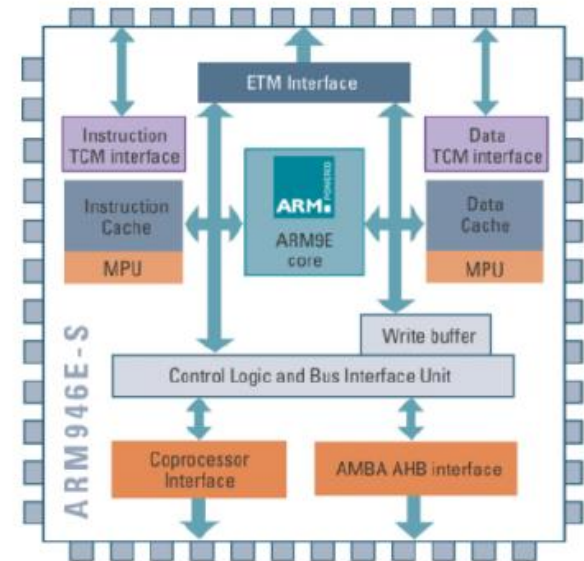


ARM 9TDMI Core



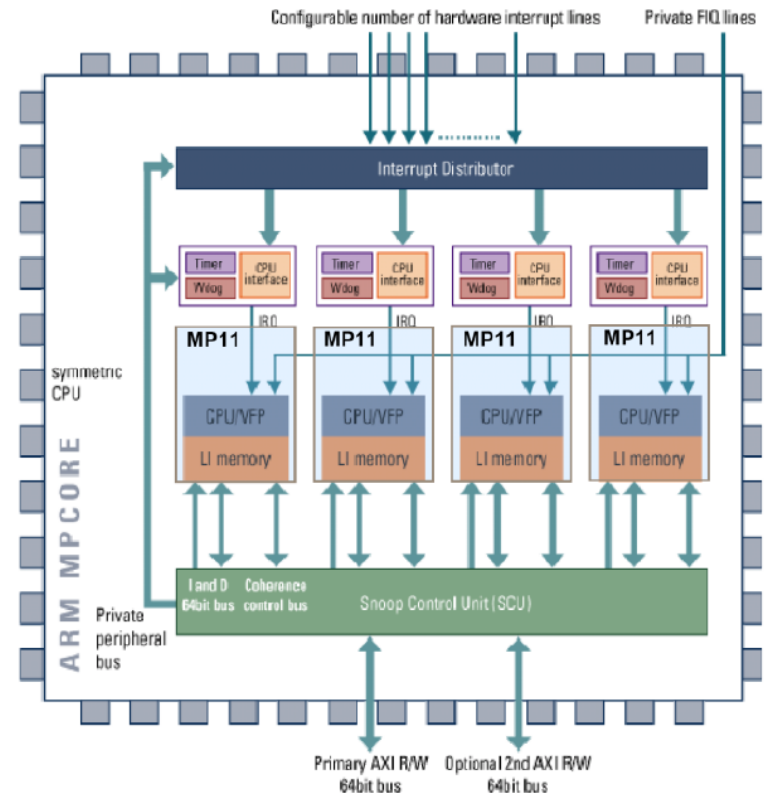
ARM 9E Cores

- **ARM9E is based on the ARM9TDMI core**
 - Core implementation differences
 - Single cycle 32x16 multiplier implementation
 - Embedded-ICE Logic RT
- **ARM926EJ-S / ARM946E-S**
 - Configurable Instruction and Data caches
 - Instruction and Data TCM Interfaces (Tightly-Coupled Memories)
 - AHB bus interface
 - ARM926EJ-S has MMU (Memory Management Unit)
 - ARM946E-S has MPU (Memory Protection Unit)
- **ARM966E-S**
 - Instruction and Data TCM Interfaces
 - No Cache or MPU/MMU



ARM 11 MP-Core

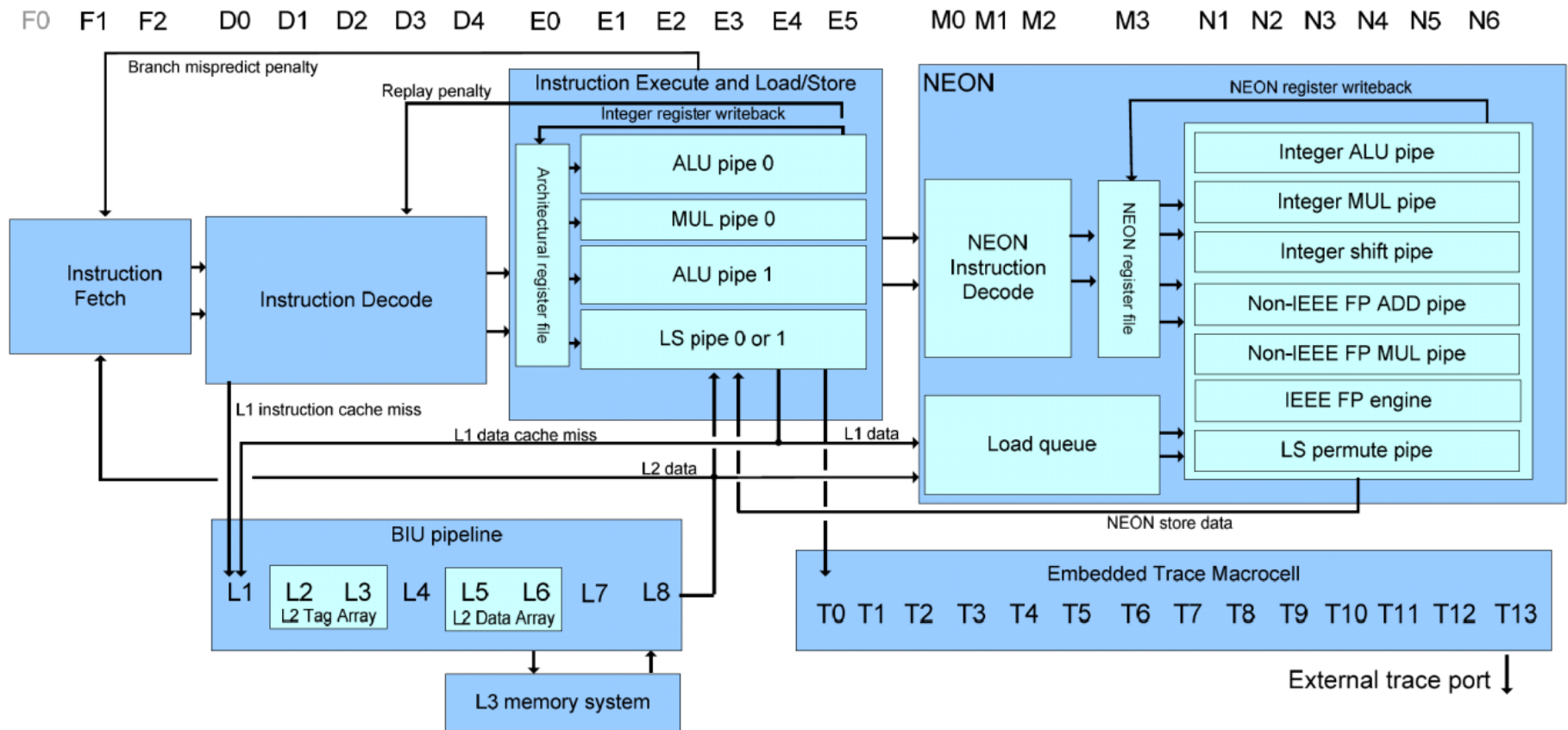
- **Synthesizable**
 - 1 –4 MP11 processors
 - With associated timers & interfaces
 - With or without VFP11 coprocessor
- **ARM v6K compliant**
- **Configurable interrupt inputs**
 - 0 –224 in steps of 32
 - Programmable distribution to MP11s
- **Support for SMP or AMP**
- **MESI-based cache coherency**
- **1 or 2 AXI interfaces to level 2**
- **64-bit data buses**
- **IEM Ready**
- **Program Trace using ETMs**



Cortex A8 Core

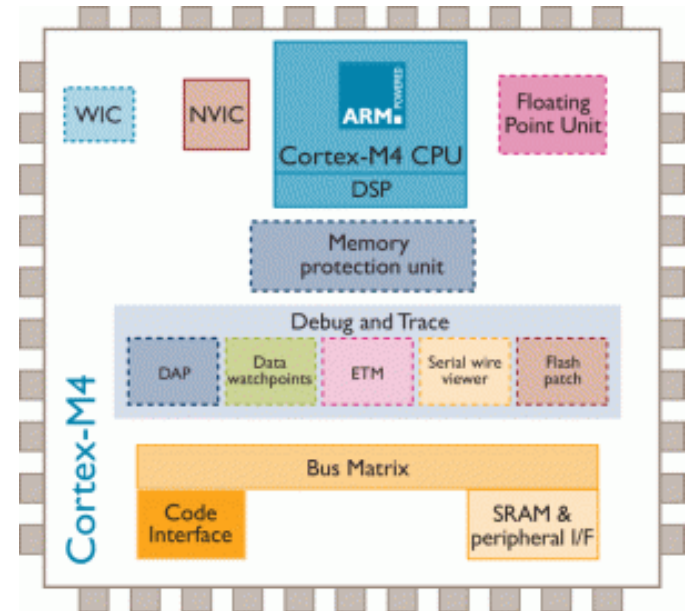
13-Stage Integer Pipeline

10-Stage NEON Pipeline



Cortex M4

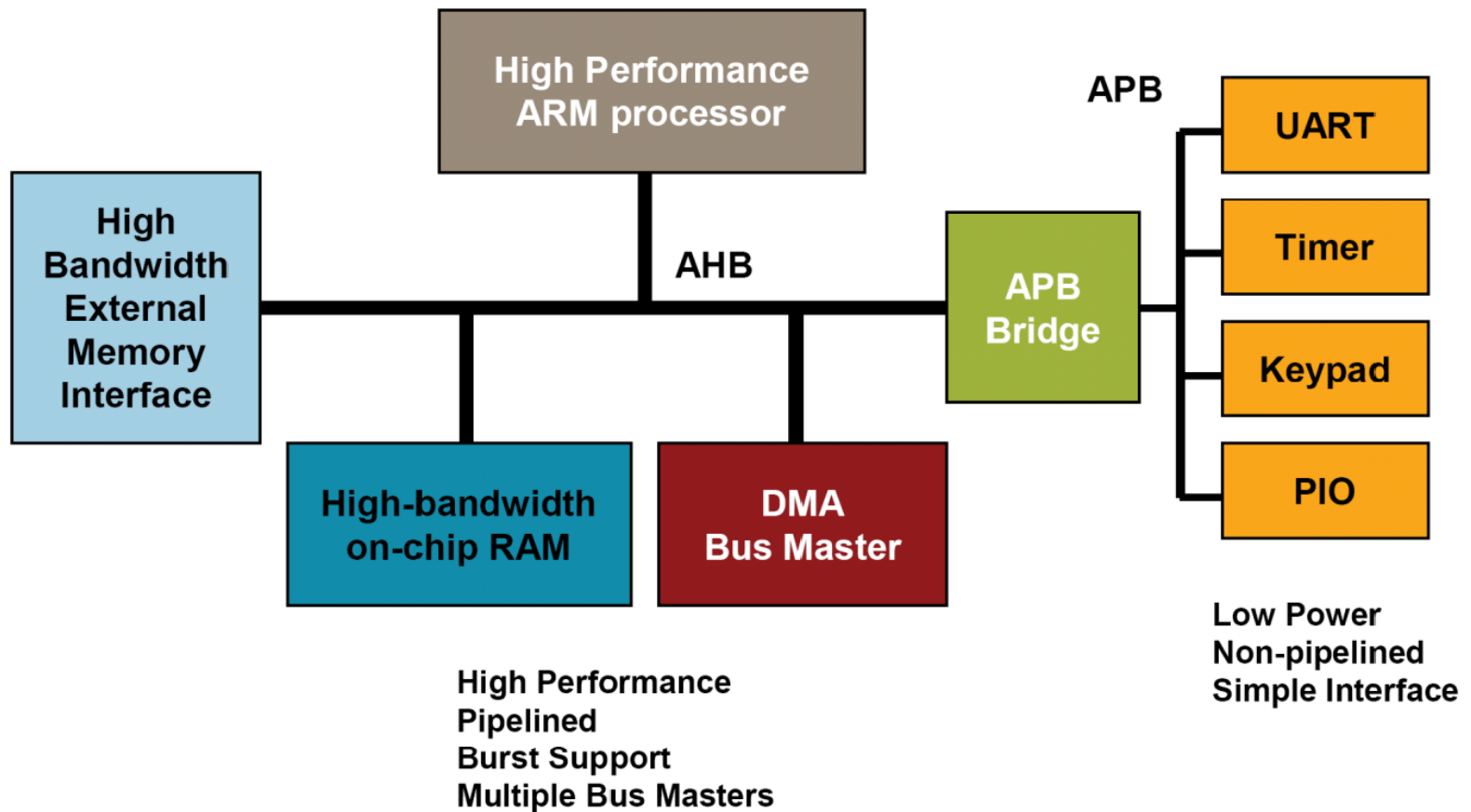
- ISA Support: Thumb® / Thumb-2
- DSP Extensions
 - Single cycle 16/32-bit MAC
 - Single cycle dual 16-bit MAC
 - 8/16-bit SIMD arithmetic
- Hardware Divide (2-12 Cycles)
- Floating Point Unit
- 3-stage pipeline + branch speculation
- Optional 8 region MPU
- Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
- Wake-up Interrupt Controller
- Optional JTAG & Serial-Wire Debug Ports. Up to 8 Breakpoints and 4 Watchpoints.



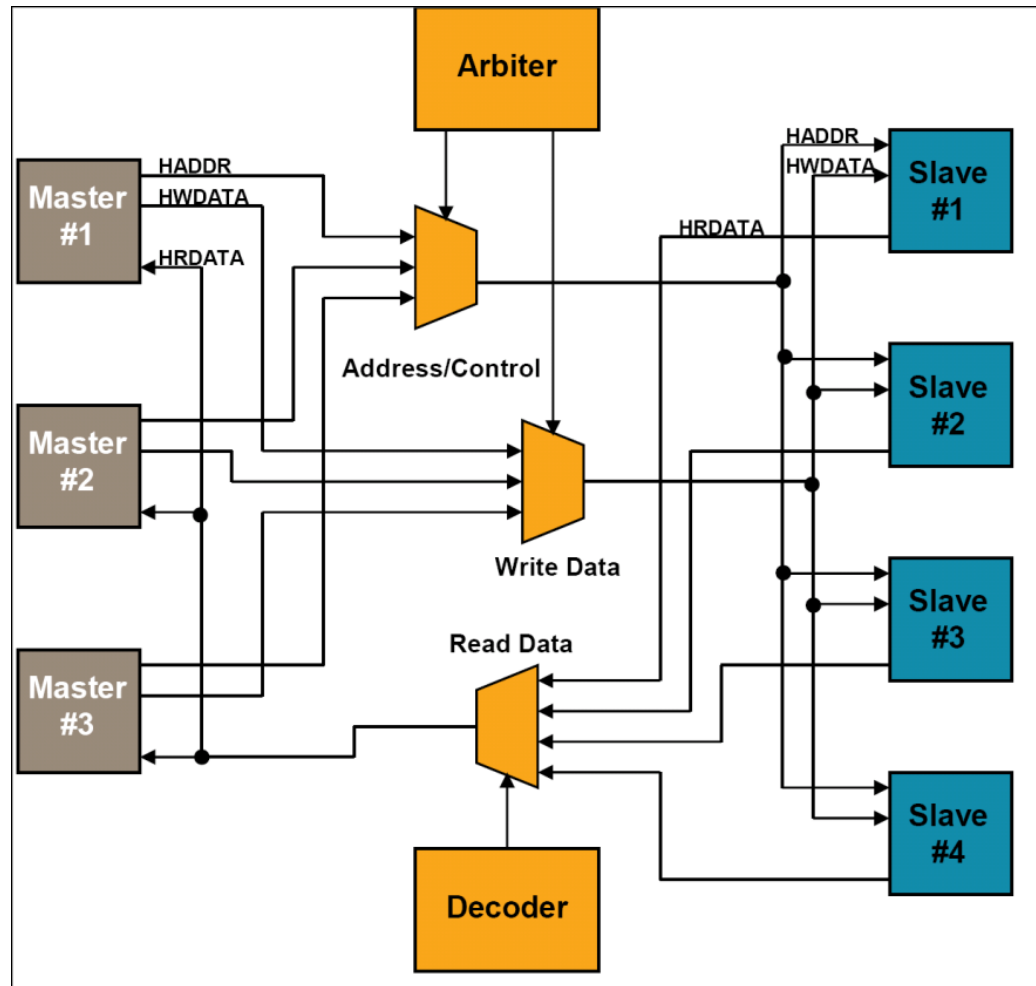
AMBA Introduction

- **Advanced Microcontroller Bus Architecture (AMBA), created by ARM as an interface for their microprocessors.**
 - **AMBA 2.0 released in 1999, includes APB and AHB**
 - **AMBA 3.0 released in 2003, includes AXI**
- **Easy to obtain documentation (free download) and can be used without royalties.**
- **Very common in commercial SoC's(e.g. Qualcomm Multimedia Cell-phone SoC)**

Typical AMBA configuration



AHB Configuration



- No three-state logic

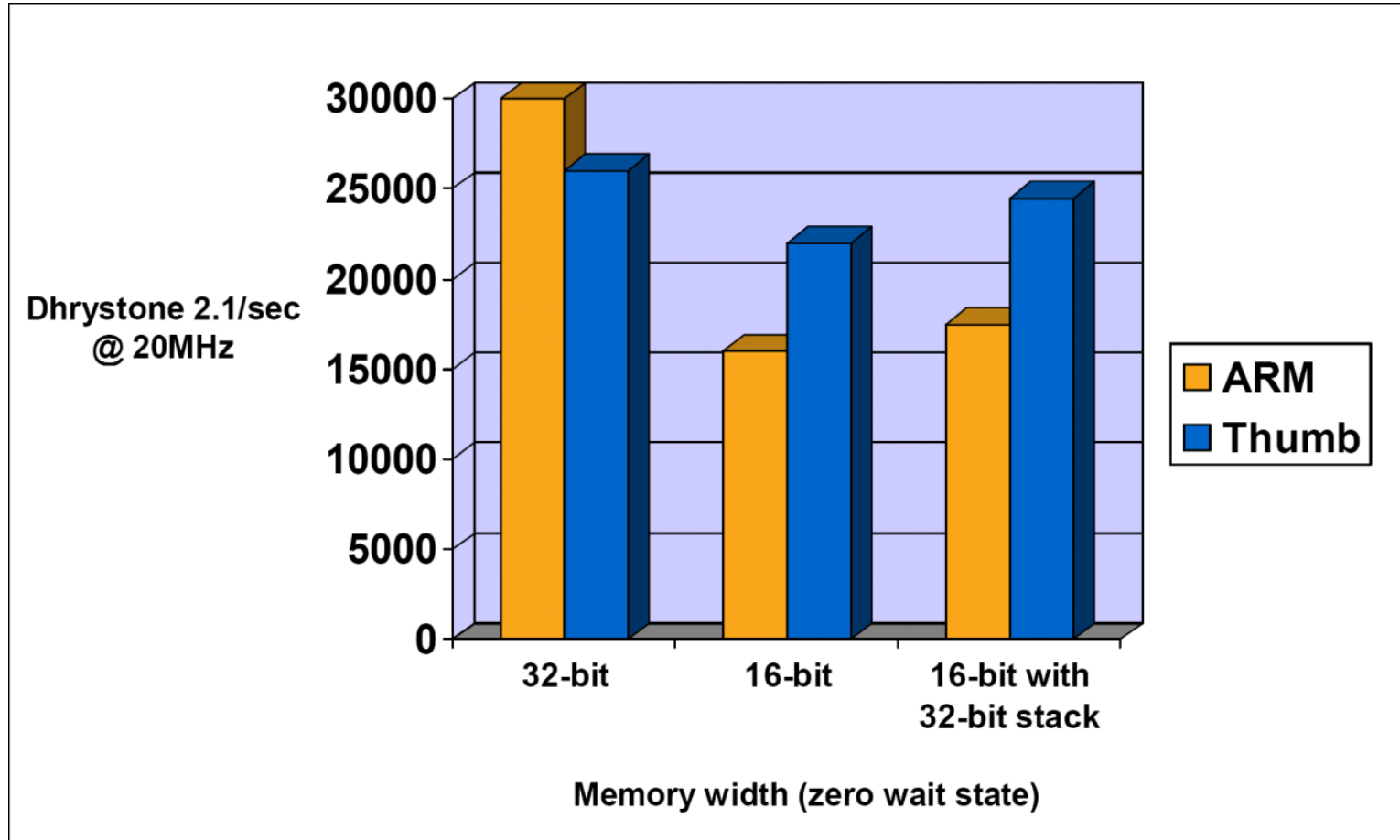
Main features of the ARM Instruction Set

- All instructions are 32 bits long.
- Most instructions execute in a single cycle.
- Most instructions can be conditionally executed.
- A load/store architecture
 - Data processing instructions act only on registers
 - Three operand format
 - Combined ALU and shifter for high speed bit manipulation
 - Specific memory access instructions with powerful auto-indexing addressing modes.
 - 32 bit and 8 bit data types
 - and also 16 bit data types on ARM Architecture v4.
 - Flexible multiple register load and store instructions
- Instruction set extension via coprocessors
- Very dense 16-bit compressed instruction set (Thumb)

Thumb

- **Thumb is a 16-bit instruction set**
 - Optimized for code density from C code
 - Improved performance from narrow memory
 - Subset of the functionality of the ARM instruction set
- **Core has two execution states – ARM and Thumb**
 - Switch between them using BX instruction
- **Thumb has characteristic features**
 - Most Thumb instruction are executed unconditionally
 - Many Thumb data process instruction use a 2-address format
 - Thumb instruction formats are less regular than ARM instruction formats, as a result of the dense encoding.

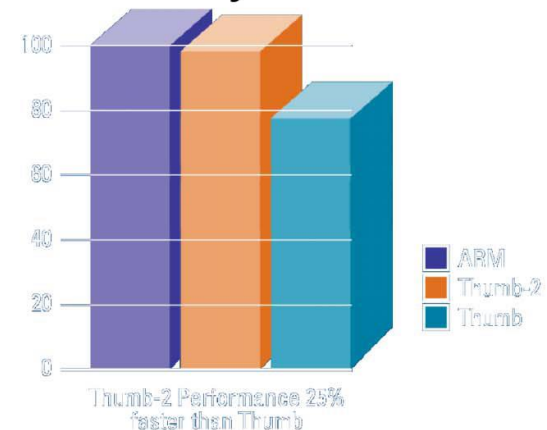
ARM & Thumb Performance Comparison



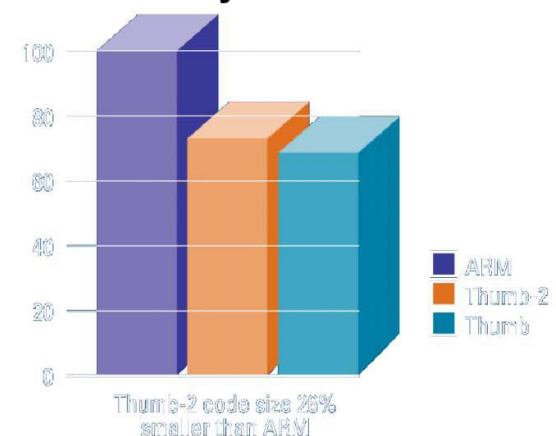
Thumb-2 Instruction Set

- **Second generation of the Thumb architecture**
 - Blended 16-bit and 32-bit instruction set (variable length encoding)
 - 25% faster than Thumb
 - 30% smaller than ARM
- **Increases performance but maintains code density**
- **Maximizes cache and tightly coupled memory usage**

EEMBC Analysis - Performance



EEMBC Analysis – Code Size



The ARM instruction set

Processor Modes

- **The ARM has six operating modes:**
 - User (unprivileged mode under which most tasks run)
 - FIQ (entered when a high priority (fast) interrupt is raised)
 - IRQ (entered when a low priority (normal) interrupt is raised)
 - Supervisor (entered on reset and when a Software Interrupt instruction is executed)
 - Abort (used to handle memory access violations)
 - Undef (used to handle undefined instructions)
- **ARM Architecture Version 4 adds a seventh mode:**
 - System (privileged mode using the same registers as user mode)

The Registers

- **ARM has 37 registers in total, all of which are 32-bits long.**
 - 1 dedicated program counter
 - 1 dedicated current program status register
 - 5 dedicated saved program status registers
 - 30 general purpose registers
- **However these are arranged into several banks, with the accessible bank being governed by the processor mode. Each mode can access**
 - a particular set of r0-r12 registers
 - a particular r13 (the stack pointer) and r14 (link register)
 - r15 (the program counter)
 - cpsr (the current program status register)
- **And privileged modes can also access**
 - a particular spsr (saved program status register)

The ARM Register Set

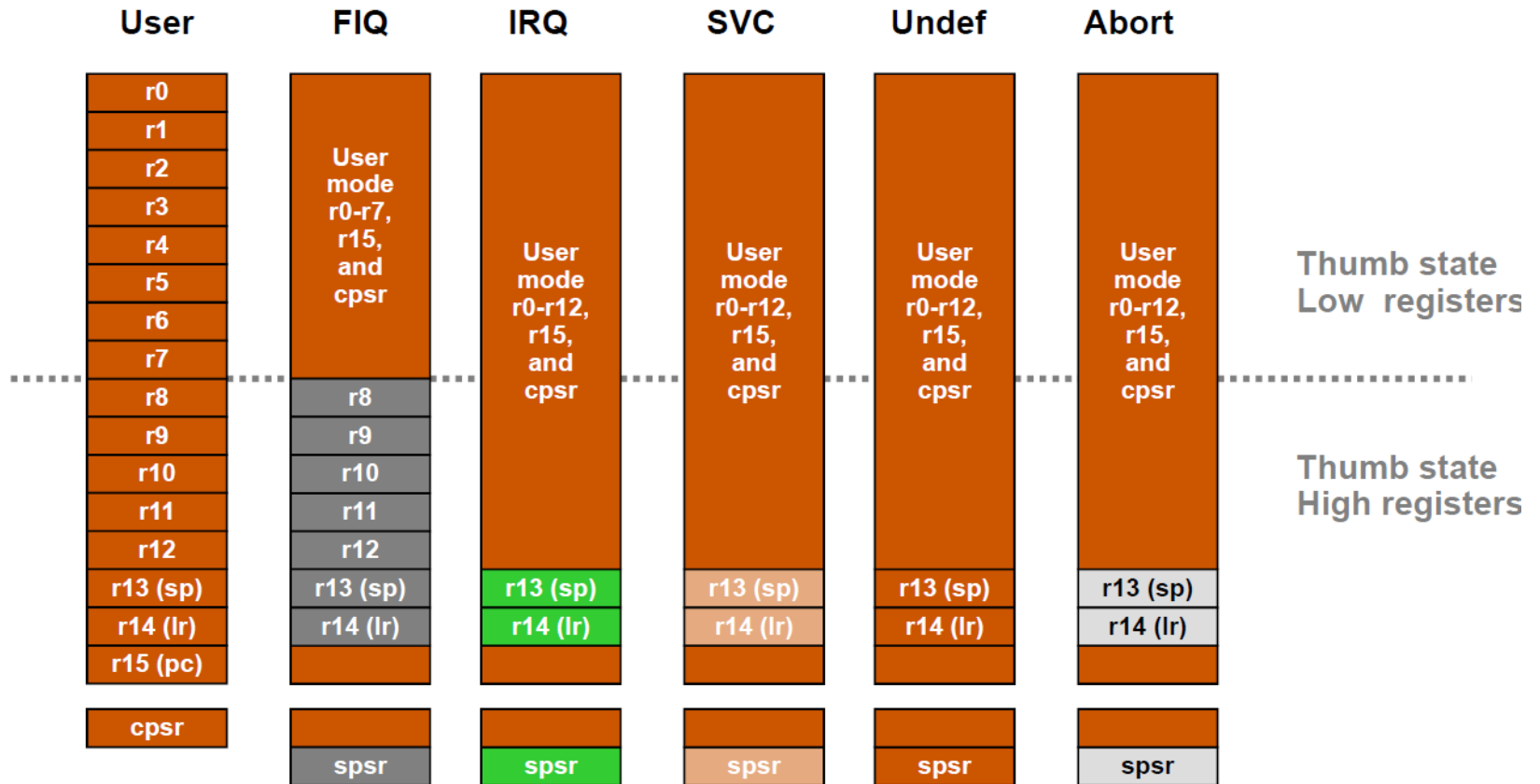
Current Visible Registers

Abort Mode	r0
	r1
	r2
	r3
	r4
	r5
	r6
	r7
	r8
	r9
	r10
	r11
	r12
	r13 (sp)
	r14 (lr)
	r15 (pc)
	cpsr
	spsr

Banked out Registers

User	FIQ	IRQ	SVC	Undef
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

Register Organization Summary

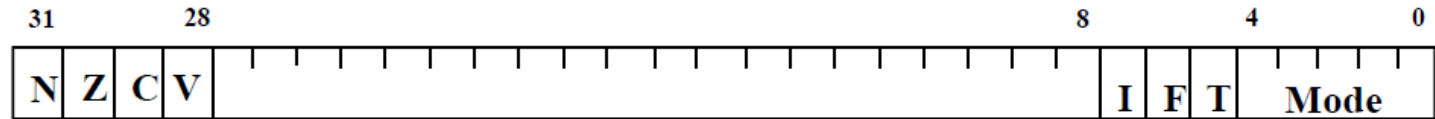


Note: System mode uses the User mode register set

Accessing Registers using ARM Instructions

- **No breakdown of currently accessible registers.**
 - All instructions can access r0-r14 directly.
 - Most instructions also allow use of the PC.
- **Specific instructions to allow access to CPSR and SPSR.**
- **Note : When in a privileged mode, it is also possible to load-store the (banked out) user mode registers to or from memory.**

The Program Status Registers (CPSR and SPSRs)



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

* Condition Code Flags

N = Negative result from ALU flag.

Z = Zero result from ALU flag.

C = ALU operation Carried out

V = ALU operation oVerflowed

* Interrupt Disable bits.

$I = 1$, disables the IRQ.

F = 1, disables the FIQ.

* T Bit (Architecture v4T only)

T = 0, Processor in ARM state

T = 1, Processor in Thumb state

* Mode Bits

M[4:0] define the processor mode.

Condition Flags

	Logical Instruction	Arithmetic Instruction
<u>Flag</u>		
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater than 32 bits
oVerflow (V='1')	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers

The Program Counter (R15) and Link Register (R14)

- When the processor is executing in ARM state:
 - All instructions are 32 bits in length
 - All instructions must be word aligned
 - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).
- R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link operations are performed, calculated from the PC.
- Thus to return from a linked branch:
 MOV r15,r14
 or
 MOV pc,lr

ARM Instruction Set Format

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0	Instruction Type	
Condition		0	0	I	OPCODE				S	Rn			Rs			OPERAND-2												Data processing					
Condition		0	0	0	0	0	0	A	S	Rd			Rn			Rs			1	0	0	1	Rm			Multiply							
Condition		0	0	0	0	1	U	A	S	Rd HIGH			Rd LOW			Rs			1	0	0	1	Rm			Long Multiply							
Condition		0	0	0	1	0	B	0	0	Rn			Rd			0	0	0	0	1	0	0	1	Rm			Swap						
Condition		0	1	I	P	U	B	W	L	Rn			Rd			OFFSET												Load/Store - Byte/Word					
Condition		1	0	0	P	U	B	W	L	Rn			REGISTER LIST															Load/Store Multiple					
Condition		0	0	0	P	U	1	W	L	Rn			Rd			OFFSET 1			1	S	H	1	OFFSET 2			Halfword Transfer Imm Off							
Condition		0	0	0	P	U	0	W	L	Rn			Rd			0	0	0	0	1	S	H	1	Rm			Halfword Transfer Reg Off						
Condition		1	0	1	L	BRANCH OFFSET																					Branch						
Condition		0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn			Branch Exchange			
Condition		1	1	0	P	U	N	W	L	Rn			CRd			CPNum			OFFSET												COPROCESSOR DATA XFER		
Condition		1	1	1	0	Op-1				CRn			CRd			CPNum			OP-2			0	CRm			COPROCESSOR DATA OP							
Condition						OP-1			L	CRn			Rd			CPNum			OP-2			1	CRm			COPROCESSOR REG XFER							
Condition		1	1	1	1	SWI NUMBER																					Software Interrupt						

Conditional Execution

- **Most instruction sets only allow branches to be executed conditionally.**
- **However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.**
 - All instructions contain a condition field which determines whether the CPU will execute them.
 - Non-executed instructions consume 1 cycle.
 - Can't collapse the instruction like a NOP. Still have to complete cycle so as to allow fetching and decoding of the following instructions.
- **This removes the need for many branches, which stall the pipeline (3 cycles to refill).**
 - Allows very dense in-line code, without branches.
 - The Time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

The Condition Field

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				0	0	I	OPCODE		S	Rn		Rs		OPERAND-2										Data processing								

0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear, or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.


Using and updating the Condition Field

- To execute an instruction conditionally, simply postfix it with the appropriate condition:
 - For example an add instruction takes the form:
 - **ADD r0,r1,r2; r0 = r1 + r2 (ADDAL)**
 - To execute this only if the zero flag is set:
 - **ADDEQ r0,r1,r2; If zero flag set then r0 = r1 + r2**
- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
 - For example to add two numbers and set the condition flags:
 - **ADDS r0,r1,r2; r0 = r1 + r2 and set flags**

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
 - This improves code density *and* performance by reducing the number of forward branch instructions.

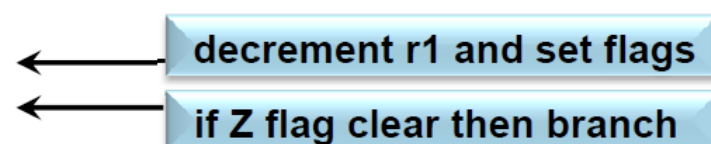
```
CMP    r3,#0
BEQ    skip
ADD    r0,r1,r2
skip
```



```
CMP    r3,#0
ADDNE  r0,r1,r2
```

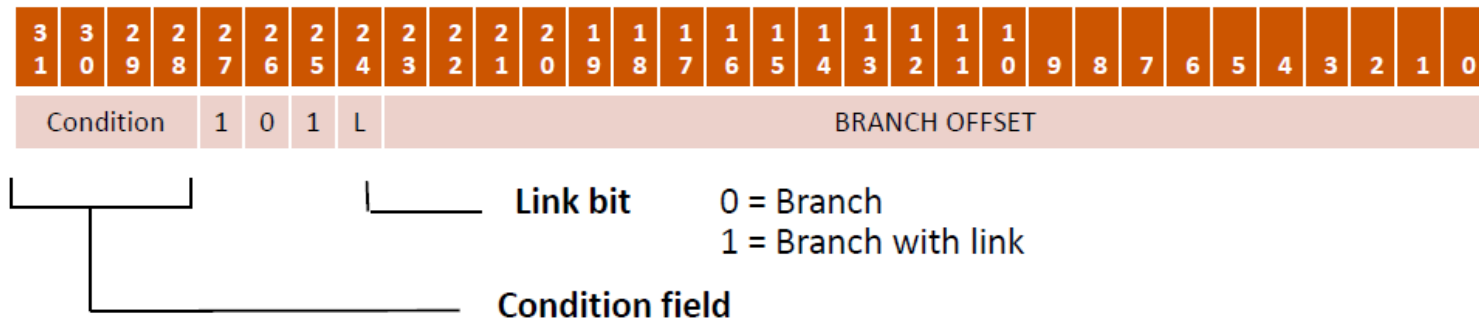
- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S”. CMP does not need “S”.

```
loop
...
SUBS   r1,r1,#1
BNE    loop
```



Branch instructions (1)

- Branch : B{<cond>} label
- Branch with Link :BL{<cond>} b_routine_label



The offset for branch instructions is calculated by the assembler:

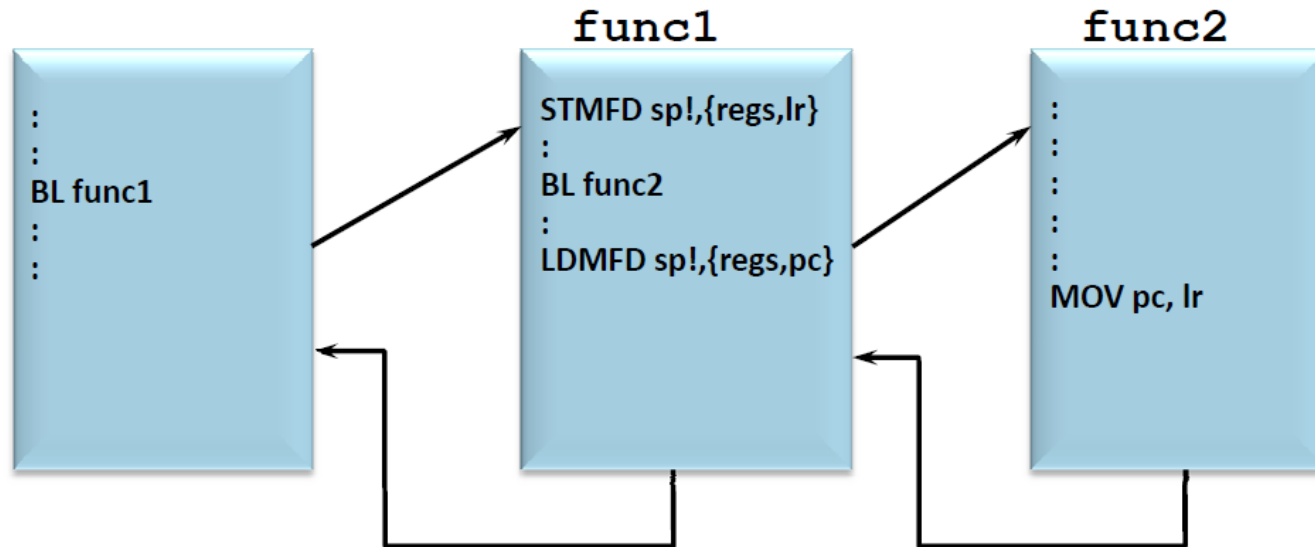
- By taking the difference between the branch instruction and the target address minus 8 (to allow for the pipeline).
- This gives a 26 bit offset which is right shifted 2 bits (as the bottom two bits are always zero as instructions are word –aligned) and stored into the instruction encoding.
- This gives a range of ± 32 Mbytes.

Branch instructions (2)

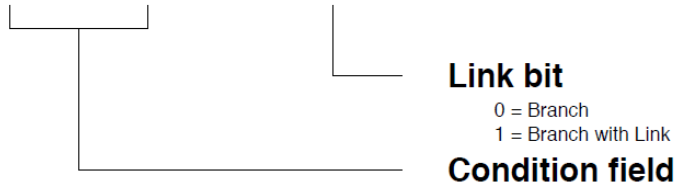
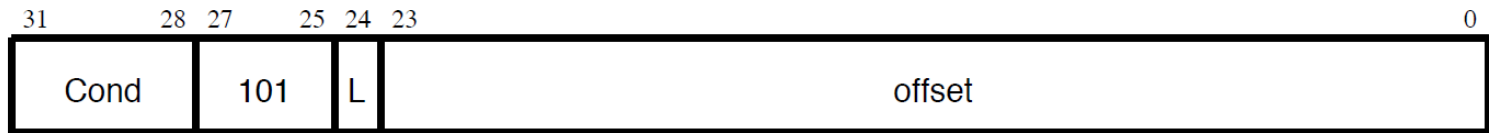
- When executing the instruction, the processor:
 - shifts the offset left two bits, sign extends it to 32 bits, and adds it to PC.
- Execution then continues from the new PC, once the pipeline has been refilled.
- The "Branch with link" instruction implements a subroutine call by writing PC-4 into the LR of the current bank.
 - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).
- To return from subroutine, simply need to restore the PC from the LR:
 - MOV pc, lr
 - Again, pipeline has to refill before execution continues.

Branch instructions (3)

- The "Branch" instruction does not affect LR.
- Note: Architecture 4T offers a further ARM branch instruction, BX
 - See Thumb Instruction Set Module for details.
- BL <subroutine>
 - Stores return address in LR
 - Returning implemented by restoring the PC from LR
 - For non-leaf functions, LR will have to be stacked



Branch instructions format resume



Data processing Instructions

- Largest family of ARM instructions, all sharing the same instruction format.
- Contains:
 - Arithmetic operations
 - Comparisons (no results -just set condition codes)
 - Logical operations
 - Data movement between registers
- Remember, this is a load / store architecture
 - These instruction only work on registers, NOT memory.
- They each perform a specific operation on one or two operands.
 - First operand always a register - Rn
 - Second operand sent to the ALU via barrel shifter.
- We will examine the barrel shifter shortly.

Arithmetic Operations

- Operations are:
 - ADD operand1 + operand2; Add
 - ADC operand1 + operand2 + carry; Add with carry
 - SUB operand1 -operand2; Subtract
 - SBC operand1 -operand2 + carry -1 ; Subtract with carry
 - RSB operand2 -operand1; Reverse subtract
 - RSC operand2 -operand1 + carry -1; Reverse subtract with carry
- Syntax:
 - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Examples
 - **ADD** **r0, r1, r2**
 - **SUBGT** **r3, r3, #1**
 - **RSBLES** **r4, r5, #5**

Comparisons

- The only effect of the comparisons is to update the condition flags. Thus no need to set S bit.
- Operations are:
 - **CMP** operand1 - operand2; Compare
 - **CMN** operand1 + operand2; Compare negative
 - **TST** operand1 AND operand2; Test
 - **TEQ** operand1 EOR operand2; Test equivalence
- Syntax:
 - <Operation>{<cond>} Rn, Operand2
- Examples:
 - **CMP** r0, r1
 - **TSTEQ** r2, #5

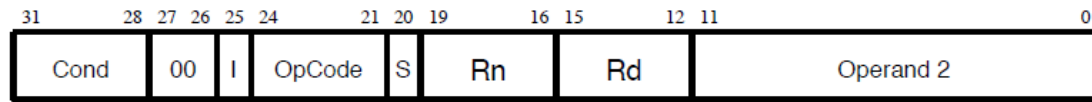
Logical Operations

- Operations are:
 - AND operand1 AND operand2
 - EOR operand1 EOR operand2
 - ORR operand1 OR operand2
 - ORN operand1 NOR operand2
 - BIC operand1 AND NOT operand2 [ie bit clear]
- Syntax:
 - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Examples:
 - **AND** **r0, r1, r2**
 - **BICEQ** **r2, r3, #7**
 - **EORS** **r1,r3,r0**

Data Movement

- Operations are:
 - MOV operand2
 - MVN NOT operand2
- Note that these make no use of operand1.
- Syntax:
 - <Operation>{<cond>}{S} Rd, Operand2
- Examples:
 - **MOV r0, r1**
 - **MOVS r2, #10**
 - **MVN EQr1,#0**

Data processing instruction format resume



Destination register
1st operand register

Set condition codes

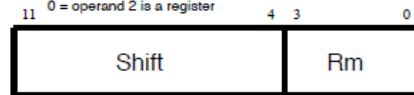
0 = do not alter condition codes
 1 = set condition codes

Operation Code

0000 = AND - Rd := Op1 AND Op2
 0001 = EOR - Rd := Op1 EOR Op2
 0010 = SUB - Rd := Op1 - Op2
 0011 = RSB - Rd := Op2 - Op1
 0100 = ADD - Rd := Op1 + Op2
 0101 = ADC - Rd := Op1 + Op2 + C
 0110 = SBC - Rd := Op1 - Op2 + C - 1
 0111 = RSC - Rd := Op2 - Op1 + C - 1
 1000 = TST - set condition codes on Op1 AND Op2
 1001 = TEQ - set condition codes on Op1 EOR Op2
 1010 = CMP - set condition codes on Op1 - Op2
 1011 = CMN - set condition codes on Op1 + Op2
 1100 = ORR - Rd := Op1 OR Op2
 1101 = MOV - Rd := Op2
 1110 = BIC - Rd := Op1 AND NOT Op2
 1111 = MVN - Rd := NOT Op2

Immediate Operand

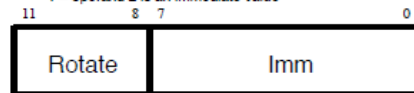
0 = operand 2 is a register



2nd operand register

shift applied to Rm

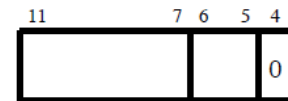
1 = operand 2 is an immediate value



Unsigned 8 bit immediate value

shift applied to Imm

Condition field

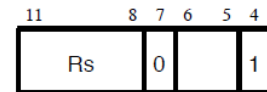


Shift type

00 = logical left
 01 = logical right
 10 = arithmetic right
 11 = rotate right

Shift amount

5 bit unsigned integer



Shift type

00 = logical left
 01 = logical right
 10 = arithmetic right
 11 = rotate right

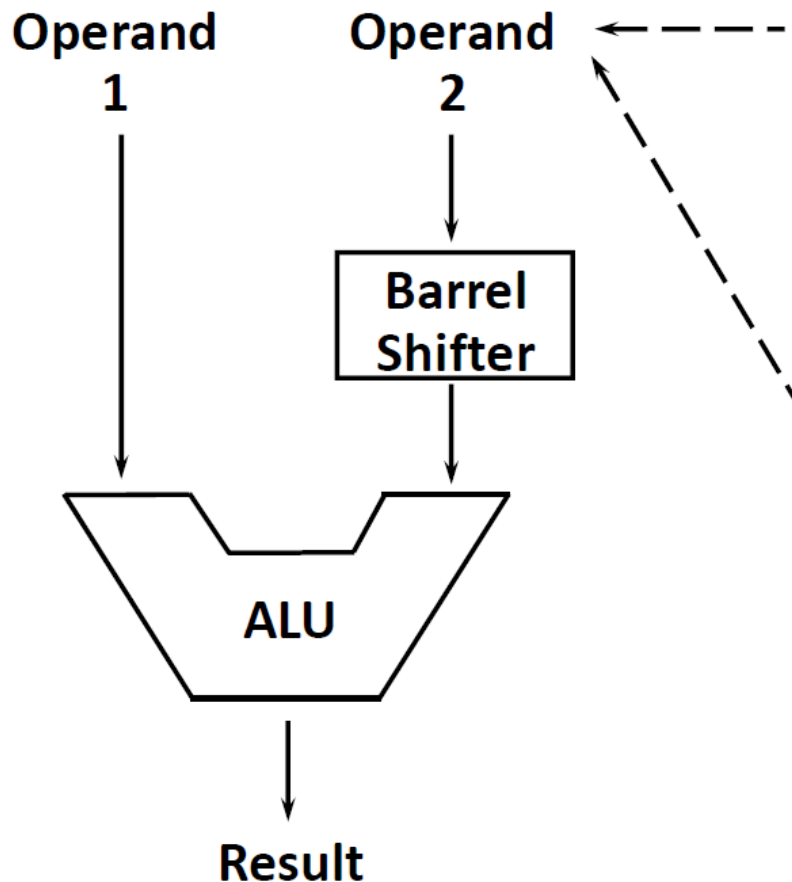
Shift register

Shift amount specified in bottom byte of Rs

The Barrel Shifter

- The ARM doesn't have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.
- So what operations does the barrel shifter support?

Using the Barrel Shifter: The Second Operand



- **Register, optionally with shift operation applied.**
- **Shift value can be either be:**
 - 5 bit unsigned integer
 - Specified in bottom byte of another register.

- * **Immediate value**
 - 8 bit number
 - Can be rotated right through an even number of positions.
 - Assembler will calculate rotate for you from constant.

Barrel Shifter -Left Shift

- Shifts left by the specified amount (multiplies by powers of two) e.g.
 - **LSL #5 => multiply by 32**

Logical Shift Left (LSL)



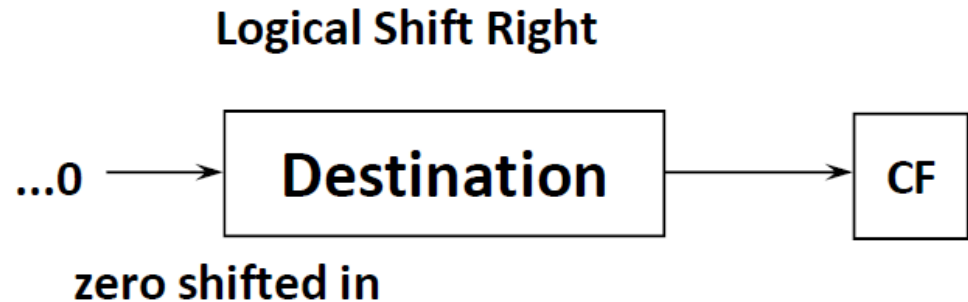
CF: Carry Flag

Barrel Shifter -Right Shifts

Logical Shift Right (LSR)

Shifts right by the specified amount (divides by powers of two) e.g.

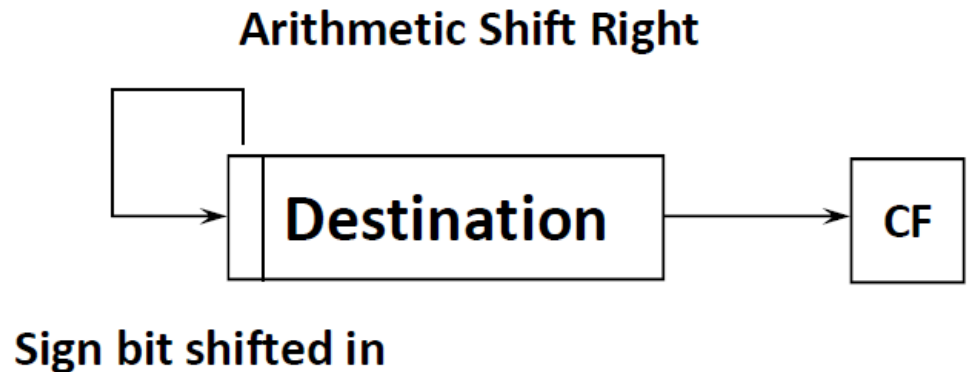
LSR #5 = divide by 32



Arithmetic Shift Right (ASR)

Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g.

ASR #5 = divide by 32



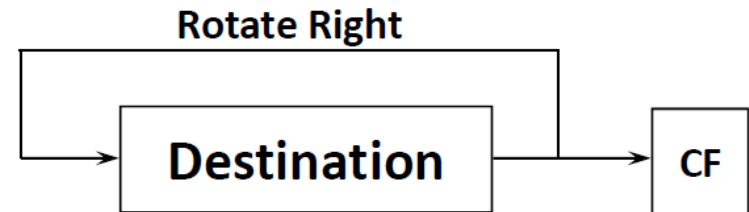
Barrel Shifter -Rotations

Rotate Right (ROR)

Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

e.g. ROR #5

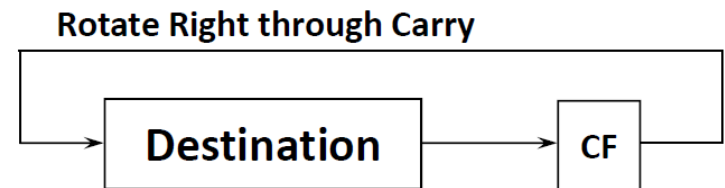
Note the last bit rotated is also used as the Carry Out.



Rotate Right Extended (RRX)

This operation uses the CPSR C flag as a 33rd bit.

Rotates right by 1 bit. Encoded as
RRX #0



Second Operand : Shifted Register

- The amount by which the register is to be shifted is contained in either:
 - the immediate 5-bit field in the instruction
 - NO OVERHEAD
 - Shift is done for free -executes in single cycle.
 - the bottom byte of a register (not PC)
 - Then takes extra cycle to execute
 - ARM doesn't have enough read ports to read 3 registers at once.
 - Then same as on other processors where shift is separate instruction.
- If no shift is specified then a default shift is applied: LSL #0
 - i.e. barrel shifter has no effect on value in register.

Second Operand: Using a Shifted Register

- Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.
- A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.
 - Multiplications by a constant equal to a $((\text{power of } 2) \pm 1)$ can be done in one cycle.

MOV R2, R0, LSL #2	; Shift R0 left by 2, write to R2, (R2=R0x4)
ADD R9, R5, R5, LSL #3	; R9 = R5 + R5 x 8 or R9 = R5 x 9
RSB R9, R5, R5, LSL #3	; R9 = R5 x 8 - R5 or R9 = R5 x 7
SUB R10, R9, R8, LSR #4	; R10 = R9 - R8 / 16
MOV R12, R4, ROR R3	; R12 = R4 rotated right by value of R3

Second Operand: Immediate Value (1)

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

Second Operand: Immediate Value (2)

- This gives us:
 - 0 -255 [0 -0xff]
 - 256,260,264,...,1020 [0x100-0x3fc, step 4, 0x40-0xff ror30]
 - 1024,1040,1056,...,4080 [0x400-0xff0, step 16, 0x40-0xff ror28]
 - 4096,4160, 4224,...,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror26]
- These can be loaded using, for example:
 - **MOV r0, #0x40, 26; => MOV r0, #0x1000 (ie 4096)**
- To make this easier, the assembler will convert to this form for us if simply given the required constant:
 - **MOV r0, #4096; => MOV r0, #0x1000 (ie 0x40 ror26)**
- The bitwise complements can also be formed using MVN:
 - **MOV r0, #0xFFFFFFFF ; assembles to MVN r0, #0**
- If the required constant cannot be generated, an error will be reported.

Load / Store Instructions

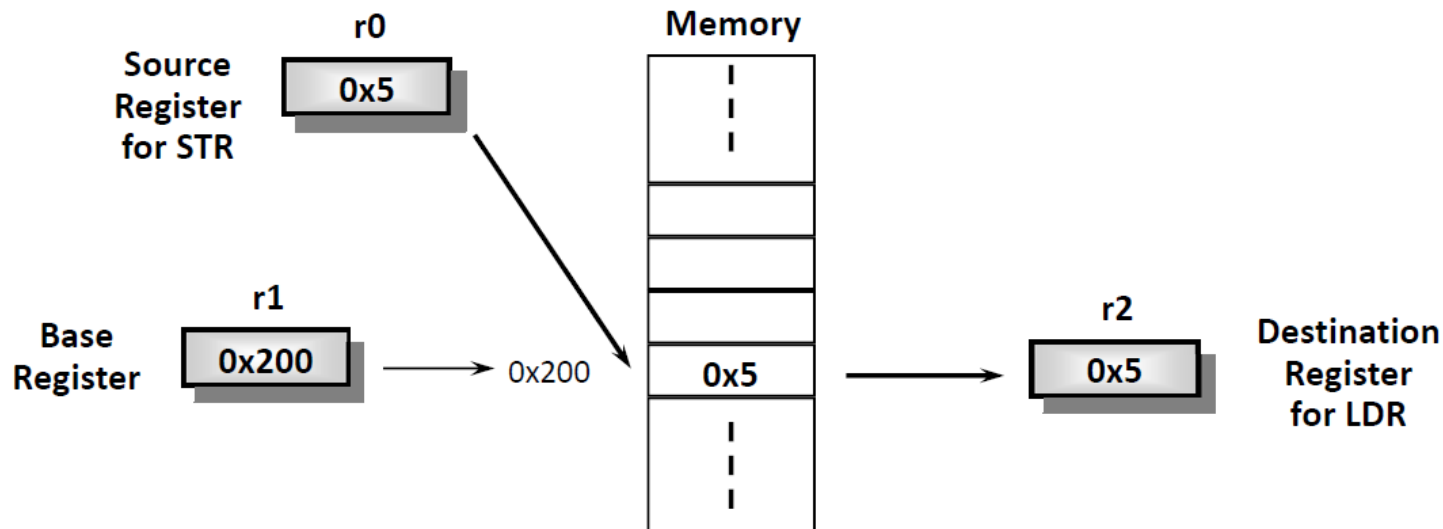
- The ARM is a Load / Store Architecture:
 - Does not support memory to memory data processing operations.
 - Must move data values into registers before using them.
- This might sound inefficient, but in practice it isn't:
 - Load data values from memory into registers.
 - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
 - Store results from registers out to memory.
- The ARM has three sets of instructions which interact with main memory. These are:
 - Single register data transfer (LDR / STR).
 - Block data transfer (LDM/STM).
 - Single Data Swap (SWP).

Single register data transfer

- The basic load and store instructions are:
 - Load and Store Word or Byte
 - LDR / STR / LDRB / STRB
- ARM Architecture Version 4 also adds support for Halfwords and signed data.
 - Load and Store Halfword
 - LDRH / STRH
 - Load Signed Byte or Halfword -load value and sign extend it to 32 bits.
 - LDRSB / LDRSH
- All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR
 - .e.g. LDREQB
- Syntax:
 - <LDR|STR>{<cond>}{<size>} Rd, <address>

Load and Store Word or Byte: Base Register

- The memory location to be accessed is held in a base register
STR r0, [r1] ; Store contents of r0 to location pointed to
; by contents of r1.
LDR r2, [r1] ; Load r2 with contents of memory location
; pointed to by contents of r1.

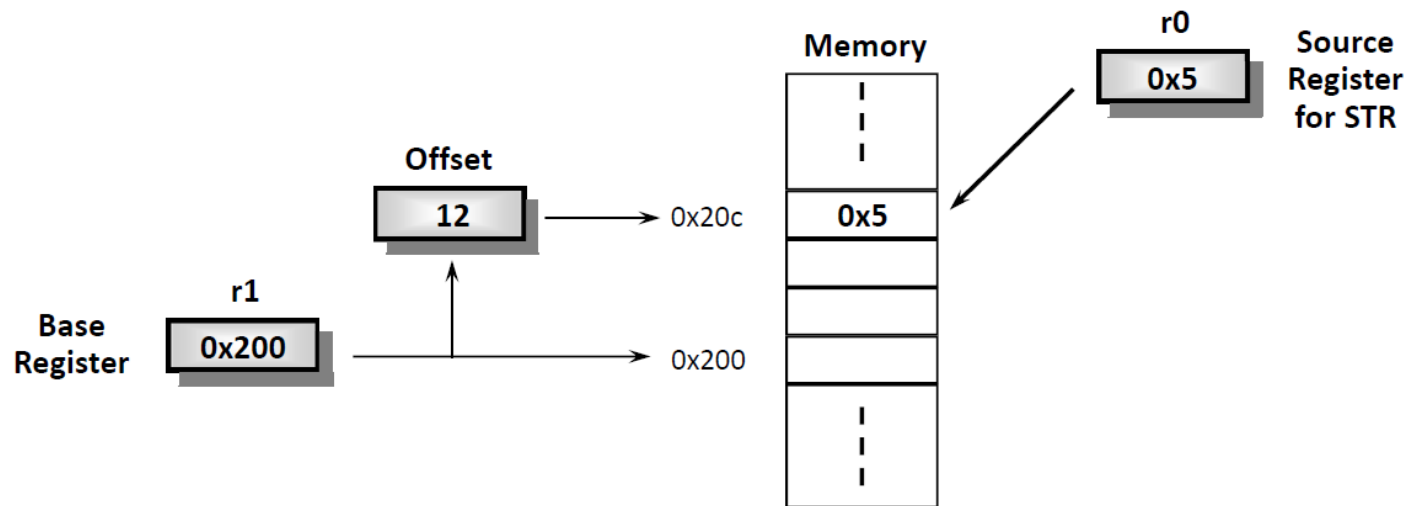


Load/Store Word or Byte: Offsets from the Base Register

- As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.
- This offset can be
 - An **unsigned 12bit immediate** value (i.e. 0 -4095 bytes).
 - A **register**, optionally shifted by an immediate value
- This can be either added or subtracted from the base register:
 - Prefix the offset value or register with '+' (default) or '-'.
- This offset can be applied:
 - before the transfer is made: **Pre-indexed** addressing
 - optionally auto-incrementing the base register, by postfixing the instruction with an '!'.
 - after the transfer is made: **Post-indexed** addressing
 - causing the base register to be auto-incremented.

Load/Store Word or Byte: Pre-indexed Addressing

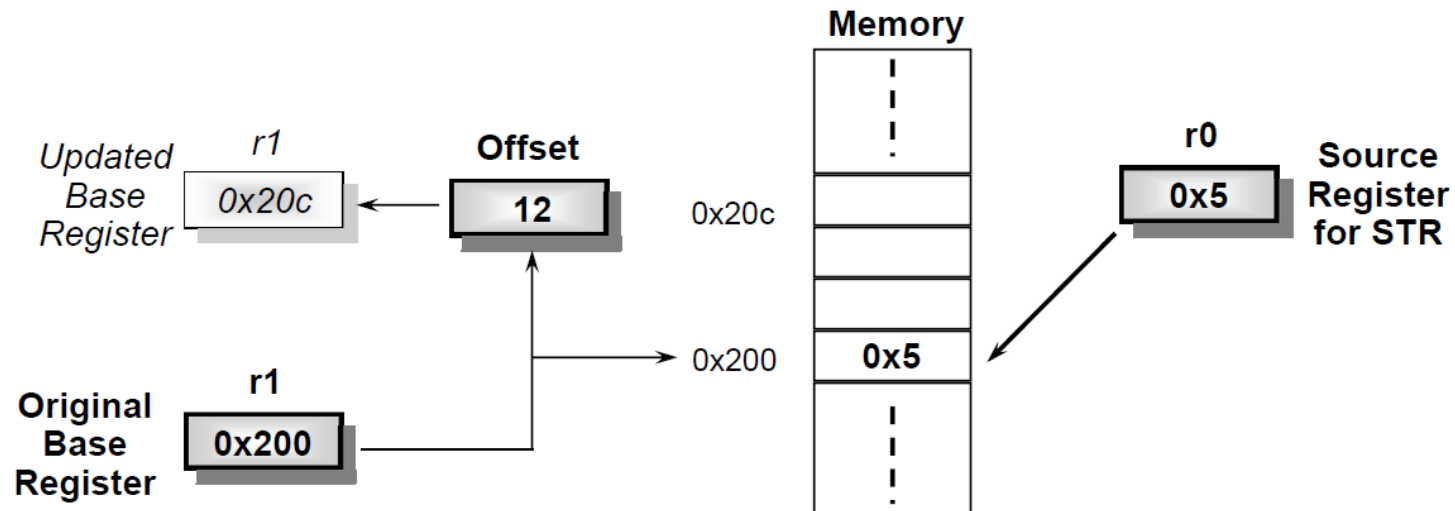
- Example: STR r0, [r1,#12]



- To store to location `0x1f4` instead use:
 - `STR r0, [r1,#-12]`
- To auto-increment base pointer to `0x20c` use:
 - `STR r0, [r1, #12]!`
- If `r2` contains 3, access `0x20c` by multiplying this by 4:
 - `STR r0, [r1, r2, LSL #2]`

Load and Store Word or Byte: Post-indexed Addressing

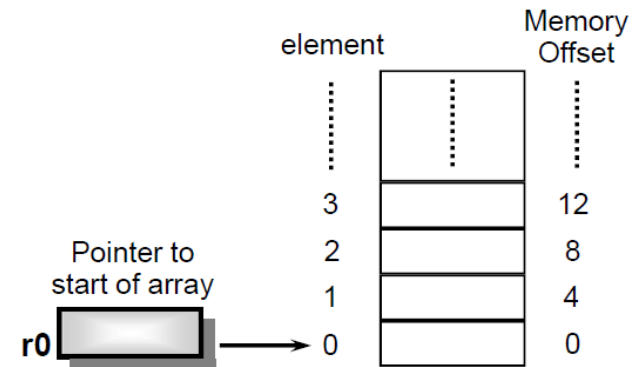
- Example: `STR r0, [r1], #12`



- To auto-increment the base register to location `0x1f4` instead use:
 - `STR r0, [r1], #-12`
- If `r2` contains 3, auto-increment base register to `0x20c` by multiplying this by 4:
 - `STR r0, [r1], r2, LSL #2`

Example Usage of Addressing Modes

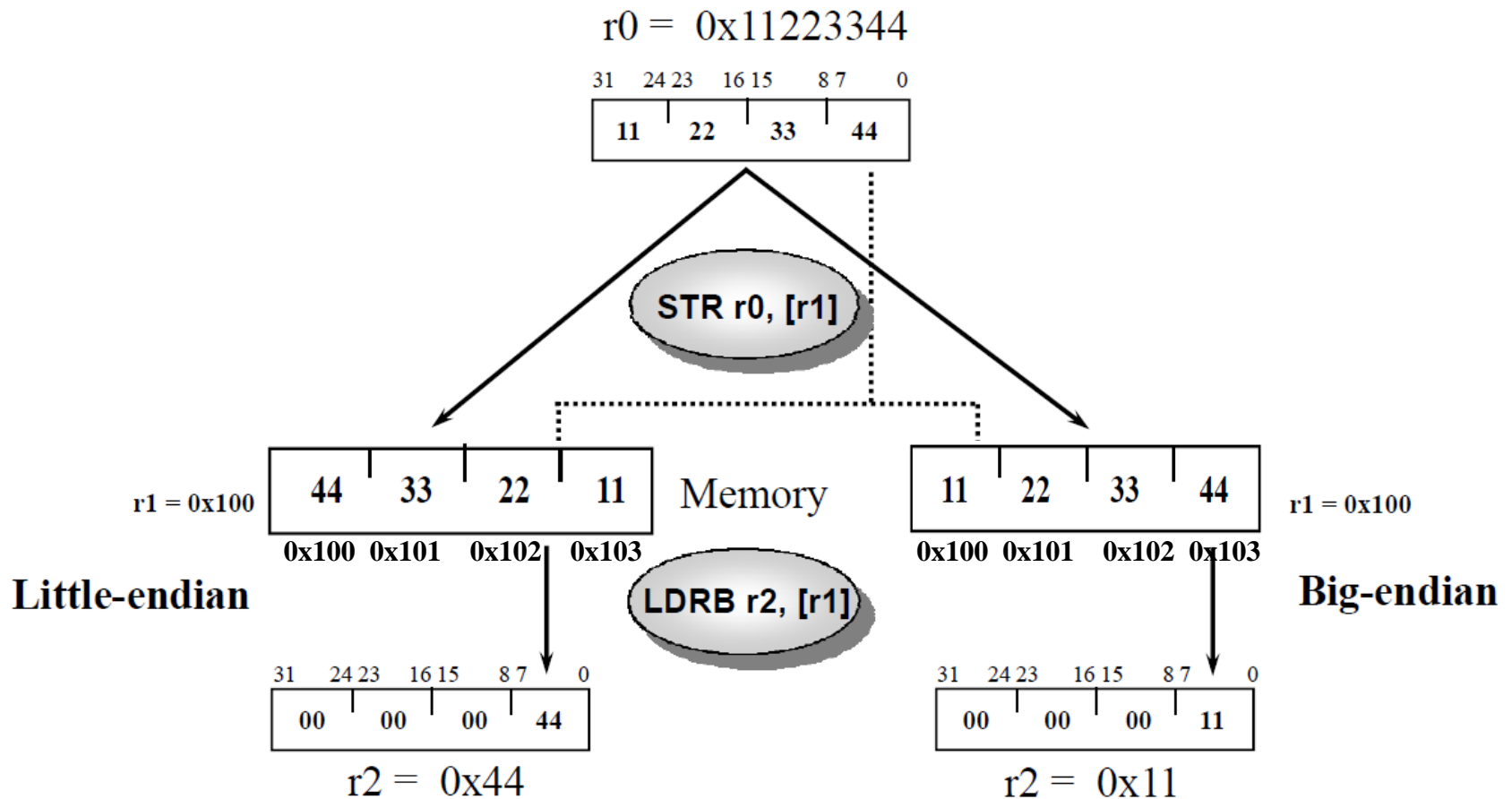
- Imagine an array, the first element of which is pointed to by the contents of r0.
- If we want to access a particular element, then we can use pre-indexed addressing:
 - r1 is element we want.
 - **LDR r2, [r0, r1, LSL #2]**
- If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:
 - r1 is address of current element (initially equal to r0).
 - **LDR r2, [r1], #4**
- Use a further register to store the address of final element, so that the loop can be correctly terminated.



Effect of endianness

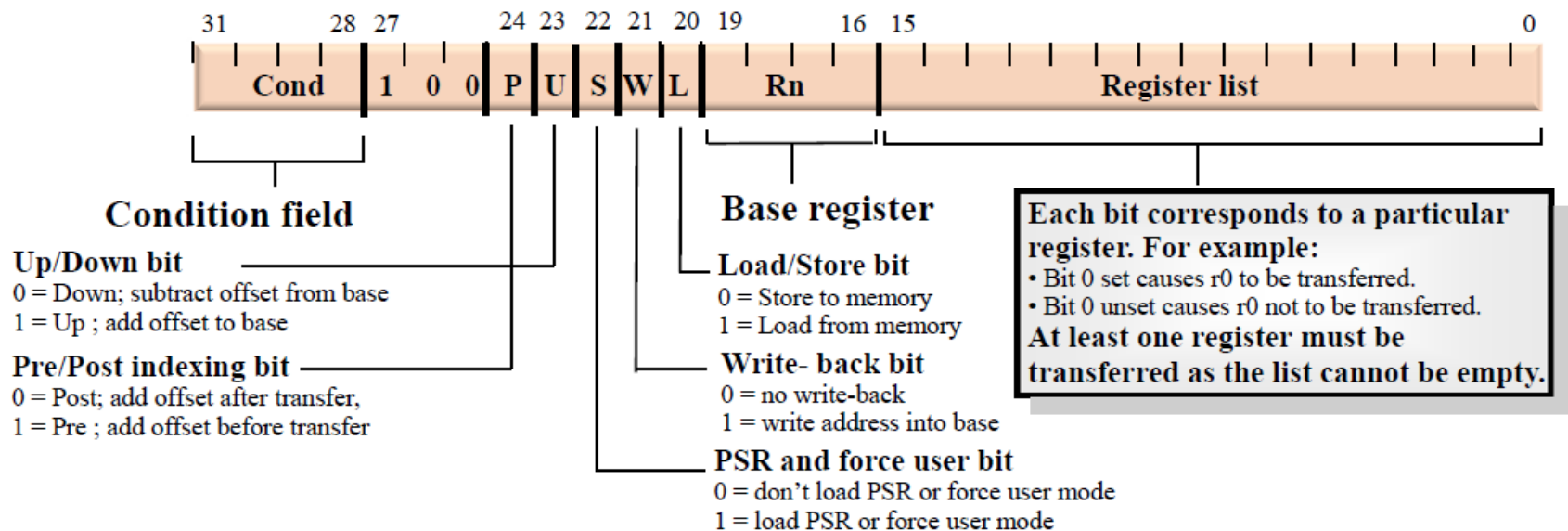
- The ARM can be set up to access its data in either little or big endian format.
- Little endian:
 - Least significant byte of a word is stored in bits 0-7 of an addressed word.
- Big endian:
 - Least significant byte of a word is stored in bits 24-31 of an addressed word.
- This has no real relevance unless data is stored as words and then accessed in smaller sized quantities (halfwords or bytes).
 - Which byte / halfword is accessed will depend on the endianness of the system involved.

Endianess Example



Block Data Transfer (1)

- The Load and Store Multiple instructions (LDM / STM) allow between 1 and 16 registers to be transferred to or from memory.
- The transferred registers can be either:
 - Any subset of the current bank of registers (default).
 - Any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a '^').



Block Data Transfer (2)

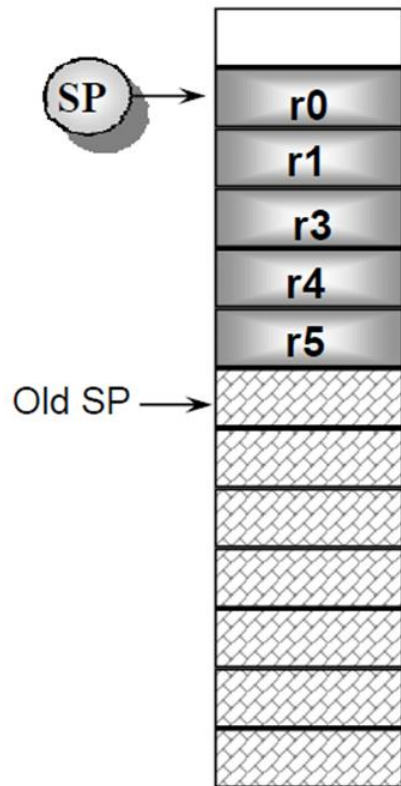
- Base register used to determine where memory access should occur.
 - 4 different addressing modes allow increment and decrement inclusive or exclusive of the base register location.
 - Base register can be optionally updated following the transfer (by appending it with an '!').
 - Lowest register number is always transferred to/from lowest memory location accessed.
- These instructions are very efficient for
 - Saving and restoring context
 - For this useful to view memory as a stack.
 - Moving large blocks of data around memory
 - For this useful to directly represent functionality of the instructions.

Stack Operation

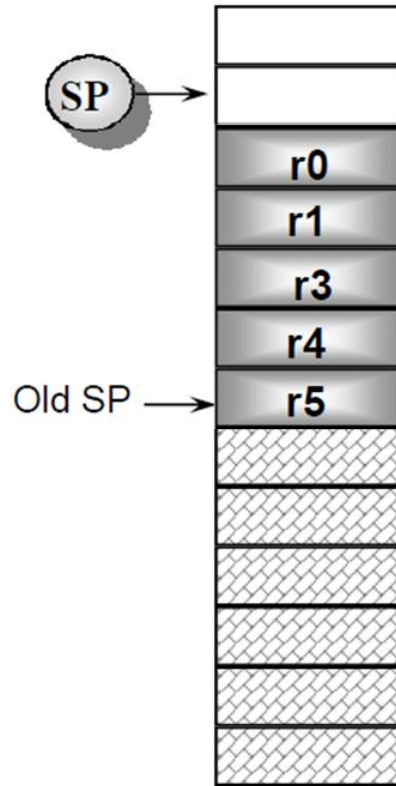
- Traditionally, a stack grows down in memory, with the last “pushed” value at the lowest address. The ARM also supports ascending stacks, where the stack structure grows up through memory.
- The value of the stack pointer can either:
 - Point to the last occupied address (Full stack) and so needs pre-decrementing (ie before the push)
 - Point to the next occupied address (Empty stack) and so needs post-decrementing (ie after the push)
- The stack type to be used is given by the postfix to the instruction:
 - **STMFD / LDMFD** : Full Descending stack
 - **STMFA / LDMFA** : Full Ascending stack.
 - **STMED / LDMED** : Empty Descending stack
 - **STMEA / LDMEA** : Empty Ascending stack
- Note: ARM Compiler will always use a Full descending stack.

Stack Examples

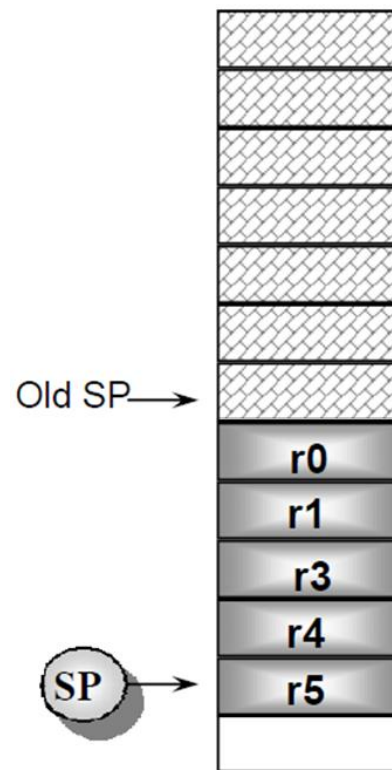
STMFD sp!,
{r0,r1,r3-r5}



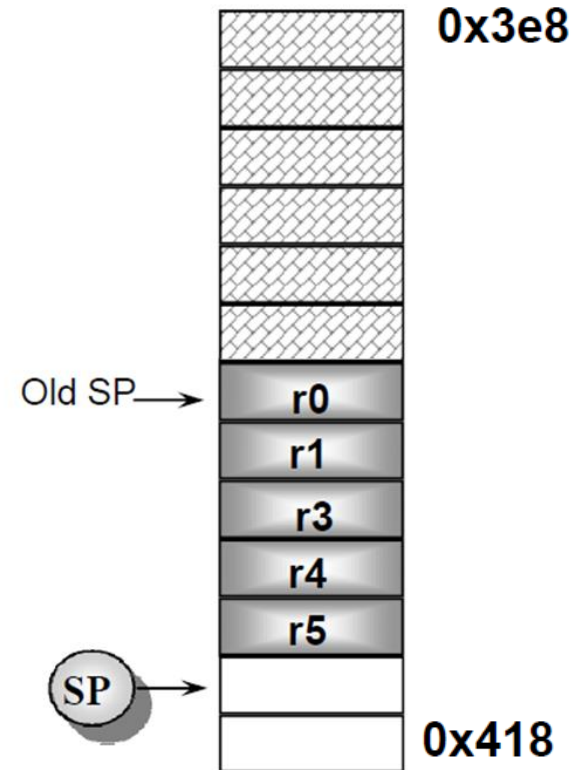
STMED sp!,
{r0,r1,r3-r5}



STMFA sp!,
{r0,r1,r3-r5}



STMEA sp!,
{r0,r1,r3-r5}



Stacks and Subroutines

- One use of stacks is to create temporary register workspace for subroutines. Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the end so as to restore them before return to the caller:

```
STMFD sp!,{r0-r12, lr}      ; stack all registers
.....                      ; and the return address
.....
LDMFD sp!,{r0-r12, pc}      ; load all the registers
                           ; and return automatically
```

- See the chapter on the ARM Procedure Call Standard in the SDT Reference Manual for further details of register usage within subroutines.
- If the pop instruction also had the 'S' bit set (using '^') then the transfer of the PC when in a privileged mode would also cause the SPSR to be copied into the CPSR (see exception handling module).

Direct functionality of Block Data Transfer

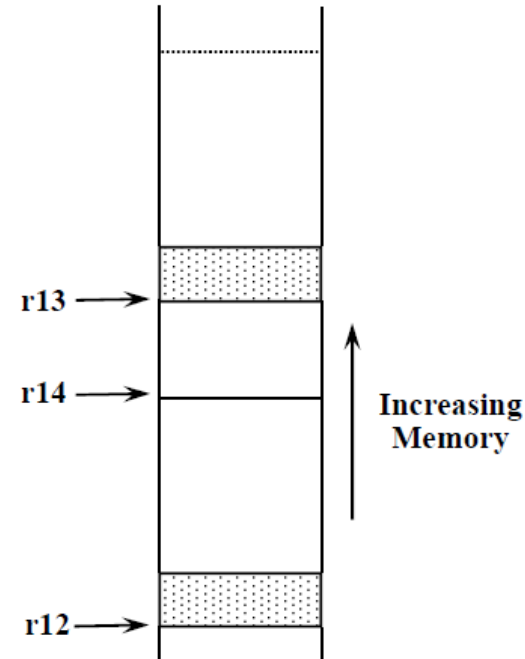
- When LDM / STM are not being used to implement stacks, it is clearer to specify exactly what functionality of the instruction is:
 - i.e. specify whether to increment / decrement the base pointer, before or after the memory access.
- In order to do this, LDM / STM support a further syntax in addition to the stack one:
 - **STMIA / LDMIA : Increment After**
 - **STMIB / LDMIB : Increment Before**
 - **STMDA / LDMDA : Decrement After**
 - **STMDB / LDMDB : Decrement Before**

Example: Block Copy

- Copy a block of memory, which is an exact multiple of 12 words long from the location pointed to by r12 to the location pointed to by r13. r14 points to the end of block to be copied.

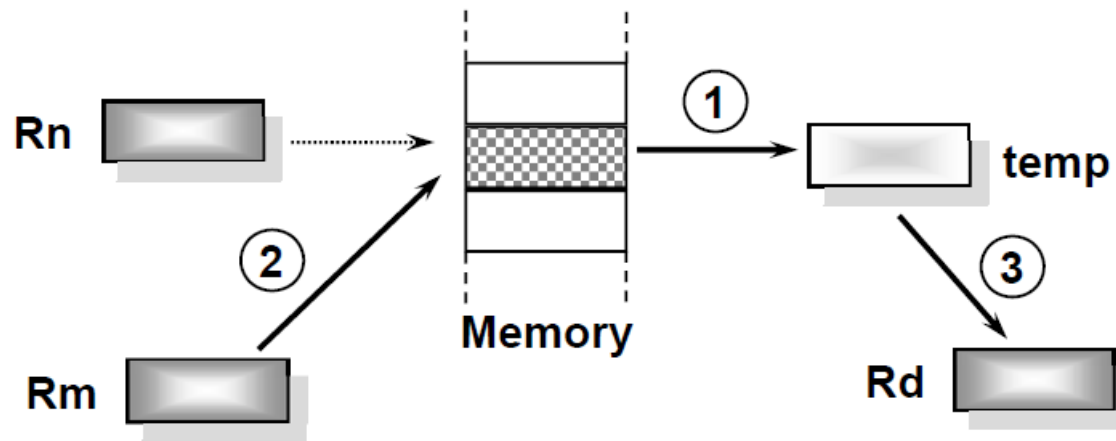
```
; r12 points to the start of the source data
; r14 points to the end of the source data
; r13 points to the start of the destination data
loop    LDMIA    r12!, {r0-r11}; load 48 bytes
        STMIA    r13!, {r0-r11}; and store them
        CMP      r12, r14; check for the end
        BNE      loop; and loop until done
```

- This loop transfers 48 bytes in 31 cycles
- Over 50 Mbytes/sec at 33 MHz



Swap and Swap Byte Instructions

- Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.
- Syntax:
 - **SWP{<cond>}{B} Rd, Rm, [Rn]**



- To implement an actual swap of contents make $Rd = Rm$.
- The compiler cannot produce this instruction.

Multiplication Instructions

- The Basic ARM provides two multiplication instructions.
- Multiply
 - **MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs**
- Multiply Accumulate-does addition for free
 - **MLA{<cond>}{S} Rd, Rm, Rs, Rn ; Rd = (Rm * Rs) + Rn**
- Restrictions on use:
 - Rd and Rm cannot be the same register
 - Can be avoided by swapping Rm and Rs around. This works because multiplication is commutative.
 - Cannot use PC.
- These will be picked up by the assembler if overlooked.
- Operands can be considered signed or unsigned
 - Up to user to interpret correctly.

Software Interrupt (SWI)

[illegible]

- In effect, a SWI is a user-defined instruction.
- It causes an exception trap to the SWI hardware vector (thus causing a change to supervisor mode, plus the associated state saving), thus causing the SWI exception handler to be called.
- The handler can then examine the comment field of the instruction to decide what operation has been requested.
- By making use of the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- See Exception Handling Module for further details.

References

- ARM Instruction Set:
ARM7TDMI-S Data Sheet Chapter 4
- *Thumb Instruction Set:*
ARM7TDMI-S Data Sheet Chapter 5