Microcontrollers and Embedded Systems – Module 1

Differences between Microprocessors and Microcontrollers

Parameter	Microprocessor	Microcontroller
Definition	IC that has only CPU inside it and RAM, ROM and other peripherals need to be added externally	IC that has integration of CPU RAM, ROM and other peripherals
Tasks used	General purpose	Specific purpose
Preference	Relationship between input and output is not clearly defined	Relationship between input and output is clear defined
Suitability for embedded applications	Not suitable	Ideally suitable due to lo power consumptions
Price	Costlier	Cheaper
Speed	High (1Ghz)	Slow (20-50Mhz)
Memory	External (in GBs)	KBs of internal for RAM and ROM

ARM Features:

- **ARM** stands for **Advanced RISC Machine**. It is one of the most licensed and extensive processor cores in the world.
- Specially used in portable devices like digital cameras, mobile phones, home network modules, wireless communication technologies, etc..
- Advanced RISC Machine Architecture is better than x86.
- ARM Processor is not only limited to mobile phones but is also used in Fugaku, the world's fastest supercomputer.
- ARM can be used as a microprocessor, microcontroller and both.

RISC vs CISC architectures:

RISC and CISC are two different types of computer architectures that are used to design the microprocessors that are found in computers. The fundamental difference between RISC and CISC is that **RISC (Reduced Instruction Set Computer)** includes simple instructions and takes one cycle, while the **CISC (Complex Instruction Set Computer)** includes complex instructions and takes multiple cycles.

Sl. No.	RISC	CISC
1.	It stands for Reduced Instruction Set Computer.	It stands for Complex Instruction Set Computer.
2.	It is a microprocessor architecture that uses small instruction set of uniform length.	This offers hundreds of instructions of different sizes to the users.
3.	These simple instructions are executed in one clock cycle.	This architecture has a set of special purpose circuits which help execute the instructions at a high speed.
4.	These chips are relatively simple to design.	These chips are complex to design.
5.	They are inexpensive.	They are relatively expensive.
6.	Examples of RISC chips include SPARC, POWER PC.	Examples of CISC include Intel architecture, AMD.
7.	It relies on the intelligence of software and has less complex hardware	It relies on hardware complexity
8.	It has fixed-length encodings for instructions.	It has variable-length encodings of instructions.
9.	Simple addressing formats are supported.	The instructions interact with memory using complex addressing modes.
10.	It doesn't support arrays.	It has a large number of instructions. It supports arrays.
11.	It doesn't use condition codes.	Condition codes are used.
12.	Registers are used for procedure arguments and return addresses.	The stack is used for procedure arguments and return addresses.

Design Rules of RISC Processor

The different major design rules that a RISC processor includes are as follows:

1. **Instructions**: RISC exhibit reduced instruction sets approach. In this case, there are various simple instructions each having a fixed length so one instruction will get executed in a single cycle. This supports the parallel operation. In CISC, the instructions are of multiple sizes and this makes the parallelism in operation quite difficult.

2. **Synthesis**: Since few instructions are supported, many other instructions are synthesized from existing instructions. This can also lead to code bloating. However, CISC supports multiple instructions and makes programs simpler.
3. **Fixed instruction size**: RISC was designed with all instructions of the same length. This helps in prefetching of instructions while current instructions are being executed. However, CISC has variable instruction length and multiple clock cycles might be required.
4. **Pipelining**: RISC facilitates instruction pipelining. Basically, here the various subdivided instructions into simple ones are executed parallelly in pipelined format. Through pipelining, during decoding of previous instruction, the next one can be fetched and this provides advancement in operation in each cycle thereby offering maximal throughput.
5. **Registers**: The way the RISC processor operates there exists a requirement of large memory space and thus, it includes a large general-purpose register set. There is no specified register for data or address as here the registers act as a local storage destination for all operations.
6. **Load-store technique**: The whole operation takes place through the data existing in the register. In this case, using individual load and store instructions data is moved between register and memory. This provides an advantage in terms that unnecessarily, multiple memory accesses will not be required. Direct memory access is prohibited in RISC.

ARM Design Philosophy

There are a number of physical features that have driven the ARM processor design.

1. Small to reduce power consumption and extend battery operation
2. High code density
3. Price sensitive and use slow and low-cost memory devices.
4. Reduce the area of the die taken up by the embedded processor.
5. Hardware debug technology
6. ARM core is not a pure RISC architecture

Instruction Set for Embedded Systems

The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications:

■ Variable cycle execution for certain instructions—Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred.

■ Inline barrel shifter leading to more complex instructions—The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction.

■ Thumb 16-bit instruction set—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions.

■ Conditional execution—An instruction is only executed when a specific condition has been satisfied.
■ Enhanced instructions—The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16-bit multiplier operations and saturation.

Embedded Systems Hardware based on ARM

An embedded system is a combination of computer hardware and software designed for a specific function. Embedded systems may also function within a larger system. The systems can be programmable or have a fixed functionality.

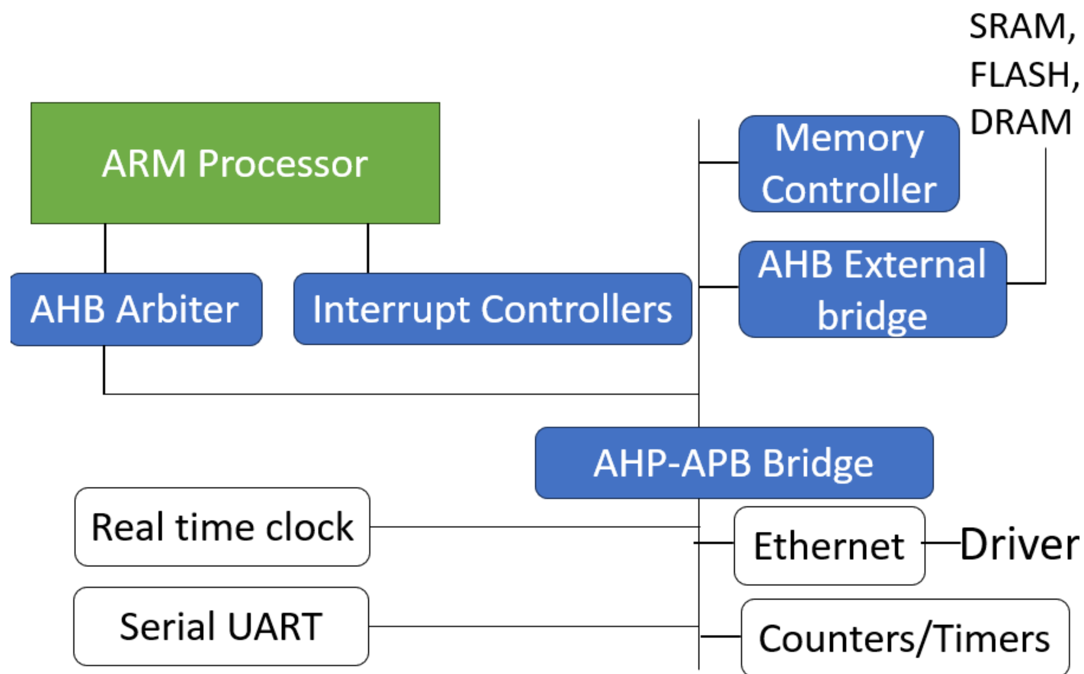


Figure: Embedded Systems Hardware architecture

The architecture for ARM based Embedded systems hardware is depicted. It consists of ARM processor for execution of instructions. The data flow from ARM can be with external memory. External memory can be in the form of SRAM, FLASH and DRAM. The external memory is accessed using ARM High Speed bus (AHB). Since multiple requests for same bus might exist, there is an AHB Arbiter who will decide the accessibility to the AHB. Further, interrupts might be caused by devices and are managed through Interrupt controllers. The low speed devices like the peripherals, UART devices, Ethernet drivers and the like are accessed through ARM Peripheral bus. There exists a bridge between AHB and APB for transition of bus speeds.

ARM Bus Technology

Embedded systems use different bus technologies. The most common PC bus technology, the Peripheral Component Interconnect (PCI) bus, logy is external or

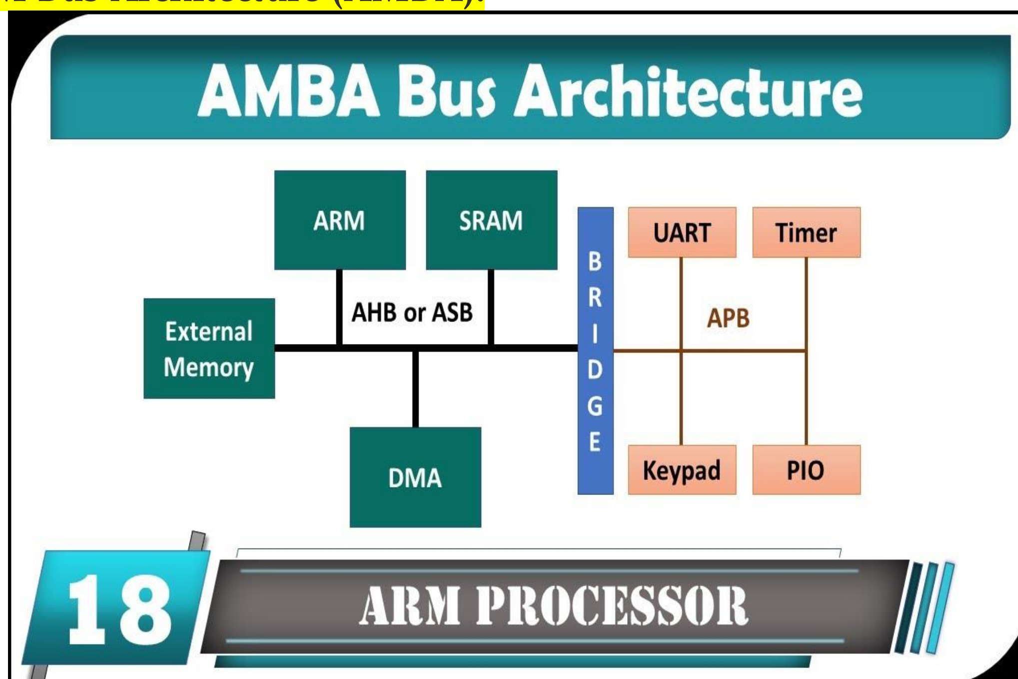
off-chip. However, embedded devices use an on-chip bus that is internal to the chip and allows different peripheral devices to be interconnected with an ARM core.
There are two different classes of devices attached to the bus.

- **Bus master**(**ARM processor core**)—a logical device capable of initiating a data transfer with another device across the same bus.
- **Bus slaves**(**Peripherals**)—logical devices capable only of responding to a transfer request from a bus master device.

A bus has two architecture levels.

1. physical level — that covers the electrical characteristics and bus width (16, 32, or 64 bits).
2. Second level deals with protocol—the logical rules that govern the communication between the processor and a peripheral.

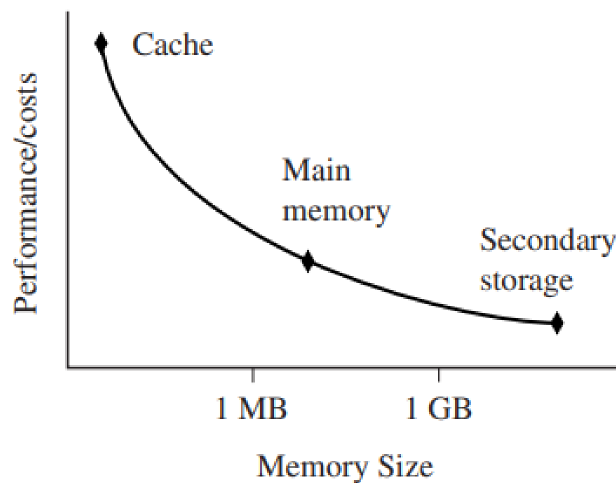
ARM Bus Architecture (AMBA):



ARM System Bus is used for connecting internal communication in the embedded systems designed with ARM. AHB – ARM High Performance Bus with larger bandwidths (64 /128 bit) is used to connect with external memory. Multi-layer AHB and AHB Lite is also provided to support variable speed accesses to external components. APB – ARM Peripheral Bus with slower bandwidth to devices like peripherals. AHB New interconnects supports multiple processors, supporting operations in parallel

Memory of Embedded System

An embedded system has to have some form of memory to store and execute code. There exists a storage tradeoff in the memory as shown. If memory size is increased, storage capacity increased but performance with respect to memory access time degrades.



The number of cycles required for fetching ARM and Thumb instructions is given as:

Fetching instructions from memory.

Instruction size	8-bit memory	16-bit memory	32-bit memory
ARM 32-bit	4 cycles	2 cycles	1 cycle
Thumb 16-bit	2 cycles	1 cycle	1 cycle

Types of Memory

- Read-only memory (ROM) is the least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed. Many devices also use a ROM to hold boot code.
- Flash ROM can be written to as well as read. It is slow to write. Its main use is for holding the device firmware. The erasing and writing of flash ROM are completely software controlled.
- Dynamic random-access memory (DRAM) is the most commonly used RAM for devices. It has the lowest cost per megabyte. DRAM is dynamic it needs to have its storage cells refreshed and given a new electronic charge every few milliseconds, so you need to set up a DRAM controller before using the memory.
- Static random-access memory (SRAM) is faster. The RAM does not require refreshing. The access time is shorter. Higher cost.
- Synchronous dynamic random-access memory (SDRAM) run at much higher clock speeds and it synchronizes itself.

Peripherals

- Forms the outside world interaction of Embedded Systems.

- A peripheral device performs input and output functions for the chip by connecting to other devices that are off-chip.
- Each peripheral device usually performs a single function.
- Peripherals range from a simple serial communication to complex 802.11 wireless device.
- All ARM peripherals are memory mapped—the programming interface is a set of memory-addressed registers.
- Specialized peripherals called as Controllers that implement higher levels of functionality. Two important types

1. Memory controllers
2. Interrupt controllers.

Memory Controllers

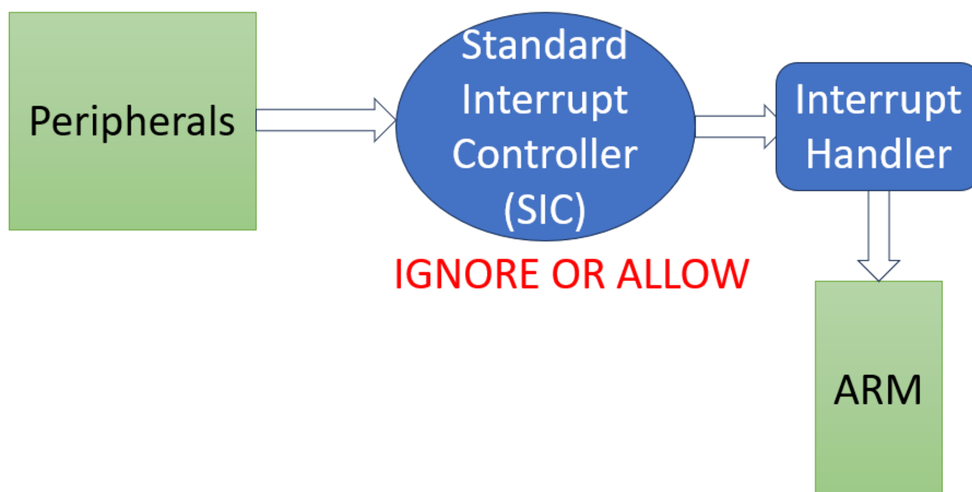
- Connect different types of memory to the processor bus.
- On power-up a memory controller is configured in hardware to allow certain memory devices to be active.
- Some memory devices must be set up by software.

Interrupt Controllers

Interrupts are raised to gain attention. In this case, interrupts raised by peripherals to get attention of ARM processor. Interrupts go through Interrupt controllers. There are two types of interrupt controller available for the ARM processor:

1. Standard interrupt controller (SIC).
2. Vector interrupt controller (VIC).

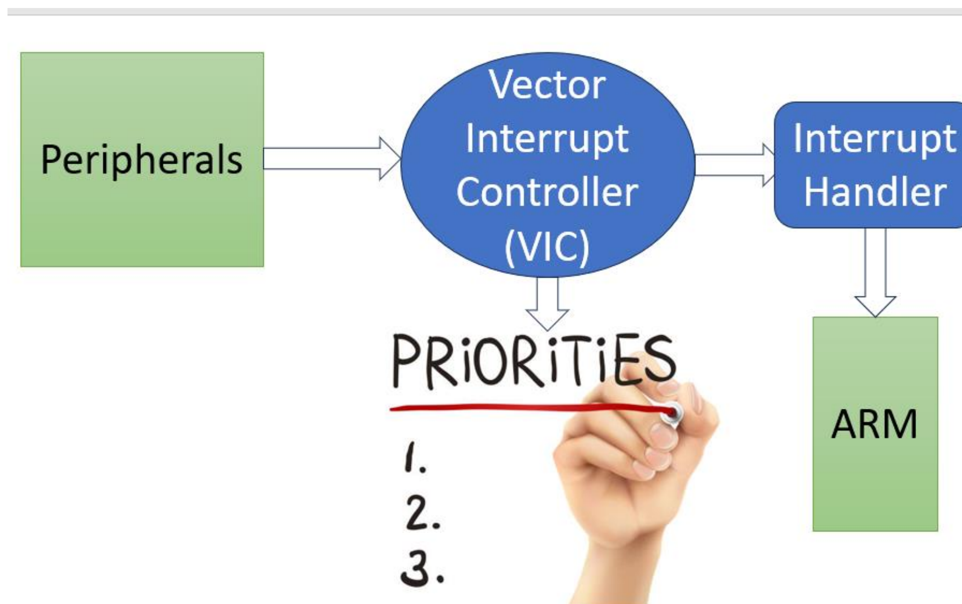
Working of Standard Interrupt Controller (SIC):



Hence SIC has option to either ignore or allow interrupts being raised by peripherals.

Working of Vector Interrupt Controller (VIC):

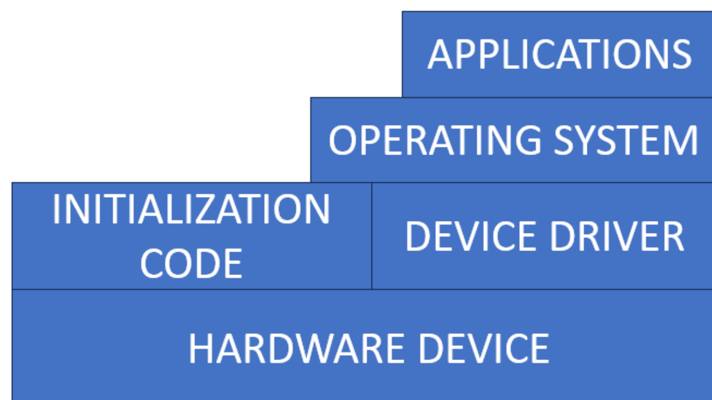
The working of Vector Interrupt Controller is as follows:



Hence interrupts are masked or serviced based on priorities. There exists a vector table which is looked up for interrupt handling.

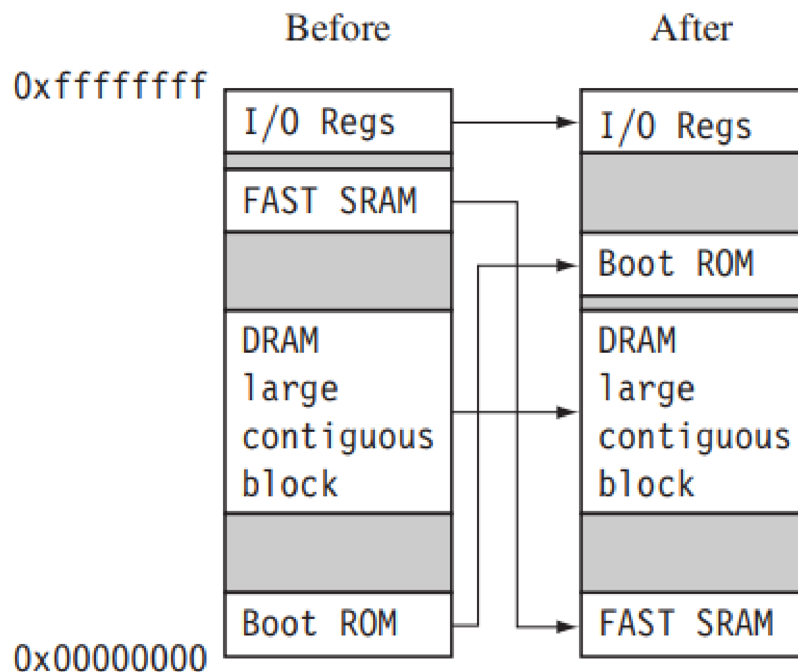
Embedded System Software

The software stack of embedded system is depicted as shown:



An embedded system gets into action by first executing boot code (initialization code). t sets up memory devices, caches, peripheral devices Deals with administrative work before actual OS image is loaded.

Phase-1: Initialize hardware configuration be reorganizing memory



As can be seen, the memory types are reorganized as FAST SRAM at lowest address, followed by DRAM, Boot ROM and I/O Registers.

Phase-2: Diagnostics

In this phase, diagnosis happens whether hardware is working or not. This helps in isolating faults.

Phase-3: Booting an image

In this phase, relevant OS image is loaded. Copy code and/or data into RAM. Sometimes image decompression is required. This is complex when task is to boot from multiple OS images.

Operating Systems

OS is like manager which manages various resources like memory, processor and peripherals. ARM has more than 50 OS, but 2 categories: RTOS and PSOS.

Real Time Operating Systems (RTOS):

- It has deadlines and guaranteed response.
- Hard RTOS provides guaranteed response
- Soft RTOS provides good response
- RTOS do not have secondary storage.

Platform Specific Operating Systems (PSOS):

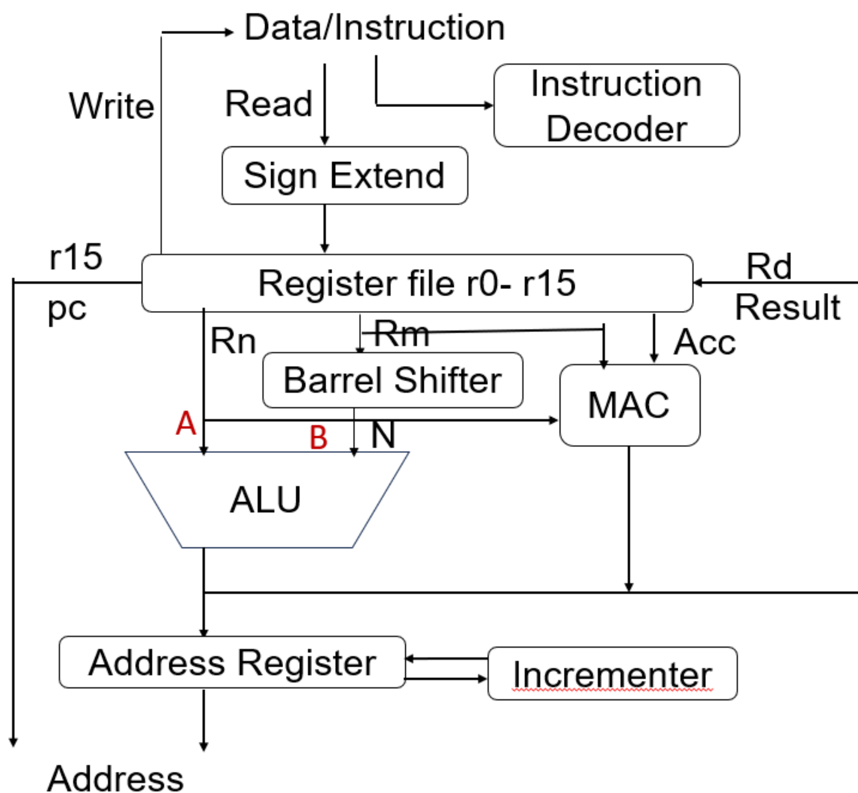
- No deadlines
- Large exclusive memory manager for real time applications
- Tend do have secondary storage

Applications

An application implements a processing task; the operating system controls the environment. An embedded system can have one active application or several applications running simultaneously. In contrast, ARM processors are not found in applications that require leading-edge high performance. Because these applications tend to be low volume and high cost, ARM has decided not to focus designs on these types of applications.

ARM Core Processor Components

It is the view of programmer. Data flow through various functional units is shown. Bus communication is also depicted in the ARM Core processor component architecture. The data or instruction can be fed. In case of instruction, decoding happens through instruction decoder. The data is sign extended if required or loaded into general purpose registers (r0-r15). Instructions are executed by Arithmetic and Logical Units. They are coupled with barrel shifting operations. For some of the instructions, accumulator contents may also be added. Program Counter (PC) contains the address of the next instruction to be executed. Separate buses are used to communicate between registers and ALU/MAC units.



ARM Registers

Basically, there are two types of ARM registers – General purpose registers and Special purpose registers. General-purpose registers hold either data or an address. The letter r is prefixed to the register number to identify them. For example, the label r4 is assigned to register 4. The registers are 32 bits in size. Up to 18 active registers

are available: 16 data registers and 2 processor status registers. The data registers are labeled r0 through r15 by the programmer. The ARM processor contains three registers: r13, r14, and r15, each of which is allocated to a specific duty or unique function.

- Register r13 is traditionally used as the stack pointer (SP) and stores the head of the stack in the current processor mode.
- Register r14 is called the link register (LR) and is where the core puts the return address whenever it calls a subroutine.
- Register r15 is the program counter (PC) and contains the address of the next instruction to be fetched by the processor.

CPSR (Current Processor Status Register)

•Current processor status register (CPSR) contains the current status of the processor.

•This includes various conditional code flags, Interrupt Status Processor mode and other status and control information.

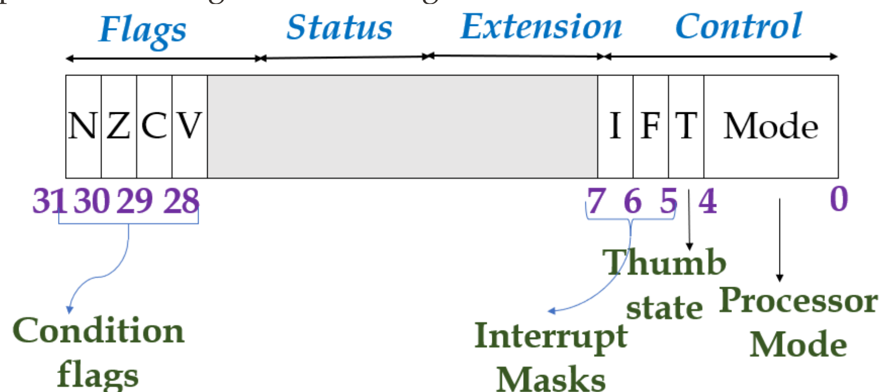
•The exception modes also have a saved processor status register (SPSR), that is used to preserve the value of CPSR when the associated exception occurs.
•Because the User and System modes are not exception modes, there is no SPSR available.

SPSR (Saved Processor Status Register)

In the exception modes there is an additional Saved Processor Status register (SPSR) which holds information on the processor's state before the system changed into this mode i.e., the processor status just before an exception.

Program Status Register

The composition of Program Status Register is as shown.

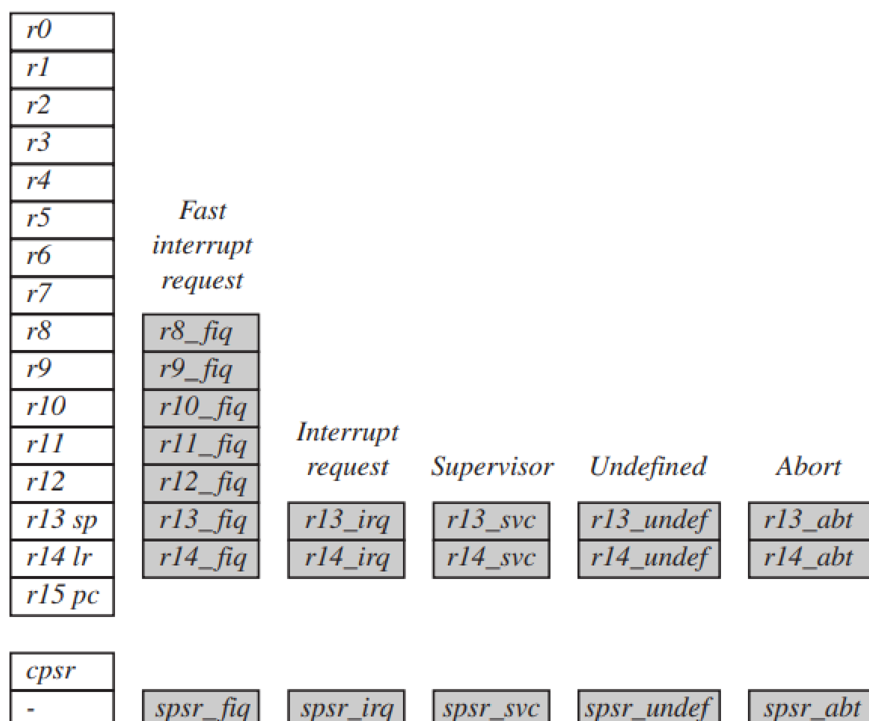


Processor modes can be Privileged or Non-privileged. In privileged mode- full read-write access to CPSR is provided. In non-privileged mode read only access is provided.

Processor Modes

Processor Mode	Explanation	Type
Abort	Failure to access memory	Privileged
Fast Interrupt Request	Peripherals signaling attention of processor with high priority	Privileged
Interrupt Request	Peripherals signaling attention of processor with normal priority	Privileged
Supervisor	OS mode	Privileged
System	Special version of user mode that allows full readwrite access to the user	Privileged
undefined	Wrong instruction	Privileged
User	Enter user mode	Non-Privileged

Banked Registers

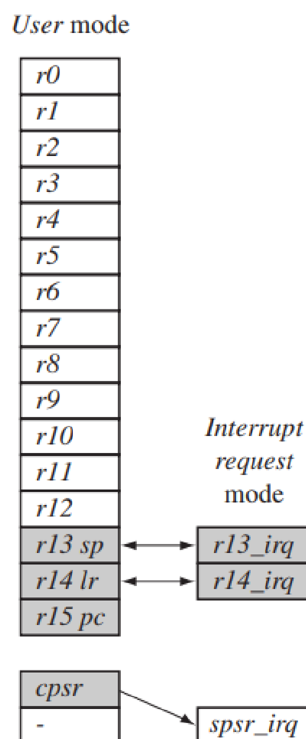


There exists 37 registers in the register file. Of those, 20 registers are hidden from a program at different times. These registers are called *banked registers* and are

identified by the shading in the diagram. They are available only when the processor is in a particular mode; for example, *abort* mode has banked registers *r13_abt, r14_abt* and *spsr_abt.* Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or *_mode.*

Changing mode on Exception

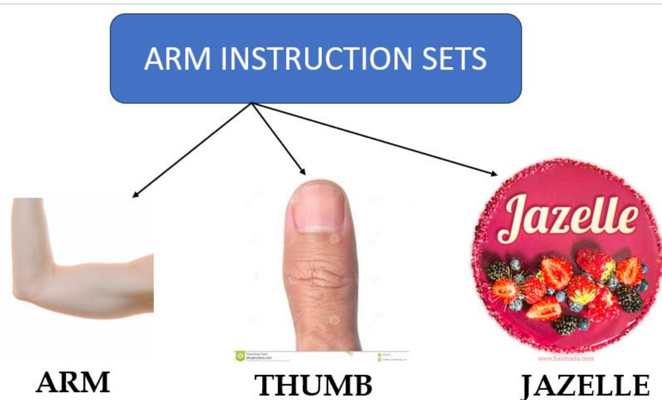
Interrupts and exceptions changes mode. Reset, interrupt request, fast interrupt request, software interrupt, data abort, prefetch abort, undefined instruction. Every processor mode except user mode can change mode by directly writing to CPSR. Processor modes can be changed by program or hardware. Saved program status register stores previous modes CPSR. SPSR is only modified with privileged mode and only done when hardware changes processor mode.



Processor Mode values

Each processor mode is assigned a 5 bit binary value as shown:

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000



In state field T=1 => Thumb Mode,
J=1 => Jazelle Mode

Hence there are 3 types of ARM Instruction Sets: ARM, THUMB and JAZELLE.

ARM vs THUMB

Parameter	**ARM**	**THUMB**
Instruction size	32 bit	16 bit
Core instructions	58	30
Conditional Execution	most	Only branch
Data processing	Combined access to barrel shifter and ALU	Separate access to barrel shifter and ALU
Program status register	Read-write in privileged access	No direct access
Register usage	15 GP + pc	8GP +7high+pc

	Jazelle (*cpsr* $T = 0, J = 1$)
Instruction size	8-bit
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.

Interrupt Mask

An internal switch setting that controls whether an interrupt can be processed or not. The mask is a bit that is turned on and off by the program. If interrupt is allowed it is said to be serviced.

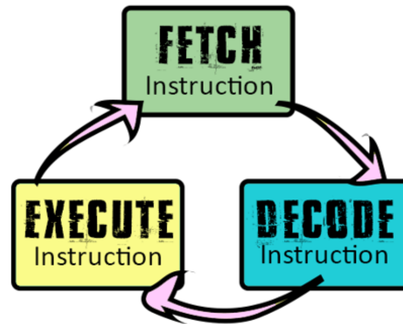
Condition Flags

There exist many condition flags affected directly and indirectly by the instructions. Majority of these flags are affected by ALU operations. Conditional execution will determine whether ARM processor will execute a specific instruction or not.

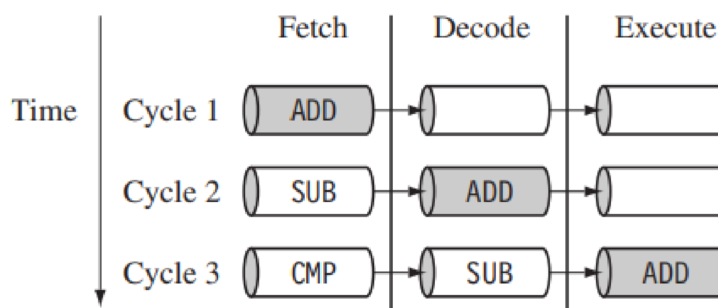
Flag	Flag Name	When
Q	Saturation	DSP instruction overflow
V	oVerflow	Signed overflow
C	Carry	Unsigned carry
Z	Zero	Comparison results in equality
N	Negative	Bit 31 is 1

ARM 3 Stage Pipeline

3 Stage pipeline follows Fetch-Decode and Execute model as shown. Fetch process fetches an instruction from memory. Decode identifies the instruction. Execute executes the instruction and writes result back to register

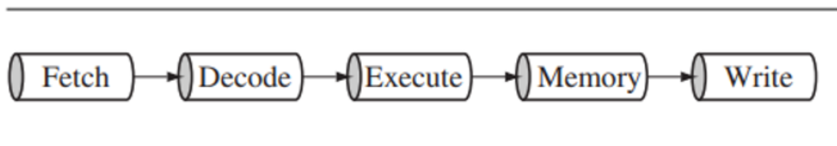


Example pipeline sequence:

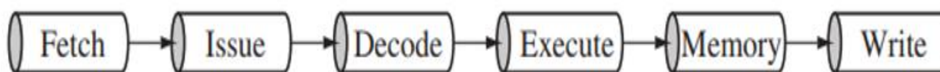


It can be inferred from pipeline sequence that 3 instructions are in pipeline ADD, SUB and CMP. It can be observed that when ADD enters Decode stage, SUB is fetched. Further when ADD moves to Execute Stage, SUB is decoded and CMP is fetched. Similarly, 5 stage and 7 stage pipelines can be created.

5 stage pipeline:

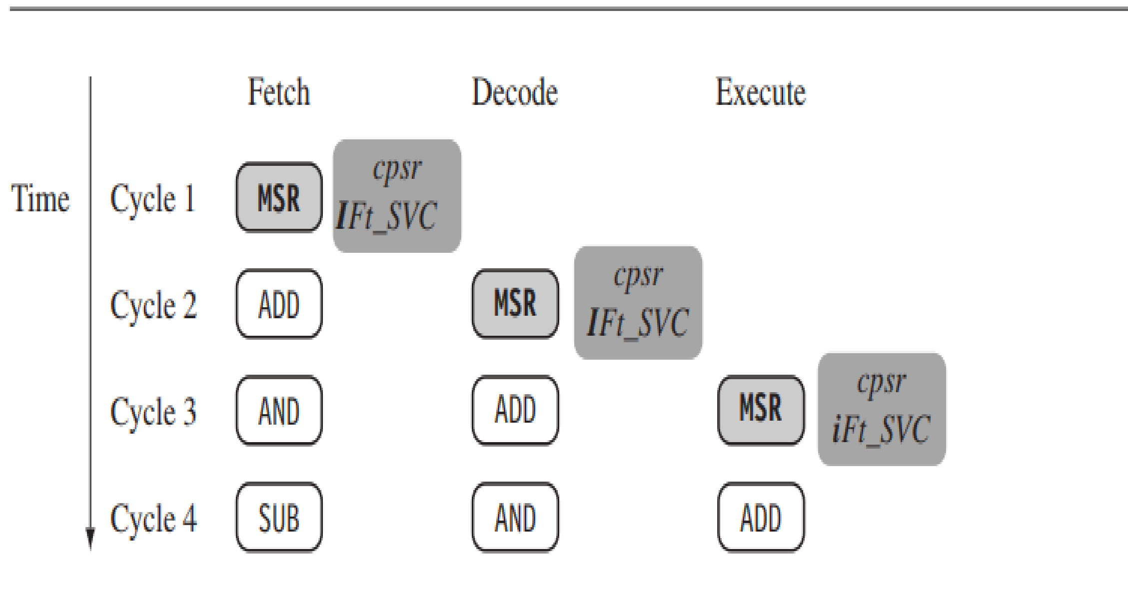


7 stage pipeline:



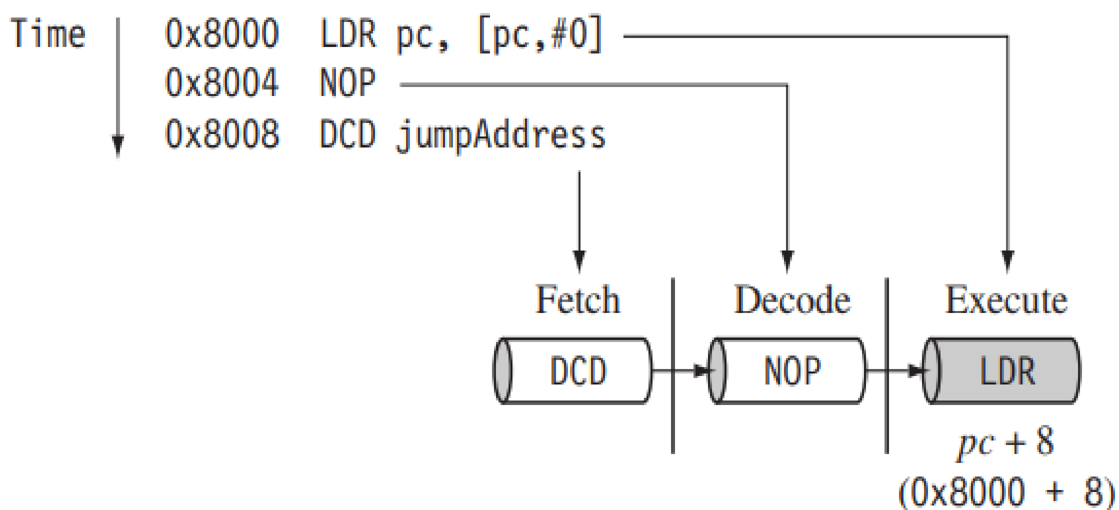
Increased pipeline length is a tradeoff between performance and latency. If pipeline length is increased performance improves, but an instruction requires more waiting time while moving between stages.

Pipeline Executing characteristics



It can be observed that MSR – an instruction that moves contents from status register to normal register is an interrupt instruction as user mode is switched to OS mode. Interrupt also forms part of pipeline and interrupt handling happens after Execution stage.

Pipelines and Program counter:



It can be observed that Program Counter gets updated to next instruction plus the number of instructions in pipeline. Execution of branch instruction causes ARM core to flush its pipeline. Loading the new branch address prior to the execution of instruction. Instruction in the execute stage will complete even though an interrupt has been raised.

Exceptions and Interrupts

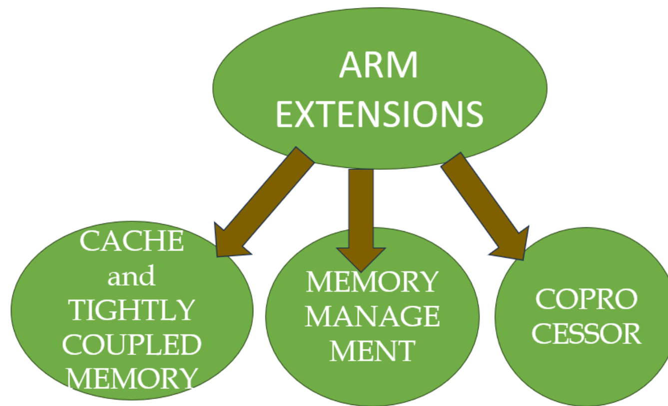
Exceptions and interrupts are unexpected events that disrupt the normal flow of instruction execution. An exception is an unexpected event from within the processor. An interrupt is an unexpected event from outside the processor. When an exception or interrupt occurs, the hardware begins executing code that performs an action in response to the exception. This action may involve killing a process, outputting a error message, communicating with an external device, or horribly crashing the entire computer system by initiating a "Blue Screen of Death" and halting the CPU. The instructions responsible for this action reside in the **operating system kernel**, and the code that performs this action is called the interrupt handler code. You can think of handler code as an operating system subroutine. After the handler code is executed, it may be possible to continue execution after the instruction where the execution or interrupt occurred.
When exception or interrupt, processor sets pc to special address from vector table.

The vector table.

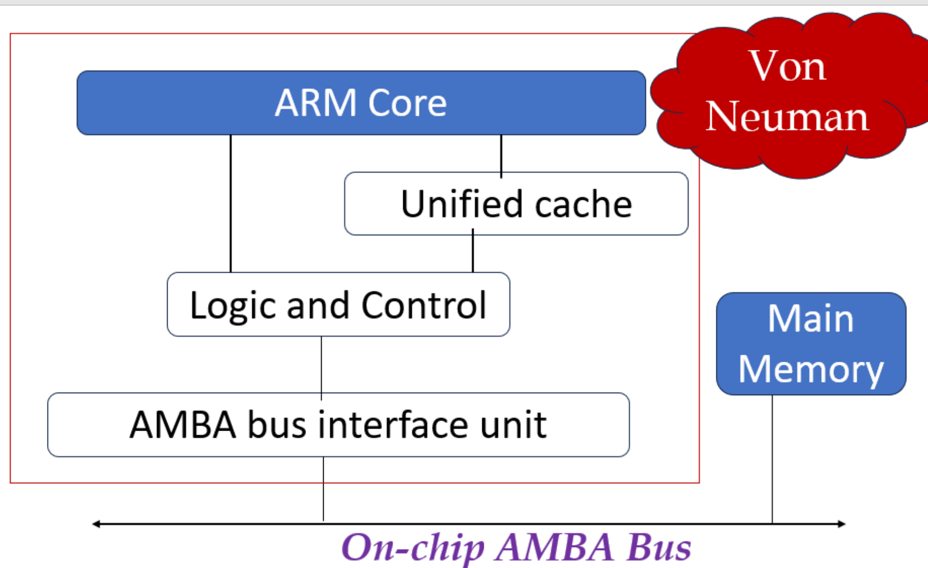
Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

ARM Extensions

ARM processor is given extensions to improve performance, do extra functionality and provide flexibility. 3 ARM extensions exist and are: (i) Cache and Tightly coupled memory (ii) Memory Management and (iii) Coprocessors

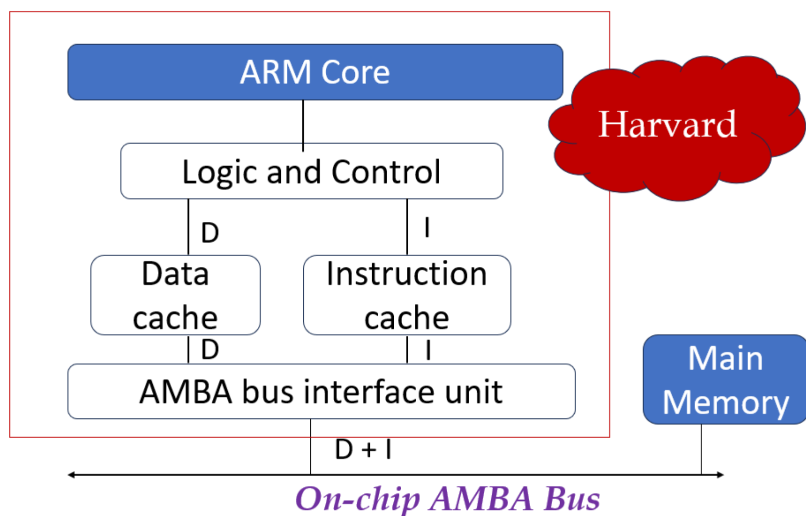


Unified Cache Management using Von-Neuman architecture:



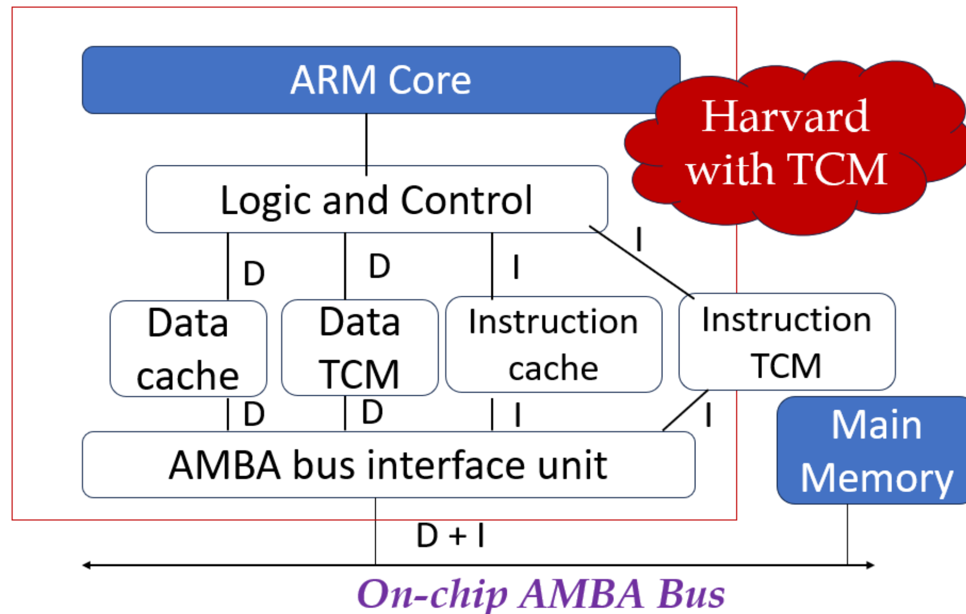
In this cache management technique, single cache is shared by data and instruction. Hence it is said to follow Von-Nueman architecture and a single bus is provided to access the cache for data or instruction.

Separate cache management based on Harvard Architecture:



In this cache management scheme, 2 separate caches are provided for data and instruction. Data and instruction cache are accessed by ALU using separate bus and hence this cache management scheme is based on Harvard Architecture. However to access external main memory common bus is used.

Cache Management with Tightly Coupled Memory:



In this cache management, tightly coupled cache is additionally provided which acts like shared memory for processes. This Tightly coupled cache is separately provided for data and instructions. There also exists separate data and instruction cache for high speed access.

ARM Core Memory Management Hardwares

- Nonprotected memory
- Memory Protection Units (MPU) with limited protection.
- Memory Management Units (MMU) with full protection

Coprocessors

- A co-processor is many times referred as a Math Processor. As the coprocessor performs routine mathematical tasks, the core processor is freed up from this computation and its time is saved. By taking specialized processing tasks from core CPU, coprocessor reduces the strain on the main microprocessor, so that it can run at a greater speed.
- A coprocessor can perform special tasks like complex mathematical calculations or graphical display processing. They perform such jobs faster than core CPU. As a result, overall computer speed of the system increases.
- To an ARM processor, we can attach the coprocessors. A coprocessor when added, we need to expand instruction set of Core CPU or add configurable registers, to increase the processing power. The coprocessor interface permits a couple of coprocessors to be connected to the ARM CPU.

Microcontrollers and Embedded Systems – Module 2

ARM Instructions:

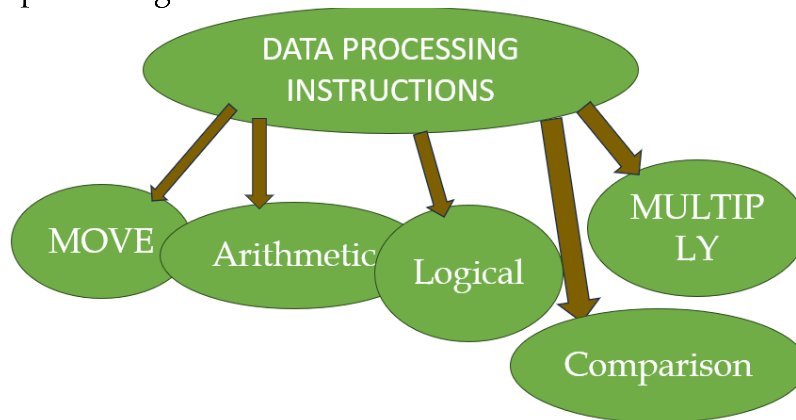
All ARM instructions are 32 bits long. Different ARM revisions have various instruction sets. These instruction sets express capabilities of ARM processor. Every ARM instruction follows this format:

PRE <preconditions>
INSTRUCTION(s)
POST<postconditions>

There are no direct memory access instructions. In case of registers, usually destination first is applied.

Data Processing Instructions:

Chart of data processing instructions is:



Data processing instructions operate on Registers. If suffix S is added, flags in cpsr is updated. MOV and Logical instructions update C, N and Z flags

MOV Instruction

It is the simplest ARM instruction used to set initial values and transfer data between registers.

Syntax:

instruction{cond}{S} Rd, N

In this syntax, condition and S are optional. Rd is the destination register and N can be register or immediate value with/without barrel shifter.

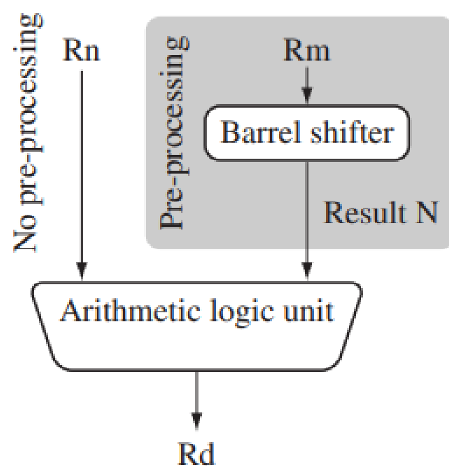
In addition to MOV instruction, there also exists MVN which negates the data in the register and moves to another register.

MOV example:

```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5     ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

Role of Barrel Shifter:

Barrel shifters are used to preprocess the data in the register before movement. As shown in the figure, data in the register can be given to barrel shifter and then sent to ALU. Unique and powerful feature of ARM to shift data left or right by specified positions before used in instructions. However, instructions MUL, CLZ, and QADD does not use barrel shifter.



MOV example with Barrel shifter:

```
PRE     r5 = 5
        r7 = 8

MOV     r7, r5, LSL #2


POST    r5 = 5
        r7 = 20
```

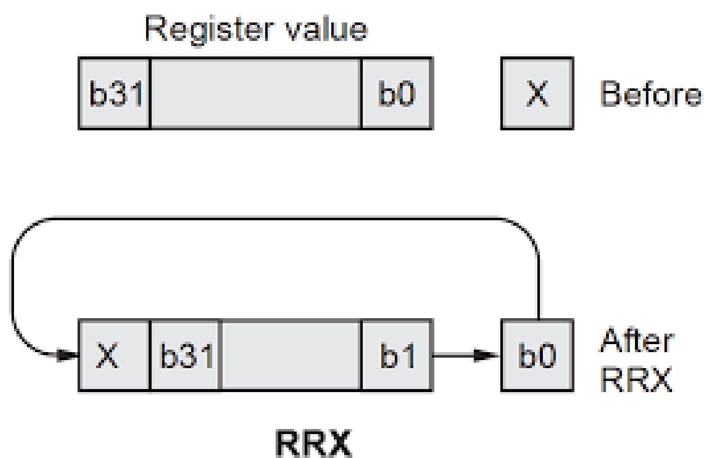
Various Barrel Shifter operations:

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	xLSL y	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	xLSR y	$(\text{unsigned})x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	xASR y	$(\text{signed})x \gg y$	#1–32 or *Rs*
ROR	rotate right	xROR y	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$	#1–31 or *Rs*
RRX	rotate right extended	xRRX	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	none

Hence in right shifting, either LSR or ASR can be used. LSR is used when sign extension is not required. However, ASR extends sign bit after shifting, Further it is noteworthy that LSL is equivalent to multiply by 2 and LSR is equivalent to divide by 2. Rotation will preserve the shifted data from right to left.

Working of RRX:

RRX instruction shifts one bit right and hence lsb (b0) will be rotated to msb if carry flag is on and not if carry flag is off.

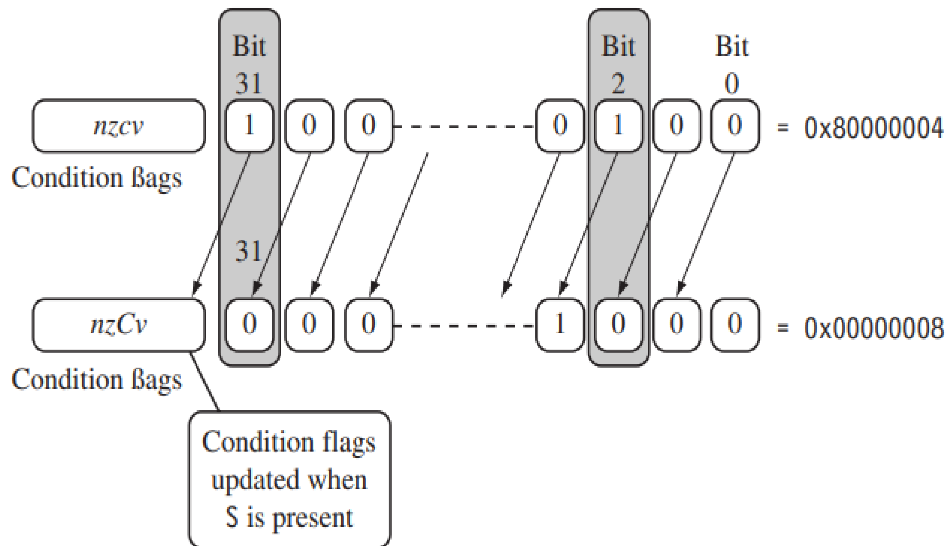


Eg:

```
 Before  R2=0x00000031
  MOV   R0, R2, RRX    C=1
After   R0=0x80000018
        R2=0x00000031
```

Impact of Barrel Shifter on Condition flags:

It can be observed that carry flag is affected when left shifting is done. Condition flags are only updated when MOV instruction contains suffix S.



Variants of barrel shift:

Shifting amount can be an immediate value or given through a register.

Barrel shift operation syntax for data processing instructions.

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

Barrel shifter with cpsr example:

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

MOVS     r0, r1, LSL #1
```

```
POST    cpsr = nzCvqiFt_USER
        r0 = 0x00000008
        r1 = 0x80000004
```

Arithmetic Instructions:

Arithmetic Instructions are used to perform arithmetic operations like addition and subtraction.
Syntax:
Syntax: instruction{<cond>}{S} Rd, Rn, N

Condition can be specified on addition and suffix S can be given for updation of cpsr. Computation of arithmetic instruction is:

Rd=Rn operator N
where N can be immediate value, register or scaled value.

Following are the various arithmetic instructions:

ADC	Add with Carry	Rd=Rn+N+carry
ADD	Add without carry	Rd=Rn+N
RSB	Reverse Subtract	Rd=N-Rn
RSC	Reverse Subtract with Carry	Rd=N-Rn-carry
SBC	Subtract with Carry	Rd=Rn-N-carry
SUB	Subtract without carry	Rd=Rn-N

Eg:
(i)

```
PRE    r0 = 0x00000000
       r1 = 0x00000002
       r2 = 0x00000001

SUB r0, r1, r2

POST   r0 = 0x00000001
```

Tracing:
r0=r1-r2=2-1=1

(ii)

```
PRE     r0 = 0x00000000
        r1 = 0x00000077

RSB r0, r1, #0

POST    r0 = -r1 = 0xffffff89
```

Tracing:
r0=0-r1=0-0x00000077 = 0xffffff89 (hexadecimal subtraction)

(iii)

```
PRE     cpsr = nzcvqiFt_USER
        r1 = 0x00000001

SUBS r1, r1, #1

POST    cpsr = nZCvqiFt_USER
        r1 = 0x00000000
```

Tracing:
r1=r1-1=1-1=0 (Zero and carry flag is affected)

(iv)

```
PRE     r0 = 0x00000000
        r1 = 0x00000005

ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

Tracing:
r0=r1+r1<<1=5+(5<<1)=5+10=15=0xf

Logical Instructions:

ARM *logical operations* include AND, ORR (OR), EOR (XOR), and BIC (bit clear). These each operate bitwise on two sources and write the result to a destination register. The first source is always a register and the second source is either an immediate or another register.

Syntax: instruction{<cond>}{S} Rd, Rn, N

Following is the list of Logical Instructions:

AND	Logical bitwise AND	Rd=Rn & N
ORR	Logical bitwise OR	Rd=Rn | N
EOR	Exclusive OR	Rd=Rn ^ N
BIC	Logical bit clear	Rd=Rn & ~N

Eg:
(i)

```
PRE     r0 = 0x00000000
        r1 = 0x02040608
        r2 = 0x10305070

 ORR    r0, r1, r2

 POST   r0 = 0x12345678
```

Tracing: r0=r1 | r2= 0x02040608 | 0x10305070 =0x12345678 (as anything OR with 0=anything)

(ii)

```
PRE     r1 = 0b1111
        r2 = 0b0101

  BIC   r0, r1, r2

POST   r0 = 0b1010
```

Tracing: r0=r1 & ~r2= 1111 & ~(0101)= 1111 & 1010 = 1010

Comparison Instructions:

These instructions are used to compare the contents of registers. They affect the condition flags in the CPSR.
Syntax:
instruction{<cond>}{S} Rn, N

Following are the various comparison instructions:

CMN	Compare negated	Rn + N
CMP	Compare	Rn - N
TEQ	Test for equality	Rn ^ N
TST	Test bits	Rn & N

Eg:
(i)

```
PRE    cpsr = nzcvqiFt_USER
       r0 = 4
       r9 = 4

CMP   r0, r9

POST   cpsr = nZcvqiFt_USER
```

Multiplication Instructions:

These instructions are intended to find the product of registers. Multiplication can be augmented with addition of accumulator. Further, there are signed and unsigned multiplication instructions exclusively. Multiplication of 32 bit numbers can result in 64bit product and such product is stored in low and high registers.

Syntax:

MLA{<cond>}{S} Rd, Rm,Rs, Rn
MUL {<cond>}{S} Rd, Rm, Rs

MLA	Multiply and accumulate	Rd=(Rm*Rs)+Rn
MUL	Multiply	Rd=Rm*Rs

Syntax for signed and long multiplication:
instruction{<cond>}{S} RdLo, ,RdHi, Rm, Rs

SMLAL	Signed multiply accumulate long	[RdHi, RdLo]= [RdHi, RdLo] +(Rm*Rs)
SMULL	Signed multiply long	[RdHi, RdLo]=Rm*Rs
UMLAL	Unsigned multiply accumulate long	[RdHi, RdLo]= [RdHi, RdLo] +(Rm*Rs)
UMULL	Unsigned multiply long	RdHi, RdLo]=Rm*Rs

Eg:
(i)

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

MUL   r0, r1, r2   ; r0 = r1*r2

POST   r0 = 0x00000004
       r1 = 0x00000002
       r2 = 0x00000002
```

Tracing: r0=r1*r2=2*2=4

(ii)

```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

UMULL   r0, r1, r2, r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

Tracing:

[r0,r1]=r2*r3=0xf0000002 * 0x00000002
=0b 1111 000000 0010 << 000000001
r0=0b1110 000000 0100 =0xe0000004
r1=shifted contents=1

Branch Instructions:

Branch Instructions are used in ARM to branch to labels and this alters the sequential execution of the programs. Branches can be unconditional or conditional (based on the value of conditional flags in CPSR). Further, one can branch to the instructions ahead (forward branch) or instructions before (backward branch).

Syntax: instruction{<cond>} label | Rm

B	Branch	pc=label
BL	Branch with link	pc=label lr-return address
BX	Branch exchange	pc=Rm & 0xfffffffe
BLX	Branch exchange with link	pc=Rm & 0xfffffffe lr=return address

Eg:

(i) Forward branch:

```
        B      forward
        ADD    r1, r2, #4
        ADD    r0, r6, #2
        ADD    r3, r7, #4
forward
        SUB    r1, r2, #4
```

(ii) Backward branch:

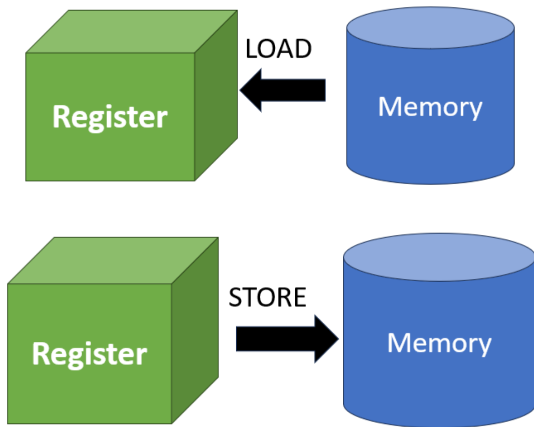
```
backward
        ADD    r1, r2, #4
        SUB    r1, r2, #4
        ADD    r4, r6, r7
        B      backward
```

(iii) Branch to subroutines

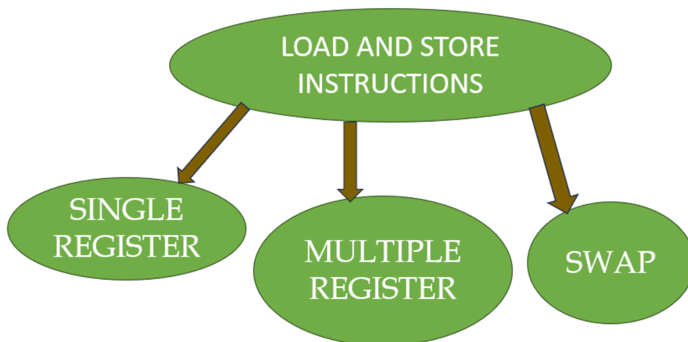
```
        BL      subroutine      ; branch to subroutine
        CMP     r1, #5          ; compare r1 with 5
        MOVEQ   r1, #0          ; if (r1==5) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV     pc, lr          ; return by moving pc = lr
```

Load and Store instructions:

There are no direct memory access instructions in ARM. The contents have to be transferred to and from the memory with the register. **Load** instruction loads the data **from memory** to register. **Store** instruction writes the data from register **to the memory**.



Categories of Load and Store Instructions:



Single Register Transfer:

Single Register load can be LDRB (Load byte), LDRSB (Load Signed Byte), LDRH (Load half word-16 bits), LDR (Load word – 32 bits)

Single Register store can be STRB (Store byte), STRSB (Store Signed Byte), STRH (Store half word-16 bits), STR (Store word – 32 bits)

Single Register Load-Store Addressing Modes:

These modes help to compute the address of the memory for loading and storing.

Index method	Data	Base address register
Preindex with writeback	*mem[base + offset]*	*base + offset*
Preindex	*mem[base + offset]*	*not updated*
Postindex	*mem[base]*	*base + offset*

Eg:
(i) Preindex with write back:

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00009000] = 0x01010101
        mem32[0x00009004] = 0x02020202

LDR     r0, [r1, #4]!

POST(1) r0 = 0x02020202
        r1 = 0x00009004
```

(ii) Preindex only

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00009000] = 0x01010101
        mem32[0x00009004] = 0x02020202

LDR     r0, [r1, #4]

POST(2) r0 = 0x02020202
        r1 = 0x00009000
```

(iii) Post index

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00009000] = 0x01010101
        mem32[0x00009004] = 0x02020202

LDR     r0, [r1], #4

POST(3) r0 = 0x01010101
        r1 = 0x00009004
```

Offset class of Single Register Load-Store addressing modes:
The offset can be:
(i) Signed or Unsigned
(ii) Offset address can be given through another register – Register based
(iii) Offset can be an immediate value of max 12 bits
(iv) Offset address can be scaled with barrel shift register operations

Load-Store Multiple Register transfer:

Addressing mode for load-store multiple instructions.

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	Rn	$Rn + 4*N - 4$	$Rn + 4*N$
IB	increment before	$Rn + 4$	$Rn + 4*N$	$Rn + 4*N$
DA	decrement after	$Rn - 4*N + 4$	Rn	$Rn - 4*N$
DB	decrement before	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

Eg:
(i)

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
```

```
mem32[0x80010] = 0x01
r0 = 0x00080010
r1 = 0x00000000
r2 = 0x00000000
r3 = 0x00000000
```

```
LDMIA   r0!, {r1-r3}
```

Tracing:

Before LDMIA

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
***r0* = 0x80010** →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

After LDMIA: (First load multiple and then increment)

Address pointer	Memory address	Data	
	0x80020	0x00000005	
***r0* = 0x8001c** →	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000003
	0x80014	0x00000002	*r2* = 0x00000002
	0x80010	0x00000001	*r1* = 0x00000001
	0x8000c	0x00000000	

After LDMIB: (First increment and then load multiple)

Address pointer	Memory address	Data	
	0x80020	0x00000005	
***r0* = 0x8001c** →	0x8001c	0x00000004	*r3* = 0x00000004
	0x80018	0x00000003	*r2* = 0x00000003
	0x80014	0x00000002	*r1* = 0x00000002
	0x80010	0x00000001	
	0x8000c	0x00000000	

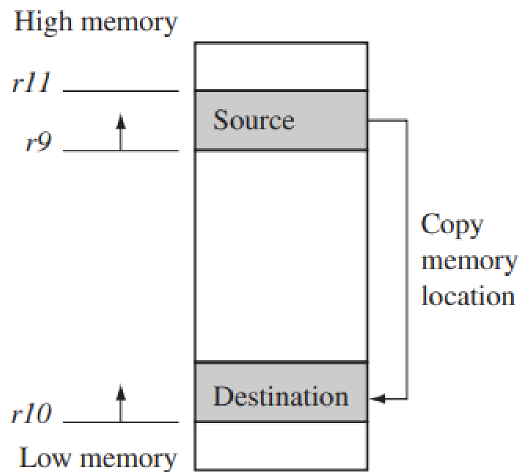
Similarly, LDMDB and LDMDA works by decrementing addresses and goes to the lower address of the memory.

Block memory copy:

The load-store multiple instructions can help in performing copying one block of memory to another. An ALP for block memory copy is:

```
loop:
    LDMIA r9!, {r0-r7}
    STMIA r10, {r0-r7}
    CMP r9, r11
    BNE loop
```

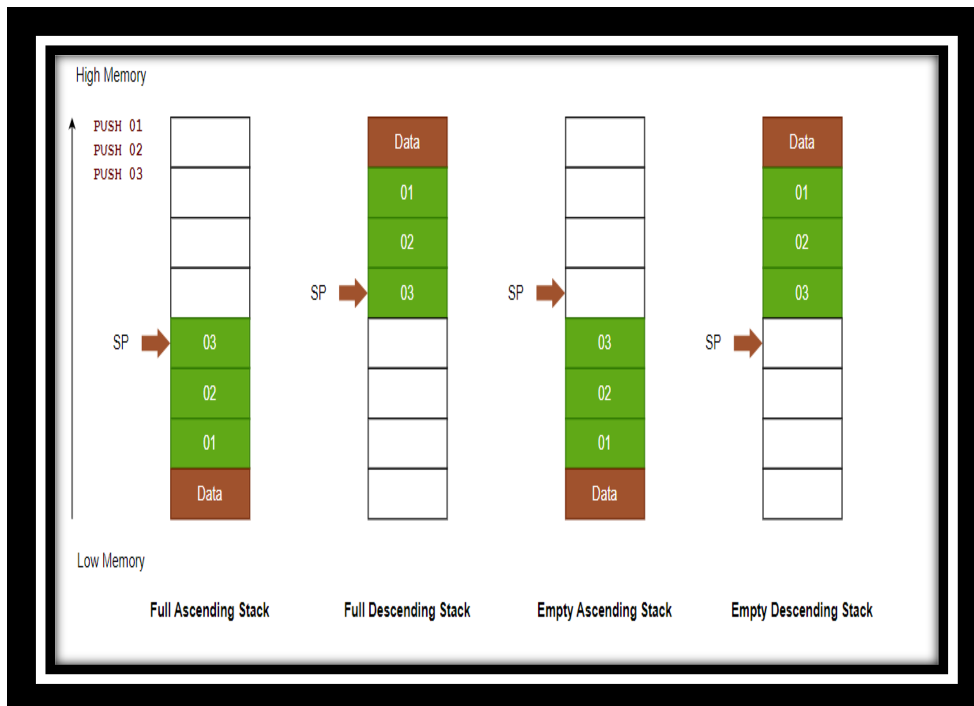
The corresponding memory map can be depicted as:



Stacks:

Stacks are data structures and memory locations which can be operated in Last-In-First-Out (LIFO) order. Stacks are supported with PUSH (writing to stack-Store-STRM) and POP (reading from stack-Load-LDM) operations. Stacks are operated through stack pointers (sp). Stack Base is a pointer that points to the starting address of the stack. Stack pointer points to the top of the stack. Maximum size of the stack beyond which overflow occurs is called Stack limit.

Stacks are called Full Stack if sp points to the last inserted location and are called Empty Stack if sp points the next available location. Stacks are called Descending stack if they grow from higher memory address to lower memory address and Ascending stack if they grow from lower memory address to higher memory address. Hence we have four combinations of Stacks:



Addressing modes for Stack operations:

FA	Full Ascending	POP-LDMFA PUSH-STMFA
FD	Full descending	POP-LDMFD PUSH - STMFD
EA	Empty Ascending	POP-LDMEA PUSH-STMEA
ED	Empty Descending	POP- LDMED PUSH - STMED

Eg:

(i) Descending Full Stacks

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080014

STMFD   sp!, {r1,r4}
```

PRE

	Address	Data
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	*Empty*
	0x8000c	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002

(ii) Descending Empty Stacks

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080010

STMED   sp!, {r1,r4}
```

PRE

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	*Empty*

SWP Instructions:

SWP instruction is used to swap the contents of the memory and register.

Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

Eg:

```
PRE     mem32[0x9000] = 0x12345678
        r0 = 0x00000000
        r1 = 0x11112222
        r2 = 0x00009000

SWP     r0, r1, [r2]

POST    mem32[0x9000] = 0x11112222
        r0 = 0x12345678
        r1 = 0x11112222
        r2 = 0x00009000
```

Software Interrupts

They are the call to the operating system for specific tasks. Each task is identified by SWI number.

Format:

SWI{<cond>} SWI number

Eg:

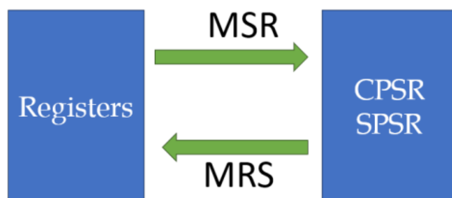
PRE cpsr:nzcvqift_user
pc=0x00008000
lr=0x003fffff
r0=0x12

SWI 0x123456

POST cpsr:nzcvqIft_svc
spsr: nzcvqift_user
pc=0x00000008
lr= 0x00008004
r0=0x12

SWI_Number= <SWI Instruction> AND NOT (0xff000000)

Program Status Register Instructions



MSR copies contents from general purpose registers to the status register. MRS operates in the reverse direction. Further the specific fields to copied or saved in the status register can also be mentioned in the MRS and MSR instructions. The MRS and MSR instructions also change processor mode from user to OS as they need elevated privileges.

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```

Eg:

```
PRE    cpsr = nzcvqIFt_SVC

MRS    r1, cpsr
BIC    r1, r1, #0x80
MSR    cpsr_c, r1

POST   cpsr = nzcvqiFt_SVC
```

Overview of C compilers and optimization:

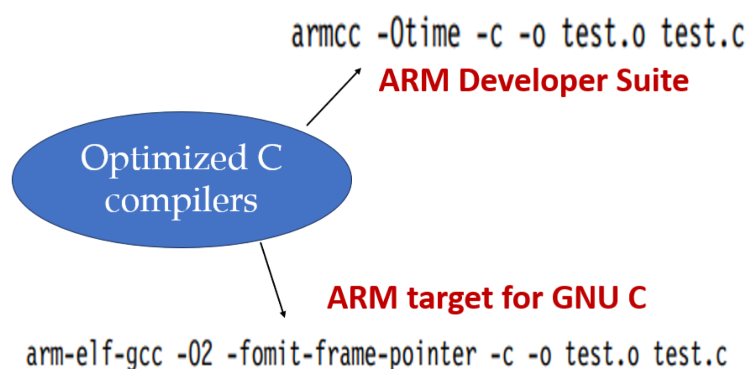
There exists tradeoff in Optimizing code takes and code readability. Hence one needs to optimize the functions that are frequently executed. Further, optimize the functions that are important. It is necessary to document non-obvious optimizations. C compilers need to be efficient yet conservative of ARM processor architecture.

Eg:

```
void memclr(char *data, int N)
{
    for(;N>0;N--)
   {
       *data=0;
        data++;
    }
}
```

This program clears N bytes of data. However, compiler does not know:

(i) whether N=0, as if N=0, condition checking in loop can be avoided.
(ii) Data array pointer is 4 byte aligned, otherwise masking would be required
(iii) N is a multiple of 4 so that aligned addressing is possible



Load and Store Instructions based on ARM versions:

Pre-ARMv4: LDRB, STRB, LDR, STR

ARMv4: LDRSB, LDRH, LDRSH, STRH, STRSH

ARMv5: LDRD, STRD

Hence prior to ARMV4, only byte and word loading/storing was supported. In ARMV4, signed and unsigned byte and half words load-store was added. ARMV5 also supports double word (64 bit) load-stores.

C compiler datatype mappings.

C Data Type	Implementation
`char`	unsigned 8-bit byte
`short`	signed 16-bit halfword
`int`	signed 32-bit word
`long`	signed 32-bit word
`long long`	signed 64-bit double word

It is recommended to use int and long (32 bit words) in the C program as they are friendly to ARM architecture which has 32 bit registers and uses 32 bit memory.

Eg:
Consider a C program that computes checksum. Checksum is the sum of 64 data elements.

```
int checksum_v1(int *data)
{
    char i;
    int sum=0;
    for(i=0;i<64;i++)
   {
       sum+=data[i];
   }
   return sum;
}
```

Assume program name is csum.c, then
Command
armcc -Otime -c -o csum.o csum.c
will generate object code csum.o which can be executed in ARM devices.
Command
fromelf --text -c csum.o > csum.txt
will generate a text file that contains ALP of the corresponding C program.

Hence, resulting ALP is:

```
checksum_v1

MOV r2,r0; r2->data
MOV r0,#0; r0->sum
MOV r1, #0; r1->i
checksum_v1_loop
  ;r3=data[i]
  LDR r3,[r2, r1, LSL #2]
  ADD r1,r1,#1 ; i++
  AND r1, r1,#0xff
```

```
 ; i=(char) casting
 CMP r1, #0x40; i<64
 ADD r0,r3,r0; sum+=data[i]
 BCC checksum_v1_loop
 MOV pc, r14 ;return sum
```

Thus an AND instruction is added to make sure char data type boundary is not violated. However AND is costlier and can be avoided by changing data type to unsigned int.

```
int checksum_v2(int *data)
{
    unsigned int i;
    int sum=0;
    for(i=0;i<64;i++)
   {
       sum+=data[i];
    }
   return sum;
}
```

and its converted ALP:

```
checksum_v2
MOV r2,r0; r2->data
MOV r0,#0; r0->sum
MOV r1, #0; r1->i
checksum_v2_loop
  ;r3=data[i]
  LDR r3,[r2, r1, LSL #2]
  ADD r1,r1,#1 ; i++
  CMP r1, #0x40; i<64
  ADD r0,r3,r0; sum+=data[i]
  BCC checksum_v1_loop
  MOV pc, r14 ;return sum
```

Short checksum computation:

Short checksum is a checksum of 64 elements of 16-bit data instead of 32 bit. Corresponding C program is:

```
short checksum_v3(short *data)
{
    unsigned int i;
    short sum=0;
    for(i=0;i<64;i++)
   {
       sum=(short)(sum+data[i]);
```

```
    }
   return sum;
}
```

In this program, casting to short is required as sum is 32 bit and data element is 16 bit. Since loop iterates 64 times, 64 times casting would be done which will be costlier. This can also be emphasized with the following converted ALP:

```
checksum_v3
MOV r2,r0; r2->data
MOV r0,#0; r0->sum
MOV r1, #0; r1->i
checksum_v3_loop
  ADD r3,r2, r1, LSL #1;r3=&data[i]
  LDRH r3, [r3,#0];r3=data[i]
  ADD r1,r1,#1 ; i++
  CMP r1, #0x40; i<64
  ADD r0,r3,r0; sum+=data[i]
  MOV r0,r0, LSL #16
  MOV r0, r0, ASR #16
  BCC checksum_v1_loop
  MOV pc, r14 ;return sum
```

It can be observed that in then ALP, first address of loading is computed and the loading is done with LDRH. Separate steps are necessary as LDRH does not support barrel shifting. Further in each iteration of the loop, casting of sum is done by left shifting lower half word to upper half word and then right shifting upper half word to lower half word with sign extension.

This casting and shifting which is costlier can be avoided by processing data as a pointer rather than an array. The next version of checksum illustrates this:

```
short checksum_v4(short *data)
{
    unsigned int i;
    int sum=0;
    for(i=0;i<64;i++)
   {
       sum+=*(data++);
    }
   return (short)sum;
}
```

Consider the generated ALP:

```
checksum_v4
MOV r2,#0; r2->sum
```

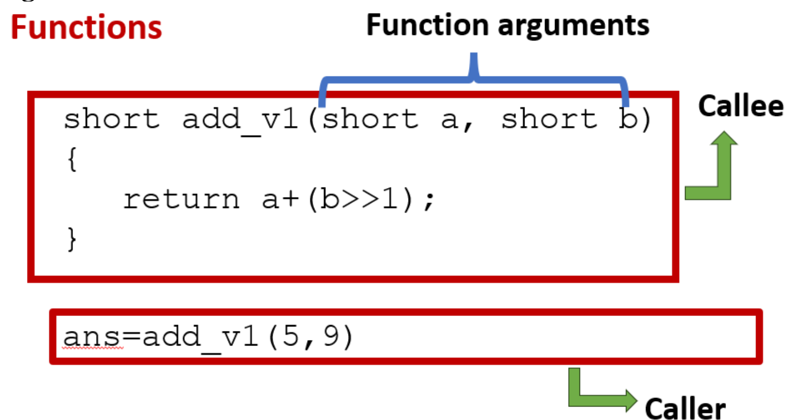
```
MOV r1, #0; r1->i
checksum_v4_loop
  LDRSH r3, [r0],#2;r3=*(data++)
  ADD r1,r1,#1 ; i++
  CMP r1, #0x40; i<64
  ADD r2,r3,r2; sum+=r3
  BCC checksum_v4_loop
  MOV r0,r2, LSL #16
  MOV r0, r0, ASR #16
  MOV pc, r14 ;return sum
```

Hence two improvements can be pointed out. Separate steps of address computation and loading is combined as single load with post index. Further casting is done only on return value and not in each iteration. Thus, only once left and right barrel shifters are used.

Functions:

Functions are written in C for modularization of the code. A function is written once and called many times. Every function has a name, return type and arguments. Function return values of the appropriate type. Function definition is known as callee statement and function invocation is known as caller.

Eg:



Function arguments are said to be passed wide if they are not reduced to range of type. Else they are narrow. Following rules prevail with respect to widening and narrowing:

(i) If compiler passes wide, callee should reduce
(ii) If compiler passes narrow, caller should reduce
(iii) If compiler returns wide, caller should reduce
(iv)If compiler returns narrow, callee should reduce

Eg: Consider the following C snippet:

```
short add_v1(short a, short b)
{
```

```
  return a+(b>>1);
}
```

The corresponding ALP is:

```
add_v1
ADD r0,r0,r1,ASR #1
MOV r0,r0, LSL #16
MOV r0, r0, ASR #16
MOV pc, r14 ;return sum
```

It has shifting code (LSL and ASR) because of type casting issues.

Signed and unsigned numbers:

Signed numbers have an exclusive sign bit as the most significant bit. The value of 1 in the sign bit indicates it is a negative number else it is a positive number. Unsigned numbers are always positive and no special bit is required.

For addition, subtraction and multiplication no difference in signed or unsigned. For division unsigned is better. Consider the following C example:

```
int average_v1(int a, int b)
{
  return (a+b)/2;
}
```

Here division by 2 can be implemented with ASR. However for negative numbers 1 should be added before division. Thus following ternary condition illustrates the division complexity of signed numbers:

```
(x<0) ? ((x+1) >> 1): (x >> 1)
```

Hence generated ALP of signed numbers average is as follows:

```
average_v1
ADD r0,r0,r1
ADD r0,r0,r0, LSR #31
MOV r0, r0, ASR #1
MOV pc, r14
```

Thus we bring sign bit to lsb by performing LSR 31 times and add it to the number before performing division. If unsigned number was used, no such shifting was required.

Loops:

Loops are mechanism deployed in C to repeat certain sequence of statements.

Multiple looping mechanisms like for, do-while and while exist. However, efficient way of coding loops in C is required such that it will be suitable for ARM architecture. Two cases shall be considered:

- ***(i)*** Loops with fixed number of iterations
- ***(ii)*** Loops with variable number of iterations

(i)Loops with fixed number of iterations:

Consider the following C code to compute checksum that use a loop with upcounter:

```
int checksum_v5(int *data)
{
    unsigned int i;
    int sum=0;
    for(i=0;i<64;i++)
   {
       sum+=*(data++);
    }
   return sum;
}
```

The generated ALP has 3 instructions required for looping: ADD, CMP and BCC:

```
checksum_v5
MOV r2,r0; r2->data
MOV r0,#0; r0->sum
MOV r1, #0; r1->i
checksum_v5_loop
  LDR r3,[r2],#4
  ADD r1,r1,#1 ; i++
  CMP r1, #0x40; i<64
  ADD r0,r3,r0; sum+=data[i]
  BCC checksum_v5_loop
  MOV pc, r14 ;return sum
```

We can reduce this to instructions if downcounter is used as shown in the following C code:

```
int checksum_v6(int *data)
{
    unsigned int i;
    int sum=0;
    for(i=64;i!=0;i--)
   {
       sum+=*(data++);
    }
   return sum;
}
```

The generated ALP has only 2 instructions for looping: SUBS and BNE. Further it is always better to use post indexing technique while operating with arrays which helps to operate array like pointer and avoid explicit index variable.

```
checksum_v6
MOV r2,r0; r2->data
MOV r0,#0; r0->sum
MOV r1, #0x40;r1->i
checksum_v6_loop
  LDR r3,[r2],#4
  SUBS r1,r1,#1 ; i--
  ADD r0,r3,r0; sum+=data[i]
  BNE checksum_v6_loop
  MOV pc, r14 ;return sum
```

(ii)Loops with variable number of iterations:

Consider another checksum example where only first N elements are added and thus induces variable number of iterations. (N can be varied)

```
int checksum_v7(int *data, unsigned int N)
{
    int sum=0;
    for(;N!=0;N--)
   {
       sum+=*(data++);
    }
   return sum;
}
```

However in the generated ALP there is an unnecessary condition checking of N:

```
checksum_v7
MOV r2,#0; r2->data
CMP r1,#0; r0->sum
BEQ checksum_v7_end
checksum_v7_loop
  LDR r3,[r0],#4
  SUBS r1,r1,#1 ; i--
  ADD r2,r3,r2; sum+=data[i]
  BNE checksum_v7_loop
Checksum_v7_end
   MOV r0,r2
   MOV pc,r14
```

This unnecessary condition checking can be avoided if do-while is used instead of for loop:

```
int checksum_v8(int *data, unsigned int N)
{
    int sum=0;
```

```
 do
 {
   sum+=*(data++);
 }while(--N!=0);
 return sum;
}
```

and in the resulting ALP the prechecking code is avoided:

```
checksum_v8
MOV r2,#0; r2->data
checksum_v8_loop
  LDR r3,[r0],#4
  SUBS r1,r1,#1 ; i--
  ADD r2,r3,r2; sum+=data[i]
  BNE checksum_v8_loop
  MOV r0,r2
  MOV pc,r14
```

Loop Unrolling:

Every loop incurs an overhead. Considering the previous example, even with down counter loop, every loop iteration uses 2 instructions: SUBS and BNE . SUBS requires 1 instruction cycle and BNE requires 3 instruction cycle. Hence each iteration has an overhead of 4 instruction cycle. Hence for loop with N iterations, there shall be 4N cycles overhead. This can be reduced if same loop statement is written multiple times and step decrement is reduced by the times it is unrolled.

Eg:

```
int checksum_v9(int *data, unsigned int N)
{
 int sum=0;
 do
 {
  sum+=*(data++);
sum+=*(data++);
sum+=*(data++);
sum+=*(data++);
N-=4;
 }while(N!=0);
  return sum;
}
```

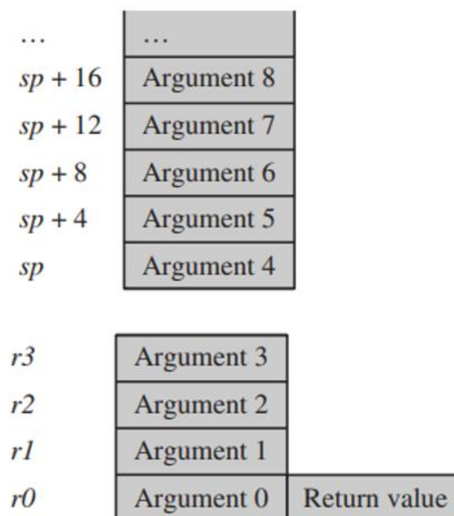
Hence in this example, loop has been unrolled 4 times by writing same loop statement 4 times and consequently N is reduced by 4 each time than one. Now, each loop iteration requires 4 cycles (SUBS-1, BNE-3) and we will have N/4 iterations. Hence loop overhead is reduced to 4*N/4 = N. Thus, we can reduce loop overhead 4 times.

Register Allocation:

If more variables exist than available registers, some variables are stored in process stack and are called Spilled variables. For efficient implementation it is advised to minimize the number of spilled variables. Compilers chose most important and frequently accessed variables are stored in registers. Programmers are advised to limit loop variables to 12 (even though in theory 14 registers are possible). Compiler decides spilling of variables on the basis of frequency of use. C has register keyword which hints the compiler to use register to store the variable than stack. However it is compiler dependent decision on whether it will store or not.

Function calls:

- APCS – ARM Procedure Call Standard – A standard for arguments and return values
- In case of Thumb registers, APCS is redefined as ATPCS – ARM Thumb Procedure Call Standard
- As per APCS, 4 function arguments can be stored in registers and rest of the arguments find place in the stack. Hence for C functions having more than 4 arguments Stack will be accessed by caller and callee
- To optimize, use structures in C which can pack arguments and make sure that only registers are used.



Consider the following example that demonstrates the use of creating structures of function arguments:

```
char *queue_bytes_v1(char *Qstart, char *Qend, char *Qptr, char *data, unsigned int
N)
{
    do{
      *(Qptr++)=*(data++);
      if(Qptr==Qend)
         Qptr=Qstart;
    }while(--N);
```

```
  return Qptr;
}
```

In this C program data is transferred from location data to a destination queue which is managed by start pointer Qstart, end pointer Qend and current pointer Qptr. The transfer is limited by the value of N. The queue is modelled as a circular queue in which insertion starts from beginning, when end is reached.

However the C program written requires 5 arguments and hence will need the use of stack. However this can be avoided if queue pointers are packed inside the following structure and structure is sent in the function argument:

```
typedef struct
{
   char *qptr;
   char *qstart;
   char *qend;
}Queue;


char *queue_bytes_v2(Queue q, char *data, unsigned int N)
{
    char *Qptr=q->qptr;
    char *Qend=q->qend;
    char *Qstart=q->qstart;
    do{
       *(Qptr++)=*(data++);
       if(Qptr==Qend)
          Qptr=Qstart;
      }while(--N);
  return Qptr;
}
```

Further function optimizations possible are:

- Write before functions before calling them. This allows compiler to make them inline
- Critical functions can be coded as __inline keyword

Pointer Aliasing:

2 pointers are said to be alias, if they point to same address. Changing one affects another. The problem of pointer aliasing in C is that compiler uses a pessimistic attitude towards them. Compiler thinks all pointers are aliased and results in repeated loading of same memory locations in resulting ALP.

Eg:Consider the following C code that uses pointers as function arguments:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
```

```
  *timer1+=*step;
  *timer2+=*step;
}
```

In this compiler treats timer1 and step as aliases. Hence it assumes timer1 will update step and hence step will be reloaded. To avoid reloading, step is used as a local variable. Consider the example in which step is used as a local variable and hence will not be reloaded.

```
void timers_v1(int *timer1, int *timer2)
{
   int step=4;
  *timer1+=step;
  *timer2+=step;
}
```

Consider another example in which pointer aliasing can occur:

```
int checksum_next_packet()
{
   int *data;
   unsigned int i;
   int N, sum=0;
    data=get_next_packet(&N);
   for(i=0;i<N;i++)
  {
      sum+=*(data++);
   }
  return sum;
}
```

Here, compiler assumes the pointers data and &N as aliases and reloads N again and again. Hence it is recommended to never send address of local variables and instead make a copy and send.