

L	T	P	C
3	0	3	4

19CSE303 – Embedded Systems

L	T	P	C
3	0	3	4

Course Introduction

Unit 1

Basics of Embedded Systems –Definition, Characteristics, Architecture of Microprocessors: General definitions of computers, micro-processors, micro controllers and digital signal processors. ARM Architecture: RISC Machine, Architectural Inheritance, Programmers model, CPSR, ARM Organization and Implementation. 3 Stage pipeline, 5 Stage pipeline, ARM Instruction execution, Co-processor interface, ARM Assembly language Programming, Data processing instructions, Data Transfer Instructions, Control flow instructions, Architectural support for high level programming, Thumb Instruction set.

Unit 2

Interrupt structure -Vector interrupt table, Interrupt service routines, Asynchronous and Synchronous data transfer schemes, ARM memory interface, AMBA interface, Microcontroller ARM7-LPC2148 ports: GPIO, A/D Converters, PWM, timer / counter, UART and its interfacing – Embedded Programming Concepts: Role of Infinite loop, Compiler, Assembler, Interpreter, Linker, Loader, Debugger-Application development using Keil IDE.

Unit 3

Introduction to ARM Cortex M4 Microcontroller –GPIO and other Peripherals -System development process: Requirements, Design, Development, Testing and Deployment. Prototyping: Analog circuit design and construction on a solderless breadboard, Hardware and Software design, Programming simple logic and testing PLL and Systick Timers, Design strategy for building Finite State machine.

Course Introduction

➤ Text Book(s):

Furber SB. ARM system-on-chip architecture. Pearson Education; 2000. Martin T. The Insider's guide to the Philips ARM7-based microcontrollers. Coventry, Hitex, UK, Ltd. 2005.

➤ Reference(s):

- Valvano JW. Embedded Systems: Introduction to ARM Cortex-M Microcontrollers. Jonathan W. Valvano; 2016.
- Valvano JW. Embedded microcomputer systems: real time interfacing. Cengage Learning; 2012.

Course Introduction

➤ Evaluation Pattern (65 : 35) (TENTATIVE)

19CSE303 Embedded Systems 3-0-3-4	Internal	Components	Max Marks	Weightage
		Quiz-1	10	
		Quiz-2	10	
		Lab Evaluation-1	15	
		Lab Evaluation-2	15	
		Periodical Exam 1	50	
		Periodical Exam 2	50	
	External	End Semester Exam	100	35

Introduction

➤ What is a System?

A system is a way of working organizing or doing one or many tasks according to a fixed plan, program or set of rules

A system is also an arrangement in which all its units assemble and work together according to the plan or program

SYSTEM EXAMPLES



WATCH

It is a time display **SYSTEM**

Parts: Hardware, Needles, Battery, Dial, Chassis and Strap

Rules

All needles move clockwise only

A thin needle rotates every second

A long needle rotates every minute

A short needle rotates every hour

All needles return to the original position after 12 hours



SYSTEM EXAMPLES

WASHING MACHINE

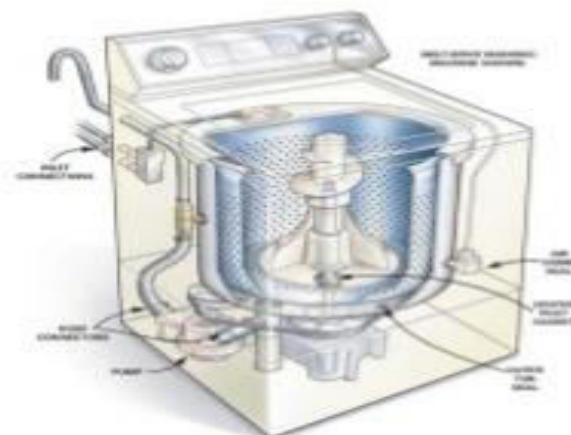
It is an automatic clothes washing **SYSTEM**

Parts: Status display panel, Switches & Dials, Motor, Power supply & control unit, Inner water level sensor and solenoid valve.



Rules

- . Wash by spinning
- . Rinse
- . Drying
- . Wash over by blinking
- . Each step display the process stage
- In case interruption, execute only the remaining



Computing Systems

Computing systems are everywhere

Most of us think of “desktop” computers

- PC's
- Laptops
- Mainframes
- Servers

designed to be flexible and to meet a wide range of end-user needs.

But there's another type of computing system

- Far more common...

Embedded Systems

➤ We are surrounded by Embedded Systems

➤ Cell Phones

➤ Automatic Washing Machines

➤ Traffic Signals with Timers

➤ Automobile Electronics

Find a System that contains no electronic system

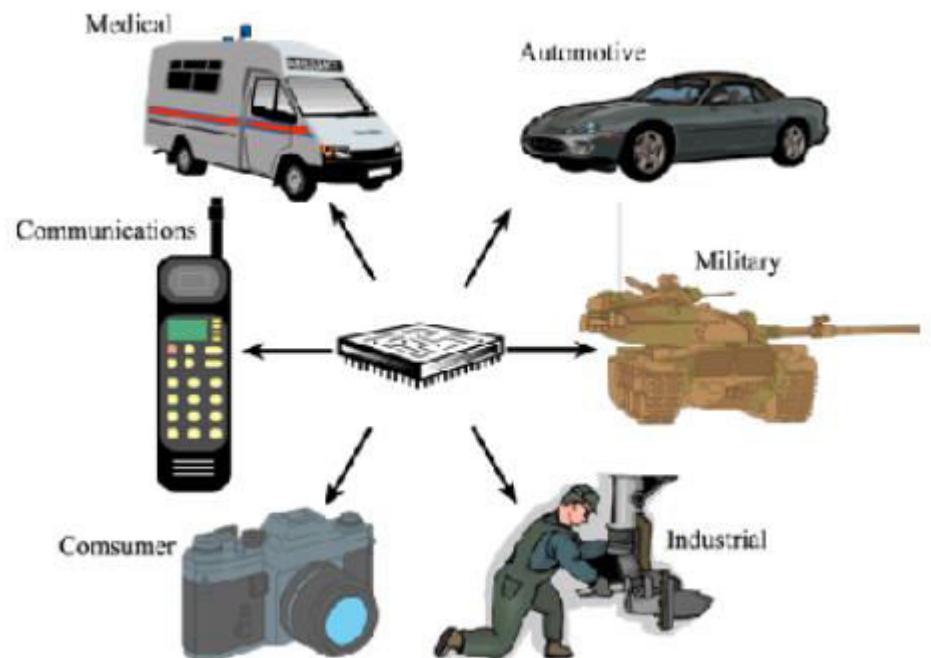
How an electronic system improve the functionality / efficiency of that system

Over 95% of software systems are actually embedded !!!



Embedded Systems

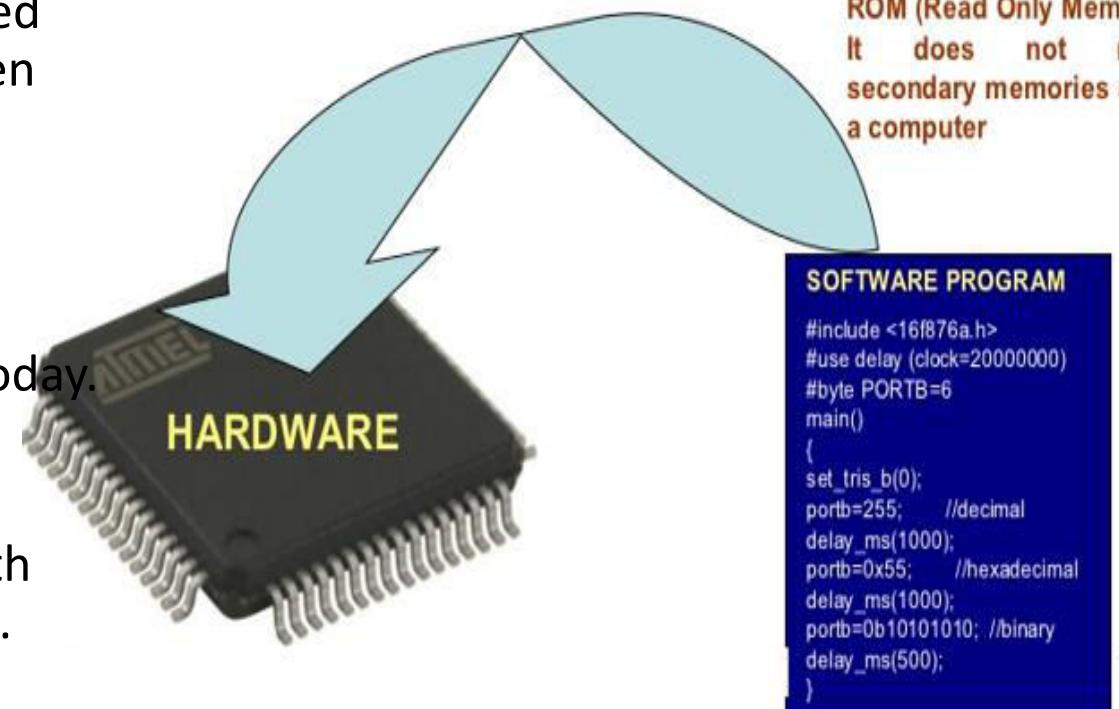
- A Special purpose computer designed to perform certain dedicated functions
- Embedded Systems are everywhere
- Hidden (computer inside)
- Dedicated purpose



Embedded Systems

- An **embedded system** is a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints.
- It is *embedded* as part of a complete device often including hardware and mechanical parts.
- Embedded systems control many devices in common use today.
- **Definition:**

An Embedded System is one that has computer hardware with software embedded in it as one of its important components.



Embedded Systems - Characteristics

- Single-functioned –

An embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.

- Tightly constrained –

All computing systems have constraints on design metrics, but those on an embedded system can be especially tight.

Design metrics is a measure of an implementation's features such as its cost, size, power, and performance.

It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.

Embedded Systems - Characteristics

- Reactive and Real time –

Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay.

Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors.

It must compute acceleration or de-accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.

- Microprocessors based –

It must be microprocessor or microcontroller based.

Embedded Systems - Characteristics

- Memory –

It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.

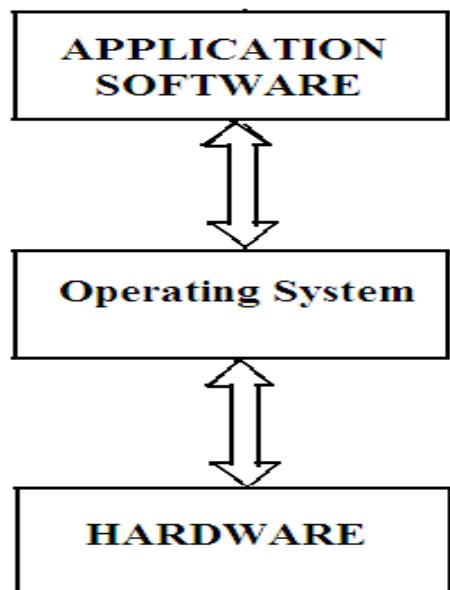
- Connected –

It must have connected peripherals to connect input and output devices.

- HW-SW systems –

Software is used for more features and flexibility. Hardware is used for performance and security.

What makes an Embedded Systems



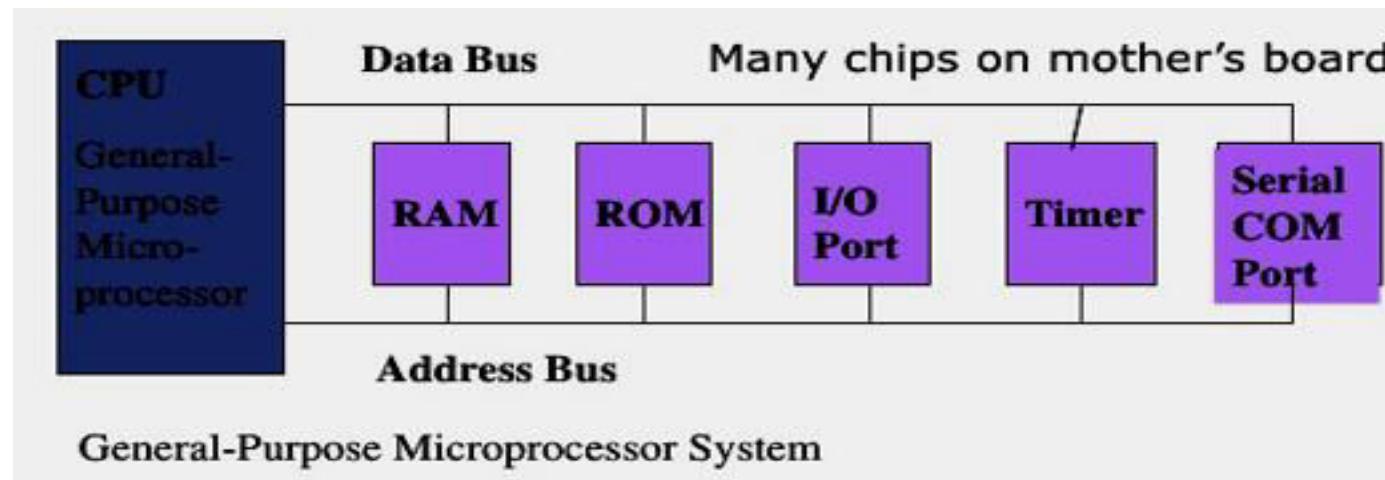
COMPONENTS OF EMBEDDED SYSTEM

- **It has Hardware**
Processor, Timers, Interrupt controller, I/O Devices, Memories, Ports, etc.
- **It has main Application Software**
Which may perform concurrently the series of tasks or multiple tasks.
- **It has Real Time Operating System (RTOS)**
RTOS defines the way the system work. Which supervise the application software. It sets the rules during the execution of the application program. A small scale embedded system may not need an RTOS.

Embedded Systems – Microcontrollers vs Microprocessors

Microprocessors:

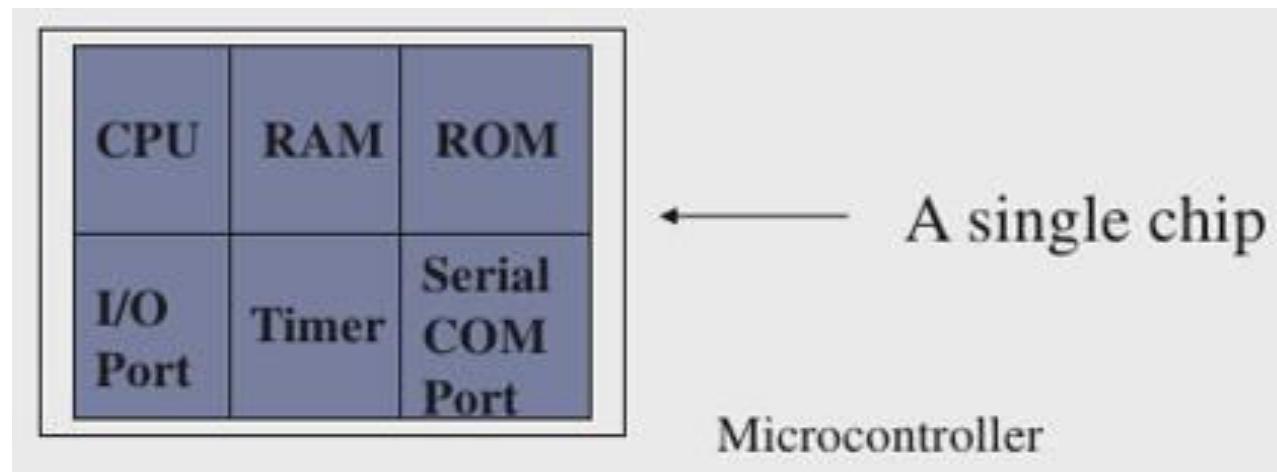
- CPU for Computers
- No RAM, ROM, I/O on CPU chip itself
- Example: Intel's x86, Motorola's 680x0



Embedded Systems – Microcontrollers vs Microprocessors

Microcontroller:

- A smaller computer
- On-chip RAM, ROM, I/O ports...
- Example: Motorola's 6811, Intel's 8051, Zilog's Z8 and PIC 16X

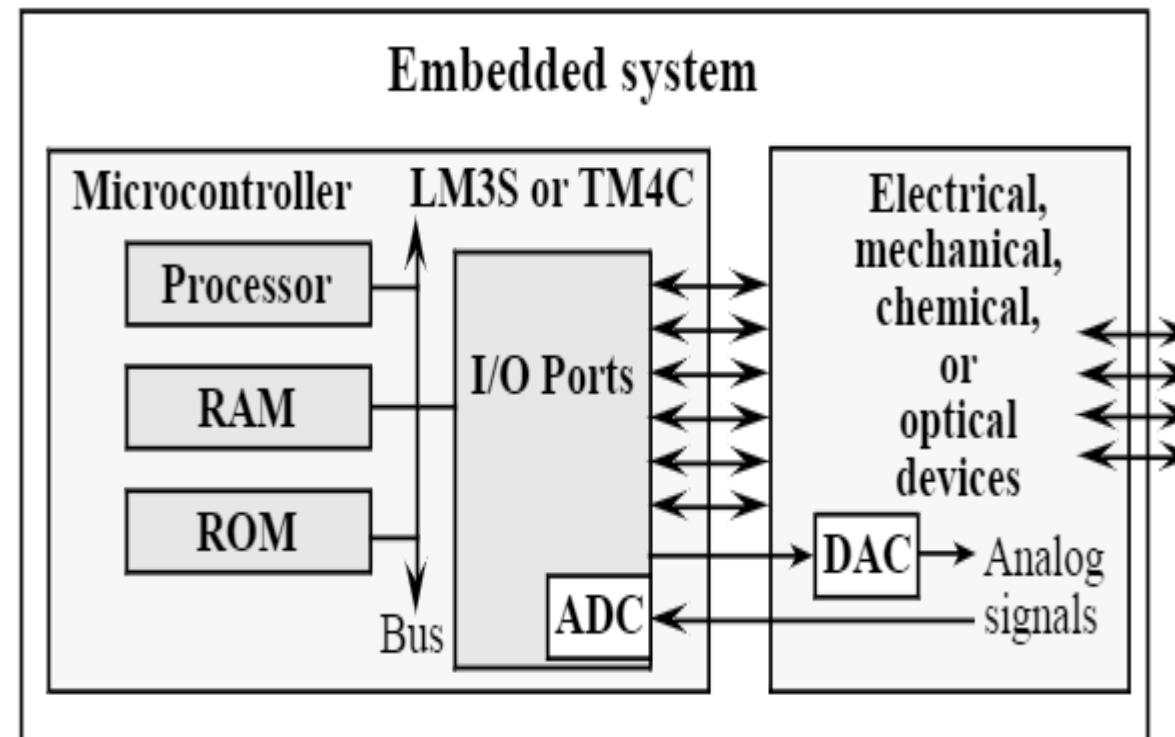


Embedded Systems – Microcontrollers vs Microprocessors

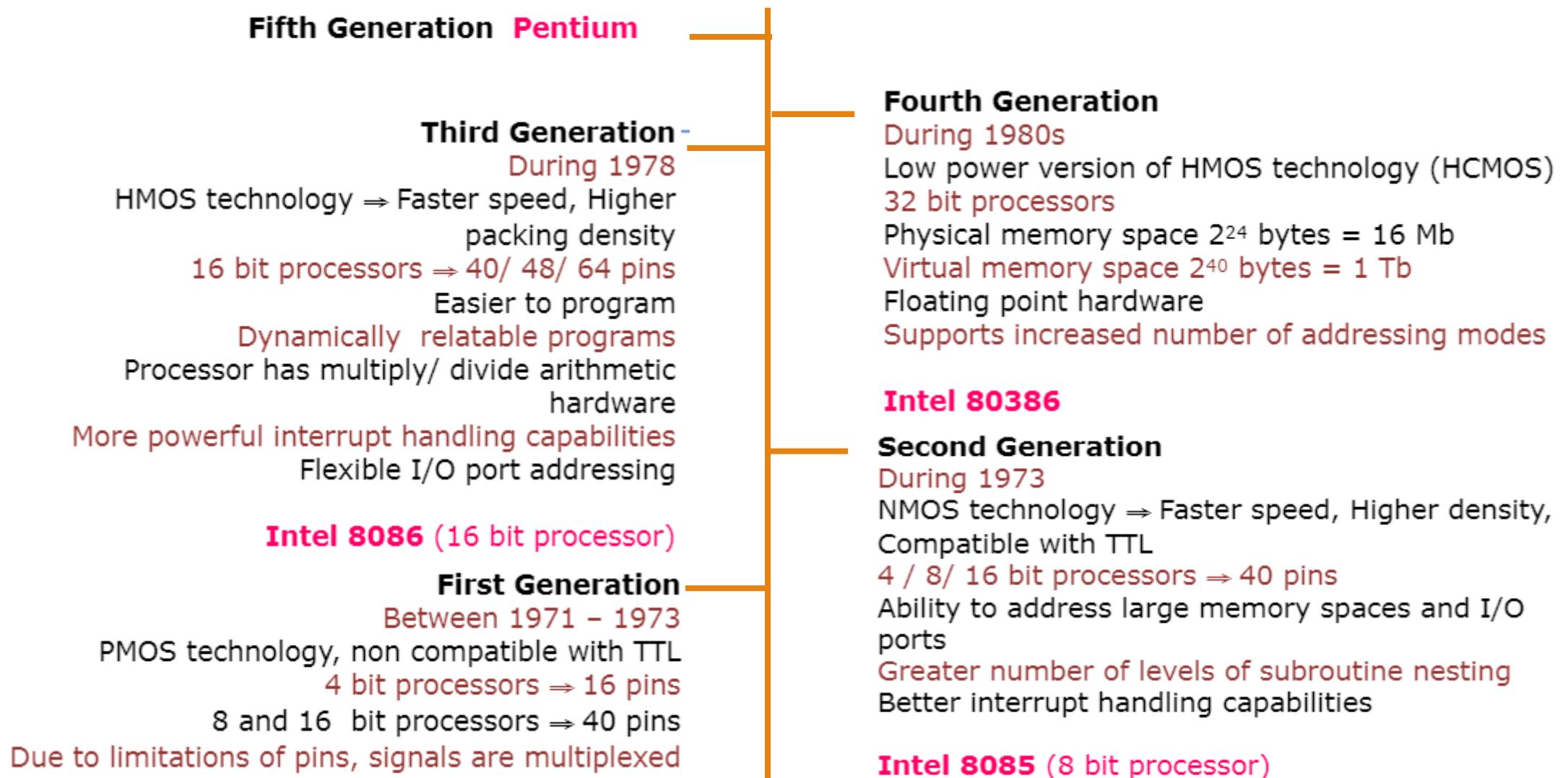
Microprocessors	Microcontrollers
<p>Microprocessors are multitasking in nature. Can perform multiple tasks at a time.</p> <p>For example, on computer we can play music while writing text in text editor.</p>	<p>Single task oriented.</p> <p>For example, a washing machine is designed for washing clothes only.</p>
<p>RAM, ROM, I/O Ports, and Timers can be added externally and can vary in numbers.</p>	<p>RAM, ROM, I/O Ports, and Timers cannot be added externally. These components are to be embedded together on a chip and are fixed in</p>
<p>Designers can decide the number of memory or I/O ports needed.</p>	<p>Fixed number for memory or I/O makes a microcontroller ideal for a limited but specific task.</p>
<p>External support of external memory and I/O ports makes a microprocessor-based system heavier and costlier.</p>	<p>Microcontrollers are lightweight and cheaper than a microprocessor.</p>
<p>External devices require more space and their power consumption is higher.</p>	<p>A microcontroller-based system consumes less power and takes less space.</p>

Embedded Systems

- Microprocessor
 - Intel: 4004, ..8080,.. x86
 - Freescale: 6800, .. 9S12,.. PowerPC
 - ARM, DEC, SPARC, MIPS, PowerPC, Natl. Semi.,...
 - Applications: Desktop PC's, Laptops, notepads etc.
- Microcontroller
 - Processor+Memory+I/O Ports (Interfaces)
 - Applications: Keyboards, mouse, pendrive, mobiles etc



Evolution of Microprocessors

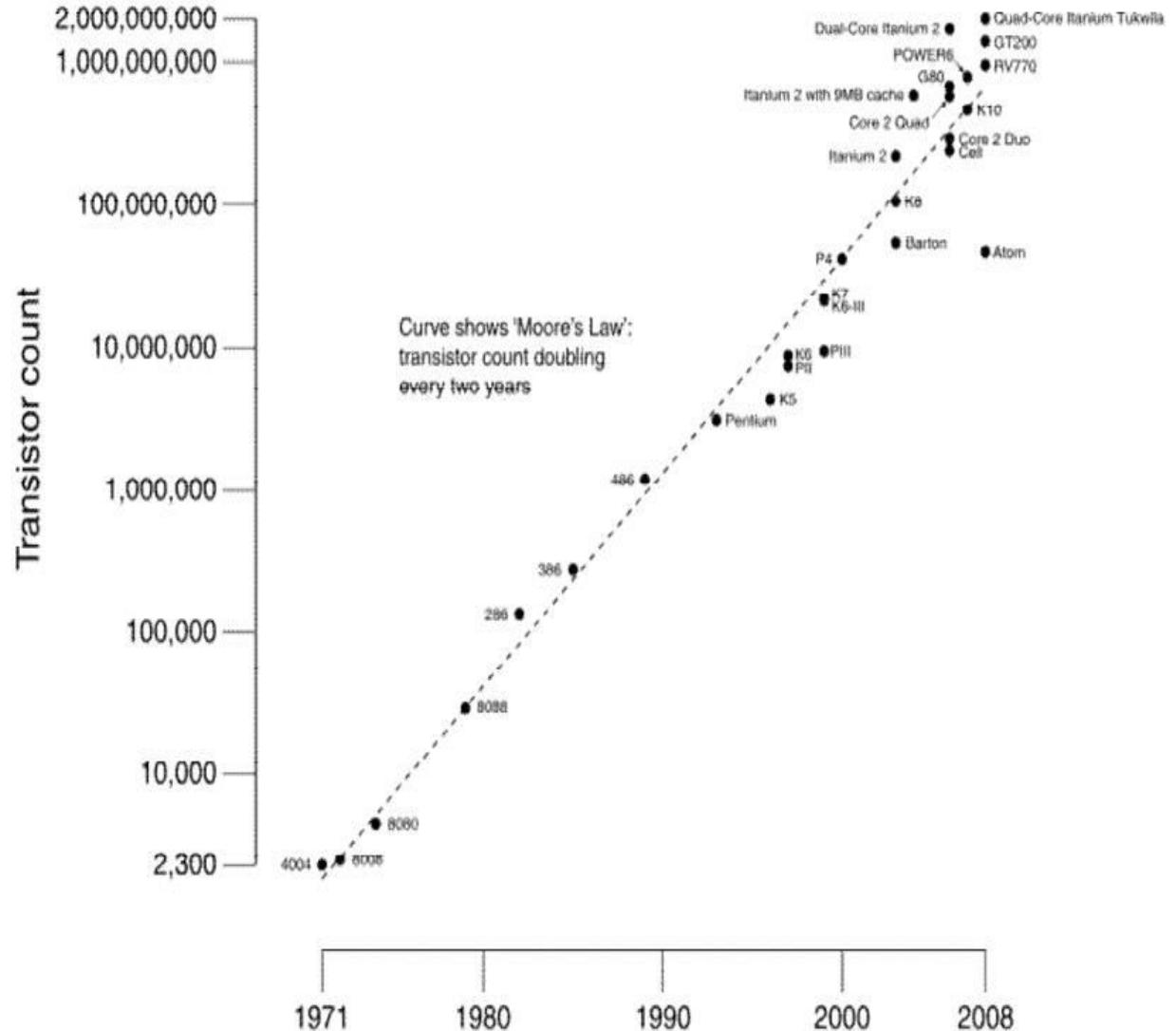


Evolution of Microprocessors

NAME	YEAR	TRANSISTORS	DATA WIDTH	CLOCK SPEED
8080	1974	6,000	8 bits	2 MHz
8085	1976	6,500	8 bits	5 MHz
8086	1978	29,000	16 bits	5 MHz
8088	1979	29,000	8 bits	5 MHz
80286	1982	134,000	16 bits	6 MHz
80386	1985	275,000	32 bits	16 MHz
80486	1989	1,200,000	32 bits	25 MHz
PENTIUM	1993	3,100,000	32/64 bits	60 MHz
PENTIUM II	1997	7,500,000	64 bits	233 MHz
PENTIUM III	1999	9,500,000	64 bits	450 MHz
PENTIUM IV	2000	42,000,000	64 bits	1.5 GHz

Moore's Law:

- Moore's Law refers to Moore's perception that the number of transistors on a microchip doubles every two years, though the cost of computers is halved.
- Moore's Law states that we can expect the speed and capability of our computers to increase every couple of years, and we will pay less for them.



Digital Signal Processing

Definition:

Digital signal processing (DSP) is the process of analyzing and modifying a signal to optimize or improve its efficiency or performance.

It involves applying various mathematical and computational algorithms to analog and digital signals to produce a signal that's of higher quality than the original signal.

Digital Signal Processing



- Architecture optimized for signal processing applications
 - Large number of mathematical operations on a series of data samples
- Hardware implementation of Multiply/Accumulate function
 - Critical for FFT (Fast Fourier Transform) type applications
- The Texas Instruments TMS 5100

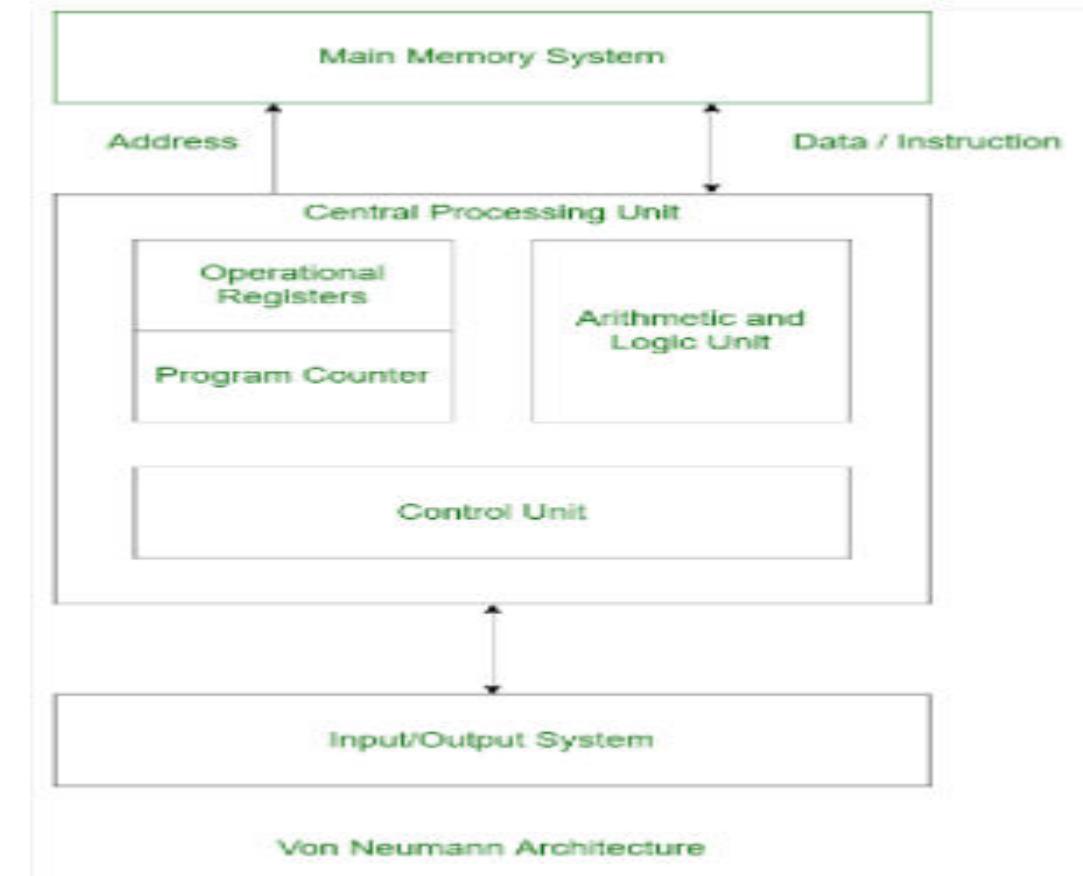


Architecture vs. Organization

Computer Architecture	Computer Organization
Computer Architecture is concerned with the way hardware components are connected together to form a computer system.	Computer Organization is concerned with the structure and behaviour of a computer system as seen by the user.
It acts as the interface between hardware and software.	It deals with the components of a connection in a system.
Computer Architecture helps us to understand the functionalities of a system.	Computer Organization tells us how exactly all the units in the system are arranged and interconnected.
A programmer can view architecture in terms of instructions, addressing modes and registers.	Organization expresses the realization of architecture.
While designing a computer system architecture is considered first.	An organization is done on the basis of architecture.
Computer Architecture deals with high-level design issues.	Computer Organization deals with low-level design issues.
Architecture involves Logic (Instruction sets, Addressing modes, Data types, Cache optimization)	Organization involves Physical Components (Circuit design, Adders, Signals, Peripherals)

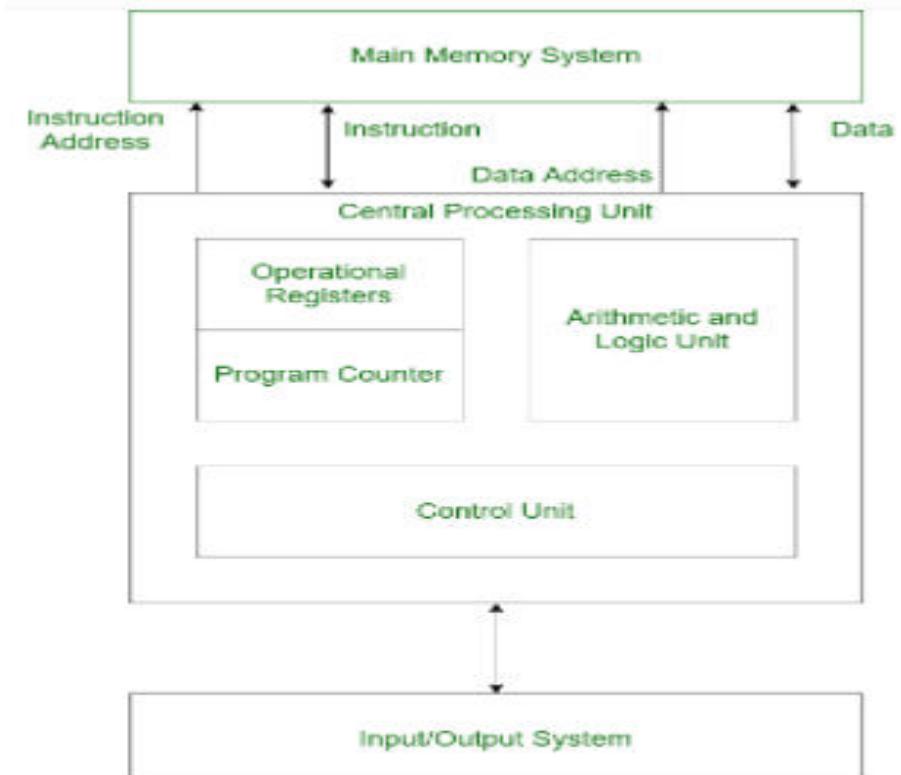
Von-Neumann Architecture

- Von Neumann Architecture is a digital computer architecture whose design is based on the concept of stored program computers where program data and instruction data are stored in the same memory.
- This architecture was designed by the famous mathematician and physicist John Von Neumann in 1945.



Harvard Architecture

- Harvard Architecture is the digital computer architecture whose design is based on the concept where there are separate storage and separate buses (signal path) for instruction and data.
- It was basically developed to overcome the bottleneck of Von Neumann Architecture



Harvard Architecture

Von-Neumann vs Harvard Architecture

Von-Neumann Architecture	Harvard Architecture
Single memory to be shared by both code and data.	Separate memories for code and data.
Processor needs to fetch code in a separate clock cycle and data in another	Single clock cycle is sufficient, as separate buses
Higher speed, thus less time consuming.	Slower in speed, thus more time-consuming.
Simple in design	Complex in design.

RISC vs CISC

RISC:

- Reduced Instruction Set Architecture
- The main idea behind is to make hardware simpler by using an instruction set composed
- Just like a load command will load data, store command will store the data.
- Mostly all have the same format.
- Reduce the number of memory accesses required by increasing the number of registers
- Reduce the number of addressing modes
- Allow pipelining of instructions
- To increase the CPU performance: Reduce the cycles per instruction at the cost of the number of instructions per program.

RISC vs CISC

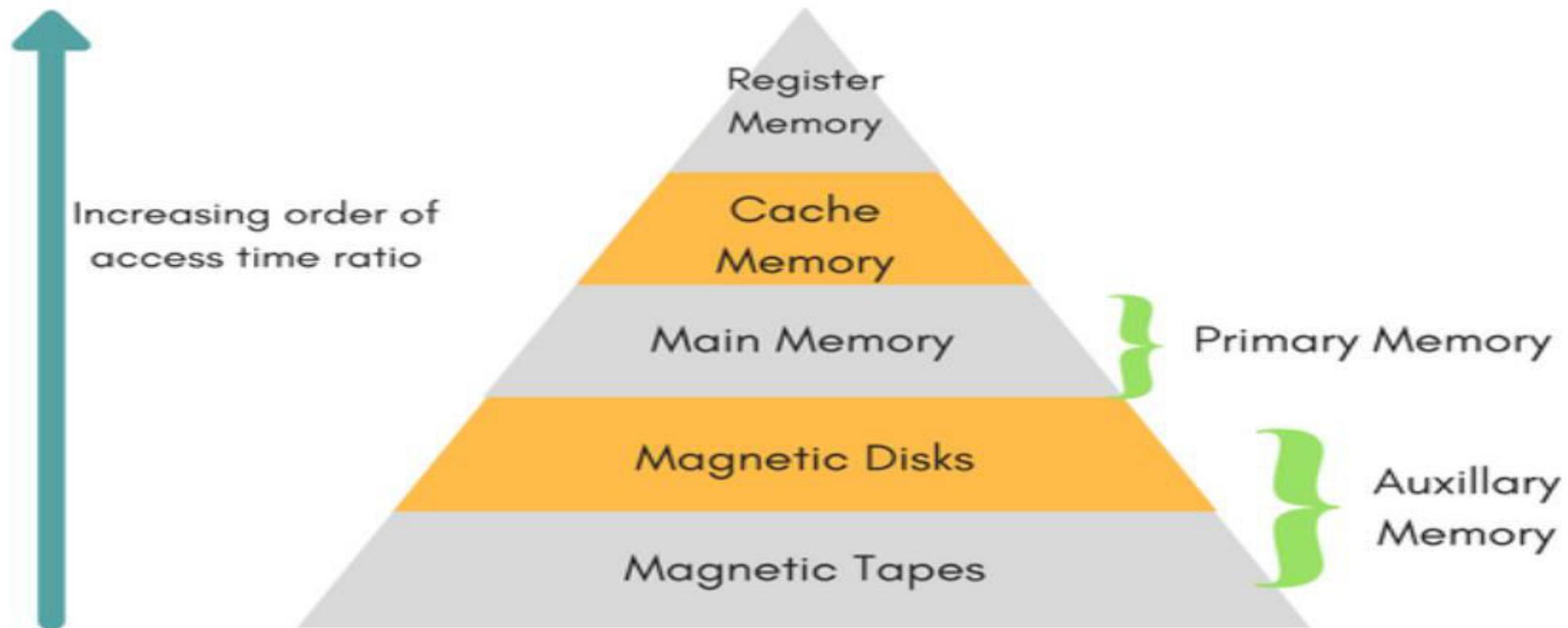
CISC:

- Complex Instruction Set Architecture
- The main idea is that a single instruction will do more operations
- Just like a multiplication command which does; loading data, evaluating, and storing it, hence it's complex.
- Number of instructions are reduced by having *multiple operations* within a single instruction
- In turn making instruction length variable and fetch-decode-execute time unpredictable – making it more complex
- Provide more addressing modes
- Less number of general-purpose registers as operation get performed in memory itself.
- To increase the CPU performance: Minimize the number of instructions per program but at the cost of increase in number of cycles per instruction.

RISC vs CISC

CISC	RISC
Larger set of instructions. Easy to Program.	Smaller set of Instructions. Difficult to program.
Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.
Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware.
Pipelining is not possible.	Pipelining of instructions is possible,

Memory Organization



Memory Organization

- Byte organized memory:

Each address on the address bus points to a memory location where a byte (8 bit) is stored.

- Word organized memory:

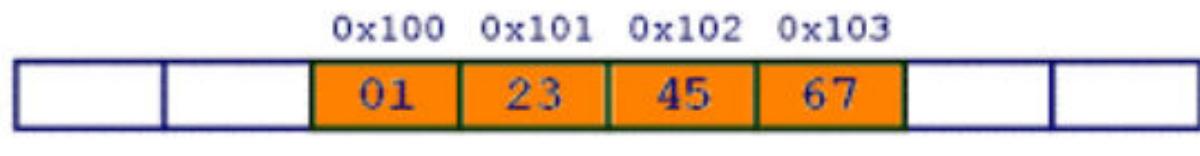
Each address on the address bus points to a memory location where a word (multiple of 8 bit) is stored.

Memory Organization

How data is stored in memory?

- LittleEndian –

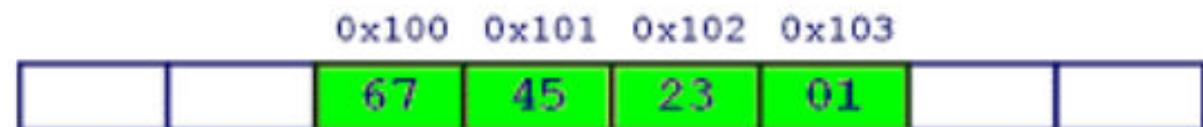
LSB to Lower memory address



Big Endian

- BigEndian –

LSB to Higher memory address



Little Endian

- 0x01234567 will be stored as

Thank You

L	T	P	C
3	0	3	4

19CSE303 – Embedded Systems

ARM (History)

- ARM (Acorn RISC Machine) started as a new, powerful, CPU design for the replacement of the 8-bit 6502 in Acorn Computers (Cambridge, UK, 1985)
- First models had only a 26-bit program counter, limiting the memory space to 64 MB (not too much by today standards, but a lot at that time).
- 1990 spin-off: ARM renamed Advanced RISC Machines
- ARM now focuses on Embedded CPU cores
 - IP(Intellectual Property) licensing: Almost every silicon manufacturer sells some microcontroller with an ARM core. Some even compete with their own designs.
 - Processing power with low current consumption
 - Good MIPS/Watt figure
 - Ideal for portable devices
- Compact memories: 16-bit opcodes (Thumb)

ARM (Partnership Model)



ARM (Powered Products)



ARM Architecture

- Based on RISC architecture with enhancements to meet requirements of Embedded applications
- A large uniform register file
- Load Store Architecture
- Uniform and Fixed length instruction
- 32-bit processor
- Instructions are of 32-bit long
- Good speed / Power consumption ratio
- High Code density

Enhancement to Basic Architecture

- Control over ALU and shifter for every data processing operations to maximize their usage.
- Auto-increment and Auto-decrement addressing modes to optimize program loops
- Load –Store multiply instructions to maximize data throughput
- Conditional execution of instruction to maximize execution throughput

ARM Architecture Versions

- Version 1
 - The first ARM processor, developed at Acorn Computers Limited
 - 1983-1985
- Version 2
 - Sold in volume in the Acorn Archimedes and A3000 products
 - 26-bit addressing, including 32-bit result multiply and coprocessor
- Version 2a
 - Coprocessor 15 as the system control coprocessor to manage cache and the atomic load store (SWP) instruction

ARM Architecture Versions

- Version 3
 - First ARM processor designed by ARM Limited (1990)
 - ARM6 (macro cell)
 - ARM60 (stand-alone processor)
 - ARM600 (an integrated CPU with on-chip cache, MMU, write buffer)
 - ARM610 (used in Apple Newton)
 - 32-bit addressing, separate CPSR and SPSRs
 - And the undefined and abort modes to allow coprocessor emulation and virtual memory support in supervisor mode
- Version 3M
 - Introduce the signed and unsigned multiply and multiply accumulate
 - instructions that generate the full 64-bit result

ARM Architecture Versions

- Version 4
 - Add the signed, unsigned half-word and signed byte load and store instructions
 - Reserve some of SWI space for architecturally defined operation
 - System mode is introduced
- Version 4T
 - 16-bit Thumb compressed form of the instruction set is introduced
- Version 5T
 - Introduced recently, a superset of version 4T adding the BLX, CLZ and
 - BRK instructions
- Version 5TE
 - Add the signal processing instruction set extension

ARM Architecture Versions

- Version 6
 - Media processing extensions (SIMD)
 - 2x faster MPEG4 encode/decode
 - 2x faster audio DSP
 - Improved cache architecture
 - Physically addressed caches
 - Reduction in cache flush/refill
 - Reduced overhead in context switches
 - Improved exception and interrupt handling
 - Important for improving performance in real-time tasks
 - Unaligned and mixed-endian data support
 - Simpler data sharing, application porting and saves memory

ARM Cores

Core	Architecture
ARM1	v1
ARM2	v2
ARM2as, ARM3	v2a
ARM6, ARM600, ARM610	v3
ARM7, ARM700, ARM710	v3
ARM7TDMI, ARM710T, ARM720T, ARM740T	v4T
StrongARM, ARM8, ARM810	v4
ARM9TDMI, ARM920T, ARM940T	V4T
ARM9E-S, ARM10TDMI, ARM1020E	v5TE
ARM10TDMI, ARM1020E	v5TE
ARM11 MPCore, ARM1136J(F)-S, ARM1176JZ(F)-S	v6
Cortex-A/R/M	v7

ARM

- Advanced RISC Machine / Acorn RISC Machine
- A 32-bit processor core
- On mid-1980's replacement for 6502 used in BBC Micro.
- A Load-Store architecture
- Fixed-length 32-bit instructions (exceptions!)
- 3-address instruction formats
- Condition execution of ALL instructions

- Load-Store multiple registers in one instruction
- Most data processing instructions are single-cycle
- A single cycle n-bit shift with ALU operation
- But many other instructions take multiple clock cycles
- By default, Little-endian but can be configured for Big-endian
- In mid-90s used in Apple's PDA Newton

Data Size and Instruction Sets

- The ARM is a 32-bit architecture.
- When used in relation to the ARM:
 - Byte means 8 bits
 - Half word means 16 bits (two bytes)
 - Word means 32 bits (four bytes)
- Most ARM's implement two instruction sets
 - 32-bit ARM Instruction Set
 - 16-bit Thumb Instruction Set
- Jazelle cores can also execute Java bytecode

ARM Processor Core

- Current low-end ARM core for applications like digital mobile phones
- TDMI
 - T: Thumb, 16-bit instruction set
 - D: on-chip Debug support, enabling the processor to halt in response to a debug request
 - M: enhanced Multiplier, yield a full 64-bit result, high performance
 - I: Embedded (in-circuit emulator) ICE hardware
- Von Neumann architecture
- 3-stage pipeline

Data Path - Overview

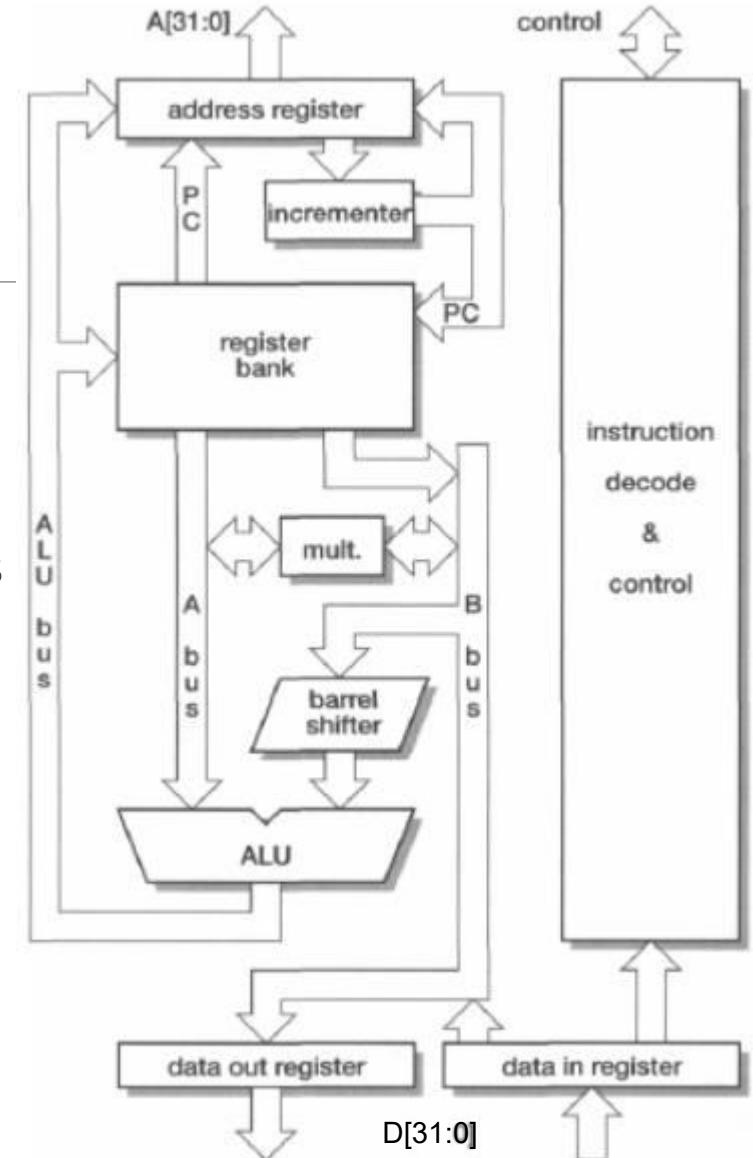
- A collection of functional units such as **arithmetic logic units or multipliers that perform data processing operations, registers, and buses**. Along with the control unit it composes the central processing unit (CPU).
- Principle: Memory will be the limiting factor
- Memory access will always take a clock cycle
- Each instruction takes exactly the number of clock cycles defined by the number of memory accesses it must take
- Datapath design with sufficient resource to allow these instructions to complete in two or one clock cycle

Data Path - Overview

- Data Items are placed in register file
- No data processing instruction directly manipulate data memory
- Instruction typically use two source registers and single result or destination register
- A Barrel shifter on data path can preprocess data before it enter ALU
- Increment/decrement logic can update register content for sequential access independent of ALU

Basic ARM Organization

- Register file –
 - 2 read ports, 1 write port + 1 read, 1 write port reserved for r15 (pc)
- Barrel shifter – shift or rotate one operand for any number of bits
- ALU – performs the arithmetic and logic functions required
- Memory address register + incrementer
- Memory data registers
- Instruction decoder and associated control logic



ARM's Programmer Model

- ARM is a flexible programmer's designed architecture with different applications
- A processor's instruction set defines the operations that the programmer can use to change the state of the system incorporating the processor
- This state usually comprises the values of the data items in the processor's visible registers and the systems memory
- Each instruction can be viewed as performing a defined transformation from the state before the instruction is executed to the state after it has completed

ARM - Registers

- ARM has 37 registers, each 32 bit long
- However, due to the applied conditions you can only use sixteen at any one time...
- 1 dedicated program counter (PC)
- 1 dedicated current program status register (CPSR)
- 5 dedicated saved program status registers (SPSR)
- 30 general purpose registers

ARM - Registers

- 16 register only visible registers
- **Register 0 to register 7** are general purpose registers and can be used for **any** purpose.
- **Register 8 to register 12** are general purpose registers, but they have shadow registers which come into use when you switch to FIQ mode
- **Register 13** is typically the OS stack pointer, but can be used as a general purpose register. This is an operating system issue, not a processor issue, so if you don't use the stack you can corrupt this freely within your own code as long as you store it afterwards. Each of the processor modes shadow this register.

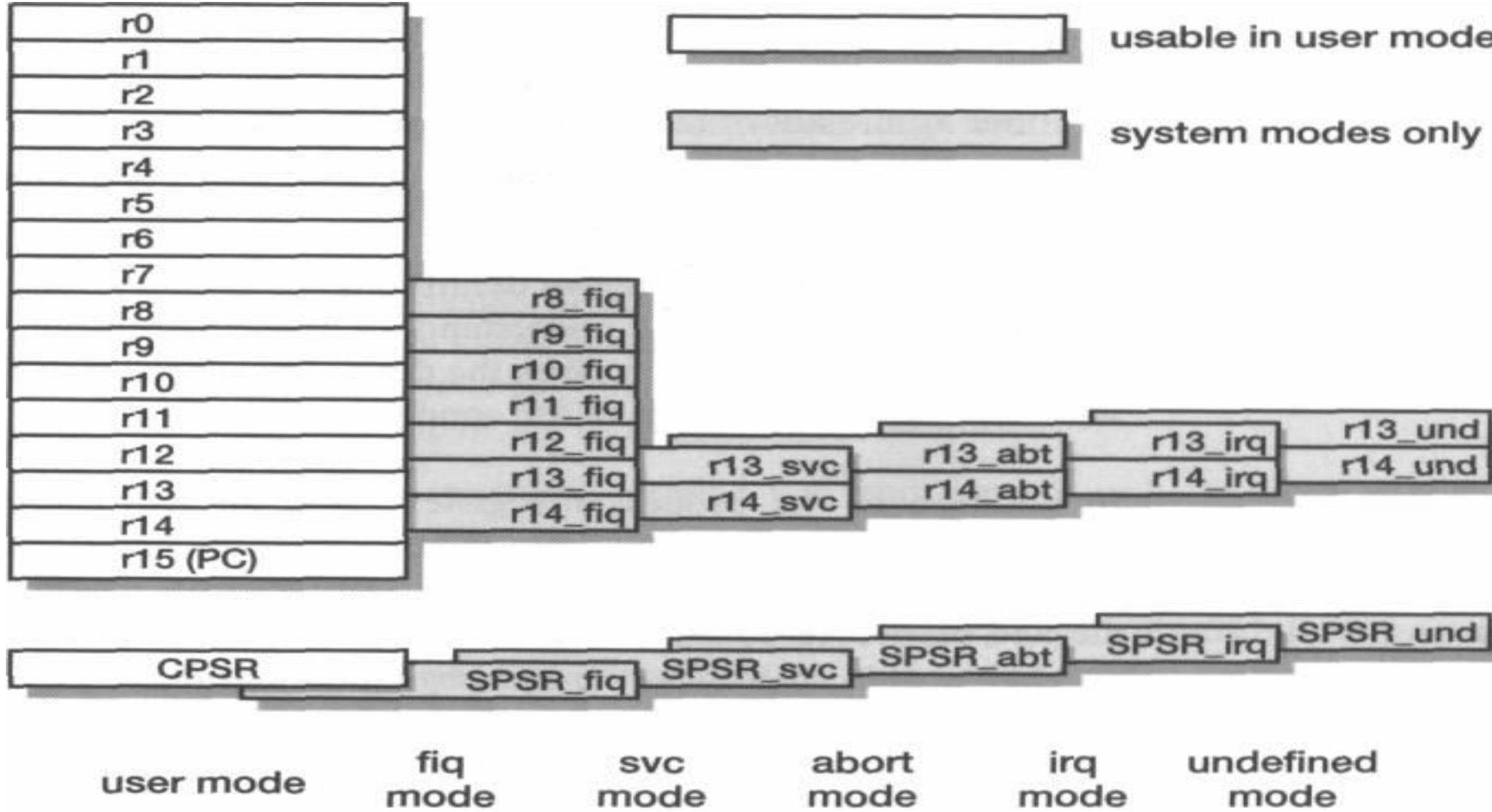
ARM - Registers

- **Register 14** is dedicated to holding the address of the return point to make writing subroutines easier. When you branch with link (BL) the return address is stored in R14
- Likewise when the program is first run, the exit address is stored in R14. All instances of R14 must be preserved in other registers (not really efficient) or in a stack. This register is shadowed across all of the processor modes. This register can be used as a general purpose register once the link address has been preserved.
- **Register 15** is the program counter
- 16 registers are user mode registers

Processor Modes

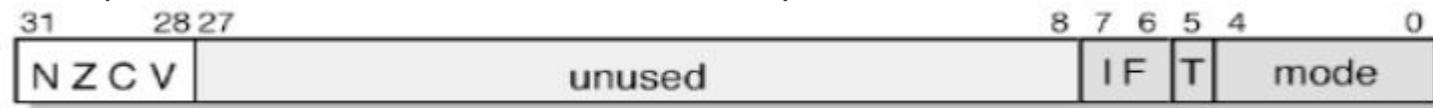
- Most ARM cores have 7 basic operating modes
 - Each mode has access to its own stack space and a different subset of registers
 - Some operations can only be carried out in a privileged mode

Mode	Description	
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



Program Status register

- **Current Program Status Register (CPSR):**
 - Monitors and controls internal operations
- **N** – Negative / sign (1 = negative & 0 = positive)
- **Z** – Zero (1 = zero & 0 = non zero)
- **C** – Carry (1 = carry occurred & 0 = no carry) – **for unsigned operations**
- **V** – Overflow (1 = overflow occurred & 0 = no overflow) – **for signed operations**
- **Mode:** Indicates which mode the processor is in
 - **I** - IRQ Enable (1 = disable & 0 = enable)
 - **F** – FIQ Enable (1 = disable & 0 = enable)
 - **T** – Thumb Mode (1 = thumb mode & 0 – ARM mode)

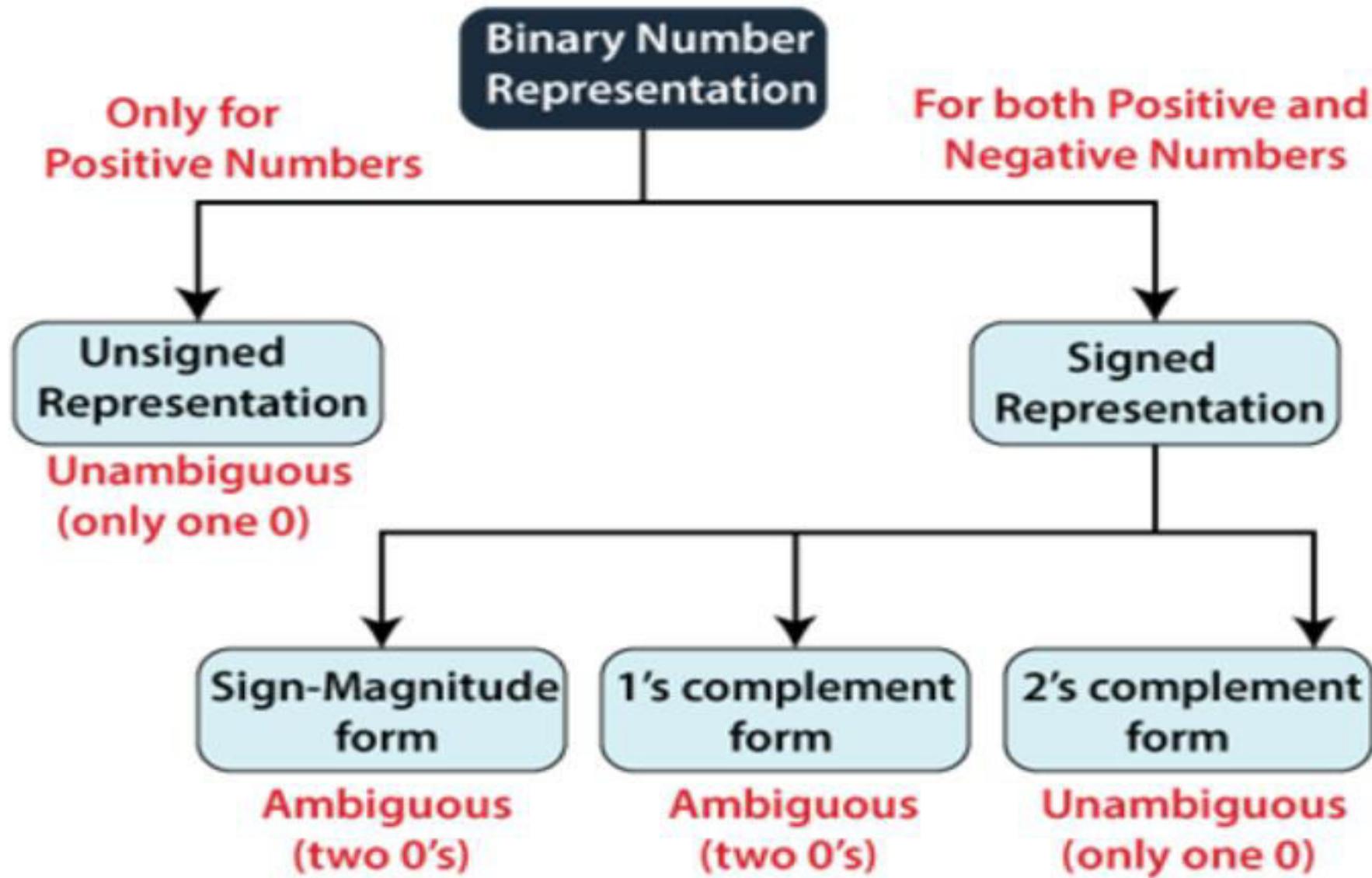


Program Status register

- N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 32-bit result was a one).
- Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 32-bit result was zero).
- C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.
- V: oVerflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

Common Number Systems

System	Base	Symbols	Used by humans?	Used in computers?
Decimal	10	0, 1, ... 9	Yes	No
Binary	2	0, 1	No	Yes
Octal	8	0, 1, ... 7	No	No
Hexa-decimal	16	0, 1, ... 9, A, B, ... F	No	No



Carry Flag for Unsigned Addition

28 + 6

Carry	1 1 1 0 0
	1 1 1 0 0
	+
	0 0 1 1 0
	1

1 0 0 0 1 0

$$\begin{array}{r} 28 \\ + 6 \\ \hline 2 \end{array}$$

Extra bit is
discarded.

5-bit result

$$28 + 6 = 2$$

Carry = 1

2s Complement

- Examples (assuming 8-bit binary numbers):

$$(14)_{10} = (00001110)_2 = (00001110)_{2s}$$

$$-(14)_{10} = -(00001110)_2 = (11110010)_{2s}$$

$$-(80)_{10} = -(?)_2 = (?)_{2s} = (10110000)_{2s}$$

2's Complement – Signed Numbers

$$\begin{array}{r} 5 \quad 0101 \\ + 2 \quad 0010 \\ \hline + 7 \quad 0111 \end{array}$$

$$\begin{array}{r} 5 \quad 0101 \\ + 4 \quad 0100 \\ \hline + 9 \quad 1001 \end{array}$$

Cin = 1
Cout = 0

Here Overflow
bit is set

Cin != Cout

$$\begin{array}{r} 2 \quad 0010 \\ - 5 \quad 1011 \\ \hline - 3 \quad 1101 \\ -3 \quad -0011 \end{array}$$

Cin = Cout , So
no overflow

$$\begin{array}{r} -5 \quad 1011 \\ - 4 \quad 1100 \\ \hline -9 \quad 10111 \end{array}$$

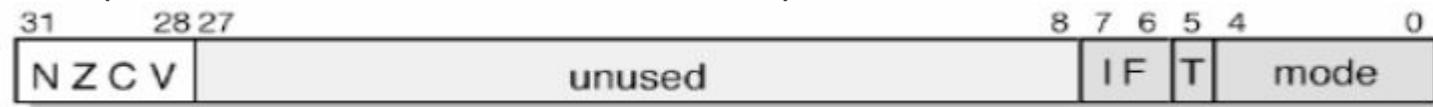
Cin != Cout , So
overflow is set
and carry flag
is set

Overflow occurs when

- Two negative numbers are added and an answer comes positive or
 - Two positive numbers are added and an answer comes as negative.
-
- So overflow can be detected by checking Most Significant Bit(MSB) of two operands
 - Overflow can also be detected using 2 Bit Comparator just by checking Carry-in(C-in) and Carry-Out(C-out) from MSB's.

Program Status register

- **Current Program Status Register (CPSR):**
 - Monitors and controls internal operations
- **N** – Negative / sign (1 = negative & 0 = positive)
- **Z** – Zero (1 = zero & 0 = non zero)
- **C** – Carry (1 = carry occurred & 0 = no carry) – **for unsigned operations**
- **V** – Overflow (1 = overflow occurred & 0 = no overflow) – **for signed operations**
- **Mode:** Indicates which mode the processor is in
 - **I** - IRQ Enable (1 = disable & 0 = enable)
 - **F** – FIQ Enable (1 = disable & 0 = enable)
 - **T** – Thumb Mode (1 = thumb mode & 0 – ARM mode)



Processor Modes

CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

Program Counter (r15)

- A register in the control unit of the CPU that is used to keep track of the address of the current or next instruction.
- When the processor is executing in ARM state:
 - All instructions are 32 bits wide
 - All instructions must be word aligned
 - Therefore the **pc** value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned).
- When the processor is executing in Thumb state:
 - All instructions are 16 bits wide
 - All instructions must be halfword aligned
 - Therefore the **pc** value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned).
- When the processor is executing in Jazelle state:
 - All instructions are 8 bits wide
 - Processor performs a word access to read 4 instructions at once

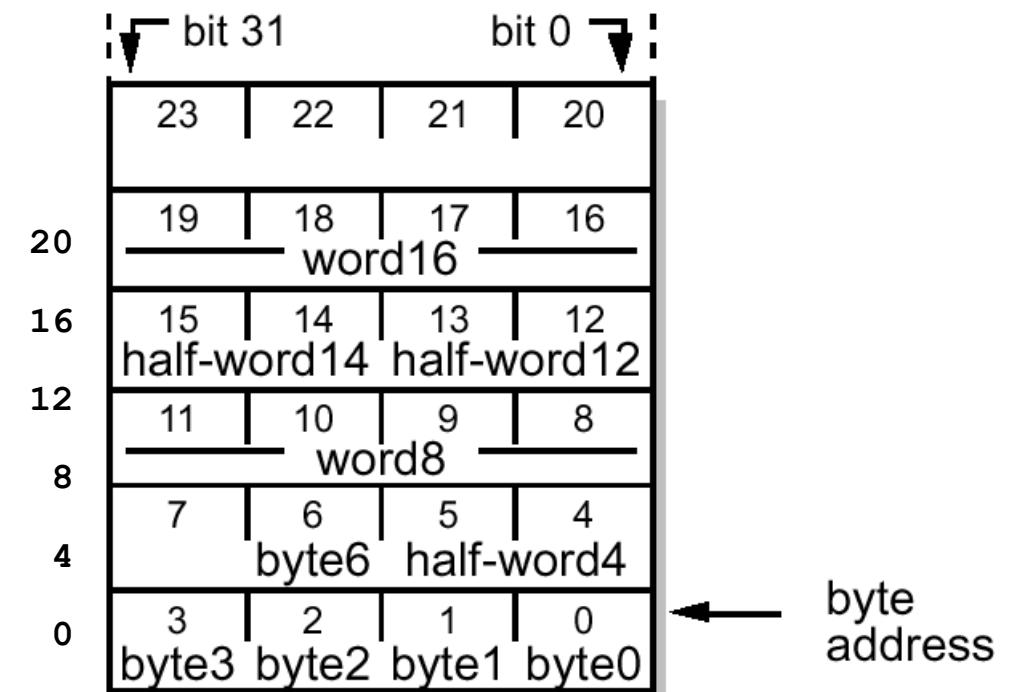
ARM's Memory Organization

- Byte addressed memory
- Maximum 2^{32} bytes of memory
- A word = 32-bits, half-word = 16 bits
- Words aligned on 4-byte boundaries

NB - Lowest byte address
= LSB of word

“Little-endian”

Word addresses follow
LSB byte address



Thank You



Amrita School Of Engineering, Bangalore Campus

ARM Assembly Language Programming

PREVIOUSLY

- Programmer's Model
- Instruction Set

TODAY...

- Data processing instruction

DATA PROCESSING INSTRUCTION

- Basically operations are arithmetic, logical or movement of data.
- All operands are 32 bit wide
- If any result is obtained that is 32 bit wide stored in destination register.
- 3 address instruction format.

ADDITION

- ADD r0,r1,r2 ; $r0 = r1 + r2$
- ADC r0,r1,r2 ; $r0 = r1 + r2 + C$ (where is C??)
- SUB r0,r1,r2 ; $r0 = r1 - r2$
- SBC r0,r1,r2 ; $r0 = r1 - r2 + C - 1$ ($B=\sim C$)
- RSB r0,r1,r2 ; $r0 = r2 - r1;$
- RSC r0,r1,r2 ; $r0 = r2 - r1 + C - 1$

BIT WISE LOGICAL OPERATION

- AND r0,r1,r2 ;r0 = r1 & r2
- ORR r0,r1,r2 ;r0 = r1 | r2
- EOR r0,r1,r2 ;r0 = r1 ^ r2
- BIC r0,r1,r2 ;r0 = r1 &(\sim r2)
- BIC – Bit Clear

REGISTER MOVEMENT OPERATIONS

- MOV r0,r1 ; $r_0 \leftarrow r_1$
- MVN r0,r1; $r_0 \leftarrow (\sim r_1)$
 - Move negated

COMPARISON OPERATION

- These instruction does not produce result instead they set conditional codes(N, Z, C, V)
- CMP r1,r2;set cc on $r1 - r2$
- CMN r1,r2 ;set cc on $r1 + r2$
- TST r1,r2 ;set cc on $r1 \& r2$ (bit test)
- TEQ r1,r2 ;set cc on $r1 \wedge r2$ (test equal)

IMMEDIATE OPERANDS

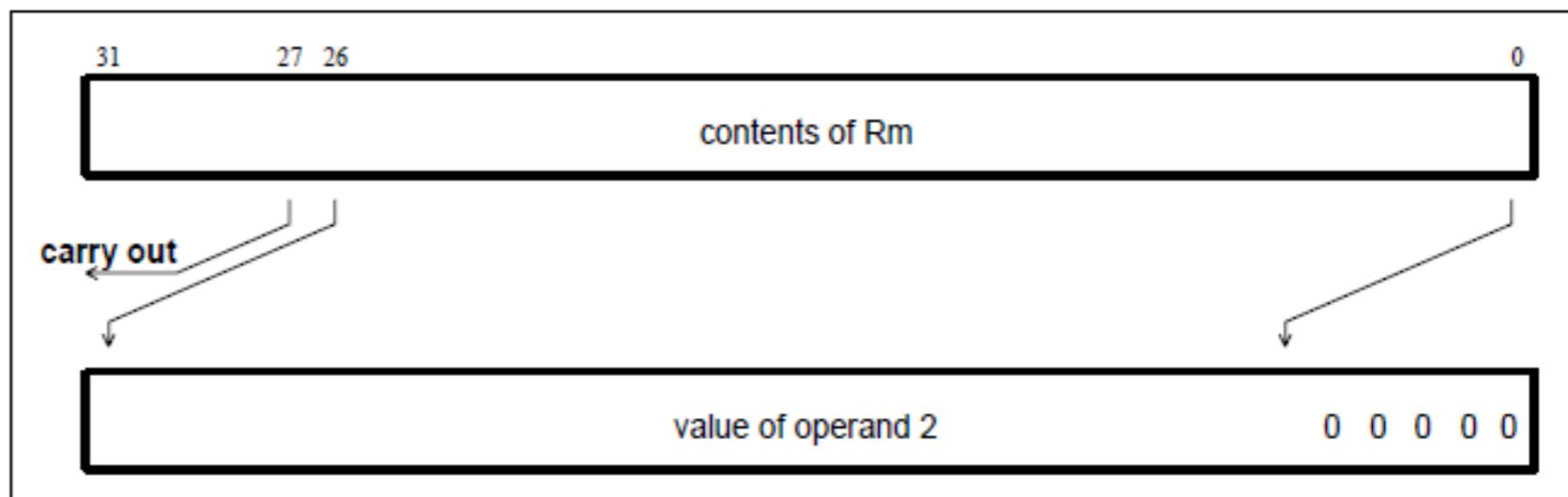
- ADD r3,r3,#1
- AND r8,r7,# &FF
- # notation used for immediate field
- & notation is used for representing hexadecimal numbers
- How many bit immediate possible ??
 - Covered later...

SHIFTED REGISTER OPERANDS

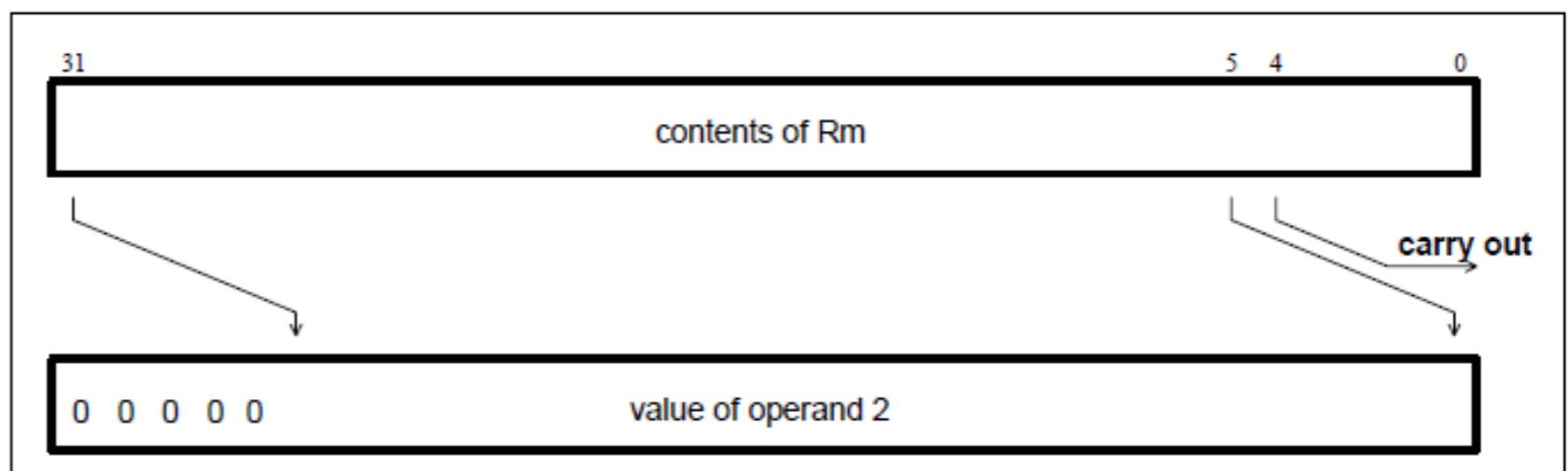
- ADD r3,r2,r1,LSL #3 ; $r3 = r2 + (8 \times r1)$
- Various shifting operations are
 - LSL – logical shift left
 - LSR – logical shift right
 - ASL – Arithmetic shift left = LSL
 - ASR – Arithmetic shift right
 - ROR – Rotate right
 - RRX – Rotate right extended by 1 place
- Register values can be used to specify the number of bits to be shifted
 - ADD r5,r5,r3, LSL r2; $r5 = r5 + r3 \times (2^r2)$

SHIFTING

- Logical Shift Left

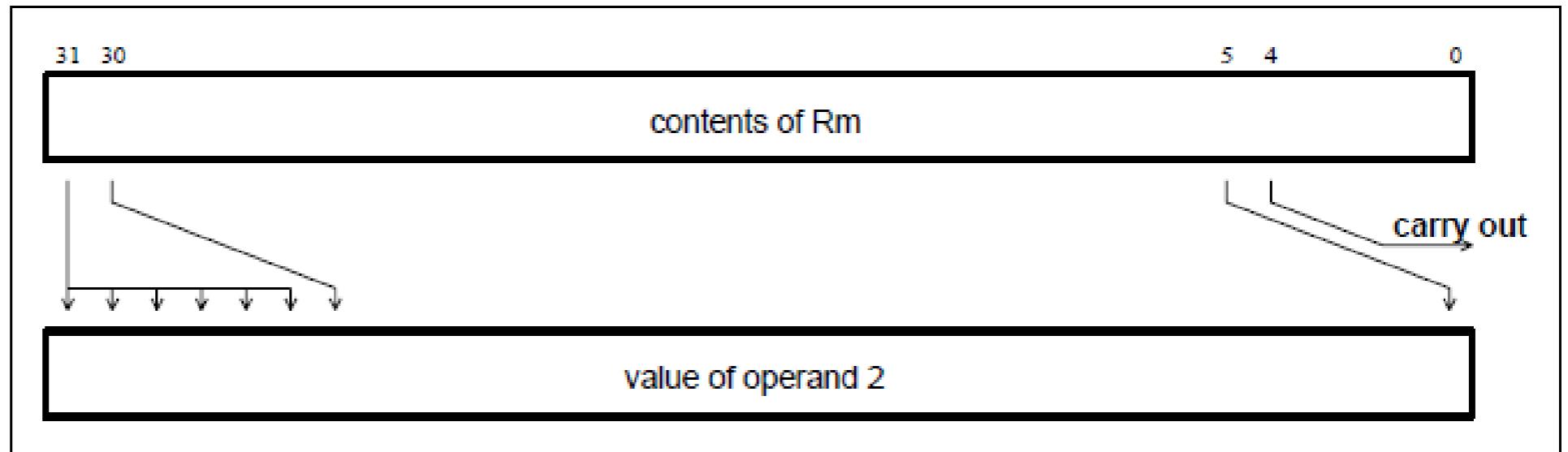


- Logical Shift Right



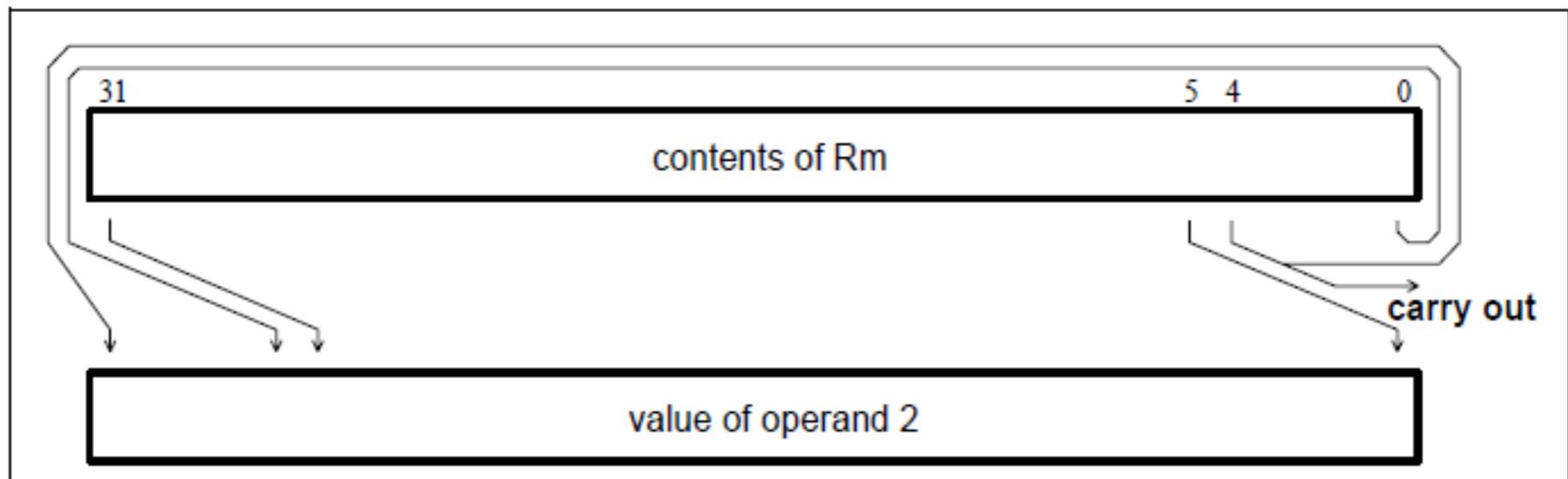
CONTD...

> Arithmetic Shift Right

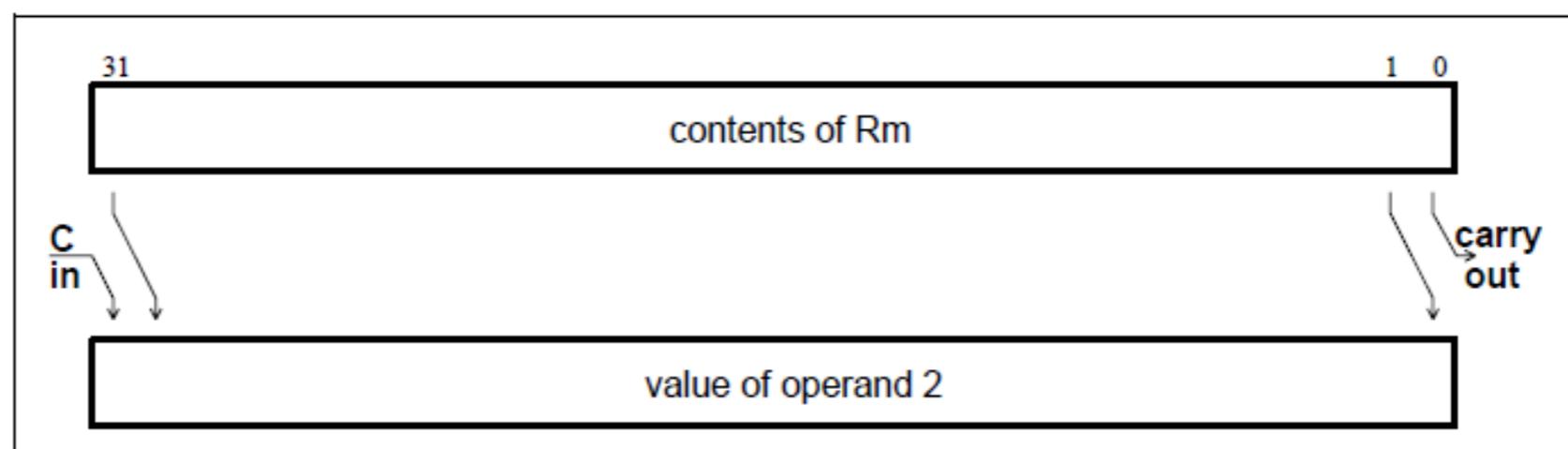


CONTD...

➤ Rotate Right



➤ Rotate Right Extended



SETTING THE CONDITION CODES

- Any data processing instruction can set the condition codes if programmer wishes to
- Just add '**S**' in the instruction
- **S** stands for set condition code
- ADD**S** r2,r2,r0
- Arithmetic instruction set – N,Z,C,V
- Logical and move instruction set – N,Z

CONDITION EXECUTION

- An unusual feature for ARM instruction is that every instruction can be executed conditionally
- Similar to conditional branches ARM extends this to all its instruction
- To execute an instruction conditionally, simply postfix it with the appropriate condition

CONTD...

- 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
  
```

← decrement r1 and set flags

← if Z flag clear then branch

COND FIELD

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Zset
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	Cset
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	Nset
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	Vset
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE-	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

COND FIELD

- NV: The 'never' condition (NV) should not be used - there are plenty of other ways to write no-ops (instructions that have no effect on the processor state) in ARM code.
- The reason to avoid the 'never' condition is that (**read the book page no. 125 just before table 5.3**)

COND FIELD

- Alternative mnemonics: There is more than one way to interpret the condition field.
- CS or HS: Both cause the instruction to be executed only if the C bit in the CPSR is set.
 - The alternatives are available because the same test is used in different circumstances. If you have previously added two unsigned integers and want to test whether there was a carry-out from the addition, you should use CS.
 - If you have compared two unsigned integers and want to test whether the first was higher or the same as the second, use HS.
- The alternative mnemonic removes the need for the programmer to remember that an unsigned comparison sets the carry on higher or the same.

EXAMPLES OF CONDITIONAL EXECUTION

➤ Use a sequence of several conditional instructions

if (a==0) func(1);

➤ Set the flags, then use various condition codes

if (a==0) x=0;

if (a>0) x=1;

➤ Use conditional compare instructions

if (a==4 || a==10) x=0;

CMP r0,#0

MOVEQ r0,#1

BLEQ func

CMP r0,#0

MOVEQ r1,#0

MOVG T r1,#1

CMP r0,#4

CMPNE r0,#10

MOVEQ r1,#0

EXAMPLE: CONDITIONAL INSTRUCTION

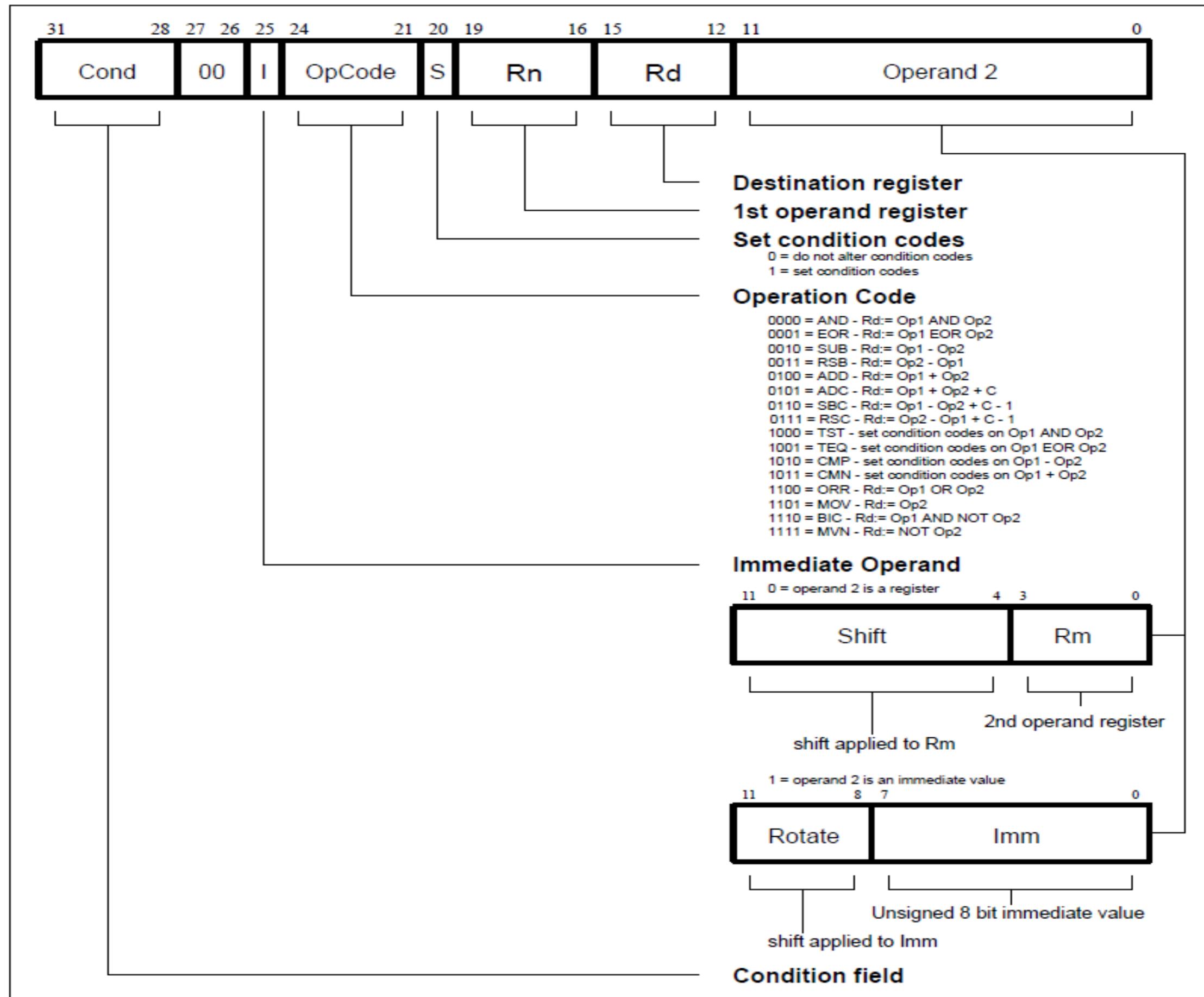
(Later)

- true block
- MOVLT r0,#5 ; generate value for x
- ADRLT r4,x ; get address for x
- STRLT r0,[r4] ; store x
- ADRLT r4,c ; get address for c
- LDRLT r0,[r4] ; get value of c
- ADRLT r4,d ; get address for d
- LDRLT r1,[r4] ; get value of d
- ADDLT r0,r0,r1 ; compute y
- ADRLT r4,y ; get address for y
- STRLT r0,[r4] ; store y

We will revisit this slide.

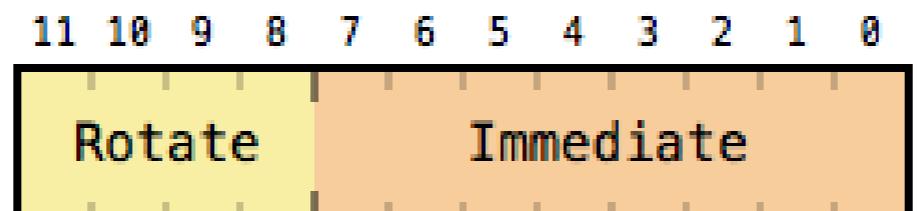


INSTRUCTION ENCODING

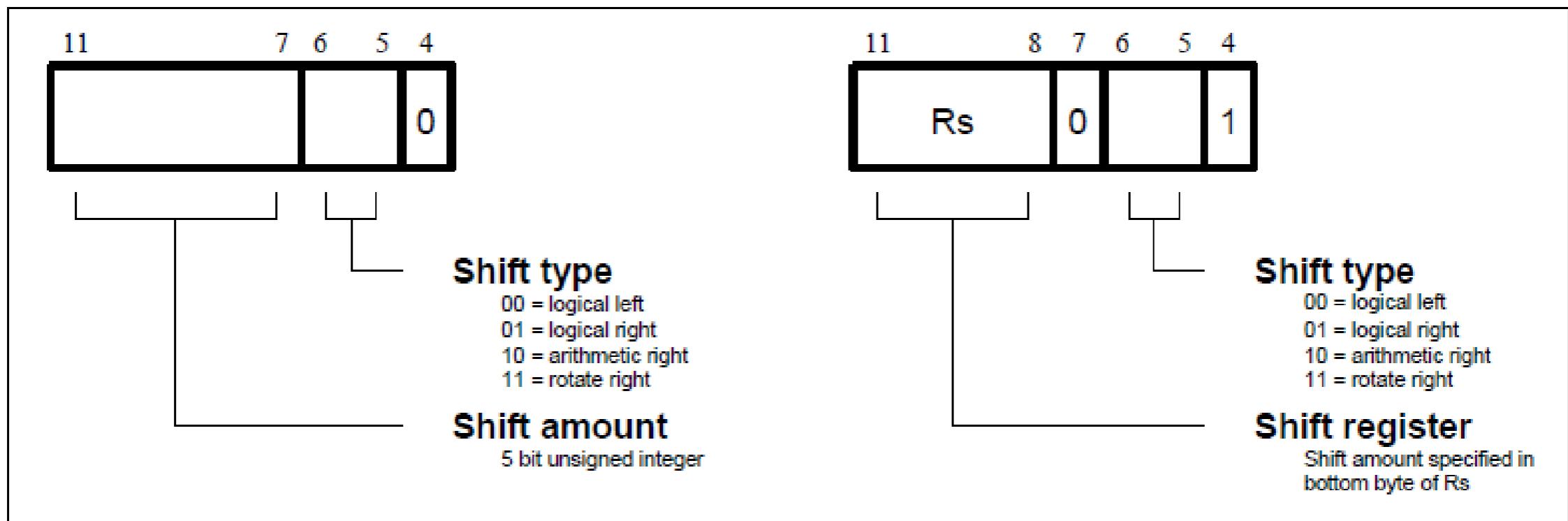


ARM IMMEDIATE VALUE ENCODING

- The [ARM instruction set](#) encodes immediate values in an unusual way.
- Despite only using 12 bits of instruction space, the immediate value can represent a useful set of 32-bit constants.
- ARM doesn't use the 12-bit immediate value as a 12-bit number. Instead, it's an 8-bit number with a [4-bit rotation](#), like this:



SHIFTING (When second operand is register)



SHIFT AMOUNT

- Only the least significant byte of the contents of Rs is used to determine the shift amount. Rs can be any general register other than R15.
- If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output.
- If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

SPECIAL CASE

- LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.
- LSR #0 & ASR #0 is Equated to LSL #0 by assembler
- If shift value is greater than or equal to 32 ??

CONTD...

- LSL by 32 has result zero, carry out equal to bit 0 of Rm.
- LSL by more than 32 has result zero, carry out zero.
- LSR by 32 has result zero, carry out equal to bit 31 of Rm.
- LSR by more than 32 has result zero, carry out zero.
- ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.

CONTD...

- ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.
- ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

(take a small example and work it out as HW)

MULTIPLY

- Difference from other Arithmetic instruction
 - Immediate second operand are not supported
 - Result register must not be same as first source register
 - Multiply give 64bit result, least 32 bit is stored in result register rest is ignored
 - However multiply long instruction can store all 64 bit result.

CONTD...

- The Basic ARM provides two multiplication instructions.
- Multiply
 - $\text{MUL}\{<\text{cond}>\}\{S\} \text{ Rd, Rm, Rs ; Rd} = \text{Rm} * \text{Rs}$
- Multiply Accumulate - does addition for free
 - $\text{MLA}\{<\text{cond}>\}\{S\} \text{ Rd, Rm, Rs,Rn ; Rd} = (\text{Rm} * \text{Rs}) + \text{Rn}$

MULTIPLY-LONG AND MULTIPLY-ACCUMULATE LONG

- Instructions are
 - MULL which gives $RdHi, RdLo := Rm * Rs$
 - MLAL which gives $RdHi, RdLo := (Rm * Rs) + RdHi, RdLo$
- However the full 64 bit of the result now matter (lower precision multiply instructions simply throws top 32bits away)
 - Need to specify whether operands are signed or unsigned
- Therefore syntax of new instructions are:
 - UMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
 - UMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs
 - SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
 - SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs

EXAMPLE

- Example (Multiply, Multiply-Accumulate)

MUL r4, r3, r2	$r4 := [r3 \times r2]_{<31:0>}$
MLA r4, r3, r2, r1	$r4 := [r3 \times r2 + r1]_{<31:0>}$

- Note
 - least significant 32-bits are placed in the result register, the rest are ignored
 - immediate second operand is not supported
 - result register must not be the same as the first source register
 - if 'S' bit is set the V is preserved and the C is rendered meaningless
- Example ($r0 = r0 \times 35$)
 -

EXAMPLE (quick Quiz)

Can you do r0 = 5 r0

Can you do r0 = 7 r0

ADD r0, r0, r0, LSL #2 :=5xr0

RSB r0, r0, r0, LSL #3 := 7xr0

SUMMARY

- MUL r4,r3,r2 ; $r4 = r3 \times r2 [31:0]$
- MLA r4,r3,r2,r1; $r4 = r3 \times r2 + r1 [31:0]$
- UMULL; $rdHi:rdLo = Rm \times Rs$
- UMLAL; $rdHi:rdLo += Rm \times Rs$
- SMULL; $rdHi:rdLo = Rm \times Rs$
- SMLAL; $rdHi:rdLo += Rm \times Rs$



THANK YOU



Amrita School Of Engineering, Bangalore Campus

INSTRUCTION SET LECTURE - 4



PREVIOUSLY

- > Data Processing Instructions



TODAY...

- Data Processing Instructions – Contd...



CPSR FLAGS

- Carry/~Borrow
 - In unsigned arithmetic, watch the carry flag to detect errors.
 - Bit goes from MSB
- Overflow
 - In signed arithmetic, watch the overflow flag to detect errors.
 - Will be set if an overflow occurs into bit 31 of the result
- Negative
 - Set to the logical value of bit 31 of the result
- Zero
 - Sets only if result is zero



CONTD...

- Logical Instructions
 - V Flag unaffected
 - C flag will be set to the carry out from the barrel shifter
 - Z flag will be set if and only if the result is all zeros
 - N flag will be set to the logical value of bit 31 of the result.



CONTD...

- Arithmetic Instructions
 - V/C flag based on Signed/Unsigned
 - Z flag will be set if and only if the result was zero
 - N flag will be set to the value of bit 31 of the result



FLAGS FOR MUL INSTRUCTIONS

- The N (Negative) and Z (Zero) flags are set correctly on the result
 - N is made equal to bit 31 of the result
 - Z is set if and only if the result is zero.
- The C (Carry) flag is set to a meaningless value
- V (oVerflow) flag is unaffected.



INSTRUCTION FORMAT

- Ex: ADD r5,r1,r2;

14	0	0	4	0	1	5	2
4-bits Cond	2 bits F	1 bit I	4 bits Opcode	1 bit S	4 bits Rn	4 bits Rd	12 bits Operand2

- Operand2 second source operand
- I If 0, second source is a register else second source is 12-bit imm.
- S set cond code
- F Instrn format, 0 = data processing instrn format



CONTD...

- What ARM instruction does this represent?

14	0	0	4	0	0	1	2
➤ 4-bits Cond	2 bits F	1 bit I	4 bits Opcode	1 bit S	4 bits Rn	4 bits Rd	12 bits Operand2

ADD R1, R0, R2



LOGICAL OPERATIONS

- AND ; &
- ORR ; |
- MVN ; ~ MOV
- LSL ; <<
- LSR ; >>



CONTD...

- Shift the second operand:
- ADD r5, r1, r2, LSL #2;
- $r5 = r1 + (r2 \ll 2)$



CONTD...

- Shift right r5 by 4-bits and place the result in r6?

MOV r6,r5, LSR #4;

MOV r6,r5, LSR r3;

- $r6 = r5 \gg r3$



CONTD...

➤ Instruction format??

31-28	27-26	25	24-21	20	19-16	15-12	11-0
4-bits Cond	2 bits F	1 bit I	4 bits Opcode	1 bit S	4 bits Rn	4 bits Rd	12 bits Operand2

➤ 12-bits operand2 field is interpreted as:

11	10	9	8	7	6	5	4	3	2	1	0
Shift_imm					Shift	0	Rm				
Rs			0	Shift	1	Rm					

Shift = Type of shift

4th bit = 0 Shift_amt = Imme

4th bit = 1 Shift_amt = Rs



SUMMARY

Opcode 124:21)	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	$Rd := Rn \text{ AND } Op2$
0001	EOR	Logical bit-wise exclusive OR	$Rd := Rn \text{ EOR } Op2$
0010	SUB	Subtract	$Rd := Rn - Op2$
0011	RSB	Reverse subtract	$Rd := Op2 - Rn$
0100	ADD	Add	$Rd := Rn + Op2$
0101	ADC	Add with carry	$Rd := Rn + Op2 + C$
0110	SBC	Subtract with carry	$Rd := Rn - Op2 + C - 1$
0111	RSC	Reverse subtract with carry	$Rd := Op2 - Rn + C - 1$
1000	TST	Test	$Sec on Rn \text{ AND } Op2$
1001	TEQ	Test equivalence	$Sec on Rn \text{ EOR } Op2$
1010	CMP	Compare	$Sec on Rn - Op2$
1011	CMN	Compare negated	$Sec on Rn + Op2$
1100	ORR	Logical bit-wise OR	$Rd := Rn \text{ OR } Op2$
1101	MOV	Move	$Rd := Op2$
1110	BIC	Bit clear	$Rd := Rn \text{ AND NOT } Op2$
1111	MVN	Move negated	$Rd := \text{NOT } Op2$



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 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).

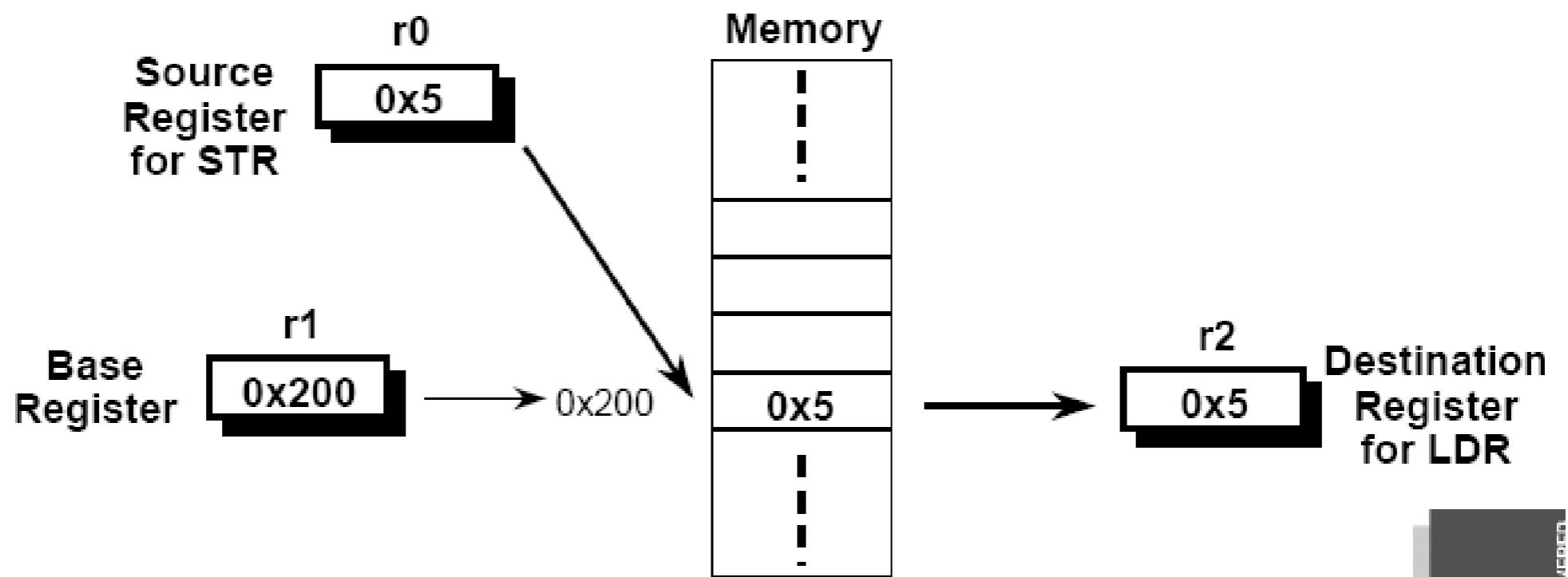


SINGLE REGISTER DATA TRANSFER

- 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
 - STRSB/STRSH is not available.
- 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.





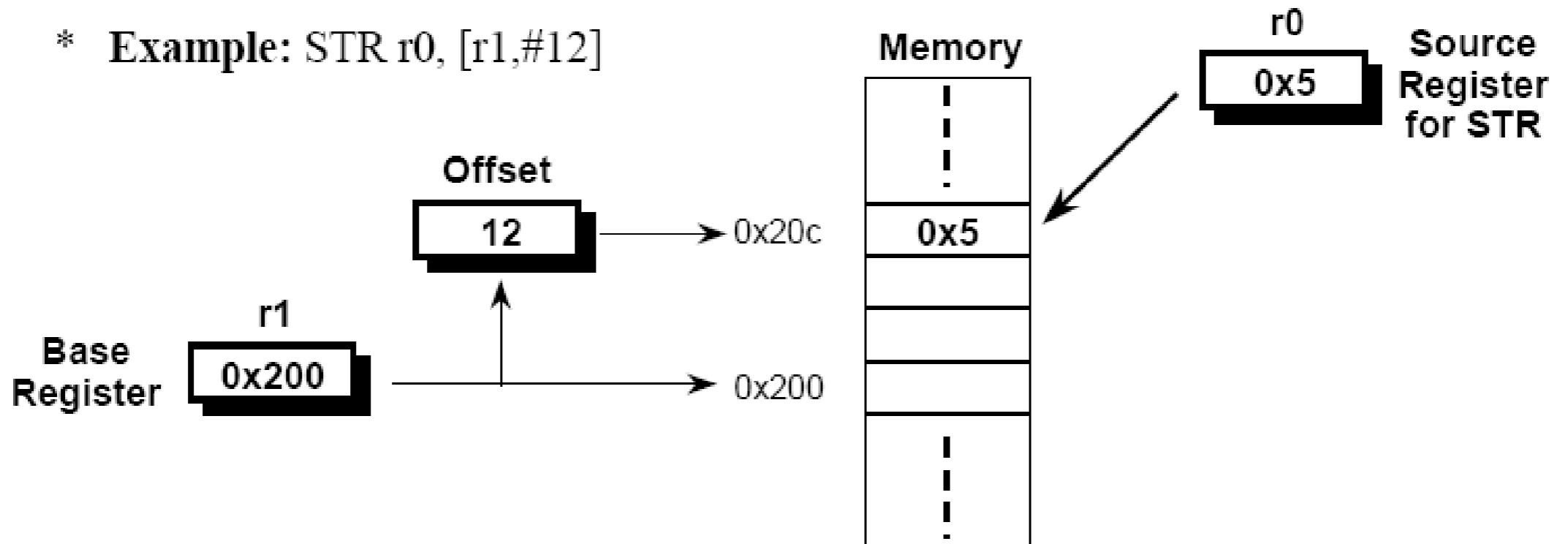
LOAD AND STORE WORD OR BYTE OFFSETS FROM THE BASE REGISTER

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 AND STORE WORD OR BYTE:PRE-INDEXED ADDRESSING

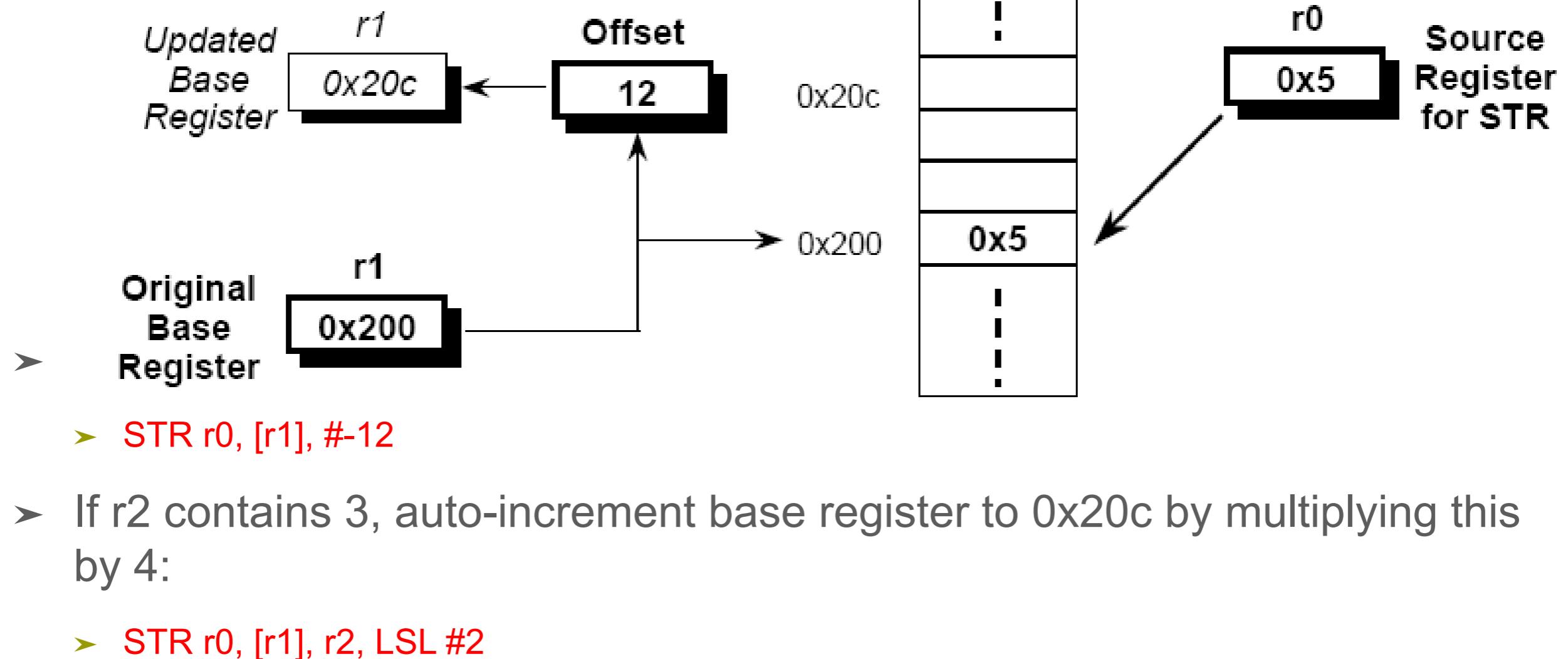
* Example: STR r0, [r1,#12]



- 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





INSTRUCTION ENCODING

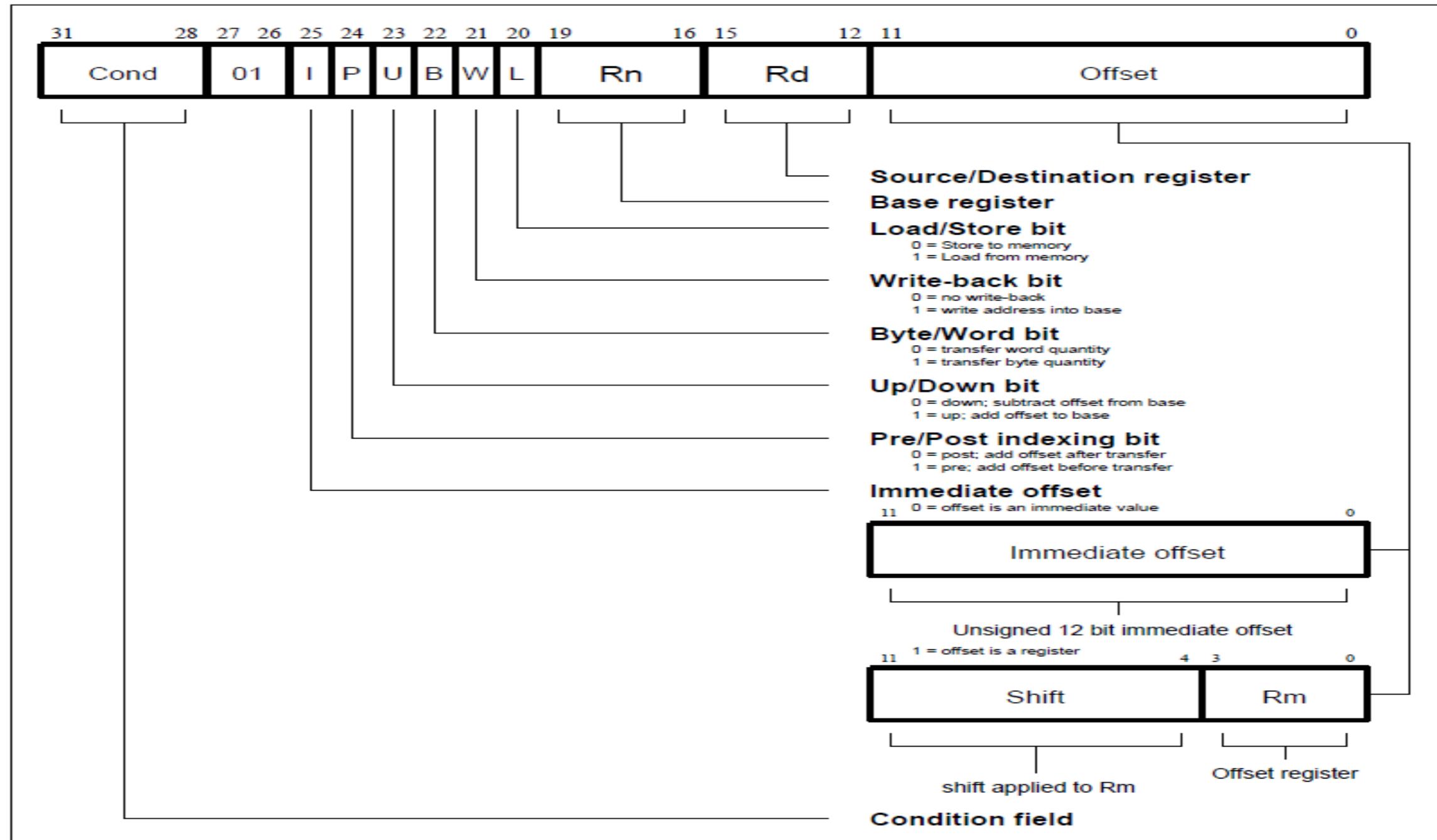


Figure 4-14: Single data transfer instructions



USE OF PC

- Write-back must not be specified if R15 is specified as the base register (Rn).
- R15 must not be specified as the register offset (Rm).



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.



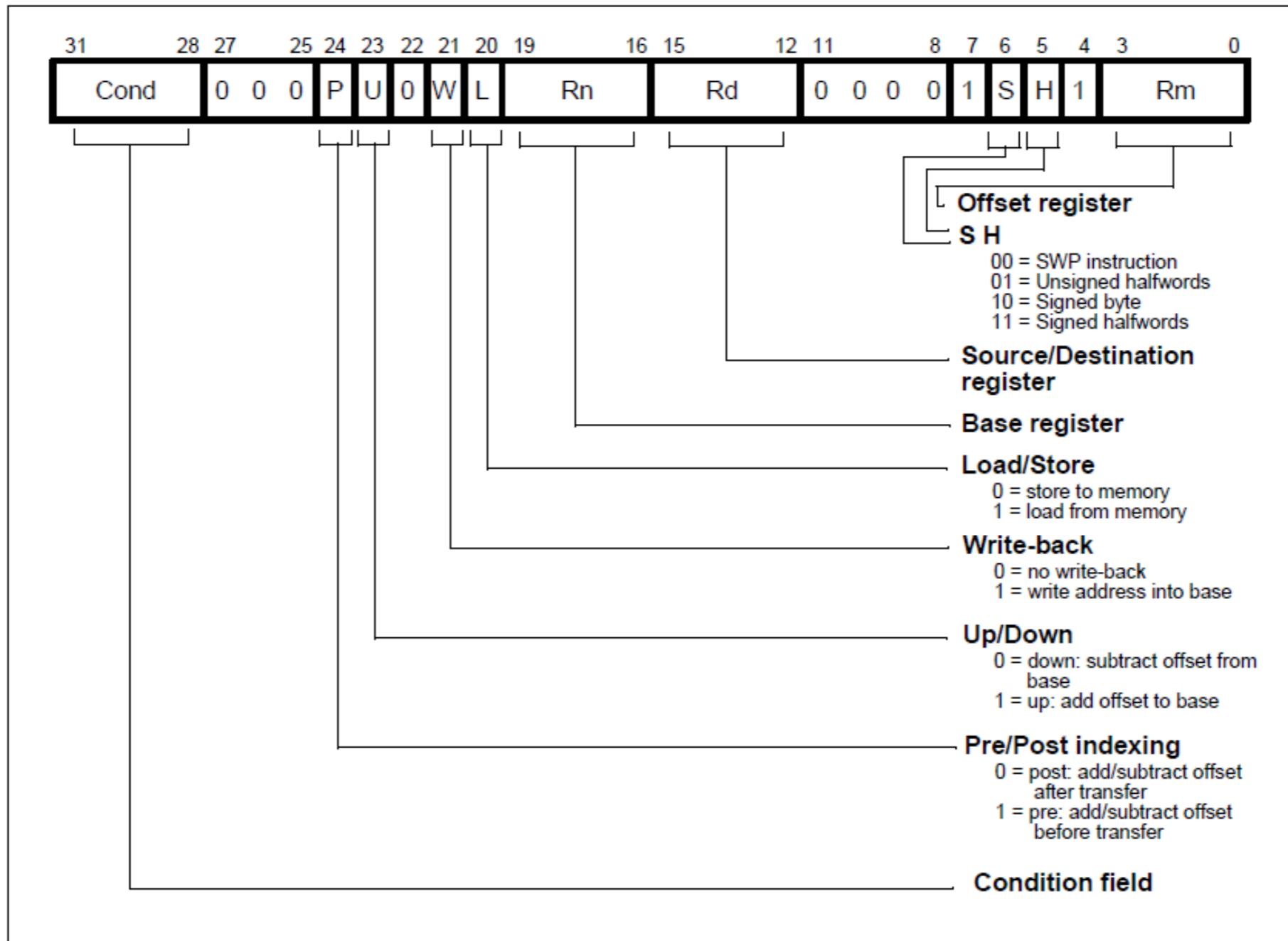
OFFSETS FOR HALFWORD AND SIGNED

HALFWORD / BYTE ACCESS

- The Load and Store Halfword and Load Signed Byte or Halfword instructions can make use of pre- and post-indexed addressing in much the same way as the basic load and store instructions.
- However the actual offset formats are more constrained:
 - The immediate value is limited to 8 bits (rather than 12 bits) giving an offset of 0-255 bytes.
 - The register form cannot have a shift applied to it.

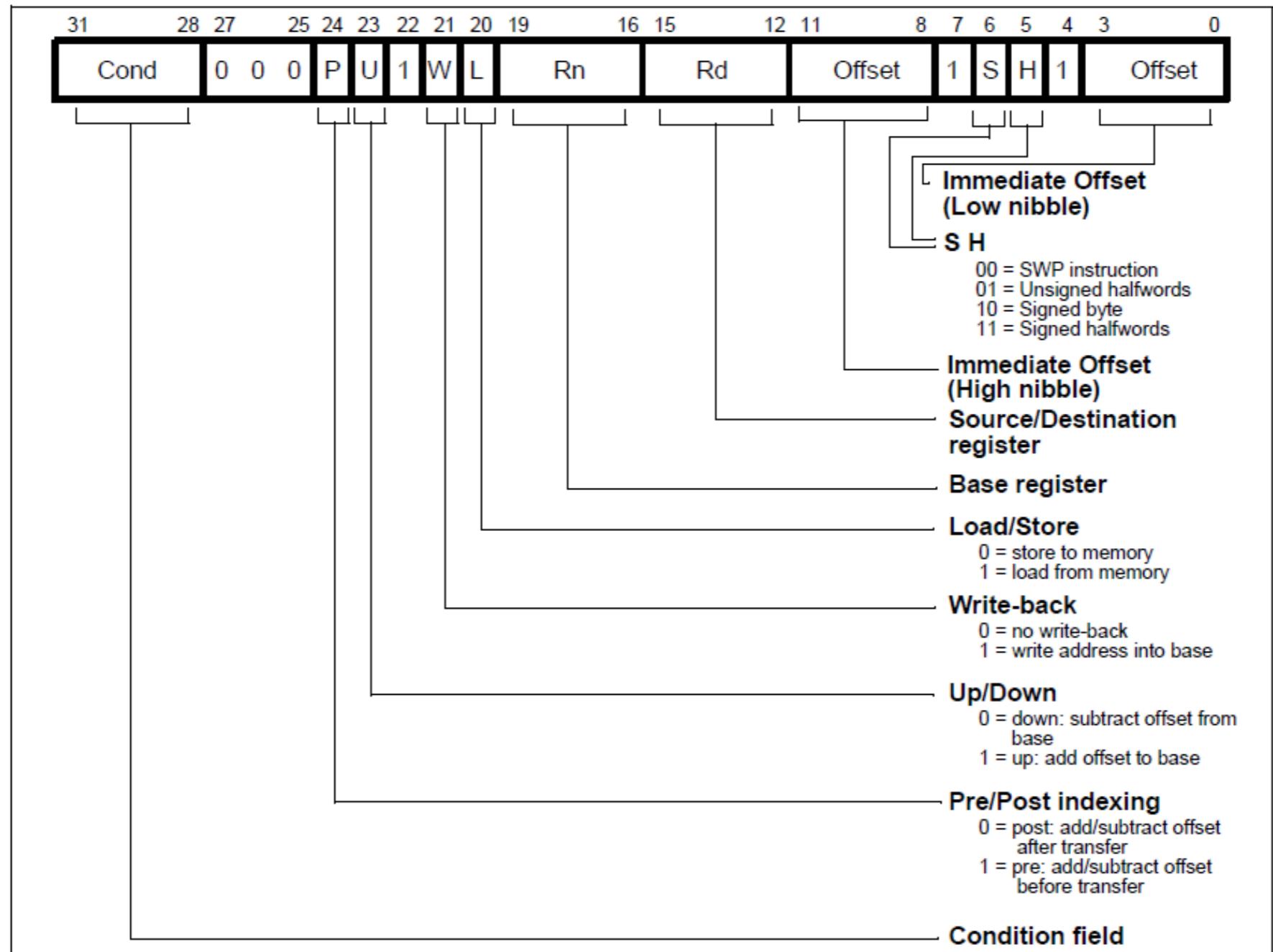


INSTRUCTION ENCODING – WITH REGISTER OFFSET





INSTRUCTION ENCODING – WITH IMMEDIATE OFFSET





BLOCK DATA TRANSFER

- 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 '^').

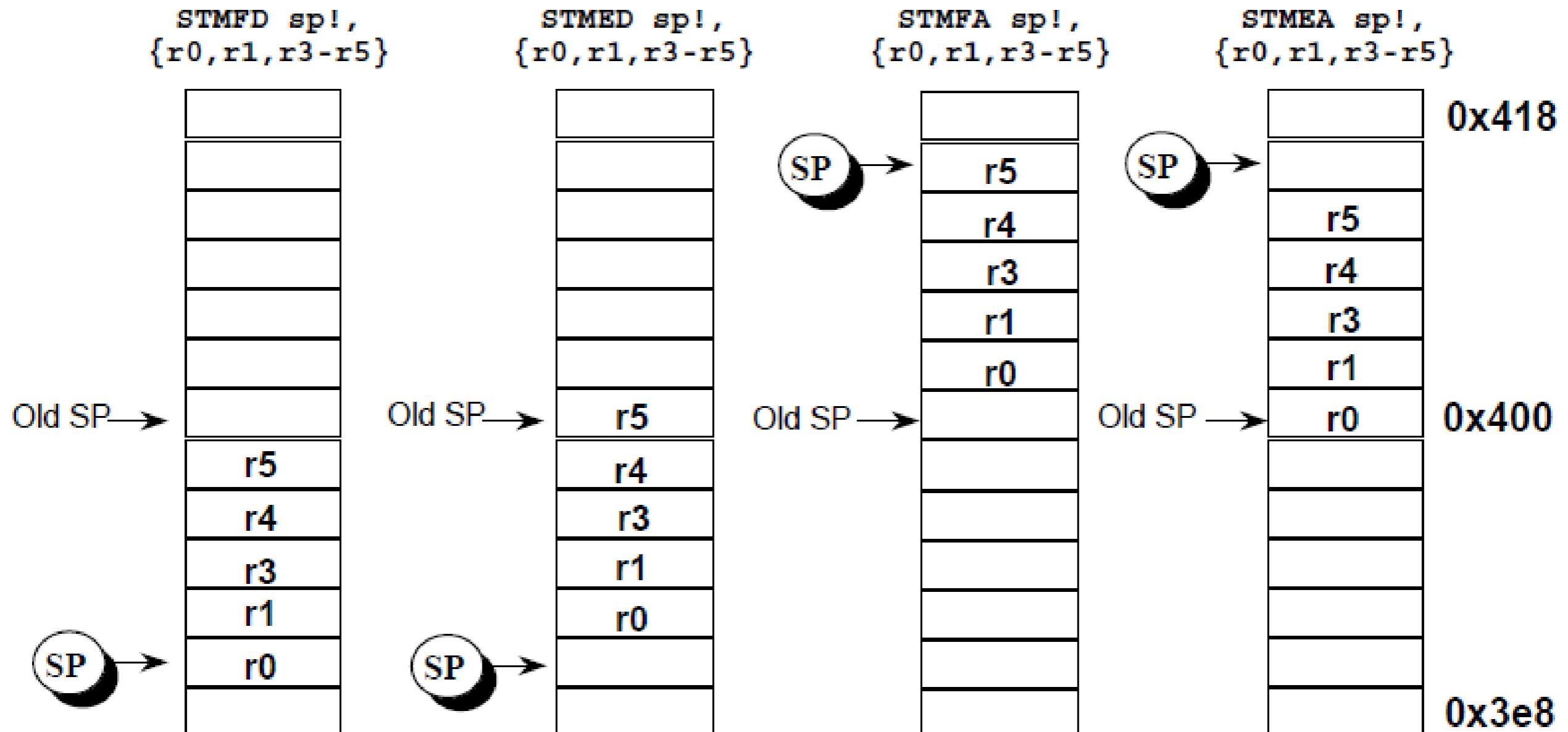


CONTD...

- 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



STACK EXAMPLES





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

- Multiple register load and store instructions
 - enable transfer of large quantities of data
 - used for procedure entry and exit, to save/restore workspace registers, to copy blocks of data around memory

Multiple register data transfers

LDMIA r1, {r0, r2, r5}	r0 := mem ₃₂ [r1] r2 := mem ₃₂ [r1 + 4] r5 := mem ₃₂ [r1 + 8]
------------------------	--

Note: any subset (or all) of the registers may be transferred with a single instruction

Note: the order of registers within the list is insignificant

Note: including r15 in the list will cause a change in the control flow



CONTD...

- 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

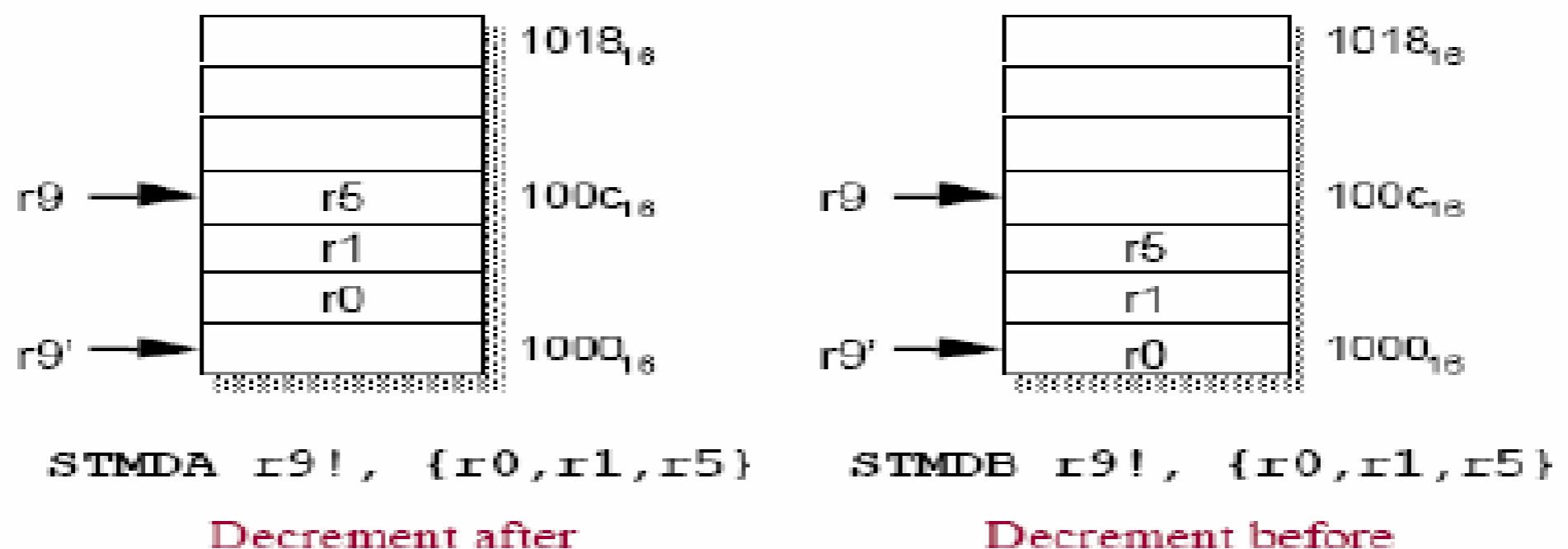
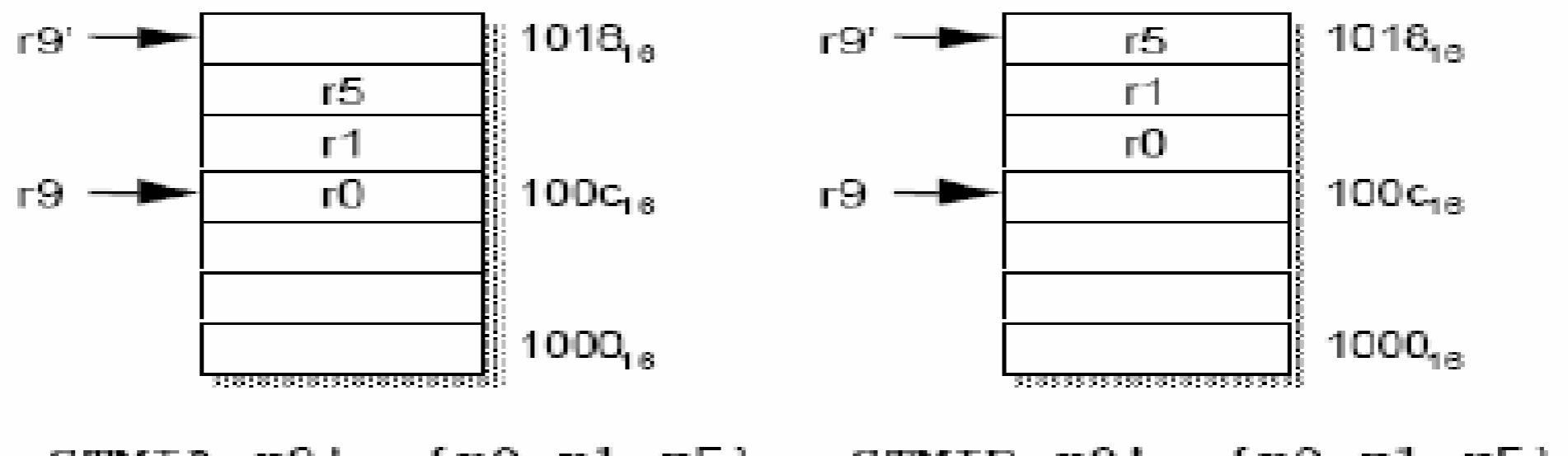
STMIA r13!, {r0-r11} ; and store them

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

MRT- ADDRESSING MODES





USE OF THE S BIT

- When the S bit is set in a LDM/STM instruction its meaning depends on whether or not R15 is in the transfer list and on the type of instruction.
- The S bit should only be set if the instruction is to execute in a privileged mode.
- **LDM with R15 in transfer list and S bit set (Mode changes)**
 - If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.



CONTD...

- **STM with R15 in transfer list and S bit set (User bank transfer)**
 - The registers transferred are taken from the User bank rather than the bank corresponding to the current mode.
 - This is useful for saving the user state on process switches.
 - Base write-back should not be used when this mechanism is employed.



CONTD...

- **R15 not in list and S bit set (User bank transfer)**
 - For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode.
 - This is useful for saving the user state on process switches.
 - Base write-back should not be used when this mechanism is employed.

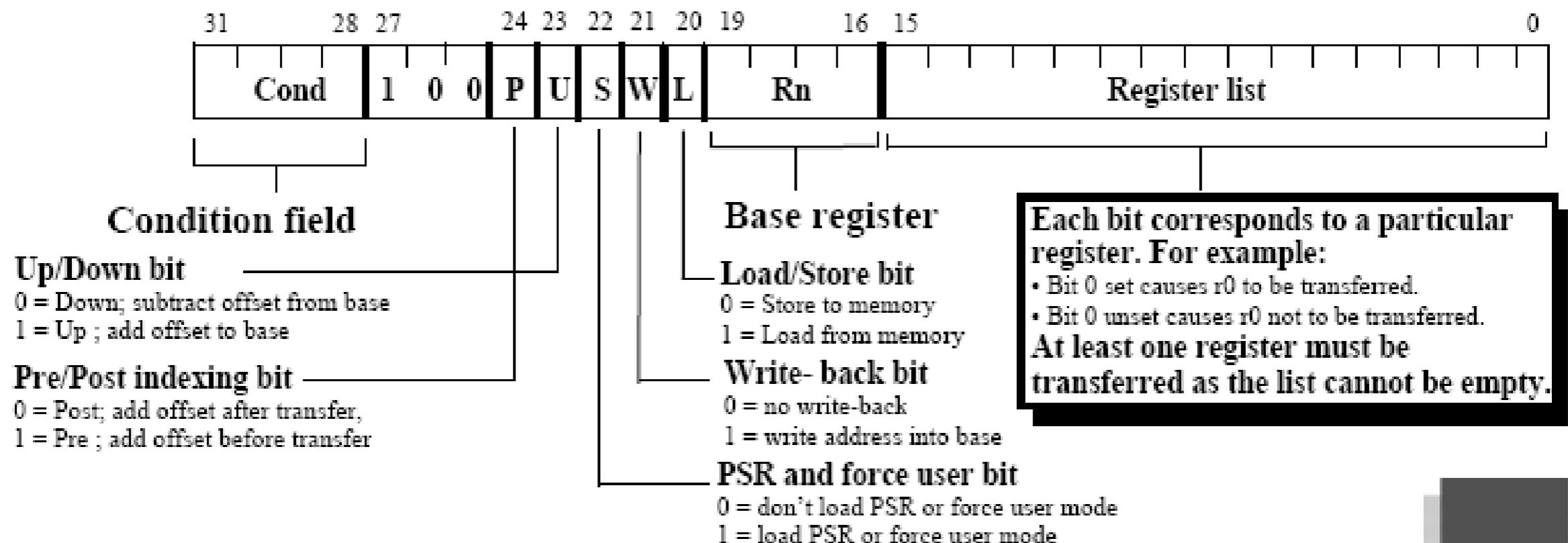


CONTD...

- 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.



INSTRUCTION FORMAT





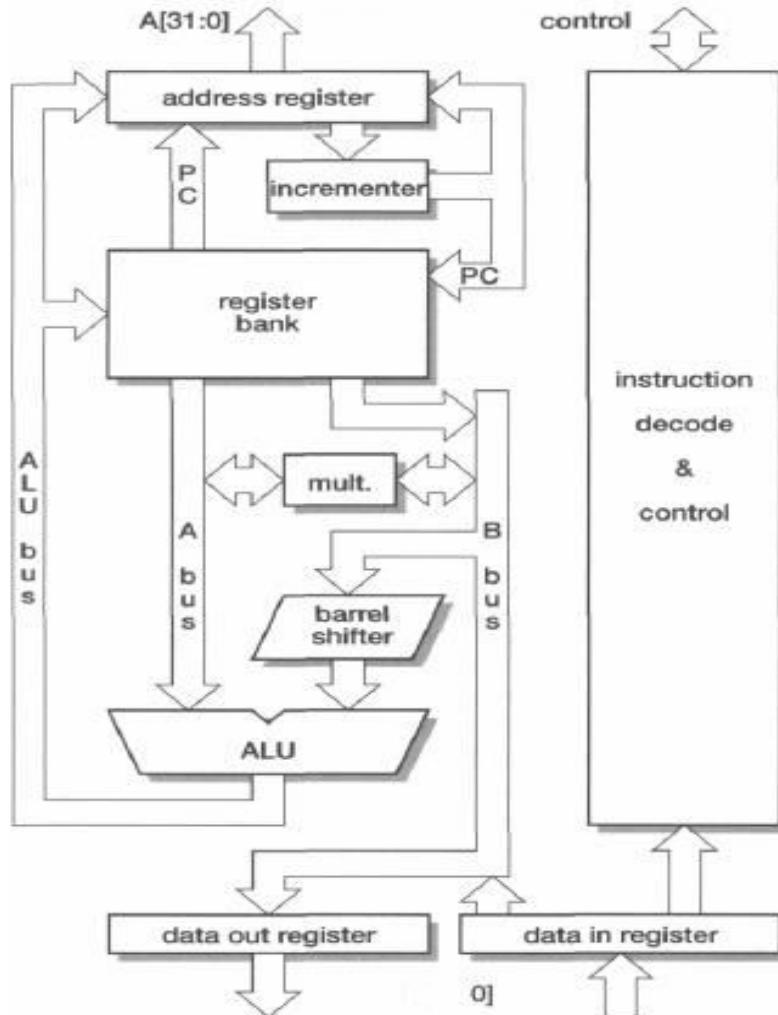
THANK YOU

Today...

- . 3 Stage Pipeline
- . 5 Stage Pipeline

3 Stage Pipeline Organization

- Principle Components
 - Register Banks
 - Ports???
 - Barrel Shifters
 - ALU
 - Address Register and Incrementer
 - Data Register
 - Instruction Decoder and Control Logic
 - Ins and data Memory???



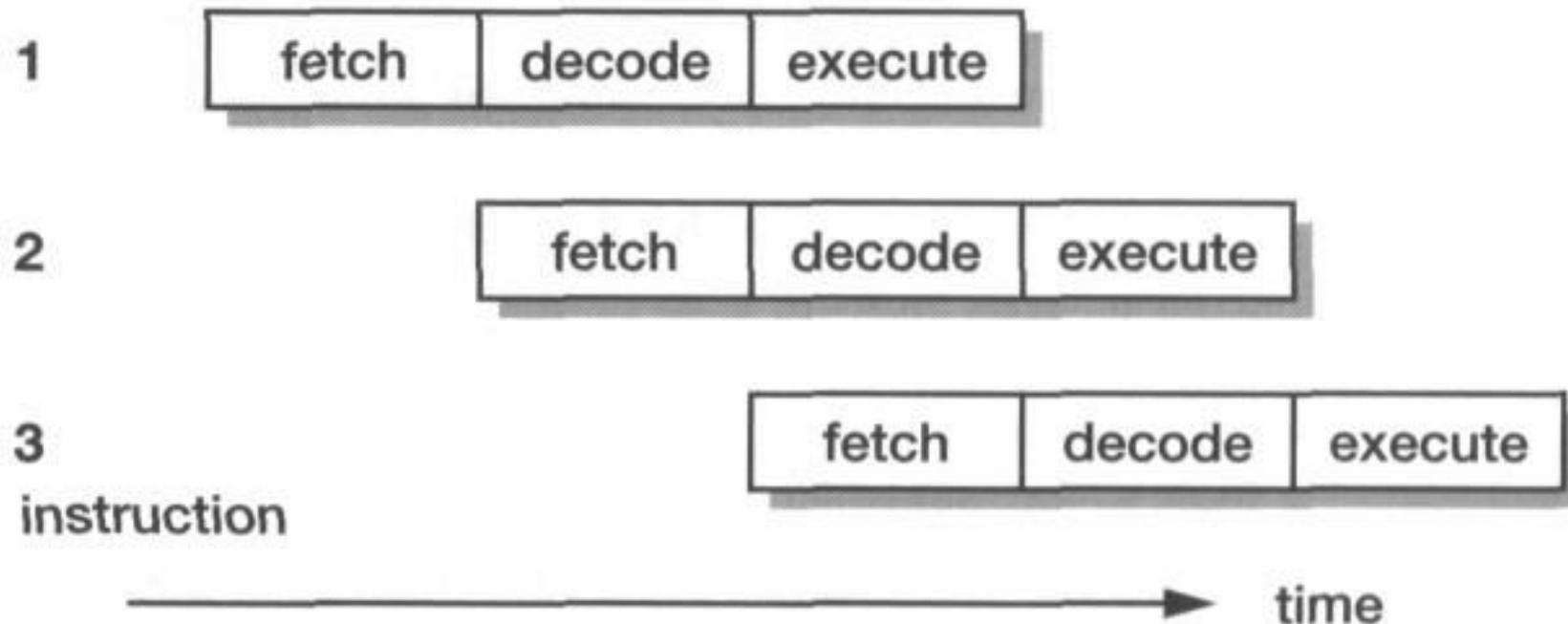
Pipeline Stages

- . **Fetch:**
 - . The instruction is fetched from memory and placed in the instruction pipeline
- . **Decode**
 - . The instruction is decoded and the datapath control signals prepared for the next cycle.
 - . In this stage the instruction 'owns' the decode logic but not the datapath.
- . **Execute**
 - . The instruction 'owns' the datapath;
 - . The register bank is read, an operand shifted, the ALU result generated and written back into a destination register.

Pipeline Stages

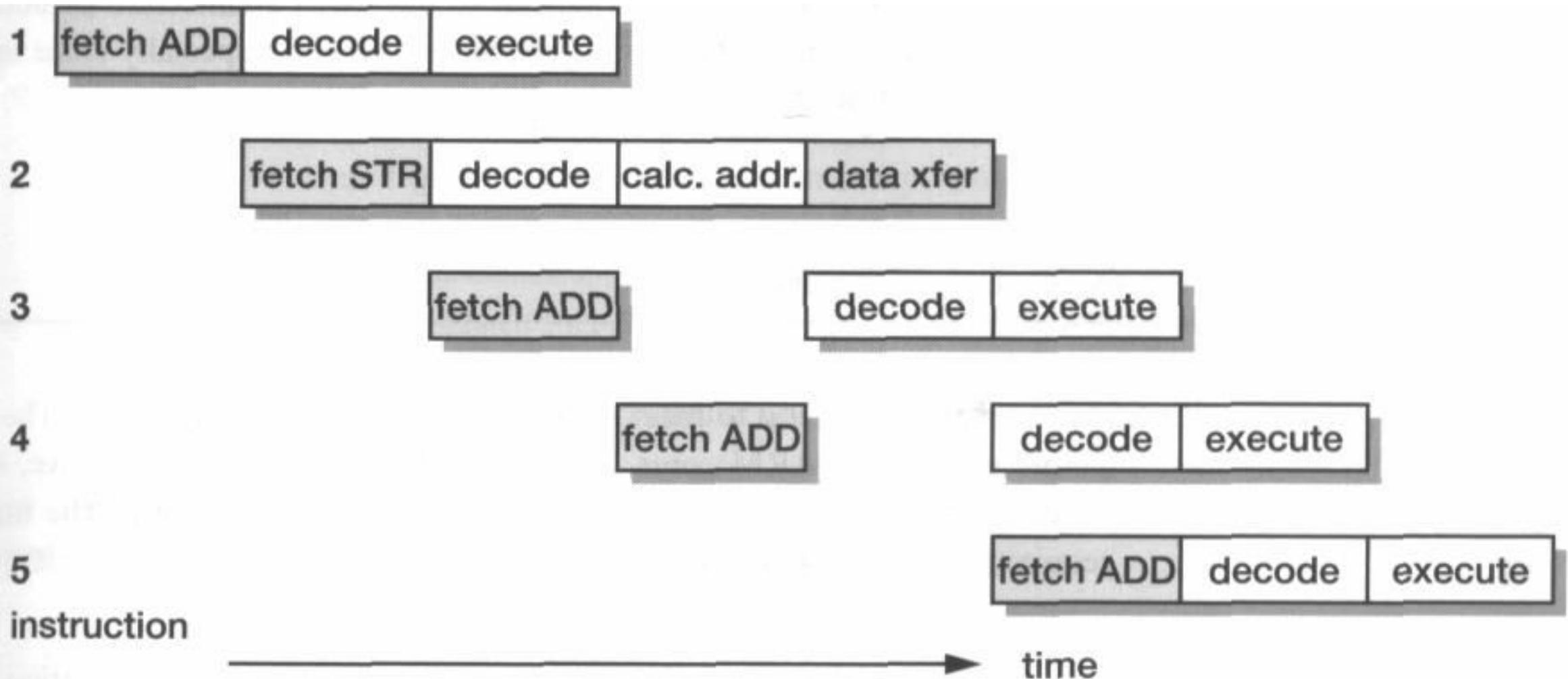
- . With simple single cycle data transfer instruction, each pipeline stage enables one instruction to be completed every clock cycle.
- . So **Latency** is **3** clock cycles
- . **Throughput** is **1** instruction per cycle.

Single Cycle Instruction



ARM single-cycle instruction 3-stage pipeline operation.

Multi-Cycle Instruction



ARM multi-cycle instruction 3-stage pipeline operation.

Multi-Cycle Instruction

- All instructions occupy the datapath for one or more adjacent cycles.
- For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.
- During the first datapath cycle each instruction issues a fetch for the next instruction but one.
- Branch instructions flush and refill the instruction pipeline.

PC??

One consequence of the pipelined execution model used on the ARM is that the program counter, which is visible to the user as r15, must run ahead of the current instruction. If, as noted above, instructions fetch the next instruction but one during their first cycle, this suggests that the PC must point eight bytes (two instructions) ahead of the current instruction.

This is, indeed, what happens, and the programmer who attempts to access the PC directly through r15 must take account of the exposure of the pipeline here. However, for most normal purposes the assembler or compiler handles all the details.

Even more complex behaviour is exposed if r15 is used later than the first cycle of an instruction, since the instruction will itself have incremented the PC during its first cycle. Such use of the PC is not often beneficial so the ARM architecture definition specifies the result as 'unpredictable' and it should be avoided, especially since later ARMs do not have the same behaviour in these cases.

3 Stage Pipeline

- The time, T , required to execute a given program is given by:

$$T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}}, \quad \text{Equation 11}$$

- How to decrease T_{prog} ?

Fclk ↑ CPI ↓

3 Stage Pipeline

- Memory bottleneck
 - Von Neumann bottleneck
- A 3-stage ARM core accesses memory on (almost) every clock cycle either to fetch an instruction or to transfer data.
- Simply tightening up on the few cycles where the memory is not used will yield only a small performance gain.
- To get a significantly better CPI the memory system must deliver more than one value in each clock cycle either by delivering more than 32 bits per cycle from a single memory or by having separate memories for instruction and data accesses.
- What's the solution then?

5 Stage Pipeline.

5 Stage Pipeline

- To meet the demand of high performance 3 stage pipeline is evolved into 5 stage in later version of ARM (after ARM7).
 - **Fetch**
 - **Decode**
 - **Execute**
 - **Buffer/Data Read**
 - **Write Back**

5 Stage Pipeline

- Fetch
 - The instruction is fetched from memory and placed in the instruction pipeline.
- Decode
 - The instruction is decoded and register operands read from the register file.
 - There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.
- Execute
 - An operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU.

5 Stage Pipeline

- Buffer/Data (**read**)
 - Data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle to give the same pipeline flow for all instructions.
- Write Back
 - The results generated by the instruction are written back to the register file, including any data loaded from memory.

5 Stage Pipeline

- This 5-stage pipeline has been used for many RISC processors and is considered to be the 'classic' way to design such a processor.
- Although the ARM instruction set was not designed with such a pipeline in mind, it maps onto it relatively simply.
 - There are three source operand read ports and two write ports in the register file (where a 'classic' RISC has two read ports and one write port), and the inclusion of address incrementing hardware in the execute stage to support load and store multiple instructions.

We will describe the ARM 5 stage Pipeline organization after discussing the instruction execution.(Simple -----→ complex)

5 Stage Pipeline

- Note:
- The coming slides discusses the data path activity for instruction execution
 - Data processing instructions
 - Data transfer instructions
 - Branch instruction
- However we discuss them with respect to either single clock cycle (data processing instructions) or 3-stage pipelining.
- We are postponing the 5-stage pipeline organization for now.

ARM instruction Execution

Data Processing Instructions

- A data processing instruction requires two operands,
 - one of which is always a **register**
 - and the other is either a **second register** or an **immediate value**.
- The second operand is passed through the barrel shifter where it is subject to a general shift operation, then it is combined with the first operand in the ALU using a general ALU operation.
- Finally, the result from the ALU is written back into the destination register (and the condition code register may be updated).
- All these operations take place in a single clock cycle.

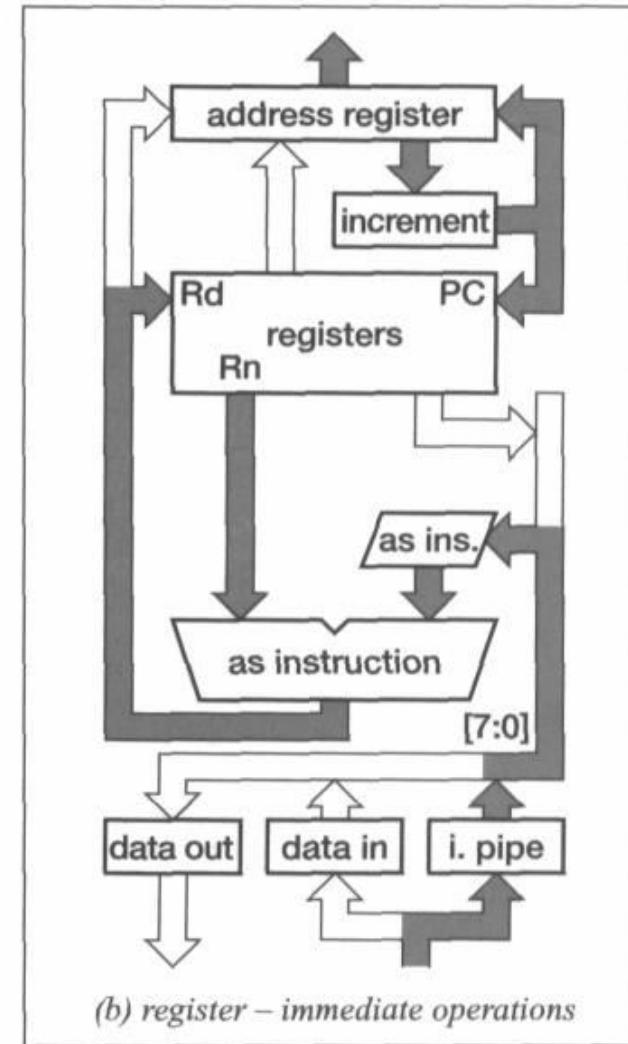
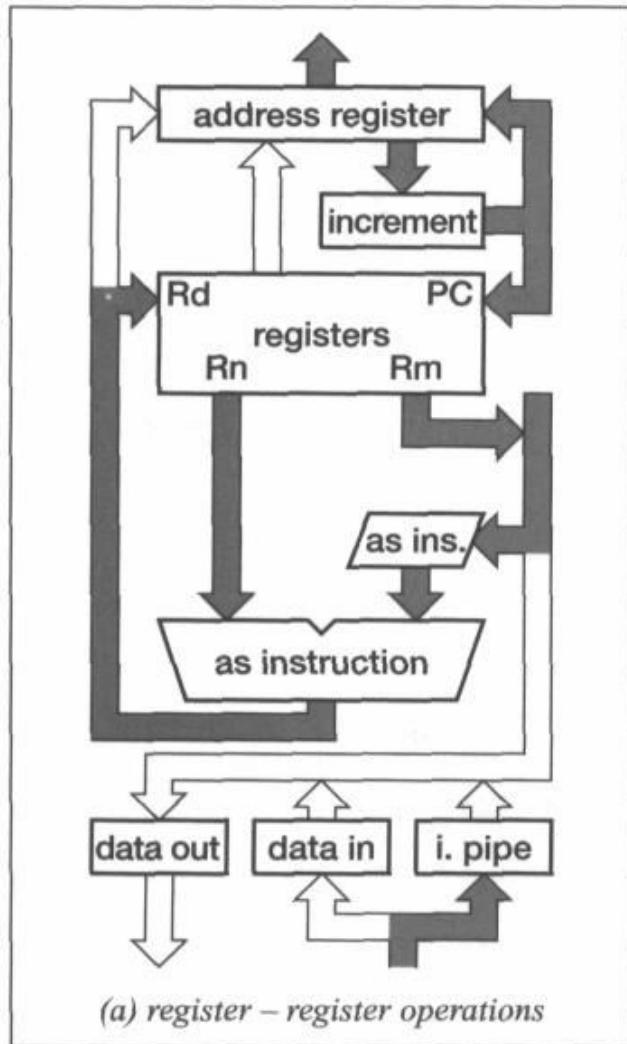
ARM instruction Execution

Data Processing Instructions

- The PC value in the address register is incremented and copied back into both the address register and r15 in the register bank, and the next instruction but one is loaded into the bottom of the instruction pipeline (*i. pipe*).
- The immediate value, when required, is extracted from the current instruction at the top of the instruction pipeline.
- For data processing instructions only the bottom eight bits (bits [7:0]) of the instruction are used in the immediate value.

ARM instruction Execution

Data Processing Instruction datapath activity (Fig 4.5)



ARM instruction Execution

Data transfer instructions

- A data transfer (load or store) instruction computes a memory address in a manner very similar to the way a data processing instruction computes its result.
- A register is used as the **base address**, to which is added (or from which is subtracted) an offset which again may be another register or an immediate value.
- This time, however, a **12-bit immediate value** is used without a shift operation rather than a shifted 8-bit value. The address is sent to the **address register**(IN 1ST CYCLE), and in a **second cycle** the data transfer takes place.

ARM instruction Execution

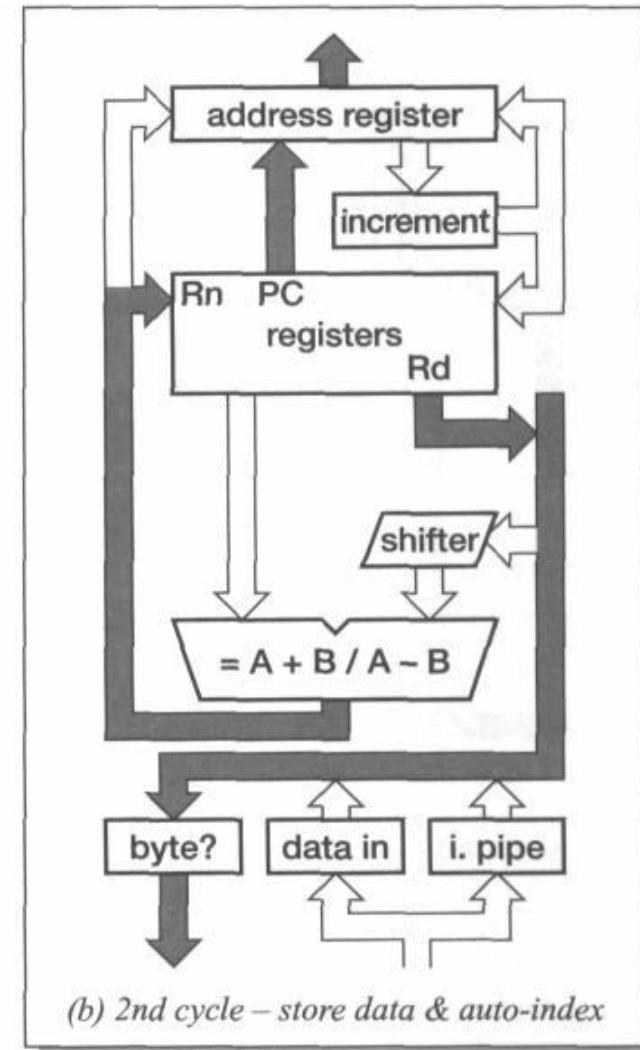
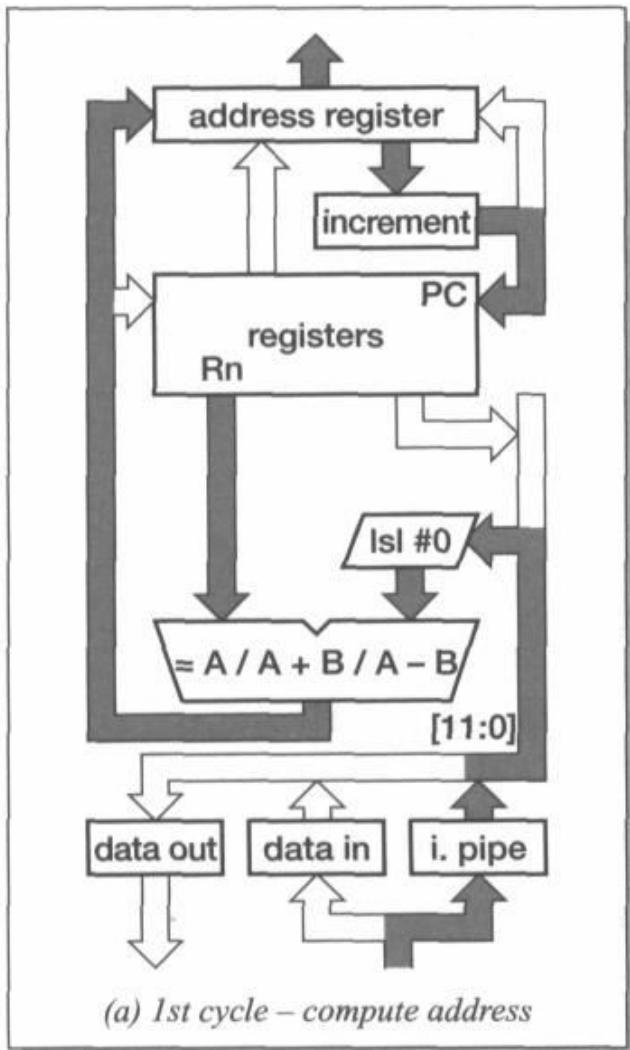
Data transfer instructions

When does the auto indexing takes place?

- Rather than leave the datapath largely idle during the data transfer cycle, the ALU holds the address components from the first cycle and is available to compute an auto-indexing modification to the base register if this is required.
- (If auto-indexing is not required the computed value is not written back to the base register in the **second cycle**.)

ARM instruction Execution

Figure 4.6 SIR (store register) datapath activity.



ARM instruction Execution

Data transfer instructions

Address register (Can it be called as the pipeline register)??

- The incremented PC value is stored in the register bank at the **end of the first cycle** so that the address register is free to accept the data transfer address for the second cycle, then at the **end of the second cycle** the PC is fed back to the address register to allow instruction prefetching to continue.
- At this stage the value sent to the address register in a cycle is the value used for the memory access in *the following* cycle.
- The address register is, in effect, a **pipeline register** between the processor datapath and the external memory.

ARM instruction Execution

Data transfer instructions

Byte date transfer:

- When the instruction specifies the store of a byte data type, the 'data out' block extracts the bottom byte from the register and replicates it four times across the 32-bit data bus.
- External memory control logic can then use the bottom two bits of the address bus to activate the appropriate byte within the memory system.

ARM instruction Execution

Data transfer instructions

How about Load?

- It takes 3 cycles.
- Load instructions follow a similar pattern like Store except that the data from memory only gets as far as the 'data in' register on the **second cycle** and a **third cycle** is needed to transfer the data from there to the destination register.

ARM instruction Execution

Branch instructions

- Branch instructions compute the target address in the first cycle.
 - A 24-bit immediate field is extracted from the instruction and then shifted left two bit positions to give a word-aligned offset which is added to the PC.
 - The result is issued as an instruction fetch address, and while the instruction pipeline refills the return address is copied into the link register (r14) if this is required (that is, if the instruction is a '**branch with link**').

ARM instruction Execution (Branch)

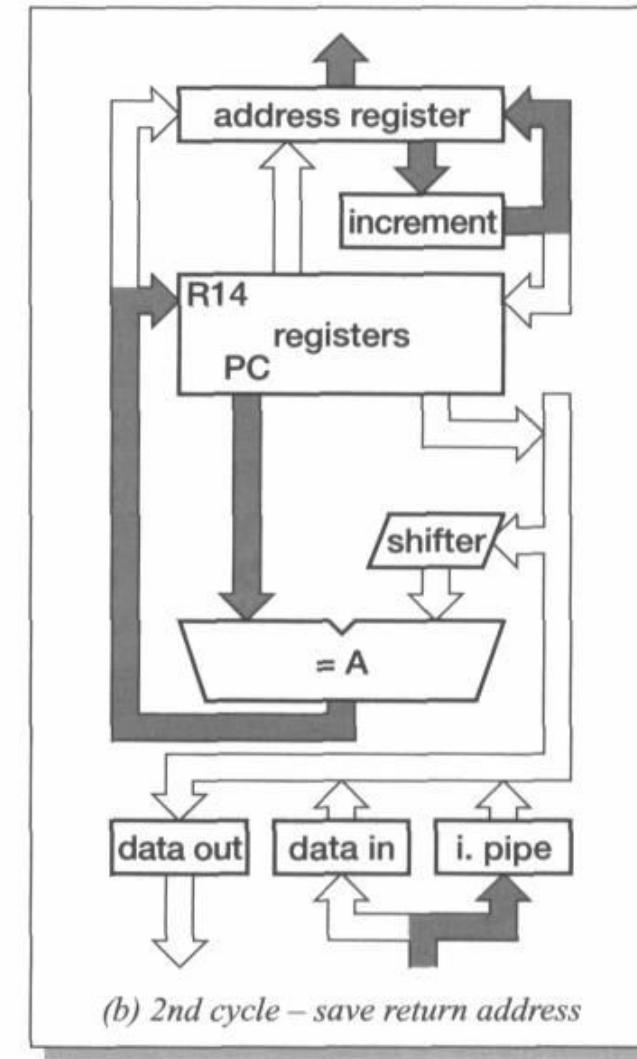
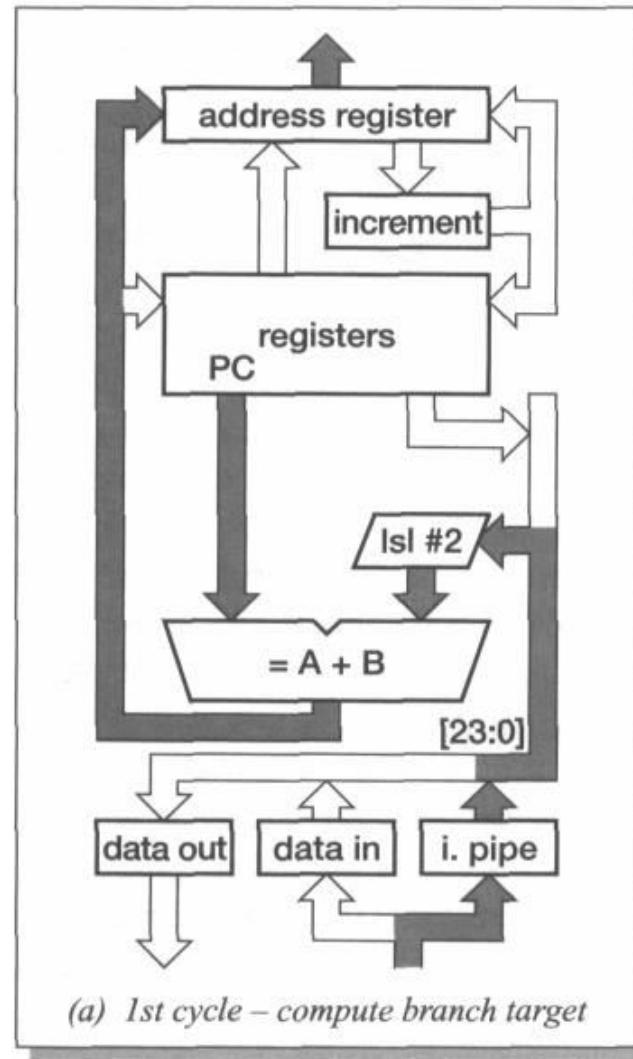


Figure 4.7 The first two (of three) cycles of a branch instruction

ARM instruction Execution

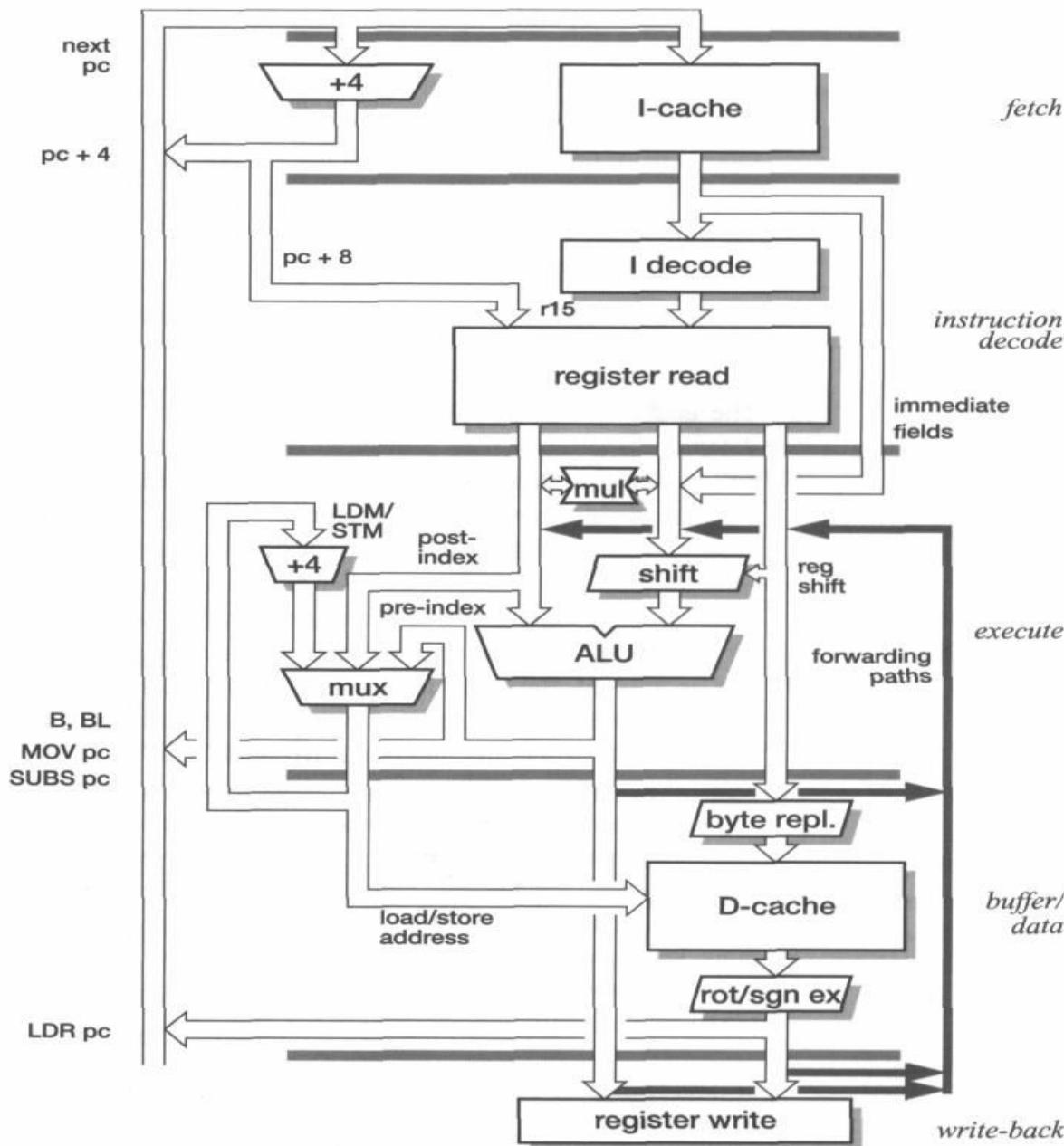
Branch instructions

- The third cycle, which is required to complete the pipeline refilling, is also used to make a small correction to the value stored in the link register in order that it points directly at the instruction which follows the branch.
- This is necessary because r15 contains $pc + 8$ whereas the address of the next instruction is $pc + 4$

Much awaited diagram

- ARM 5-stage pipeline Organization

Figure 4.4 ARM9TDMI 5-stage pipeline organization.



ARM 5-stage pipeline Organization

- Fetch
 - The instruction is fetched from memory and placed in the instruction pipeline.
 - The 5-stage pipeline reads the instruction operands one stage earlier in the pipeline, and would naturally get a different value (PC+4 rather than PC+8).
 - Referring to Figure 4.4, the incremented PC value from the fetch stage is fed directly to the register file in the decode stage, bypassing the pipeline register between the two stages. PC+4 for the next instruction is equal to PC+8 for the current instruction, so the correct r15 value is obtained without additional hardware.

ARM 5-stage pipeline Organization

- Decode
 - The instruction is decoded and register operands read from the register file.
 - There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.

ARM 5-stage pipeline Organization

- Execute
 - An operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU.

ARM 5-stage pipeline Organization

- Buffer/Data (**read**)
 - Data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle to give the same pipeline flow for all instructions.
- **Data Forwarding:**
 - A major source of complexity in the 5-stage pipeline (compared to the 3-stage pipeline) is that, because instruction execution is spread across three pipeline stages, the only way to resolve data dependencies without stalling the pipeline is to introduce forwarding paths.

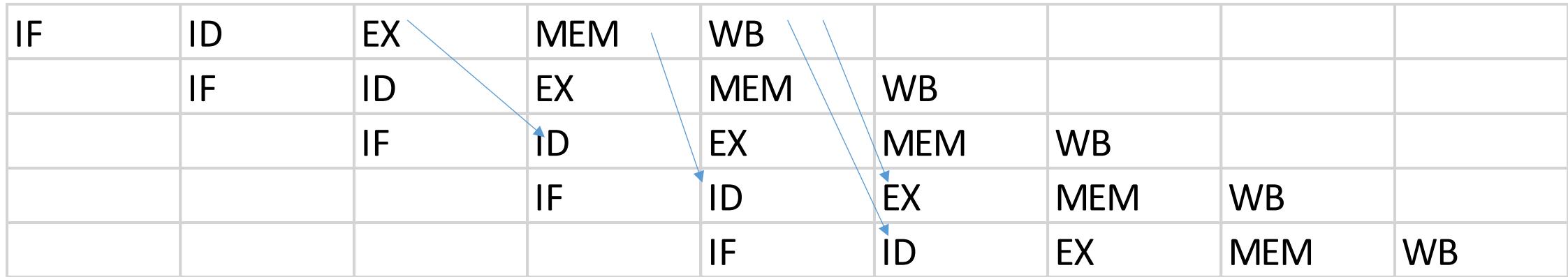
ARM 5-stage pipeline Organization

- Data Forwarding:
 - Observe the forwarding paths in the diagram (Fig 4.4)
 - Forwarding paths allow results to be passed between stages as soon as they are available, and the 5-stage ARM pipeline requires each of the three source operands to be forwarded from any of three intermediate result registers

ARM 5-stage pipeline Organization

- What are the three source Operands?
 1. BUS A
 2. BUS B
 3. PC
- What are the intermediate result registers?
 1. ALU Output
 2. Data available after store (in 4th cycle)
 3. Data available at the first half of 5th cycle (can be written in the first half of 5th CC and can be read in second half of 5th CC)
(transparent register file – you have study in MIPS)

ARM 5-stage pipeline Organization



The forwarding Paths

ARM 5-stage pipeline Organization (Stalls)

- There is one case where, even with forwarding, it is not possible to avoid a pipeline stall.
- Consider the following code sequence:
LDR rN, [. .] ; load rN from somewhere
ADD r2, r1, rN ; and use it immediately
- The processor cannot avoid a one-cycle stall as the value loaded into rN only enters the processor at the end of the buffer/data stage and it is needed by the following instruction at the start of the execute stage.
- The only way to avoid this stall is to encourage the compiler (or assembly language programmer) not to put a dependent instruction immediately after a load instruction.

ARM 5-stage pipeline Organization (Stalls)

- There is one case where, even with forwarding, it is not possible to avoid a pipeline stall.
- Consider the following code sequence:
LDR rN, [. .] ; load rN from somewhere
ADD r2, r1, rN ; and use it immediately
- The processor cannot avoid a one-cycle stall as the value loaded into rN only enters the processor at the end of the buffer/data stage and it is needed by the following instruction at the start of the execute stage.
- The only way to avoid this stall is to encourage the compiler (or assembly language programmer) not to put a dependent instruction immediately after a load instruction.

ARM 5-stage pipeline Organization (Stalls)

- Consider the following code sequence:

LDR rN, [. .] ; load rN from somewhere
ADD r2, r1, rN ; and use it immediately

Load		IF	ID	EX	MEM	WB		
ADD			IF	ID	EX	MEM	WB	
NOT POSSIBLE								
Load		IF	ID	EX	MEM	WB		
Stall			IF	ID	STALL	EX	MEM	WB
Load		IF	ID	EX	MEM	WB		
some inst			IF	ID	EX	MEM	WB	
ADD				IF	ID	EX	MEM	WB

ARM 5-stage pipeline Organization (Stalls)

- What is the role of MUX before the input to D-Cache
 - The MUX is for the address to be accessed.(see the figure)

Lecture 6

Previous Class

- . 3 Stage Pipeline
- . 5 Stage Pipeline

Today

- . Instruction Set – Contd...

.SWI

SOFTWARE INTERRUPT (SWI)

- The software interrupt instruction is used for calls to the operating system and is often called a '**supervisor call**'.
- It puts the processor into supervisor mode and begins executing instructions from **address 0x08**.

SOFTWARE INTERRUPT (SWI)

- Binary encoding

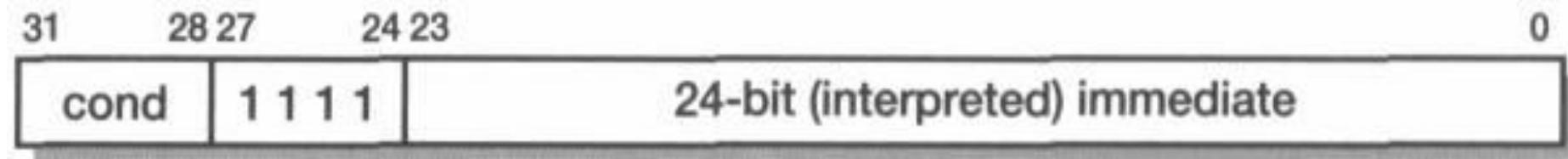
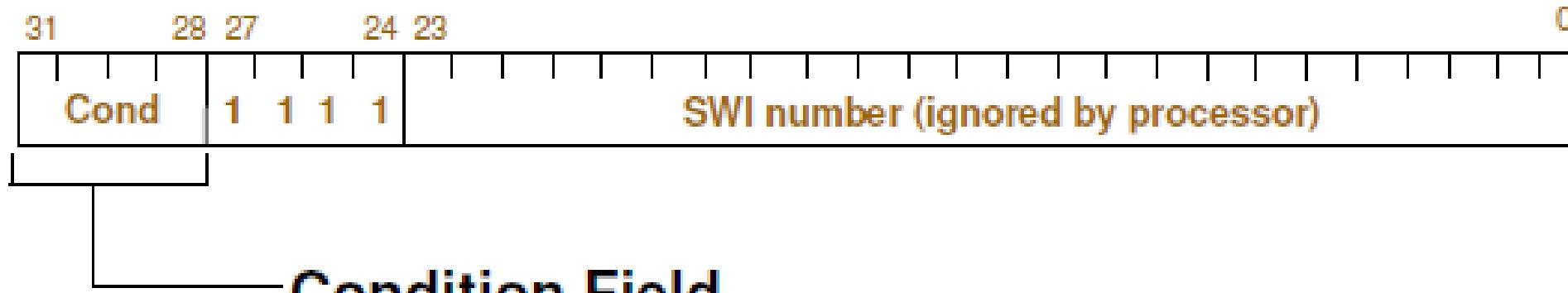


Figure 5.5 Software interrupt binary encoding.



- The 24-bit immediate field does not influence the operation of the instruction but may be interpreted by the system code.

SOFTWARE INTERRUPT (SWI)

- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.

SOFTWARE INTERRUPT (SWI)

- If the condition is passed the instruction enters supervisor mode using the standard ARM exception entry sequence.
- In detail, the processor actions are:
 1. Save the address of the instruction after the SWI in r14_svc.
 2. Save the CPSR in SPSR_svc.
 3. Enter supervisor mode and disable IRQs (but not FIQs) by setting CPSR[4:0] to $(10011)_2$ and CPSR[7] to 1.
 4. Set the PC to 08 and begin executing the instructions there.

SOFTWARE INTERRUPT (SWI)

- To return to the instruction after the SWI the system routine must not only copy `r14_svc` back into the PC, but it must also restore the `CPSR` from `SPSR_svc`.
- This requires the use of one of the special forms of the data processing instruction.

SOFTWARE INTERRUPT (SWI)

- To return to the instruction after the SWI the system routine must not only copy `r14_svc` back into the PC, but it must also restore the `CPSR` from `SPSR_svc`.
- This requires the use of one of the special forms of the data processing instruction.

SOFTWARE INTERRUPT (SWI)

- Syntax:
- **SWI{<cond>} <SWI number>**

PSR TRANSFER INSTRUCTIONS

- MRS allow contents of CPSR/SPSR to be transferred from appropriate status register to a general purpose register.
 - All of status register, or just the flags, can be transferred.
- Syntax:
 - MRS{<cond>} Rd,<psr> ; Rd = <psr>
 - Where <psr> = CPSR, CPSR_all, SPSR or SPSR_all

Examples: MRS r0, CPSR ; move the CPSR to r0 ;
 MRS r3, SPSR ;move the SPSR to r3

PSR TRANSFER INSTRUCTIONS

- **MRS** (immediate form)
 - MSR{<cond>} <psrf>, #Immediate
 - Where <psrf> = CPSR_flg or SPSR_flg
 - This immediate must be a 32-bit immediate, of which the 4 most significant bits are written to the flag bits.

PSR TRANSFER INSTRUCTIONS

- **MSR** allow contents of a general purpose register to be transferred to CPSR/SPSR.
 - All of status register, or just the flags, can be transferred.
 - Syntax:
 - `MSR{<cond>} <psr>,Rm ; <psr> = Rm`
 - `MSR{<cond>} <psrf>,Rm ; <psrf> = Rm`
- Where $<\text{psr}>$ = CPSR, CPSR_all, SPSR or SPSR_all
 $<\text{psrf}>$ = CPSR_flg or SPSR_flg

USING MRS AND MSR

- Currently reserved bits, may be used in future, therefore:
 - they must be preserved when altering PSR
 - the value they return must not be relied upon when testing other bits.
- Thus read-modify-write strategy must be followed when modifying any PSR:
 - Transfer PSR to register using MRS
 - Modify relevant bits
 - Transfer updated value back to PSR using MSR
- Note:
 - In User Mode, all bits can be read but only the flag bits can be written to.
 - The SPSR form should not be used in user or system mode since there is no accessible SPSR in those modes.

Lecture 7

Previous Class

- . SWI

Today

- . Instruction Set – Contd...

Branch

BRANCH INSTRUCTIONS

- Branch and Branch with Link (B, BL)

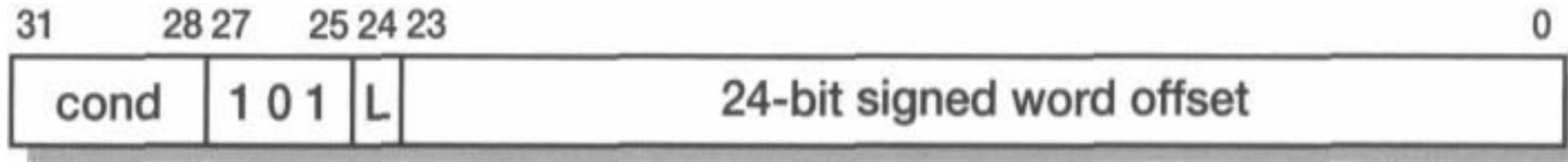


Figure 5.3 Branch and Branch with Link binary encoding.

- Branch and Branch with Link instructions cause the processor to begin executing instructions from an address computed by **sign extending the 24-bit offset** specified in the instruction, **shifting it left two places to form a word offset**, then **adding it to the program counter** which contains the address of the branch instruction plus eight bytes.

The assembler will compute the correct offset under normal circumstances.

The range of the branch instruction is +/- 32 Mbytes.

BRANCH INSTRUCTIONS

- Branch and Branch with Link (B, BL)

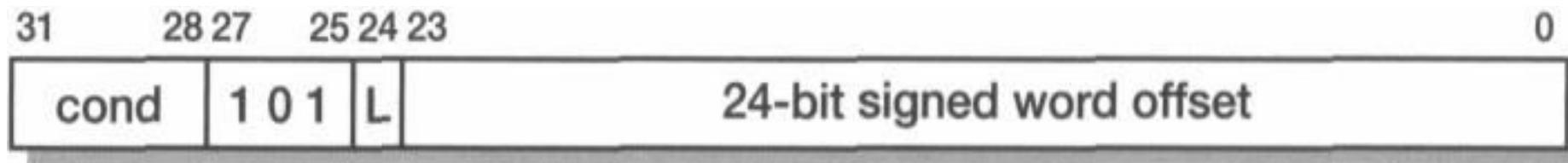


Figure 5.3 Branch and Branch with Link binary encoding.

- The **Branch with Link variant**, which has the **L bit (bit 24) set**, also moves the address of the instruction following the branch into the link register (r14) of the current processor mode. This is normally used to perform a subroutine call, with the return being caused by copying the link register back into the PC.
- Both forms of the instruction may be executed conditionally or unconditionally.

BRANCH INSTRUCTIONS

- Both forms of the instruction may be executed conditionally or unconditionally.

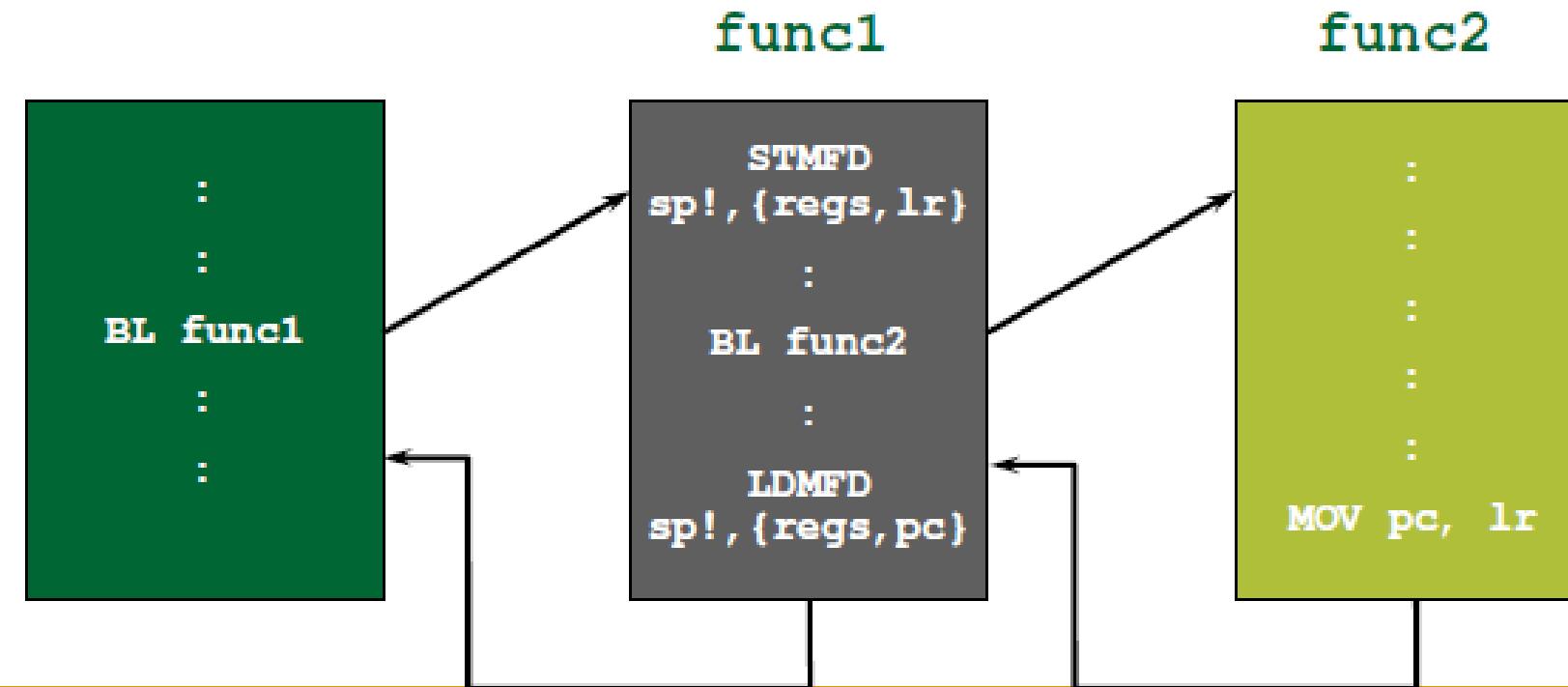
Branch : B{<cond>} label

Branch with Link : BL{<cond>} subroutine_label

'<target address>' or subroutine_label is normally a label in the assembler code; the assembler will generate the offset (which will be the difference between the address of the target and the address of the branch instruction plus 8).

ARM BRANCHES AND SUBROUTINES

- B <label>
 - PC relative. ±32 Mbyte range.
- 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



ARM SUBROUTINE LINKAGE

- Branch and link instruction:
`BL foo`
 - Copies current PC to r14.
- To return from subroutine:
 - `MOV r15,r14`

Examples

- An unconditional jump:

```
        . . . ; unconditional jump ..  
        ..  
LABEL .. ; .. to here
```

Examples

To execute a loop ten times:

```
MOV      r0, #10 ; initialize loop counter
LOOP    ...
SUBS    r0, #1  ; decrement counter setting CCs
BNE     LOOP    ; if counter <> 0 repeat loop..
...          ; .. else drop through
```

Examples

To call a subroutine:

```
..  
BL      SUB      ; branch and link to subroutine SUB  
..          ; return to here  
..  
SUB    ..          ; subroutine entry point  
MOV    PC, r14 ; return
```

Examples

Conditional subroutine call:

```
...
    CMP      r0, #5   ; if r0 < 5
    BLLT    SUB1      ; then call SUB1
    BLGE    SUB2      ; else call SUB2
...
```

- This example will only work correctly if SUB1 does not change the condition codes, since if the BLLT is taken it will return to the BLGE.

If the condition codes are changed by SUB1, SUB2 may be executed as well.

Branches

Note: Branches which attempt to go past the beginning or the end of the 32-bit address space should be avoided since they may have unpredictable results.

Conditional branches (branch conditions)

The pairs of conditions which are listed in the same row of the table (for instance BCC and BLO) are synonyms which result in identical binary code, but both are available because each makes the interpretation of the assembly source code easier in particular circumstances.

Branch	Interpretation	Normal uses
B/BAL	Unconditional Always	Always take this branch Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

ARM ADR PSEUDO-OP

- Cannot refer to an address directly in an instruction.
- Generate value by performing arithmetic on PC.
- ADR pseudo-op generates instruction required to calculate address:

ADR r1,FOO

CONTD...

```
COPY: ADR r1, TABLE1 ; r1 points to TABLE1  
      ADR r2, TABLE2 ; r2 points to TABLE2
```

```
LOOP: LDR r0, [r1]  
      STR r0, [r2]  
      ADD r1, r1, #4  
      ADD r2, r2, #4
```

...

```
TABLE1: ...  
TABLE2: ...
```

- Need to initialize address in r1 in the first place. How?
- Use ADR pseudo instruction - looks like normal instruction, but it is actually an assembler directive.
- The assembler translates it to one or more real instructions.

```
COPY: ADR r1, TABLE1 ; r1 points to  
      TABLE1
```

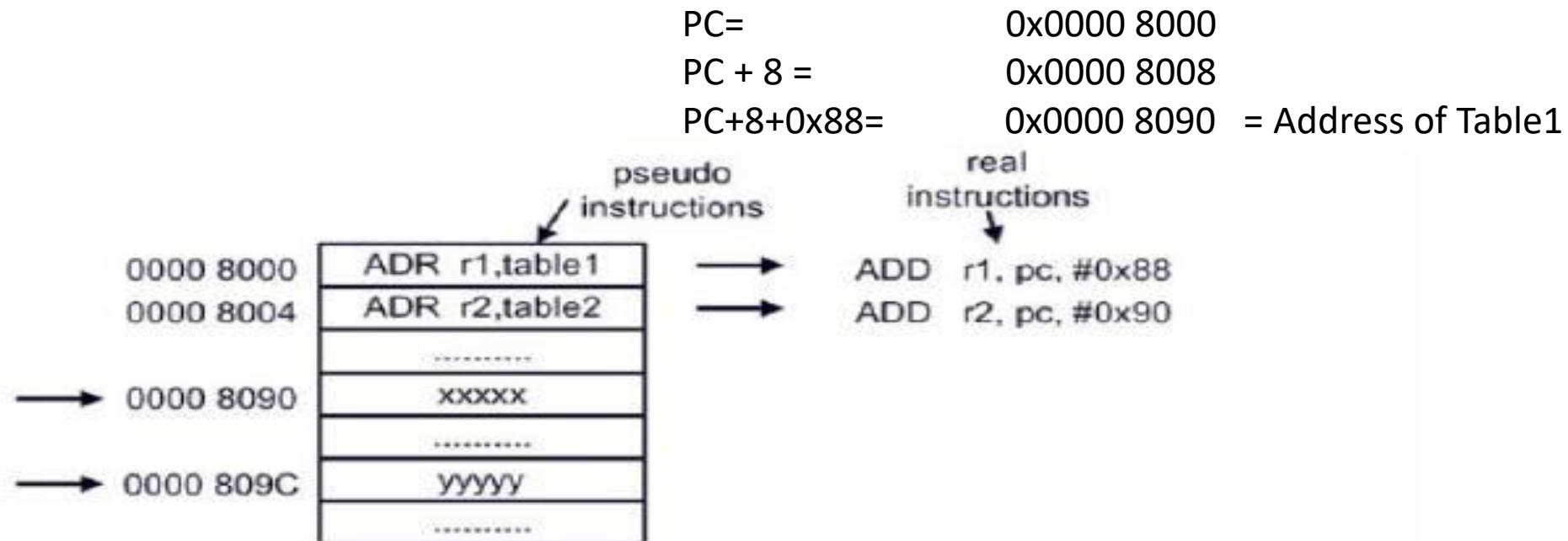
```
      ADR r2, TABLE2 ; r2 points to  
      TABLE2
```

```
LOOP: LDR r0, [r1], #4  
      STR r0, [r2], #4
```

...

```
TABLE1: ...
```

HOW ADR WORKS?



- How does the **ADR** directive work? Address is 32-bit, difficult to put a 32-bit address value in a register in the first place
- Solution: Program Counter PC (**r15**) is often close to the desired data address value
- **ADR r1, TABLE1** is translated into an instruction that adds or subtracts a constant to PC (**r15**), and puts the result in **r1**
- This constant is known as **PC-relative offset**, and it is calculated as:
 $\text{addr_of_table1} - (\text{PC_value} + 8)$

SUMMARY

- Load/store architecture
- Most instructions are RISCy, operate in single cycle.
- Some multi-register operations take longer.
- All instructions can be executed conditionally.

Lecture 8

Previous Class

- Instruction Set

Today

- Exception Handling

DEFINITIONS

- **Exception:** signal marking that something “out of the ordinary” has happened and needs to be handled. Caused by internal and external sources
- **Interrupt:** Externally asynchronous exception (by I/Os)
- **Software Interrupt (SWI):** User defined synchronous exception
- **Trap:** Processor’s diversion to a code to handle exception

EXCEPTIONS SOURCES IN ARM

- **Reset:** Occurs when the processor reset pin is asserted. (Signalling power-up)
- **Undefined Instruction:** Occurs if the processor, does not recognize the currently executing instruction.
- **Software Interrupt (SWI):** This is a user-defined intentional synchronous interrupt instruction.
- **Prefetch Abort:** Occurs when the processor attempts to execute an instruction that was not fetched, because the address was illegal.
- **Data Abort:** Occurs when a data transfer instruction attempts to load or store data at an illegal address.
- **IRQ:** Occurs when the processor external Interrupt ReQuest pin is asserted
- **FIQ:** Occurs when the processor external Fast Interrupt reQuest pin is asserted

CONTD....

- ◆ ARM processor can work in one of many **operating modes**. So far we have only considered **user mode**, which is the "**normal**" mode of operation.
- ◆ The processor can also enter "**privileged**" operating modes which are used to handle **exceptions** and **supervisor calls** (i.e. software interrupts SWI's)
- ◆ The Current Processor Status Register **CPSR** has 5 bits [bit4:0] to indicate which mode the processor is in:-

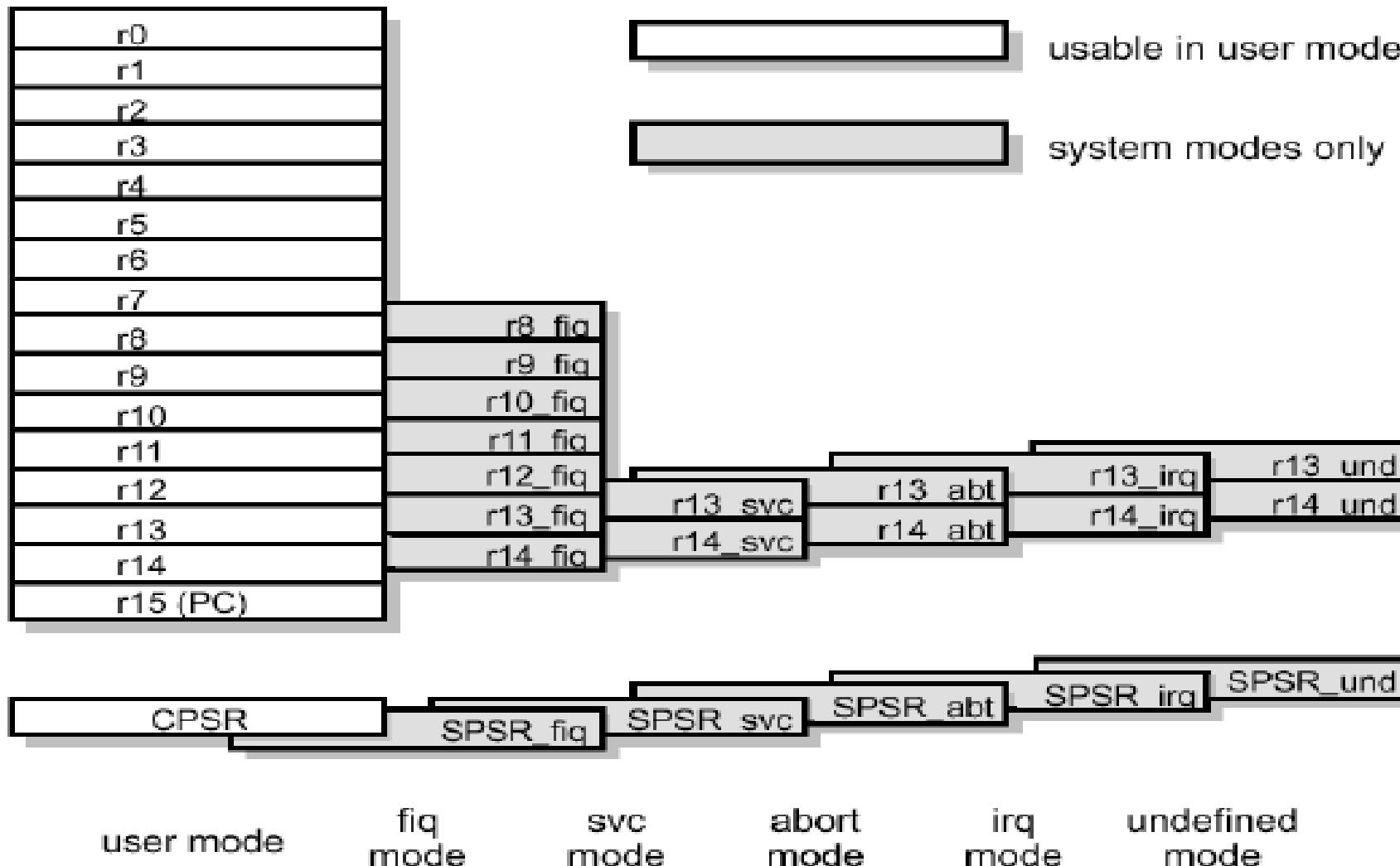
CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

HOW ARE EXCEPTIONS GENERATED

- ◆ By default, the processor is usually in user mode
- ◆ It enters one of the exception modes when unexpected events occurs.
- ◆ There are three different types of exceptions (some are called interrupts):-
 - ❖ 1. As a direct result of executing an instruction, such as:
 - Software Interrupt Instruction (SWI)
 - Undefined or illegal instruction
 - Memory error during fetching an instruction
 - ❖ 2. As a side-effect of an instruction, such as:
 - Memory fault during data read/write from memory
 - Arithmetic error (e.g. divide by zero)
 - ❖ 3. As a result of external hardware signals, such as:
 - Reset
 - Fast Interrupt (FIQ)
 - Normal Interrupt (IRQ)

SHADOW REGISTERS

- As the processor enters an exception mode, some new registers are automatically switched in:-



CONTD...

- ◆ For example, an external event (such as movement of the mouse) occurs that generates a Fast Interrupt (on the FIQ pin), the processor enters FIQ operating mode.
- ◆ It sees the same r0 - r7 as before, but sees a new set of r8 - r14, and in addition, an extra register called the Saved Processor Status Register (SPSR).
- ◆ By swapping to some new registers, it makes it easier for the programmer to preserve the state of the processor. For example, during FIQ mode, r8 - r14 can be used freely. On returning back to user mode, the original values of r8 - r14 will be automatically restored.

WHEN AN EXCEPTION OCCURS

- ◆ ARM completes current instruction as best it can.
- ◆ It departs from current instruction sequence to handle the exception by performing the following steps:-
 - ❖ 1. It **changes the operating mode** corresponding to the particular exception.
 - ❖ 2. It **saves the current PC** in the r14 corresponding to the new mode. For example, if FIQ occurs, the PC value is stored in r14(FIQ).
 - ❖ 3. It **saves the old value of CPSR** in the Saved Processor Status Register of the new mode.
 - ❖ 4. It **disables exceptions of lower priority** (to be considered later).
 - ❖ 5. It forces the PC to a new value corresponding to the exception. This is effectively a forced jump to the **Exception Handler** or **Interrupt Service Routine**.

CONTD...

- The processor's response to an exception
 - Copies the Current Program Status Register (CPSR) into the appropriate mode Saved Program Status Register(SPSR)
 - Sets the appropriate CPSR bits
 - Mode bits : set appropriately. maps in the appropriate banked registers for that mode.
 - I bit : to disable interrupts. IRQs are disabled once any other exception occurs
 - F bit: FIQs are also disabled when a FIQ occurs.
 - Stores the address of the return instruction (generally PC – 4) in LR_<mode>.
 - Sets the PC to the appropriate vector address.This forces the branch to the appropriate exception handler.

CONTD...

- ◆ Exceptions can be viewed as "forced" subroutine calls.
 - ❖ When and if an exception occurs is not predictable (unless it is a SWI exception).
 - ❖ A unique address is pre-defined for each exception handler (IRQ, FIQ, etc), and a branch is made to this address.
 - ❖ The address to which the processor is forced to branch to is called the **exception/interrupt vector**.

EXCEPTION ADDRESS

- ◆ Each vector (except FIQ) is 4 bytes long (i.e. one instruction)
- ◆ You put a branch instruction at this address:
B exception_handler
- ◆ FIQ is special in two ways:-
 - ❖ 1. You can put the actual FIQ handler (also called Fast Interrupt Service Routine) at 0x00000001C onwards, because FIQ vector occupies the highest address
 - ❖ 2. FIQ has many more shadow registers. So you don't have to save as many registers on the stack as other exceptions - faster.

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

CONTD...

- When an exception occurs, the core:
 - Copies CPSR into SPSR_<mode>
 - Sets appropriate CPSR bits
 - Interrupt disable flags if appropriate.
 - Maps in appropriate banked registers
 - Stores the “return address” in LR_<mode>
 - Sets PC to vector address

0x00000000	Reset
0x00000000	Undefined Instr
0x00000008	SWI
0x0000000C	Prefetch Abort
0x00000010	Data Abort
0x00000014	Reserved
0x00000018	IRQ
0x0000001C	FIQ

EXCEPTION RETURN

- ◆ Once the exception has been handled (by the exception handler), the user task is resumed.
- ◆ The handler program (or Interrupt Service Routine) must restore the user state exactly as it was before the exception occurred:
 - ❖ 1. Any modified user registers must be restored from the handler's stack
 - ❖ 2. The CPSR must be restored from the appropriate SPSR
 - ❖ 3. PC must be changed back to the instruction address in the user instruction stream
- ◆ Steps 1 and 3 are done by user, step 2 by the processor
- ◆ Restoring registers from the stack would be the same as in the case of subroutines
- ◆ Restoring PC value is more complicated. The exact way to do it depends on which exception you are returning from.

CONTD...

- Returning from an exception handler
 - Restore the CPSR from the SPSR_<mode>.
 - Restore the PC using the return address stored in LR_<mode>.
 - These can be achieved in a single instruction movs pc, lr or
 - subs pc, lr, #4
 - Adding the S flag (update condition codes) to a data processing instruction when in a privileged mode with the PC as the destination register, also transfers the SPSR to CPSR
 - Same thing for Load Multiple instruction (using the ^ qualifier)
`ldmfd sp! {r0-r12, pc}^`

CONTD...

- ◆ We assume that the return address was saved in r14 before entering the exception handler.
- ◆ To return from a SWI or undefined instruction trap, use:
 - ❖ MOVS pc, r14
- ◆ To return from an IRQ, FIQ or prefetch abort, use:
 - ❖ SUBS pc, r14, #4
- ◆ To return from a data abort to retry the data access, use:
 - ❖ SUBS pc, r14, #8
- ◆ Note the 'S' modifier is NOT used to set the flags, but instead to restore the CPSR, if the destination register is the PC.
- ◆ The differences between these three methods of return is due to the pipeline architecture of the ARM processor. The PC value stored in r14 can be one or two instructions ahead due to the instruction prefetch pipeline.

EXCEPTION PRIORITY

- ◆ Since exceptions can arise at the same time, a priority order has to be clearly defined. For the ARM processor this is:
 - ❖ Reset (highest priority)
 - ❖ Data abort (i.e. Memory fault in read/write data)
 - ❖ Fast Interrupt Request (FIQ)
 - ❖ Normal Interrupt Request (IRQ)
 - ❖ Prefetch abort
 - ❖ Software Interrupt (SWI), undefined instruction
- ◆ Consider the case when a FIQ and an IRQ occurring at the same time. The processor will process the FIQ handler first and “remember” that there is IRQ pending.
- ◆ On return from FIQ, the process will immediately go to the IRQ handler.

Lecture 9

Previous Class

ARM Exceptions

Today

- ARM Co-Processor interface
 - Handshake Signals
 - Instructions
 - Interface

ARM Coprocessors

- General-purpose extension of its instruction set through the addition of hardware coprocessor.
 - Example: The functionality of the ARM7TDMI instruction set may be extended by the addition of up to 16 external coprocessors.
- Software emulation of these coprocessor through the **undefined instruction trap**.
 - When the coprocessor is not present, instructions intended for it will trap, and suitable software may be installed to emulate its functions.

CONTD

- Adding the coprocessor hardware typically improves the system performance in a software compatible way.
- Some coprocessor numbers have been assigned by ARM Ltd.
- **Coprocessor Examples:**
 - (a) system coprocessor used to control on-chip functions such as cache & Memory management unit (MMU) on ARM 610.
 - (b) Floating-point ARM Coprocessor
 - (c) Application Specific Coprocessor

CONTD

- A typical coprocessor contains
 - An instruction pipeline (pipeline follower)
 - Instruction decoding logic
 - Handshake logic
 - A register bank (private; does not have to be 32 bits)
 - Special processing logic, with its own data path
- A coprocessor is connected to the same data bus as the ARM core and tracks the pipeline in the ARM processor core.
 - So a coprocessor can decode the instructions in the instruction stream and execute those that it supports

ARM7TDMI Coprocessor interface

- The ARM7TDMI coprocessor interface is based on 'bus watching' (other ARM cores use different techniques).
- The coprocessor is attached to a bus where the ARM instruction stream flows into the ARM, and the coprocessor copies the instructions into an internal pipeline that mimics the behaviour of the ARM instruction pipeline.
- As each coprocessor instruction begins execution there is a '**hand-shake**' between the ARM and the coprocessor to confirm that they are both ready to execute it.

ARM7TDMI Coprocessor interface (handshake)

- The **handshake** uses three signals:
 1. **cpi** (from ARM to all coprocessors). This signal, which stands for 'Coprocessor Instruction', indicates that the ARM has identified a coprocessor instruction and wishes to execute it.
 2. **cpa** (from the coprocessors to ARM). This is the '**Coprocessor Absent**' signal which tells the ARM that there is no coprocessor present that is able to execute the current instruction.
 3. **cpb** (from the coprocessors to ARM). This is the '**CoProcessor Busy**' signal which tells the ARM that the coprocessor cannot begin executing the instruction yet.

The timing is such that both the ARM and the coprocessor must generate their respective signals autonomously. The coprocessor cannot wait until it sees cpi before generating cpa and cpb

ARM7TDMI Coprocessor interface (handshake Outcomes)

- Once a coprocessor instruction has entered the ARM7TDMI and coprocessor pipelines, there are **four** possible ways it may be handled depending on the handshake signals:
 - The ARM may decide not to execute it, either because it falls in a **branch shadow** or because it **fails its condition code test**.

(All ARM instructions are conditionally executed, including coprocessor instructions.) ARM will not assert cpi, and the instruction will be discarded by all parties.

ARM7TDMI Coprocessor interface (handshake Outcomes contd)

- Once a coprocessor instruction has entered the ARM7TDMI and coprocessor pipelines, there are **four** possible ways it may be handled depending on the handshake signals:
 2. The ARM may decide to execute it (and signal this by asserting cpi), but no present coprocessor can take it so **cpa stays active**.

ARM will take the **undefined instruction trap** and use software to recover, possibly by emulating the trapped instruction.

ARM7TDMI Coprocessor interface (handshake Outcomes contd...)

- Once a coprocessor instruction has entered the ARM7TDMI and coprocessor pipelines, there are **four** possible ways it may be handled depending on the handshake signals:
 3. ARM decides to execute the instruction and a coprocessor accepts it, but cannot execute it yet. The coprocessor takes **cpa low but leaves cpb high**.
The ARM will '**busy-wait**' until the coprocessor takes cpb low, stalling the instruction stream at this point.
If an enabled **interrupt request** arrives while the coprocessor is busy, ARM will break off to handle the interrupt, probably returning to retry the coprocessor instruction later.

ARM7TDMI Coprocessor interface (handshake Outcomes contd...)

- Once a coprocessor instruction has entered the ARM7TDMI and coprocessor pipelines, there are **four** possible ways it may be handled depending on the handshake signals:
 4. ARM decides to execute the instruction and a coprocessor accepts it for **immediate execution**, cpi, cpa and cpb are all taken low and both sides commit to complete the instruction.

ARM7TDMI Coprocessor interface (Data Transfers)

- If the instruction is a **coprocessor data transfer instruction** the ARM is responsible for generating an initial memory address (the coprocessor does not require any connection to the address bus) but the coprocessor determines the length of the transfer;
- ARM will continue incrementing the address until the coprocessor signals completion.
- The cpa and cpb handshake signals are also used for this purpose.

ARM7TDMI Coprocessor interface (Data Transfers)

- Since the data transfer is **not interruptible** once it has started, coprocessors should limit the maximum transfer length to **16 words** (the same as a maximum length load or store multiple instruction) so as not to compromise the ARM's interrupt response.

ARM7TDMI Coprocessor interface (Pre-emptive execution)

- A coprocessor may begin executing an instruction as soon as it enters its pipeline so long as it can recover its state if the handshake does not ultimately complete.
- All activity must be repeatable with identical results up to the point of commitment.

THUMB INSTRUCTION SET

Objectives

- To understand 16-bit Thumb state operation of ARM Processor.
- To understand the features of Thumb state operation and how Thumb instructions decompress to ARM Mode.
- To know the technique of switching between ARM and Thumb mode of operations.
- To know the similarities and differences between ARM and Thumb mode of operation
- To understand exception handling and branching in Thumb mode.
- To understand operation of data processing instructions and data transfer instructions in Thumb mode.

CPU Instruction Set

ARM7TDMI processor has two instruction sets:

- the standard 32-bit **ARM** instruction set
- a 16-bit **THUMB** instruction set.

Processor Operating States

ARM state

which executes 32-bit, word-aligned *ARM* instructions.

THUMB state

which operates with 16-bit, halfword-aligned *THUMB* instructions.

Thumb Instruction Set

- ARM architecture versions v4T and above define a 16-bit instruction set called the Thumb instruction set. The functionality of the Thumb instruction set is a subset of the functionality of the 32-bit ARM instruction set.
- A processor that is executing Thumb instructions is operating in *Thumb state*. A processor that is executing ARM instructions is operating in *ARM state*.

Thumb Instruction Set

- A processor in ARM state cannot execute Thumb instructions, and a processor in Thumb state cannot execute ARM instructions. You must ensure that the processor never receives instructions of the wrong instruction set for the current state.
- Each instruction set includes instructions to change processor state.

Note: ARM processors always start executing code in ARM state.

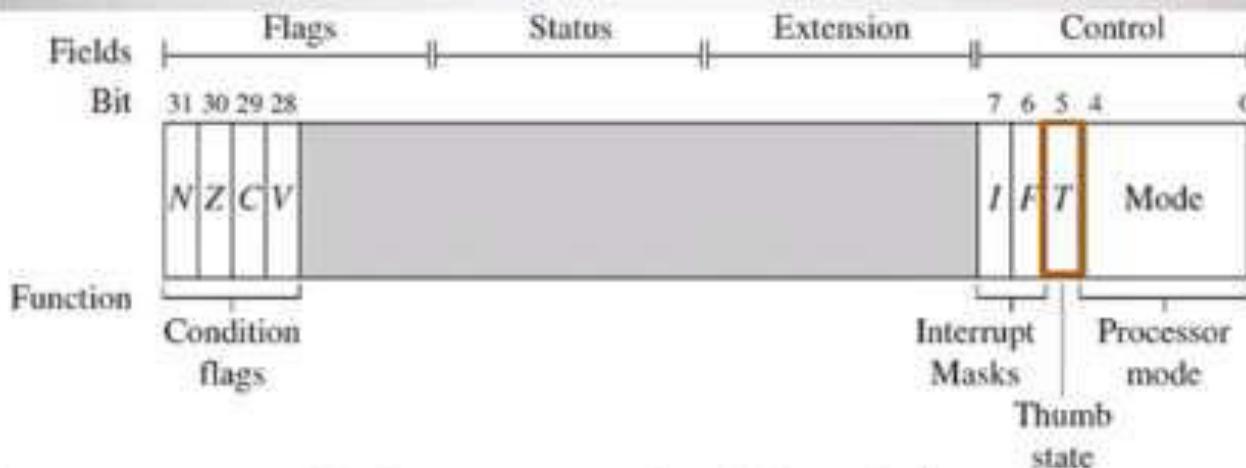
Thumb State: Philosophy

- The Thumb instruction set (**16-bit**), addresses the issue of **code density**
- It may be viewed as a **compressed** form of a subset of the **ARM instruction set**
- Thumb instructions **map** onto ARM instructions
- The Thumb **programmer's model** maps onto the ARM programmer's model
- Implementations of Thumb use **dynamic decompression** in an ARM instruction pipeline and then instructions execute as standard ARM instructions within the processor
- Thumb is not a complete architecture; it is not anticipated that a processor would execute Thumb instructions without supporting the ARM instruction set

Thumb State: Philosophy

- Therefore the Thumb instruction set need to only support common application functions
- Allowing recourse to the full ARM instruction set where necessary
 - For instance, all exceptions automatically enter ARM state
- Thumb is fully supported by ARM development tools
- An application can mix **ARM** and **Thumb** subroutines flexibly to optimize **performance** or code density on a routine-by-routine basis
- Use of the Thumb instruction set can improve **code density**, **power-efficiency**, **save cost** and **enhance performance** all at once

Thumb State bit in CPSR



- ARM processors which support the **Thumb** instruction set can also execute the standard **32-bit ARM** instruction set
- The interpretation of the instruction stream at any particular time is determined by **bit 5** of the **CPSR**, the **T bit**
- If **T is set** the processor interprets the instruction stream as **16-bit** Thumb instructions, otherwise it interprets it as standard ARM instructions
- Those ARM processors that have T in their name support Thumb mode
 - **Example:** ARM7TDMI

Thumb state entry

Thumb State bit in CPSR

- ARM cores **start up**, after **reset**, executing **ARM instructions**
- The normal way they **switch** to execute **Thumb** instructions is by executing a Branch and Exchange instruction (**BX**)
- This instruction **sets** the **T bit** if the **bottom bit (bit[0])** of the specified **register** was **set**, and switches the program counter to the address given in the remainder of the register
 - BX R1 ; if R1 = 0x100**1**, R15 = 0x1000, T = **1**
- Note that since the instruction causes a branch it **flushes** the instruction **pipeline**, removing any ambiguity over the interpretation of any instructions already in the pipeline
- Other instructions which change from ARM to Thumb code include **exception returns**
 - Either using a special form of data processing instruction or a special form of load multiple register instruction (by restoring **CPSR** with **SPSR**)

Thumb State Exit

- An explicit **switch back** to an **ARM** instruction stream can be caused by executing
 - A **Thumb BX** instruction (does the reverse of BX in ARM mode)
- An **implicit return** to an **ARM** instruction stream takes place whenever an **exception** is **taken**
 - if the Processor was running in Thumb state prior to the exception
 - Note that **exception handlers** always run in **ARM** state

BX and BLX

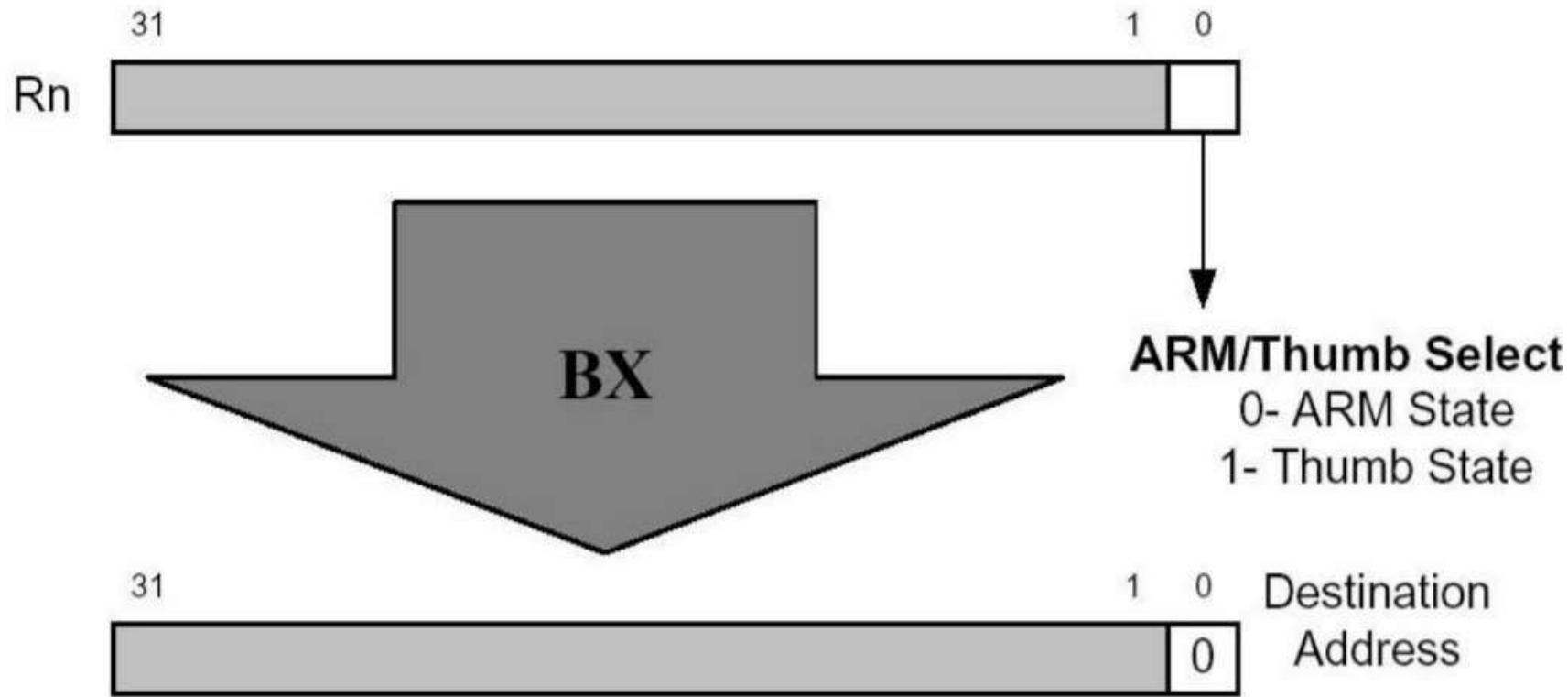
BX Rm

- Thumb version branch exchange
- $\text{pc} = \text{Rm} \& 0xffffffe$, $T = \text{Rm}[0]$

BLX Rm

- Thumb version branch exchange with link
- $\text{pc} = \text{Rm} \& 0xffffffe$, $T = \text{Rm}[0]$
- $\text{lr} = \text{address of next instruction after BLX} + 1$

Switching between States



Eg

```
;start off in ARM state
CODE32
ADR r0, Into_Thumb+1 ;generate branch target
;address & set bit 0
;hence arrive Thumb state
BX r0
;branch exchange to Thumb
...
CODE16
;assemble subsequent as Thumb
Into_Thumb ...
ADR r5, Back_to_ARM ;generate branch target to
;word-aligned address,
;hence bit 0 is cleared.
BX r5
;branch exchange to ARM
...
CODE32
;assemble subsequent as ARM
Back_to_ARM ...
```

- * **CODE32** instructs the assembler to interpret subsequent instructions as ARM instructions
- * **CODE16** instructs the assembler to interpret subsequent instructions as THUMB instructions

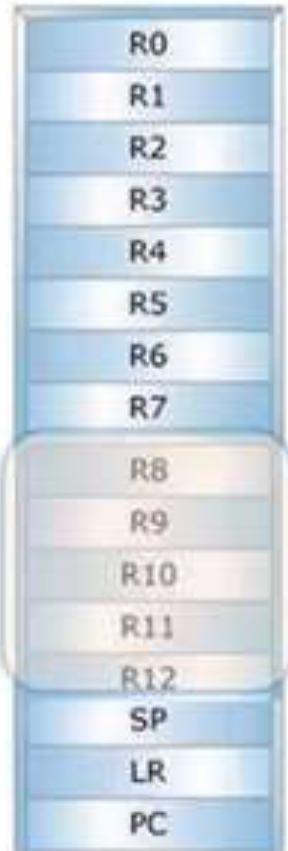
Example

□ A call to a Thumb subroutine

```
CODE32  
...  
BLX TSUB ;call Thumb subroutine  
...  
CODE16 ;start of Thumb code  
TSUB ...  
BX r14 ;return to ARM code
```

Thumb Programmer's Model

Thumb Programmer's Model



BANK REGISTERS ?????

CPSR

CPSR

ARM

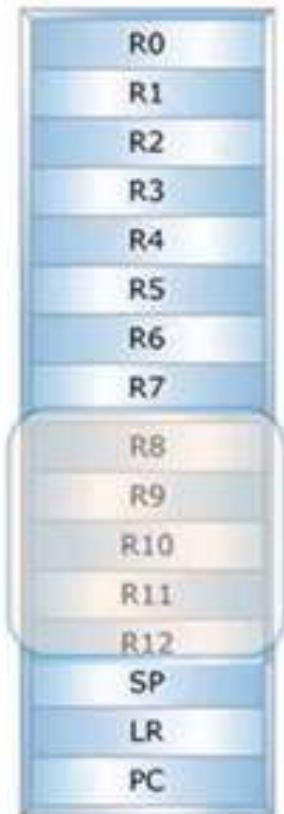
Thumb

Thumb Programmer's Model



CPSR

ARM



CPSR

Thumb

- The Thumb instruction set is a subset of the ARM instruction set
- The Thumb instructions operate on a **restricted view** of the **ARM registers**
- It gives full access to the eight 'Lo' general purpose registers **r0 to r7**
 - And makes extensive use of **r13 to r15** for special purposes
- The use of **R13** as a **stack pointer** in **ARM** code is purely a **software convention**
 - Whereas in **Thumb** code it is somewhat **hard-wired**

Register Access in Thumb

- ❑ Not all registers are directly accessible in Thumb
- ❑ Low register **r0 – r7**: fully accessible
- ❑ High register **r8 – r12**: only accessible with MOV, ADD, CMP; only CMP sets the condition code flags
- ❑ **SP** (Stack Pointer), **LR** (Link Register) & **PC** (Program Counter): limited accessibility, certain instructions have implicit access to these
- ❑ **CPSR**: only indirect access
- ❑ **SPSR**: no access

Why lo registers only and not hi registers in thumb?

Thumb inst doesnt need ?

Will Power consumption be more?

Any other reasons?

ARM-Thumb Similarities

- All **Thumb** instructions are **16 bits** long
- They map onto ARM instructions so they **inherit** many **properties** of the **ARM instruction set**
- A load-store architecture with data processing, data transfer and control flow instructions
- Support for 8-bit byte, 16-bit half-word and 32-bit word **data types**
 - Where half-words are aligned on 2-byte boundaries
 - words are aligned on 4-byte boundaries
- A **32-bit un-segmented** memory
- However, in order to achieve a **16-bit** instruction length a number of characteristic features of the **ARM instruction set** have not been **supported** in Thumb state

ARM-Thumb Differences

- Most Thumb instructions are executed **unconditionally** except branch instructions
 - Whereas All ARM instructions are executed conditionally
- Many Thumb data processing instructions use a **2-address format**
 - The **destination** register is the **same** as one of the **source** registers
 - ARM data processing instructions use a 3-address format (except 64-bit MUL instructions)
- Thumb instruction formats are **less regular** than ARM instruction formats, as a result of the **dense encoding**
- There are **no** status register access instructions (**MSR/MRS**) in Thumb state
- Many addressing modes of ARM not supported in Thumb state
- No banked registers and privileged modes in Thumb state

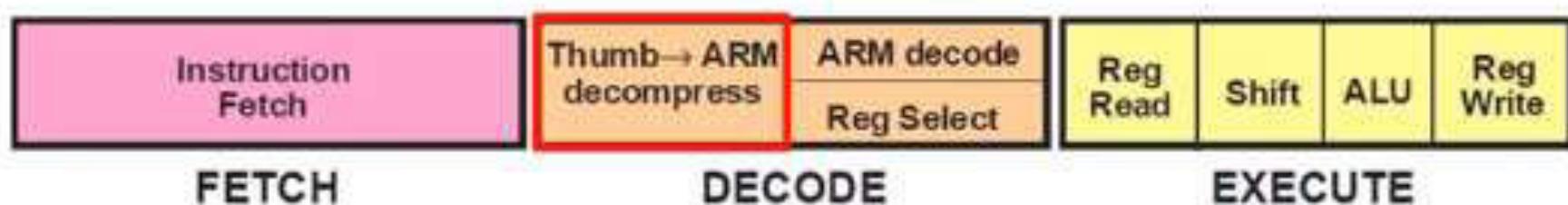
ARM-Thumb Differences

- The biggest register difference involves the **SP** register
- The Thumb state has unique stack mnemonics (**PUSH**, **POP**) that don't exist in the ARM state
- These instructions assume the existence of a stack pointer, for which **R13** is used
 - They translate into **load** and **store** instructions internally
- **No SWP** instructions in Thumb mode
- No support for Coprocessor instructions in Thumb mode
- **Barrel shifter** operations are **separate instructions**
- **All** data processing instructions **set condition flags** in CPSR
 - Except when one or more high registers are specified as operands to the MOV or ADD instructions, in these cases the flags **cannot** be set

Thumb Implementation

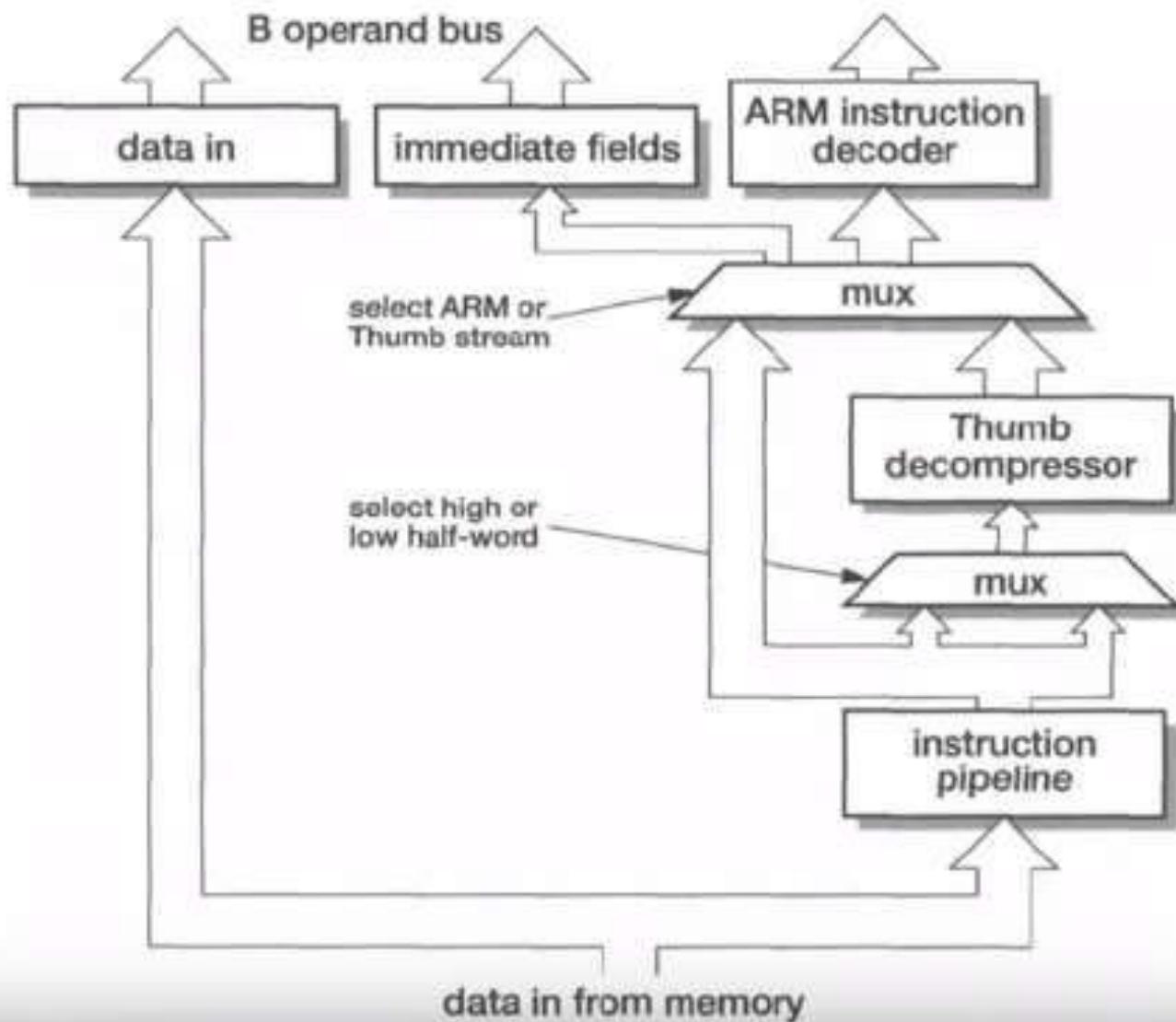
- The Thumb instruction set can be incorporated into a 3-stage pipeline ARM processor macrocell with relatively **minor changes** to most of the processor logic
 - The 5-stage pipeline implementations are trickier in ARM9
- The biggest addition is the Thumb instruction **decompressor** in the instruction pipeline
 -
- This logic **translates** a **Thumb** instruction into its **equivalent ARM** instruction
- Since ARM does relatively little work in Phase I of Decode cycle, the decompression logic can be accommodated without compromising the cycle time or increasing the pipeline latency
- ARM7TDMI **Thumb** pipeline operates in exactly the same way as that of ARM pipeline

Thumb Implementation



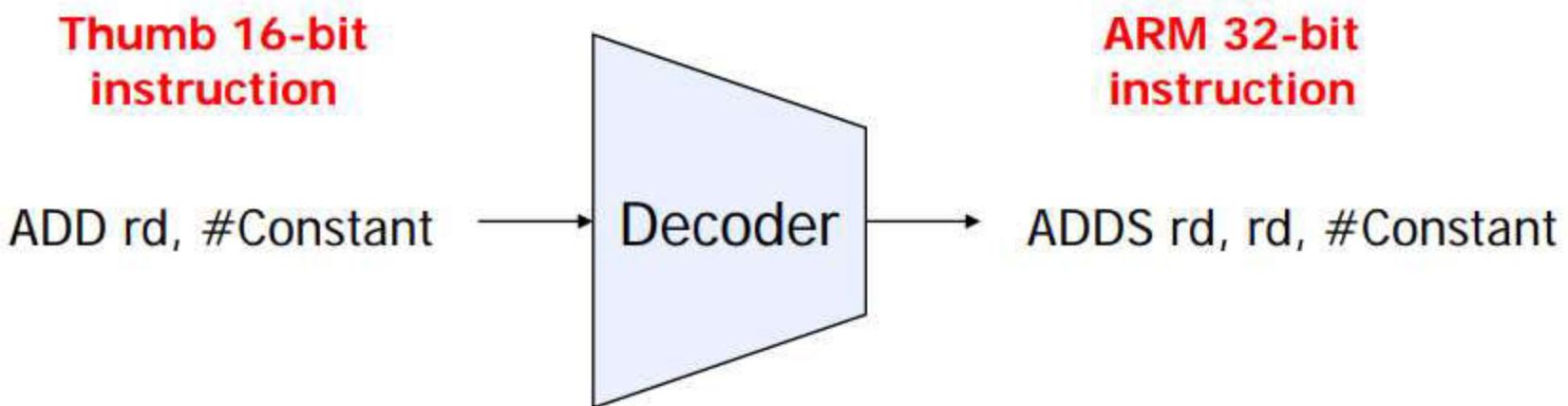
- The Thumb **decompressor** performs a **static translation** of 16-bit Thumb instructions **into** their equivalent **32-bit** ARM instructions
- This involves performing a look-up to translate the opcodes and operands
 - Zero-extending the **3-bit** register specifiers **to** give **4-bit** specifiers and mapping other fields across as required

Thumb Decompressor Organization



Decompressing process in Decoder

Thumb instruction decoding

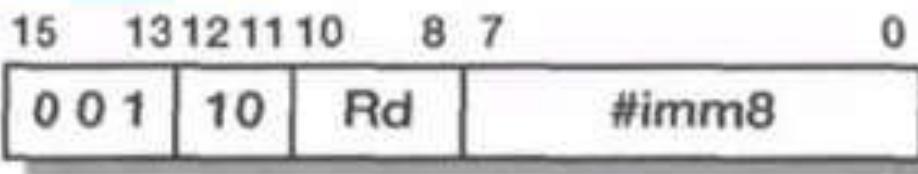


Thumb Instruction Mapping

Thumb to ARM Instruction

ADD Rd, #imm8 → **ADDS Rd, Rd, #imm8**

'always'
condition



major opcode,
format 3: MOV/
CMP/ADD/SUB
with immediate

minor opcode
denoting ADD
& set CC

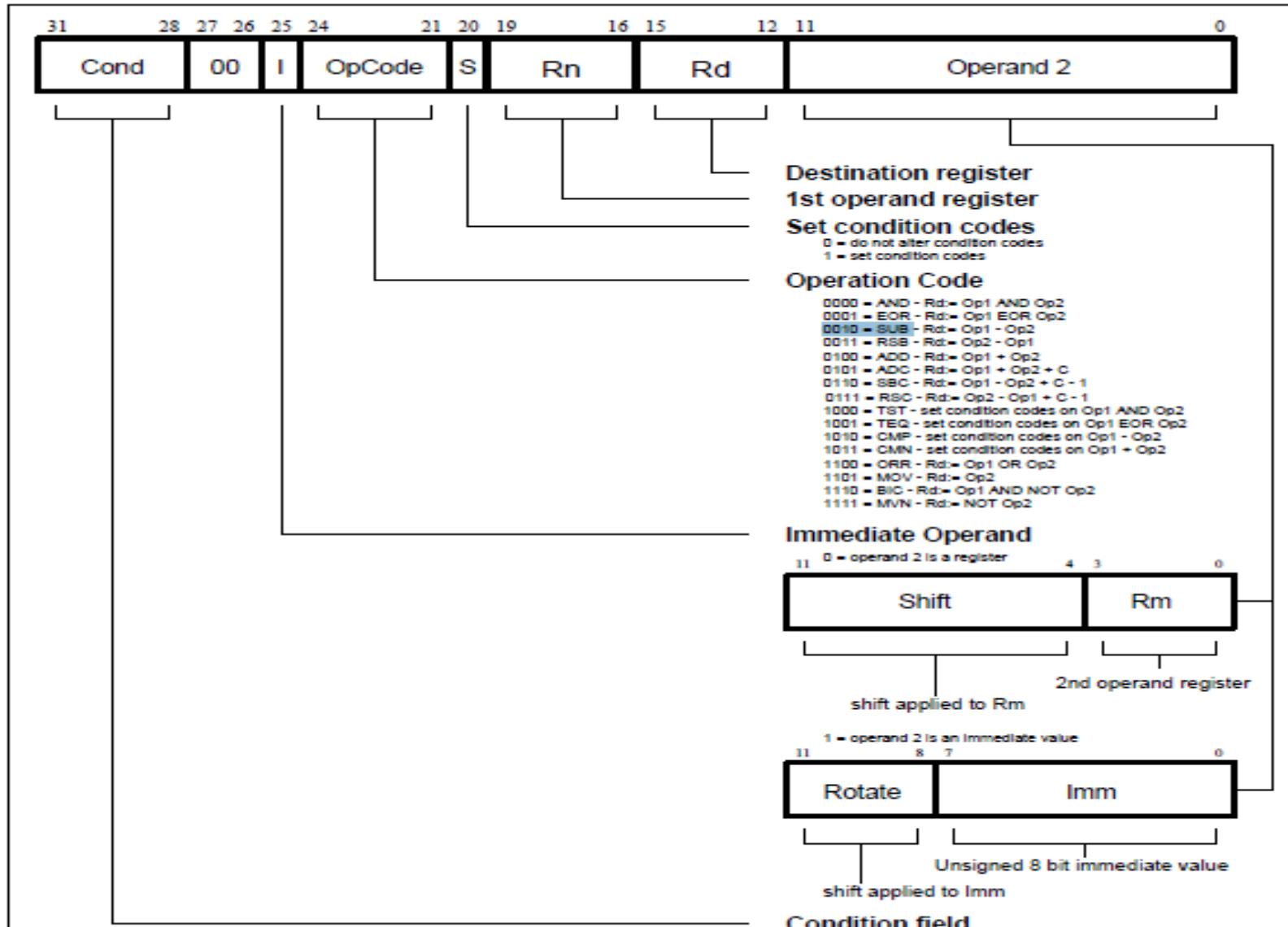
destination
and source
register

zero
shift

immediate
value



For reference – Arm instr



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

Figure 5-1: THUMB instruction set formats

Thumb instruction format

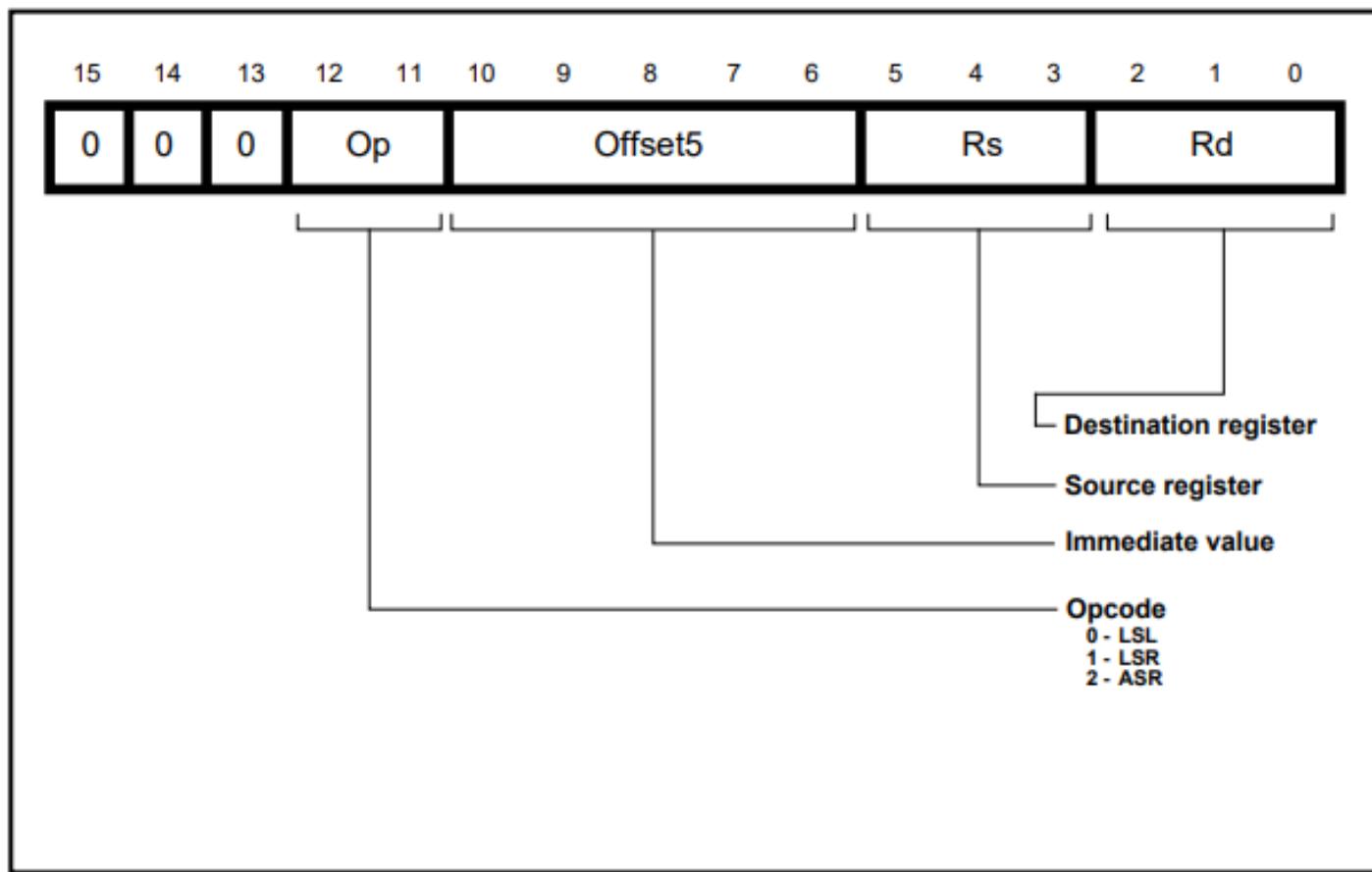


Figure 5-2: Format 1

Note All instructions in this group set the CPSR condition codes.

OP	THUMB assembler	ARM equivalent	Action
00	LSL Rd, Rs, #Offset5	MOVS Rd, Rs, LSL #Offset5	Shift Rs left by a 5-bit immediate value and store the result in Rd.
01	LSR Rd, Rs, #Offset5	MOVS Rd, Rs, LSR #Offset5	Perform logical shift right on Rs by a 5-bit immediate value and store the result in Rd.
10	ASR Rd, Rs, #Offset5	MOVS Rd, Rs, ASR #Offset5	Perform arithmetic shift right on Rs by a 5-bit immediate value and store the result in Rd.

Table 5-2: Summary of format 1 instructions

Note All instructions in this group set the CPSR condition codes.

Op	I	THUMB assembler	ARM equivalent	Action
0	0	ADD Rd, Rs, Rn	ADDS Rd, Rs, Rn	Add contents of Rn to contents of Rs. Place result in Rd.
0	1	ADD Rd, Rs, #Offset3	ADDS Rd, Rs, #Offset3	Add 3-bit immediate value to contents of Rs. Place result in Rd.
1	0	SUB Rd, Rs, Rn	SUBS Rd, Rs, Rn	Subtract contents of Rn from contents of Rs. Place result in Rd.
1	1	SUB Rd, Rs, #Offset3	SUBS Rd, Rs, #Offset3	Subtract 3-bit immediate value from contents of Rs. Place result in Rd.

Table 5-3: Summary of format 2 instructions

Note All instructions in this group set the CPSR condition codes.

Op	THUMB assembler	ARM equivalent	Action
00	MOV Rd, #Offset8	MOVS Rd, #Offset8	Move 8-bit immediate value into Rd.
01	CMP Rd, #Offset8	CMP Rd, #Offset8	Compare contents of Rd with 8-bit immediate value.
10	ADD Rd, #Offset8	ADDS Rd, Rd, #Offset8	Add 8-bit immediate value to contents of Rd and place the result in Rd.
11	SUB Rd, #Offset8	SUBS Rd, Rd, #Offset8	Subtract 8-bit immediate value from contents of Rd and place the result in Rd.

Table 5-4: Summary of format 3 instructions

The following instructions perform ALU operations on a LO register pair.

Note All instructions in this group set the CPSR condition codes.

OP	THUMB assembler	ARM equivalent	Action
0000	AND Rd, Rs	ANDS Rd, Rd, Rs	Rd := Rd AND Rs
0001	EOR Rd, Rs	EORS Rd, Rd, Rs	Rd := Rd EOR Rs
0010	LSL Rd, Rs	MOVS Rd, Rd, LSL Rs	Rd := Rd << Rs
0011	LSR Rd, Rs	MOVS Rd, Rd, LSR Rs	Rd := Rd >> Rs
0100	ASR Rd, Rs	MOVS Rd, Rd, ASR Rs	Rd := Rd ASR Rs
0101	ADC Rd, Rs	ADCS Rd, Rd, Rs	Rd := Rd + Rs + C-bit
0110	SBC Rd, Rs	SBCS Rd, Rd, Rs	Rd := Rd - Rs - NOT C-bit
0111	ROR Rd, Rs	MOVS Rd, Rd, ROR Rs	Rd := Rd ROR Rs
1000	TST Rd, Rs	TST Rd, Rs	Set condition codes on Rd AND Rs
1001	NEG Rd, Rs	RSBS Rd, Rs, #0	Rd = -Rs

Table 5-5: Summary of Format 4 instructions

Instructions with high register range only CMP sets CC (ADD, CMP, MOV)

Op	H1	H2	THUMB assembler	ARM equivalent	Action
00	0	1	ADD Rd, Hs	ADD Rd, Rd, Hs	Add a register in the range 8-15 to a register in the range 0-7.
00	1	0	ADD Hd, Rs	ADD Hd, Hd, Rs	Add a register in the range 0-7 to a register in the range 8-15.
00	1	1	ADD Hd, Hs	ADD Hd, Hd, Hs	Add two registers in the range 8-15

Table 5-6: Summary of format 5 instructions

Thumb Benefits

- The Thumb code requires only **70%** of the space of the ARM code
- The Thumb code uses **40% more instructions** than the ARM code
- With 32-bit memory, the ARM code is **40% faster** than the Thumb code
- With 16-bit memory, the Thumb code is **45% faster** than the ARM code
- Thumb code uses **30% less external memory power** than ARM code

Thumb instruction set

Mnemonic	Instruction	Example	ARM-code equivalent
ADC	Add with Carry	ADC Rd, Rs	ADCS Rd, Rd, Rs
ADD	Add	ADD Rd, Rs, Rn	ADDS Rd, Rs, Rn
AND	AND	AND Rd, Rs	ANDS Rd, Rd, Rs
ASR	Arithmetic Shift Right	ASR Rd, Rs	MOVS Rd, Rd, ASR Rs
B	Unconditional branch	B label	B label
BCC	Conditional branch	BCC label	BCC label
BIC	Bit Clear	BIC Rd, Rs	BICS Rd, Rd, Rs
BL	Branch and Link	BL label	BL label
BX	Branch and Exchange	BX Rs	BX Rs
CMN	Compare Negative	CMN Rd, Rs	CMN Rd, Rs
CMP	Compare	CMP Rd, #Offset8	CMP Rd, #Offset8
EOR	EOR	EOR Rd, Rs	EORS Rd, Rd, Rs
LDMIA	Load multiple	LDMIA Rb!, {Rlist}	LDMIA Rb!, {Rlist}
LDR	Load word	LDR Rd, [PC, #lmm]	LDR Rd, [PC, #lmm]
LDRB	Load byte	LDRB Rd, [Rb, Ro]	LDRB Rd, [Rb, Ro]
LDRH	Load halfword	LDRH Rd, [Rb, #lmm]	LDRH Rd, [Rb, #lmm]
LSL	Logical Shift Left	LSL Rd, Rs, #Offset5	MOVS Rd, Rs, LSL#Offset5

Thumb instruction set (Cont.)

LDRSB	Load sign-extended byte	LDRSB Rd, [Rb, R ₀]	LDRSB Rd, [Rb, R ₀]
LDRSH	Load sign-extended halfword	LDRSH Rd, [Rb, R ₀]	LDRSH Rd, [Rb, R ₀]
LSR	Logical Shift Right	LSR Rd, R _s	MOVS Rd, Rd, LSR R _s
MOV	Move register	MOV Rd, #Offset8	MOVS Rd, #Offset8
MUL	Multiply	MUL Rd, R _s	MULS Rd, R _s , Rd
MVN	Move NOT register	MVN Rd, R _s	MVNS Rd, R _s
NEG	Negate	NEG Rd, R _s	RSBS Rd, R _s , #0
ORR	OR	ORR Rd, R _s	ORRS Rd, Rd, R _s
POP	Pop registers	POP {Rlist}	LDMIA R13!, {Rlist}
PUSH	Push registers	PUSH {Rlist}	STMDB R13!, {Rlist}
ROR	Rotate Right	ROR Rd, R _s	MOVS Rd, Rd, ROR R _s
SBC	Subtract with Carry	SBC Rd, R _s	SBCS Rd, Rd, R _s
STMIA	Store Multiple	STMIA Rb!, {Rlist}	STMIA Rb!, {Rlist}
STR	Store word	STR Rd, [Rb, R ₀]	STR Rd, [Rb, R ₀]
STRB	Store byte	STRB Rd, [Rb, R ₀]	STRB Rd, [Rb, R ₀]
STRH	Store halfword	STRH Rd, [Rb, R ₀]	STRH Rd, [Rb, R ₀]
SWI	Software Interrupt	SWI Value8	SWI Value8

Egs

ARM vs. Thumb codes

ARM code

ARMDivide

; IN: r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

```
    MOV    r3,#0
loop
    SUBS   r0,r0,r1
    ADDGE r3,r3,#1
    BGE   loop
    ADD    r2,r0,r1
```

$5 \times 4 = 20$ bytes

Thumb code

ThumbDivide

; IN: r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

```
    MOV    r3,#0
loop
    ADD    r3,#1
    SUB    r0,r1
    BGE   loop
    SUB    r3,#1
    ADD    r2,r0,r1
```

$6 \times 2 = 12$ bytes

Thumb Instruction Set: Summary I

Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set
ADC	Add with Carry	✓		✓
ADD	Add	✓	✓	✓ ⁽¹⁾
AND	AND	✓		✓
ASR	Arithmetic Shift Right	✓		✓
B	Unconditional branch	✓		
Bxx	Conditional branch	✓		
BIC	Bit Clear	✓		✓
BL	Branch and Link			
BX	Branch and Exchange	✓	✓	
CMN	Compare Negative	✓		✓
CMP	Compare	✓	✓	✓
EOR	EOR	✓		✓

Lo: R0 to R7

Hi: R8 to R15

Thumb Instruction Set: Summary2

Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set
LDMIA	Load multiple	✓		
LDR	Load word	✓		
LDRB	Load byte	✓		
LDRH	Load halfword	✓		
LSL	Logical Shift Left	✓		✓
LDSB	Load sign-extended byte	✓		
LDSH	Load sign-extended halfword	✓		
LSR	Logical Shift Right	✓		✓
MOV	Move register	✓	✓	✓②
MUL	Multiply	✓		✓
MVN	Move Negative register	✓		✓

Lo: R0 to R7
Hi: R8 to R

Try writing with Thumb instructions wherever possible

*

To form a scalar product of two vectors:

	MOV	r11, #20	initialize loop counter
	MOV	r10, #0	initialize total
LOOP	LDR	r0, [r8], #4	get first component..
	LDR	r1, [r9], #4	. .and second
	MLA	r10, r0, r1, r10	accumulate product
	SUBS	r11, r11, #1	decrement loop counter
	BNE	LOOP	

Last Notes on THUMB

You may have noticed from the Thumb instruction set list and from the Thumb register usage table that there is no direct access to the *cpsr* or *spsr*. In other words, there are no MSR- and MRS-equivalent Thumb instructions.

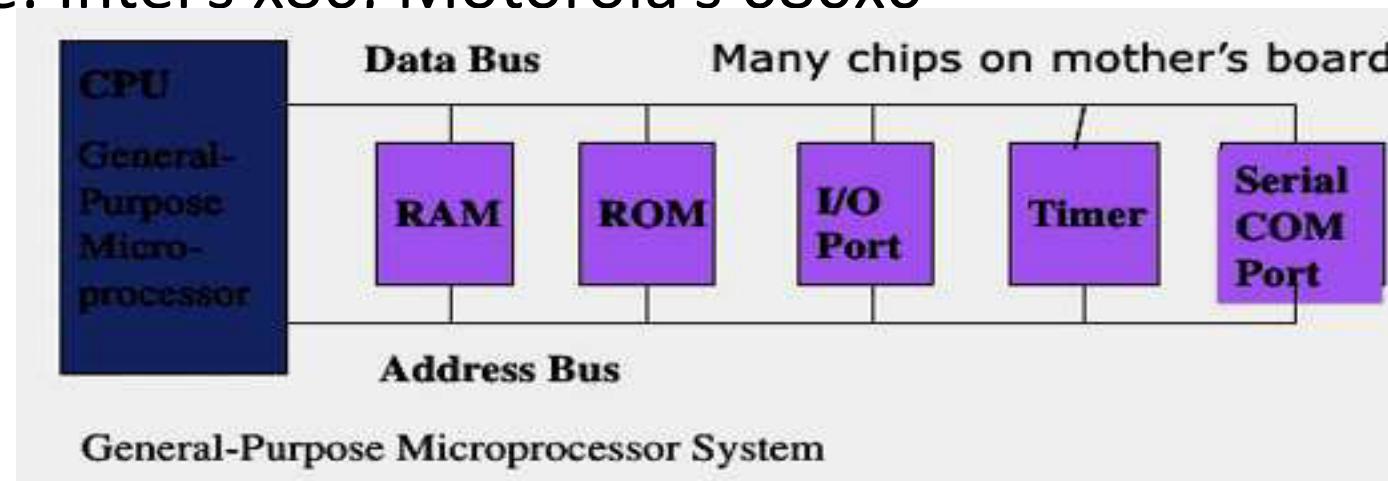
To alter the *cpsr* or *spsr*, you must switch into ARM state to use MSR and MRS. Similarly, there are no coprocessor instructions in Thumb state. You need to be in ARM state to access the coprocessor for configuring cache and memory management.

Introduction to ARM Cortex M4 Microcontroller

Embedded Systems – Microcontrollers vs Microprocessors

- **Microprocessors:**

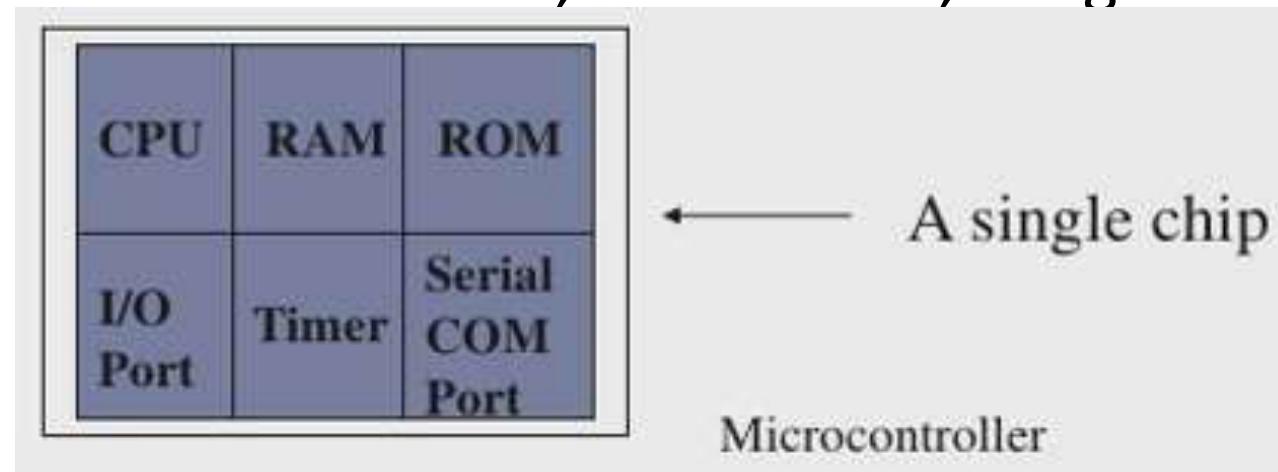
- CPU for Computers
- No RAM, ROM, I/O on CPU chip itself
- Example: Intel's x86. Motorola's 680x0



Embedded Systems – Microcontrollers vs Microprocessors

- **Microcontroller:**

- A smaller computer
- On-chip RAM, ROM, I/O ports...
- Example: Motorola's 6811, Intel's 8051, Zilog's Z8 and PIC 16X



Introduction to ARM Cortex M4 Microcontroller

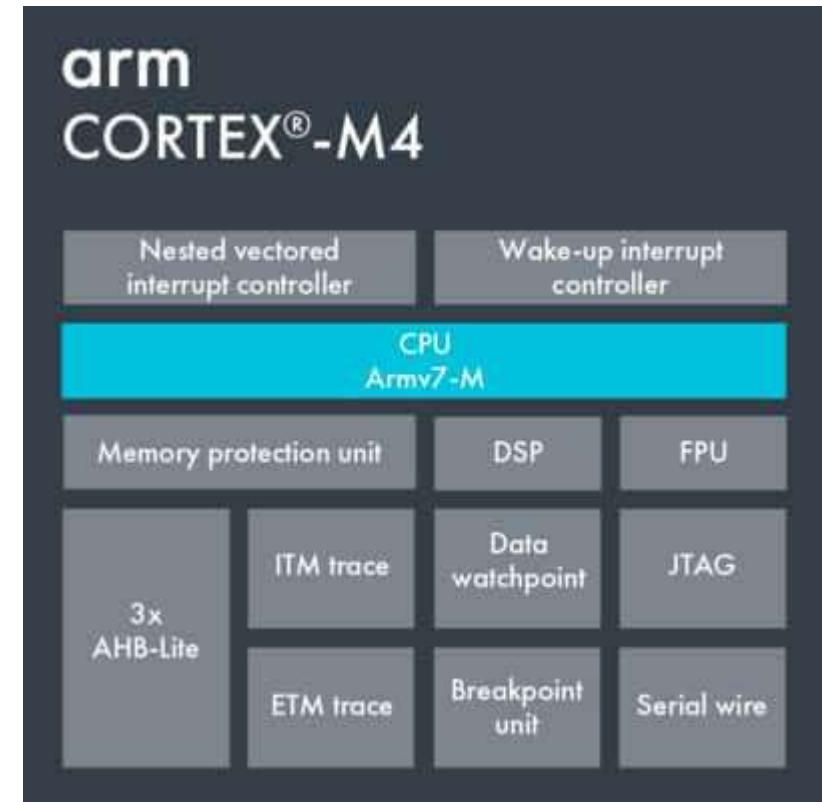
Cortex-M: Processors in these profiles are used for the **development of microcontrollers based embedded systems.** The Cortex-M family consists of Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, Cortex-M4 and Cortex-M7.

Cortex-A: Processors in this profile are used in **high performance application devices like mobile/cellular phones.**

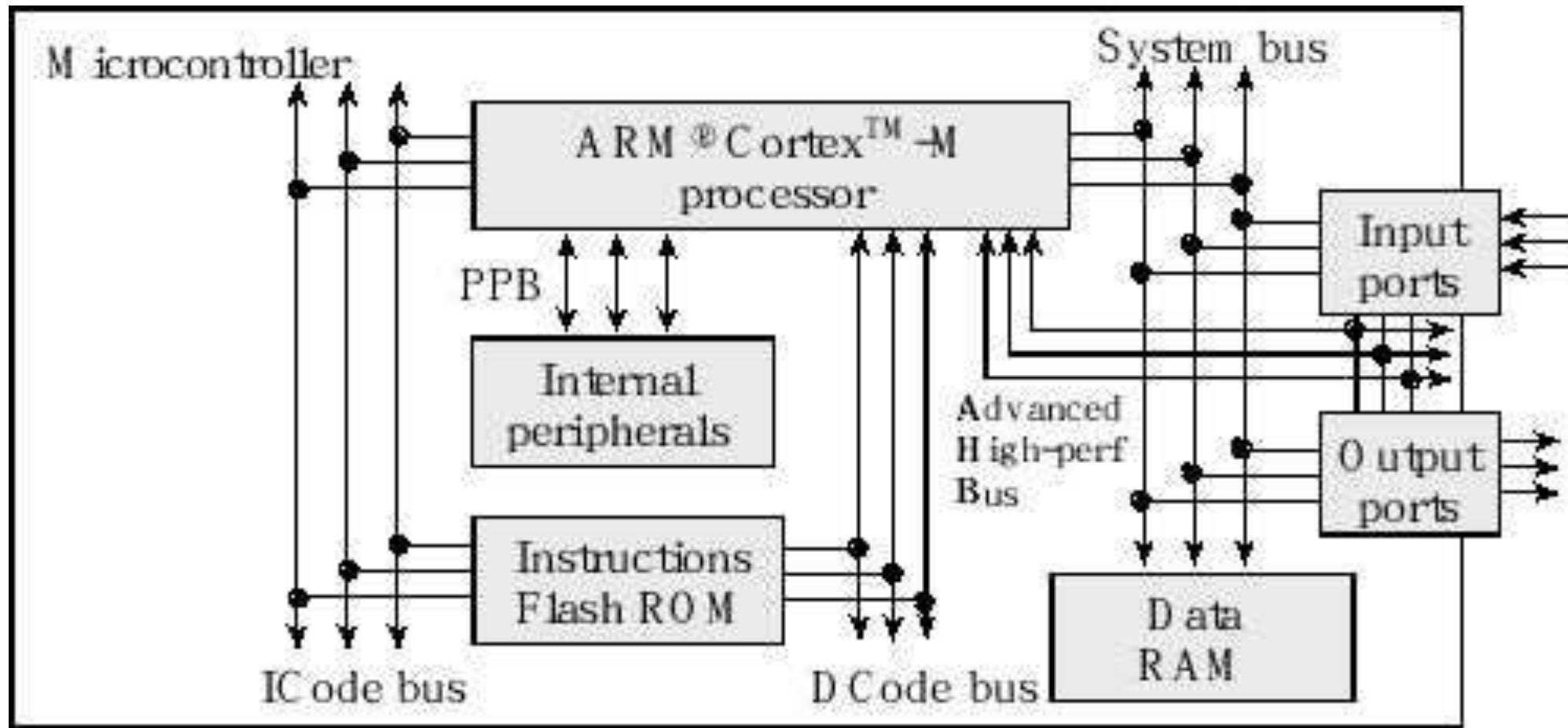
Cortex-R: Main market of processors of this profile are in **the real time application, where less response time is the main target.**

Introduction to ARM Cortex M4

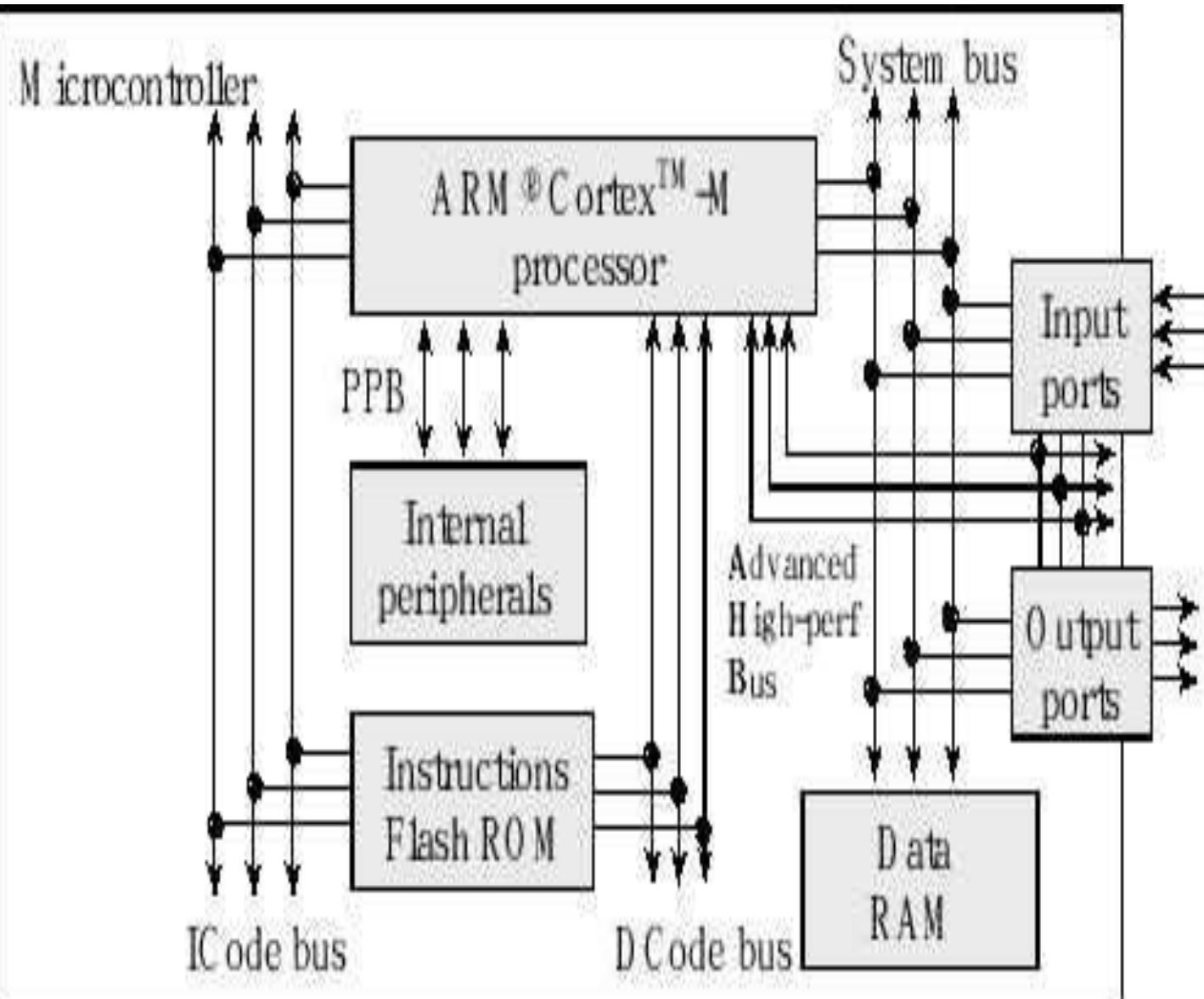
- The **32-bit Arm Cortex-M4** processor core is the first core of the Cortex-M line up to feature **dedicated Digital Signal Processing (DSP) IP blocks**, including an optional Floating-Point Unit (FPU).
- It addresses digital signal control applications that require **efficient, easy-to-use control and signal processing capabilities**, such as the IoT, motor control, power management, embedded audio, industrial and home automation, healthcare and wellness applications.



Harvard architecture of an ARM Cortex –M-4-based microcontroller

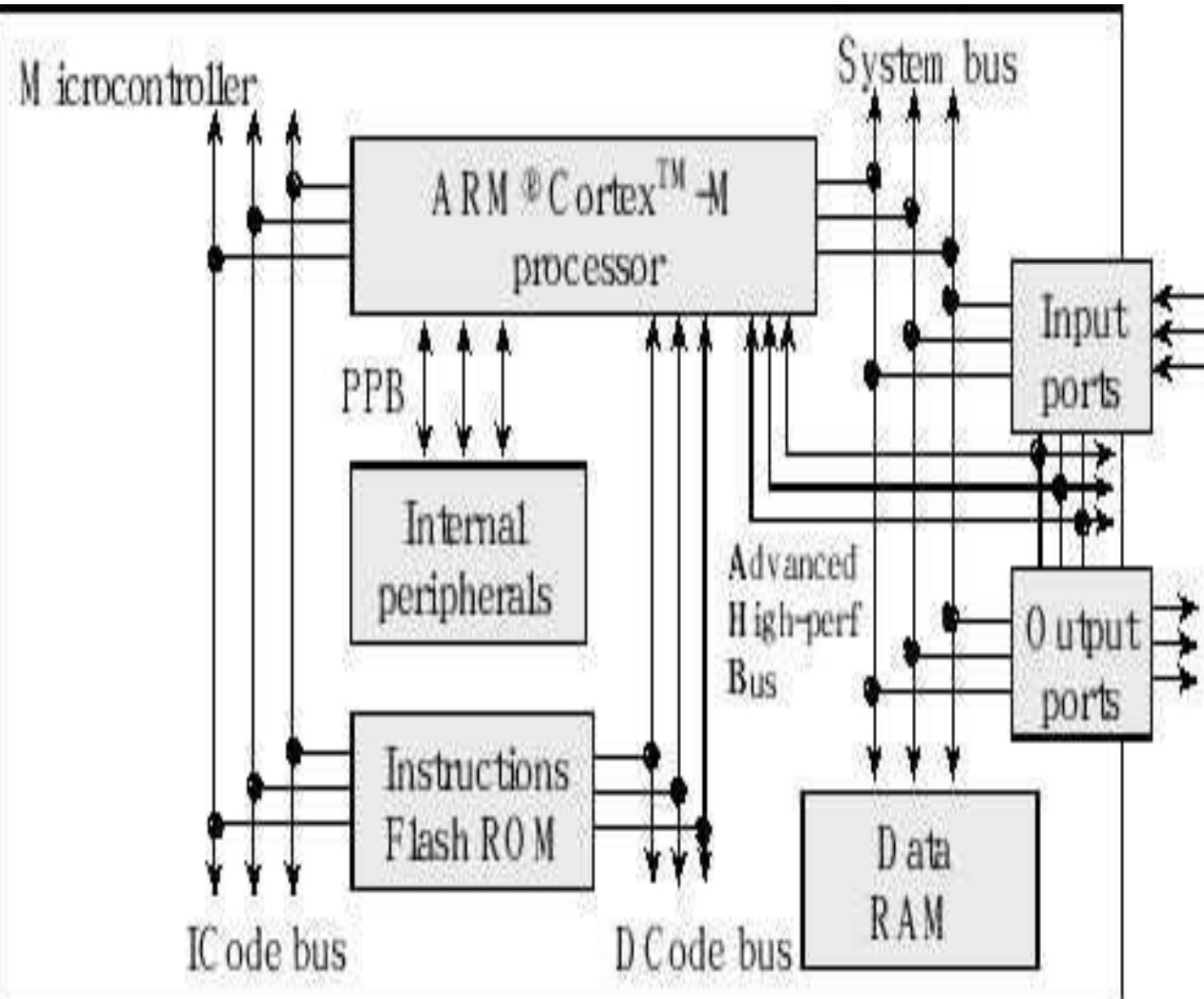


Harvard architecture of an ARM Cortex –M-4-based microcontroller



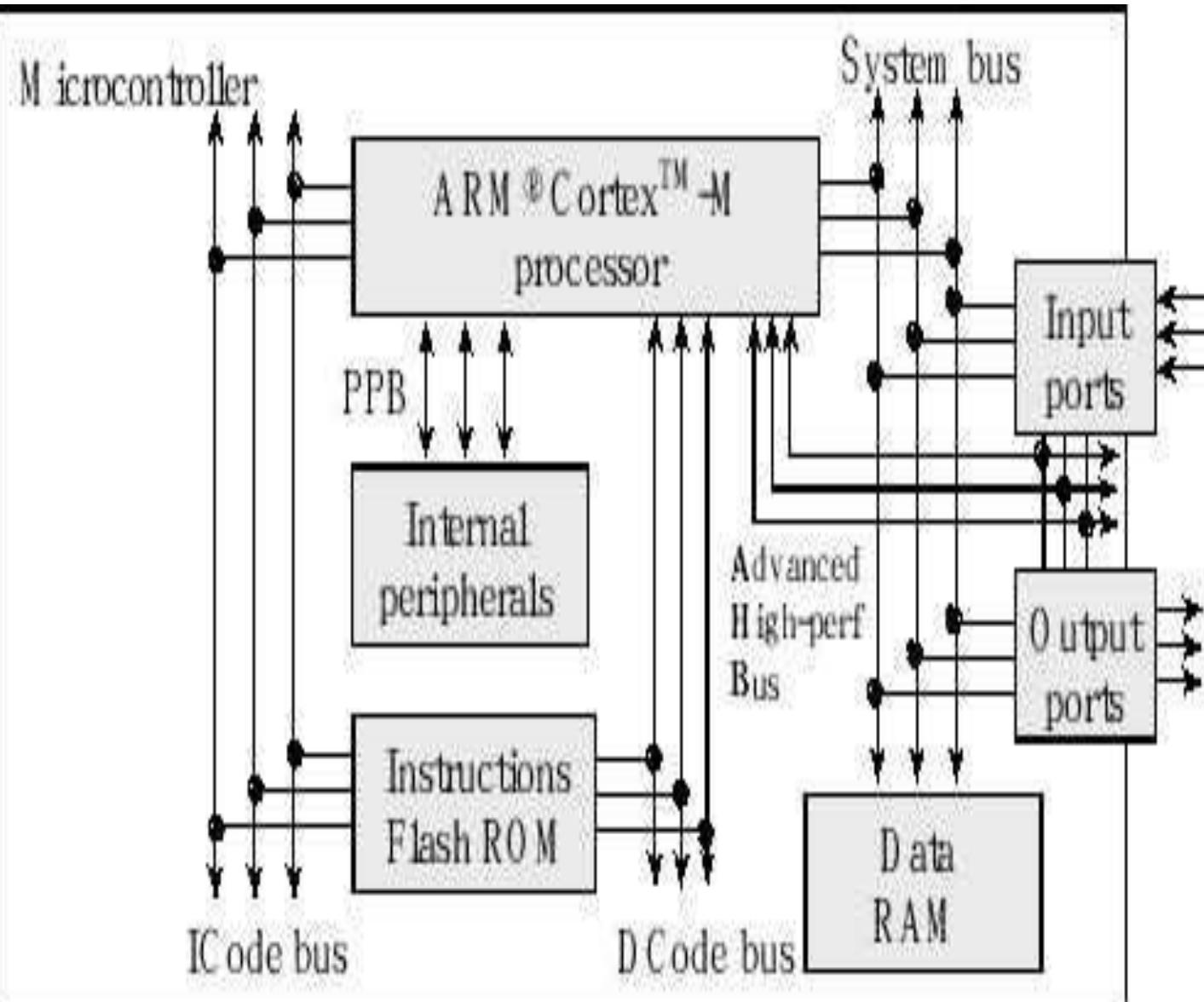
- It has separate data and instruction buses.
- Instruction set combines the high performance typical of a 32-bit processor with high code density typical of 8-bit and 16-bit microcontrollers.
- Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and I/O via the system bus interface.
- On the Cortex M4 there is a second I/O bus for high-speed devices like USB.
- There are many sophisticated debugging features utilizing the DCode bus.

Harvard architecture of an ARM Cortex –M-4-based microcontroller



- The nested vectored interrupt controller (NVIC) manages **interrupts**, which are hardware-triggered software functions.
- Some internal peripherals, like the NVIC communicate directly with the processor via the private peripheral bus (PPB).
- The tight integration of the processor and interrupt controller provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt latency.
- Even though data and instructions are fetched 32-bits at a time, each 8-bit byte has a unique address.
- This means memory and I/O ports are byte addressable. The processor can read or write 8-bit, 16-bit, or 32-bit data.

Harvard architecture of an ARM Cortex –M-4-based microcontroller



- The M4 has an advanced high-performance bus (AHB).
- Having multiple buses means the processor can perform multiple tasks in parallel.
- The following is some of the tasks that can occur in parallel.

ICode bus: Fetch opcodes from ROM

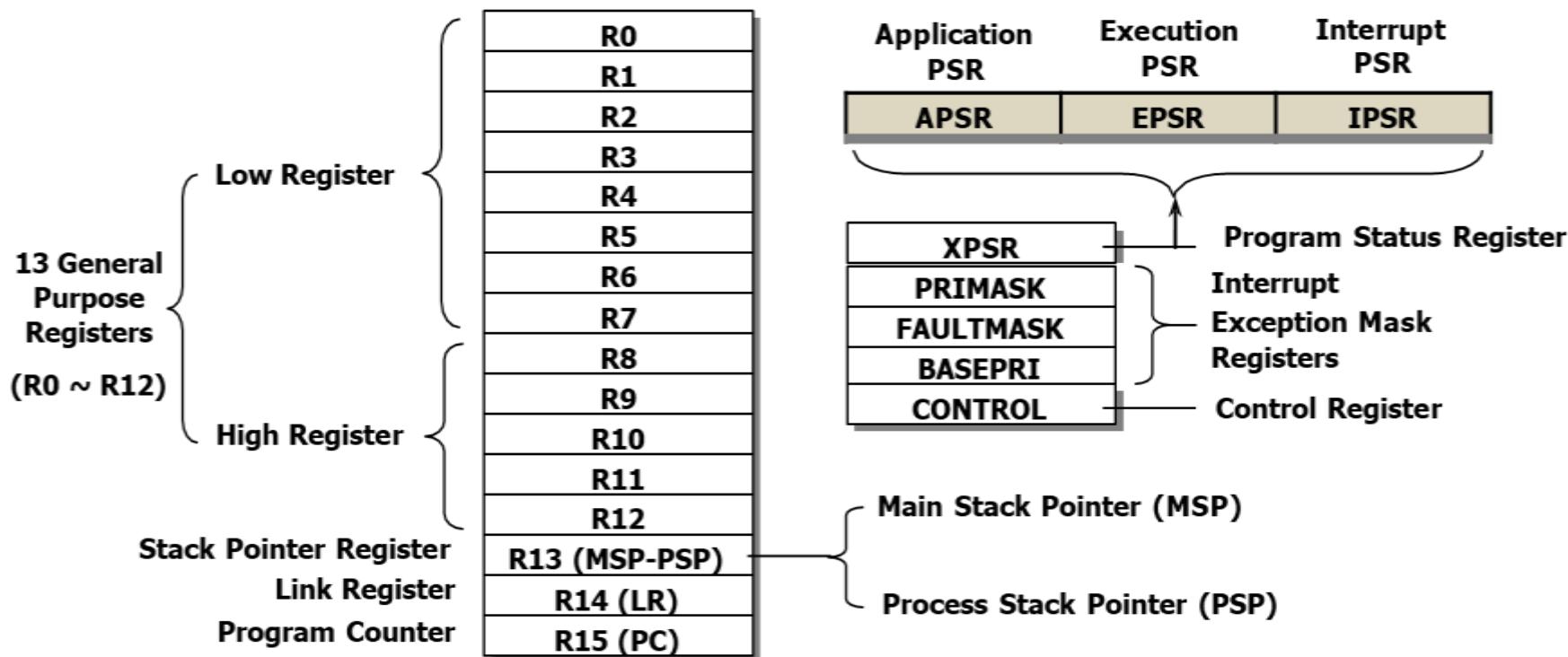
DCode bus: Read constant data from ROM

System bus: Read/write data from RAM or I/O, fetch opcode from RAM

PPB: Read/write data from internal peripherals like the NVIC

AHB: Read/write data from high-speed I/O and parallel ports (M4 only)

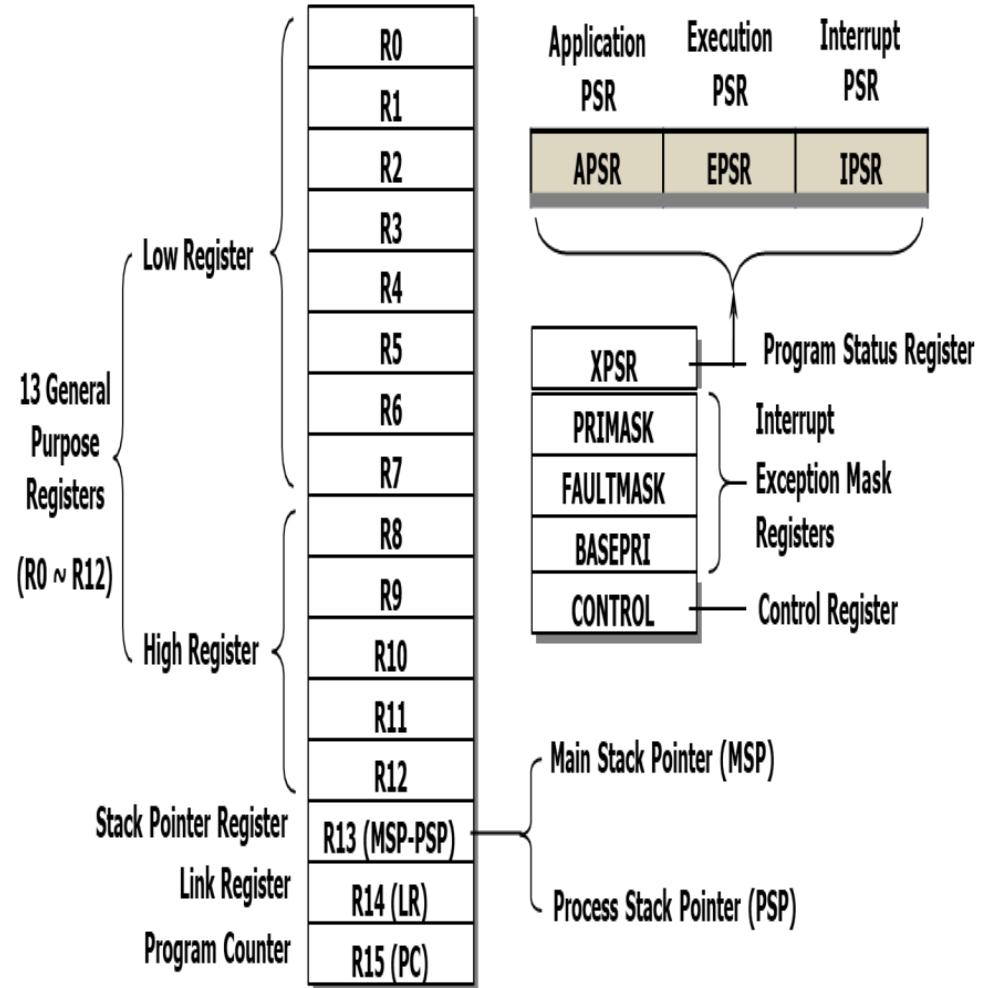
Registers of ARM Cortex –M4-based microcontroller



A structure block diagram of 21 registers in the Cortex®-M4 Core

Registers of ARM Cortex –M4-based microcontroller

- Registers are high-speed storage inside the processor.
- R0 to R12 are general purpose registers and contain either data or addresses.
- Register R13 (also called the stack pointer, SP) points to the top element of the stack.
- Register R14 (also called the link register, LR) is used to store the return location for functions.
- The LR is also used in a special way during exceptions, such as interrupts.
- Register R15 (also called the program counter, PC) points to the next instruction to be fetched from memory. The processor fetches an instruction using the PC and then increments the PC.



The Program Status Registers

- The **ARM Architecture Procedure Call Standard, AAPCS**, part of the ARM Application Binary Interface (ABI), uses registers R0, R1, R2, and R3 to pass input parameters into a C function.
- Functions must preserve the values of registers R4–R11. Also according to AAPCS we place the return parameter in Register R0. AAPCS requires we push and pop an even number of registers to maintain an 8-byte alignment on the stack. The SP will always be the main stack pointer (MSP), not the Process Stack Pointer (PSP).
- There are three status registers named :-
 1. Application Program Status Register (APSR)
 2. The Interrupt Program Status Register (IPSR)
 3. The Execution Program Status Register (EPSR)

Bits	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4	3	2	1	0		
APSR	N	Z	C	V	Q					GE*	Reserved											
IPSR	Reserved												Exception Number									
EPSR	Reserved			ICI/IT	T	Reserved			ICI/IT	Reserved												

(a) Three individual register – APSR, IPSR and EPSR.

Bits	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4	3	2	1	0
PSR	N	Z	C	V	Q	ICI/IT	T													Exception Number

The program status register of the ARM® Cortex M4 processor.

The Program Status Registers

- These registers can be accessed individually or in combination as the **Program Status Register** (PSR).
 - The N, Z, V, C, and Q bits give information about the result of a previous ALU operation.
 - In general, the **N bit** is set after an arithmetical or logical operation signifying whether or not the result is negative.
 - Similarly, the **Z bit** is set if the result is zero. The **C bit** means carry and is set on an unsigned overflow. The **V bit** signifies signed overflow.
 - The **Q bit** indicates that “saturation” has occurred.
 - The **T bit** will be 1 to indicate the ARM Cortex M processor is executing Thumb instructions.
 - The ISR NUMBER(Exception Number) indicates which interrupt if any the processor is handling.

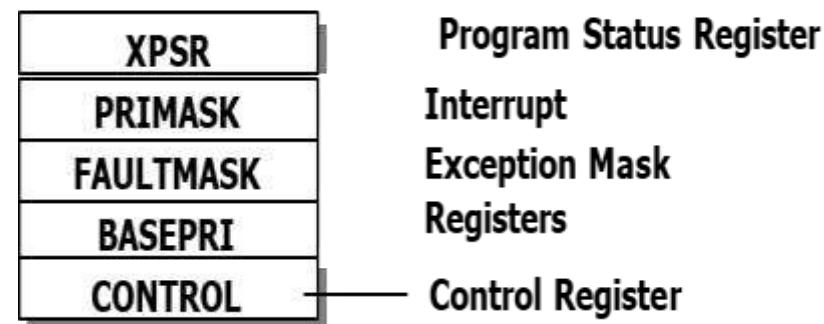
Bits	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4	3	2	1	0
APSR	N	Z	C	V	Q				GE*	Reserved										
IPSR	Reserved											Exception Number								
EPSR	Reserved		ICI/IT	T	Reserved			ICI/IT	Reserved											

(a) Three individual register – APSR, IPSR and EPSR.

Bits	31 30 29 28 27	26:25	24	23:20	19:16	15:10	9	8 7 6 5 4 3 2 1 0
PSR	N Z C V Q	ICI/IT	T		GE*	ICI/IT		Exception Number

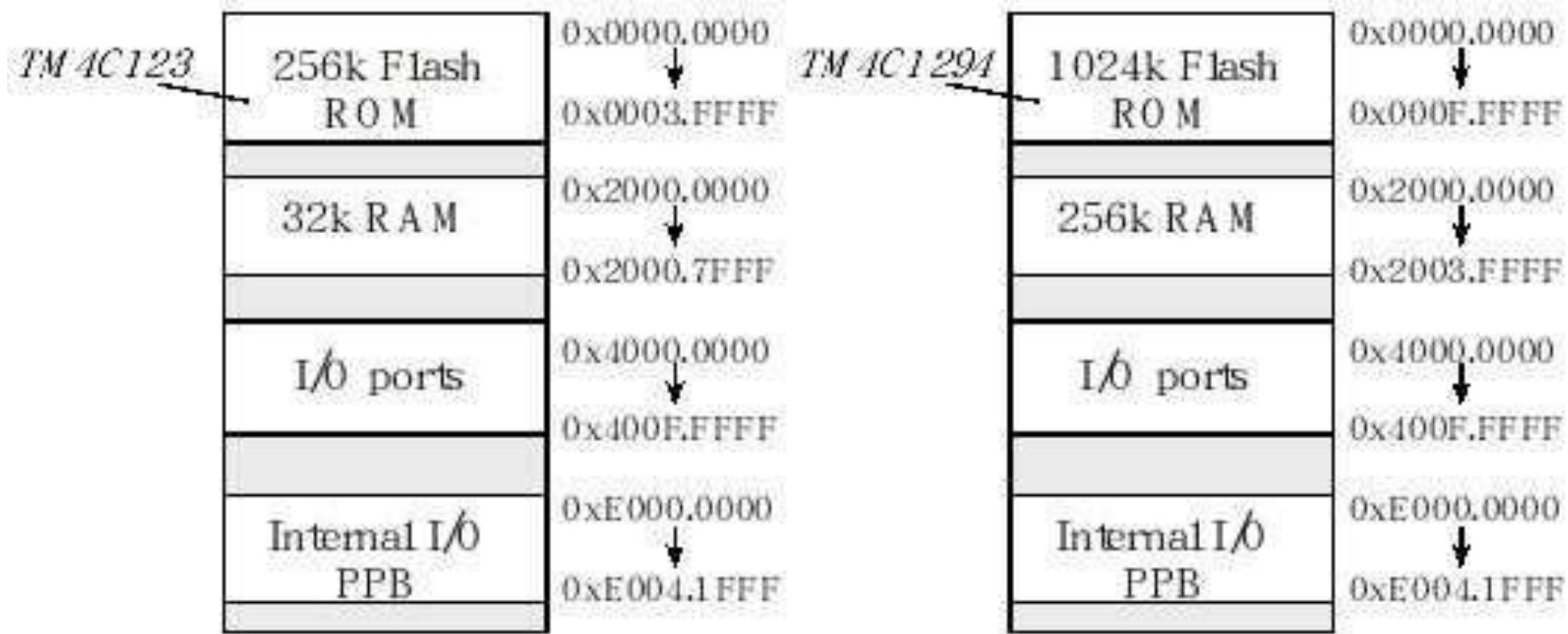
Special Registers

- Bit 0 of the special register **PRIMASK** is the interrupt mask bit. If the Bit is 1, most interrupts and exceptions are not allowed. If the Bit is 0, then interrupts are allowed.
- Bit 0 of the special register **FAULTMASK** is the fault mask bit. If this bit is 1, all interrupts and faults are not allowed. If the bit is 0, then interrupts and faults are allowed.
- The non-maskable interrupt (NMI) is not affected by these mask bits.
- The **BASEPRI** register defines the priority of the executing software. It prevents interrupts with lower or equal priority but allows higher priority interrupts.
- For example if **BASEPRI** equals 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. A lower number means a higher priority interrupt.



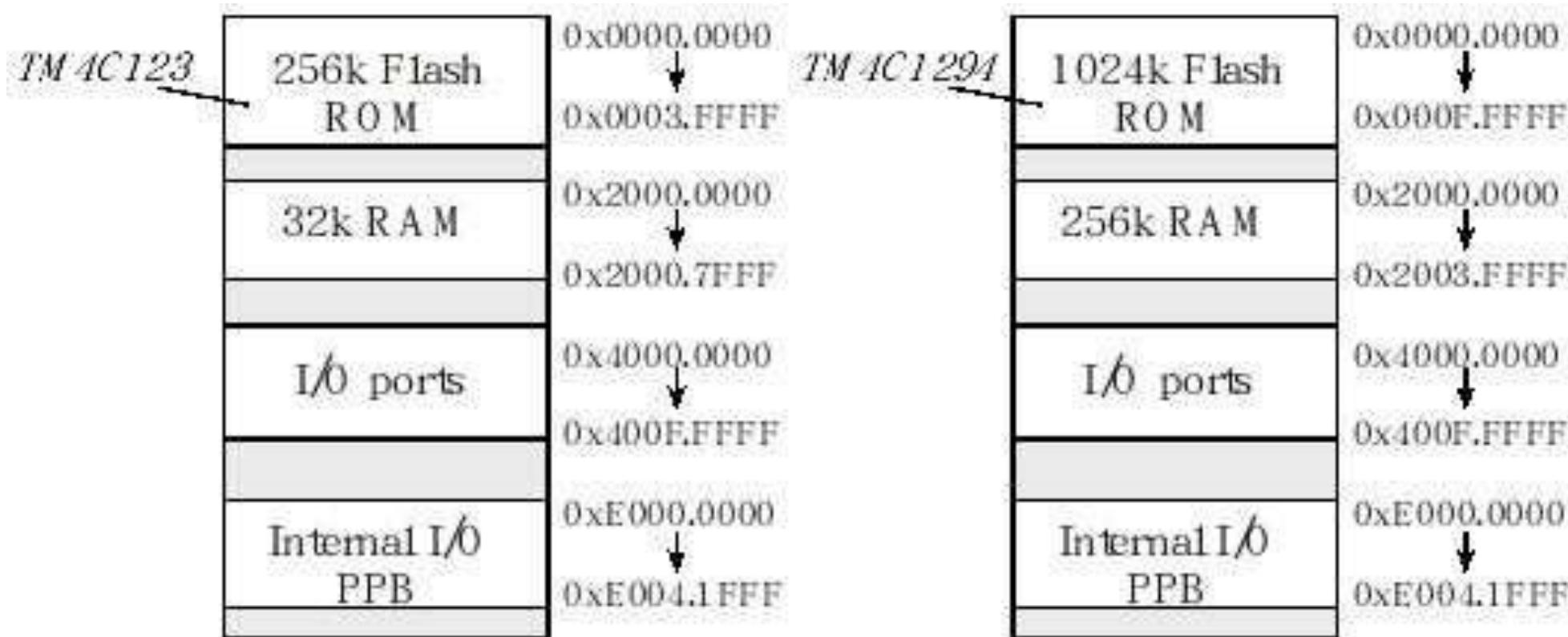
Memory

- Microcontrollers within the same family differ by the amount of memory and by the types of I/O modules.
- All LM3S and TM4C microcontrollers have a Cortex M processor.
- The memory map of TM4C123 is illustrated in Figure below. Although specific for the TM4C123, all ARM Cortex M microcontrollers have similar memory maps.



Memory maps of the TM4C123 and the TM4C1294

Memory



- In general, Flash ROM begins at address 0x0000.0000, RAM begins at 0x2000.0000, the peripheral I/O space is from 0x4000.0000 to 0x5FFFF.FFFF, and I/O modules on the private peripheral bus (PPB) exist from 0xE000.0000 to 0xE00F.FFFF.
- In particular, the only differences in the memory map for the various 180 members of the LM3S/TM4C family are the ending addresses of the flash and RAM.

Big Endian vs Little Endian Formats

- When we store 16-bit data into memory it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Freescale microcomputers implement the **big endian** approach that stores the most significant byte at the lower address.
- Intel microcomputers implement the **little endian** approach that stores the least significant byte at the lower address.
- Cortex M microcontrollers use the little endian format. Many ARM processors are **bigendian**, because they can be configured to efficiently handle both big and little endian data.
- Instruction fetches on the ARM are always little endian. The two ways to store the 16-bit number 1000 (0x03E8) at locations 0x2000.0850 and 0x2000.0851.
- Computers must choose to use either the big or little endian approach when storing 32-bit numbers into memory that is byte (8-bit) addressable.

Address	Data
0x2000.0850	0x03
0x2000.0851	0xE8

Big Endian

Address	Data
0x2000.0850	0xE8
0x2000.0851	0x03

Little Endian

Example of big and little endian formats of a 16-bit number.

Big Endian vs Little Endian Formats

- The big and little endian formats that could be used to store the 32-bit number 0x12345678 at locations 0x2000.0850 through 0x2000.0853. Again the Cortex M uses little endian for 32-bit numbers.

Address	Data
0x2000.0850	0x12
0x2000.0851	0x34
0x2000.0852	0x56
0x2000.0853	0x78

Big Endian

Address	Data
0x2000.0850	0x78
0x2000.0851	0x56
0x2000.0852	0x34
0x2000.0853	0x12

Little Endian

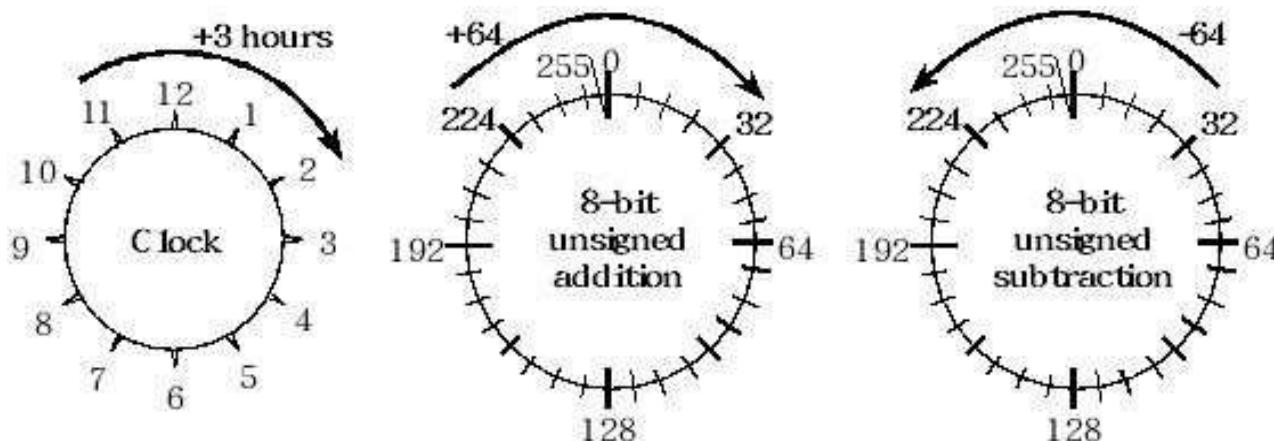
Example of big and little endian formats of a 32-bit number.

Operating Modes

- The processor knows whether it is running in the foreground (i.e., the main program) or in the background (i.e., an interrupt service routine).
- ARM processors define the **foreground** as **thread mode**, and the **background** as **handler mode**.
- Switching between **thread and handler modes occurs automatically**.
- The **processor begins in thread mode**, signified by **ISR_NUMBER=0**. Whenever it is servicing an interrupt it switches to handler mode, signified by setting **ISR_NUMBER** to specify which interrupt is being processed.
- All interrupt service routines run using the **MSP** (main stack pointer).

• Arithmetic Operations (Self Study)

- When software executes arithmetic instructions, the operations are performed by digital hardware inside the processor.
- Even though the design of such logic is complex.
- It is important to remember that arithmetic operations (addition, subtraction, multiplication, and division) have constraints when performed with finite precision on a processor.
- An overflow error occurs when the result of an arithmetic operation cannot fit into the finite precision of the register into which the result is to be stored.
- Eg: when two 32-bit numbers are added or subtracted, the result may not fit back into a 32-bit register. The same addition and subtraction hardware (instructions) can be used to operate on either unsigned or signed numbers. Although we use the same instructions, we must use separate overflow detection for signed and unsigned operations.



Eg: Full Adder

The carry bit is set on addition when crossing the 255-0 boundary.

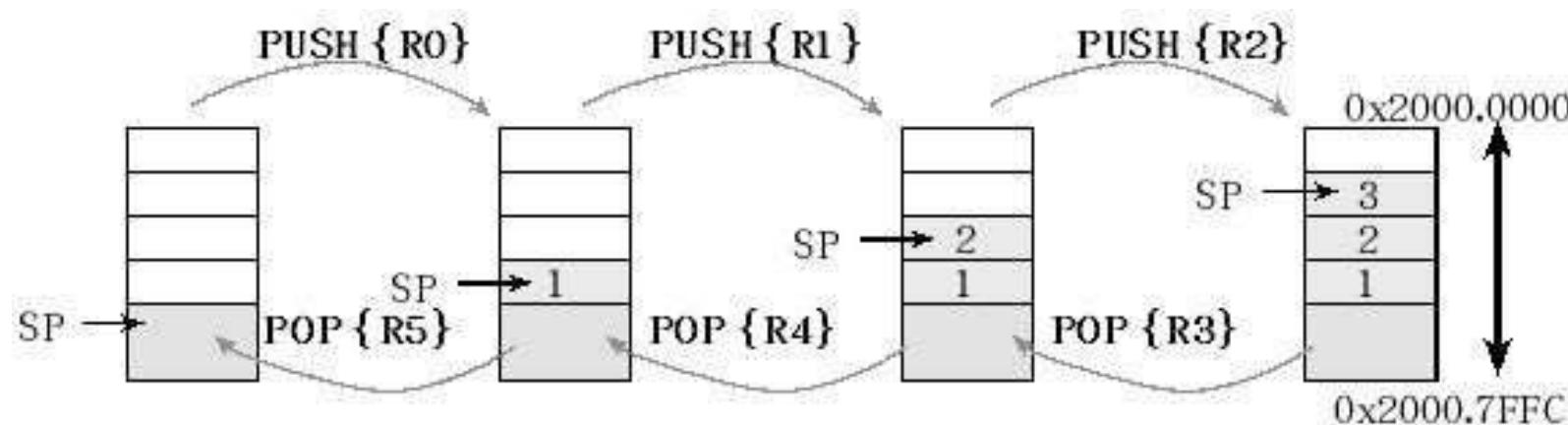
The carry bit is cleared on subtraction when crossing the 255-0 boundary.

Stack (Self Study)

- The **stack** is a last-in-first-out temporary storage. To create a stack, a block of RAM is allocated for this temporary storage.
- On the ARM ® Cortex™-M processor, the stack always operates on 32-bit data. The stack pointer (SP) points to the 32-bit data on the top of the stack.
- The stack grows downwards in memory as we push data on to it so, although we refer to the most recent item as the “top of the stack” it is actually the item stored at the lowest address!
- To push data on the stack, the stack pointer is first decremented by 4, and then the 32-bit information is stored at the address specified by SP.
- To pop data from the stack, the 32-bit information pointed to by SP is first retrieved, and then the stack pointer is incremented by 4. SP points to the last item pushed, which will also be the next item to be popped.

Stack (Self Study)

- The boxes in Figure represent 32-bit storage elements in RAM. The grey boxes in the figure refer to actual data stored on the stack, and the white boxes refer to locations in memory that do not contain stack data.
- This figure illustrates how the stack is used to push the contents of Registers R0, R1, and R2 in that order. Assume Register R0 initially contains the value 1, R1 contains 2, and R2 contains 3. The drawing on the left shows the initial stack. The software executes these six instructions in this order:
 - PUSH {R0}**
 - PUSH {R1}**
 - PUSH {R2}**
 - POP {R3}**
 - POP {R4}**
 - POP {R5}**



Stack (Self Study) Key points to remember

1. Functions should have an equal number of pushes and pops
2. Stack accesses (push or pop) should not be performed outside the allocated area
3. Stack reads and writes should not be performed within the free area
4. Stack push should first decrement SP, then store the data
5. Stack pop should first read the data, and then increment SP

ARM7 based LPC2148 Microcontroller

- The full form of an ARM is an advanced reduced instruction set computer ([RISC](#)) machine, and it is a 32-bit processor architecture expanded by ARM holdings. The applications of an ARM processor include several microcontrollers as well as processors. The architecture of an ARM processor was licensed by many corporations for designing ARM processor-based SoC products and CPUs. This allows the corporations to manufacture their products using ARM architecture. Likewise, all main semiconductor companies will make ARM-based SOCs such as Samsung, Atmel, TI etc.
- **What is an ARM7 Processor?**
- ARM7 processor is commonly used in embedded system applications. Also, it is a balance among classic as well as new-Cortex sequence. This processor is tremendous in finding the resources existing on the internet with excellence documentation offered by NXP Semiconductors. It suits completely for an apprentice to obtain in detail hardware & software design implementation.

LPC2148 Microcontroller Features

- The LPC2148 microcontroller is designed by Philips (NXP Semiconductor) with several in-built features & peripherals. Due to these reasons, it will make more reliable as well as the efficient option for an application developer. **LPC2148 is a 16-bit or 32-bit microcontroller based on ARM7 family.**

Features of LPC2148

- The LPC2148 is a 16 bit or 32 bit ARM7 family based microcontroller and available in a small LQFP64 package.
- ISP (in system programming) or IAP (in application programming) using on-chip boot loader software.
- On-chip static RAM is 8 kB-40 kB; on-chip flash memory is 32 kB-512 kB, the wide interface is 128 bit, or accelerator allows 60 MHz high-speed operation.
- It takes 400 milliseconds time for erasing the data in full chip and 1 millisecond time for 256 bytes of programming.
- Embedded Trace interfaces and Embedded ICE RT offers real-time debugging with high-speed tracing of instruction execution and on-chip Real Monitor software.
- It has 2 kB of endpoint RAM and USB 2.0 full speed device controller. Furthermore, this microcontroller offers 8kB on-chip RAM nearby to USB with DMA.

LPC2148 Microcontroller Features

Features of LPC2148 (Contd...)

- One or two 10-bit ADCs offer 6 or 14 analogs i/p/s with low conversion time as 2.44 μ s/ channel.
- Only 10 bit DAC offers changeable analog o/p.
- External event counter/32 bit timers-2, PWM unit, & watchdog.
- Low power RTC (real time clock) & 32 kHz clock input.
- Several serial interfaces like two 16C550 UARTs, two I2C-buses with 400 kbit/s speed.
- 5 volts tolerant quick general purpose Input/output pins in a small LQFP64 package.
- Outside interrupt pins-21.
- 60 MHz of utmost CPU CLK-clock obtainable from the programmable-on-chip phase locked loop by resolving time is 100 μ s.
- The incorporated oscillator on the chip will work by an exterior crystal that ranges from 1 MHz-25 MHz
- The modes for power-conserving mainly comprise idle & power down.
- For extra power optimization, there are individual enable or disable of peripheral functions and peripheral CLK scaling.

ADC in LPC2148 ARM7 Microcontroller

- The ADC in LPC2148 ARM7 Microcontroller is 10-bit successive approximation analog to digital converter.
- LPC2148 has two inbuilt ADC Modules, named as ADC0 & ADC1.
- ADC0 has 6-Channels (AD0.1-AD0.6).
- ADC1 has 8-Channels (AD1.0-AD1.7).
- ADC operating frequency is 4.5 MHz (max.), operating frequency decides the conversion time.
- Supports power down mode.
- Burst conversion mode for single or multiple inputs.
- There are several registers associated with ADC feature explained in the next slide

ADC 0		ADC 1	
ADC Channel 0	LPC2148 Pins	ADC Channel 1	LPC2148 Pins
AD0.1	P0.28	AD1.0	P0.6
AD0.2	P0.29	AD1.1	P0.8
AD0.3	P0.30	AD1.2	P0.10
AD0.4	P0.25	AD1.3	P0.12
AD0.6	P0.4	AD1.4	P0.13
AD0.7	P0.5	AD1.5	P0.15
		AD1.6	P0.21
		AD1.7	P0.22

Registers of ADC in LPC2148 Microcontroller (Self Study)

Register Name	Function
ADCR	A/D Control Register: The ADCR register must be written to select the operating mode before A/D conversion can occurs.
ADGDR	A/D Global Data Register: This register contains ADC's DONE bit and the result of the most recent A/D conversion.
ADSTAT	A/D Status Register: This register contains DONE and OVERRUN flag for all the A/D Channels, as well as the A/D interrupt flag.
ADGSR	A/D Global Start Register: This address can be written (in the AD0 address range) to start conversions in both A/D converters simultaneously.
ADINTEN	A/D Interrupt Enable Register: This register contains enable bits that allow the DONE flag of each A/D channel to be included or excluded from contributing to the generation of an A/D interrupt.
ADDRx	A/D Channel x Data: 'x' varies from 0 to 7