

ARM PROCESSOR FUNDAMENTALS

A programmer can think of an ARM core as functional units connected by data buses, as shown in the following Figure.

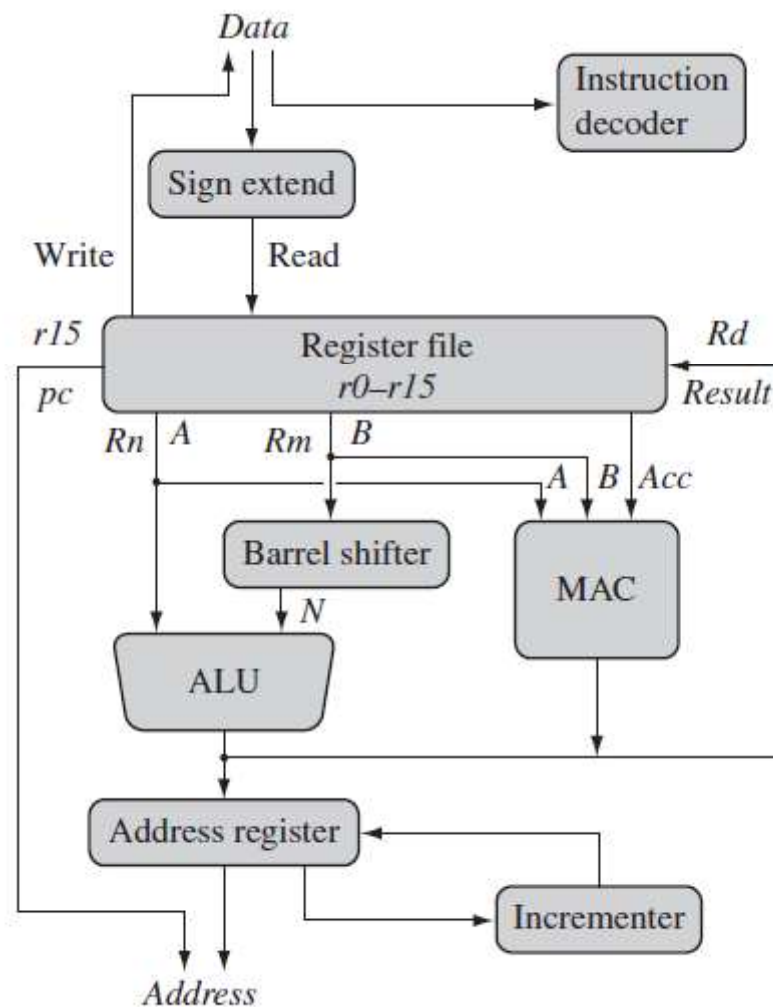



Figure: ARM Core dataflow Model

The arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area.

- ✓ Data enters the processor core through the Data bus. The data may be an instruction to execute or a data item.

- 
- Figure shows a Von Neumann implementation of the ARM—data items and instructions share the same bus. (In contrast, Harvard implementations of the ARM use two different buses).
 - ✓ The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.
 - ✓ The ARM processor, like all RISC processors, uses *load-store architecture*—means it has two instruction types for transferring data in and out of the processor:
 - load instructions copy data from memory to registers in the core
 - store instructions copy data from registers to memory
 - ✓ There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out in registers.
 - ✓ Data items are placed in the register file—a storage bank made up of 32-bit registers.
 - Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.
 - ✓ ARM instructions typically have two *source registers*, R_n and R_m , and a single result or *destination register*, R_d . Source operands are read from the register file using the internal buses A and B, respectively.
 - ✓ The *ALU (arithmetic logic unit)* or *MAC (multiply-accumulate unit)* takes the register values R_n and R_m from the A and B buses and computes a result. Data processing instructions write the result in R_d directly to the register file.
 - ✓ Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the Address bus.
 - One important feature of the ARM is that register R_m alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.
 - ✓ After passing through the functional units, the result in R_d is written back to the register file using the *Result bus*.
 - ✓ For load and store instructions the *Incrementer* updates the address register before the core reads or writes the next register value from or to the next sequential memory location.
 - ✓ The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

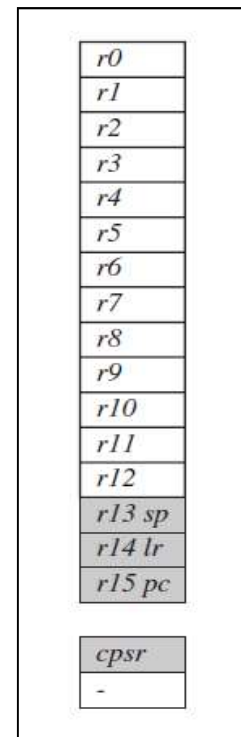
REGISTERS:

General-purpose registers hold either data or an address. They are identified with the letter r prefixed to the register number. For example, register 4 is given the label $r4$.



The Figure shows the active registers available in user mode. (A protected mode is normally used when executing applications).

- ✓ The processor can operate in seven different modes.
- ✓ All the registers shown are 32 bits in size.
- ✓ There are up to 18 active registers:
 - 16 data registers and 2 processor status registers.
 - The data registers visible to the programmer are *r0* to *r15*.
- ✓ The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are given with different labels to differentiate them from the other registers.
 - 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.
- ✓ In ARM state the registers *r0* to *r13* are orthogonal—any instruction that you can apply to *r0* you can equally well apply to any of the other registers.
- ✓ In addition to the 16 data registers, there are two program status registers: *cpsr* (**current program status register**) and *spsr* (**saved program status register**).



CURRENT PROGRAM STATUS REGISTER:

The ARM core uses the *cpsr* to monitor and control internal operations. The *cpsr* is a dedicated 32-bit register and resides in the register file. The following Figure shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

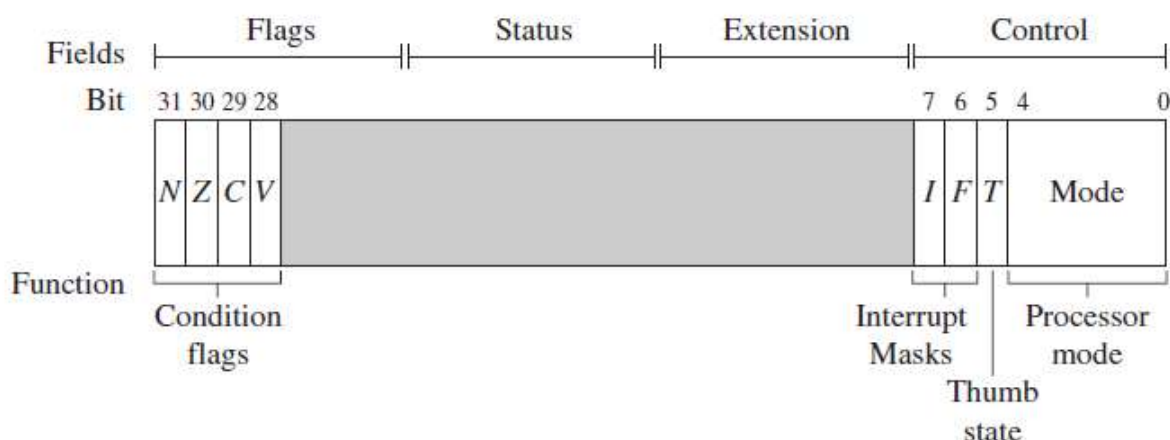
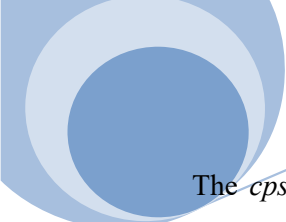


Figure: A Generic Program Status Register (psr)



The *cpsr* is divided into four fields, each 8 bits wide: *flags*, *status*, *extension*, and *control*. In current designs the extension and status fields are reserved for future use.

- ✓ The **control field** contains the *processor mode*, *state*, and *interrupt mask* bits.
- ✓ The **flags field** contains the *condition flags*.

Some ARM processor cores have extra bits allocated. For example, the *J bit*, which can be found in the flags field, is only available on *Jazelle-enabled processors*, which execute 8-bit instructions.

It is highly probable that future designs will assign extra bits for the monitoring and control of new features.

Processor Modes:

- ✓ The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or non-privileged:
 - A *privileged mode* allows full read-write access to the *cpsr*.
 - A *non-privileged mode* only allows read access to the control field in the *cpsr*, but still allows read-write access to the condition flags.
- ✓ There are *seven processor modes* in total:
 - *six privileged modes* (abort, fast interrupt request, interrupt request, supervisor, system, and undefined)
 - The processor enters **abort mode** when there is a failed attempt to access memory.
 - **Fast interrupt request** and **interrupt request modes** correspond to the two interrupt levels available on the ARM processor.
 - **Supervisor mode** is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
 - **System mode** is a special version of user mode that allows full read-write access to the *cpsr*.
 - **Undefined mode** is used when the processor encounters an instruction that is undefined or not supported by the implementation.
 - *one non-privileged mode* (user).
 - **User mode** is used for programs and applications.

Banked Registers:

The following Figure shows all 37 registers in the register file.

- ✓ Of these, 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*.
- ✓ Every processor mode except user mode can change mode by writing directly to the mode bits of the *cpsr*.
- ✓ All processor modes except system mode have a set of associated banked registers that are a subset of the main 16 registers.
- ✓ A banked register maps one-to-one onto a user mode register.
- ✓ If you change processor mode, a banked register from the new mode will replace an existing register.
 - For example, when the processor is in the interrupt request mode, the instructions you execute still access registers named *r13* and *r14*. However, these registers are the banked registers *r13_irq* and *r14_irq*. The user mode registers *r13_usr* and *r14_usr* are not affected by the instruction referencing these registers. A program still has normal access to the other registers *r0* to *r12*.

User and system					
<i>r0</i>					
<i>r1</i>					
<i>r2</i>					
<i>r3</i>					
<i>r4</i>					
<i>r5</i>					
<i>r6</i>					
<i>r7</i>					
<i>r8</i>	<i>r8_fiq</i>				
<i>r9</i>	<i>r9_fiq</i>				
<i>r10</i>	<i>r10_fiq</i>				
<i>r11</i>	<i>r11_fiq</i>				
<i>r12</i>	<i>r12_fiq</i>				
<i>r13 sp</i>	<i>r13_fiq</i>	<i>r13_irq</i>	<i>r13_svc</i>	<i>r13_undef</i>	<i>r13_abt</i>
<i>r14 lr</i>	<i>r14_fiq</i>	<i>r14_irq</i>	<i>r14_svc</i>	<i>r14_undef</i>	<i>r14_abt</i>
<i>r15 pc</i>					
<i>cpsr</i>					
-	<i>spsr_fiq</i>	<i>spsr_irq</i>	<i>spsr_svc</i>	<i>spsr_undef</i>	<i>spsr_abt</i>

Figure: Complete ARM Register Set

- ✓ The processor mode can be changed by a program that writes directly to the *cpsr* (the processor core has to be in privileged mode) or by hardware when the core responds to an exception or interrupt.
- ✓ The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*.
- ✓ Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.
- ✓ The following Figure illustrates what happens when an interrupt forces a mode change.

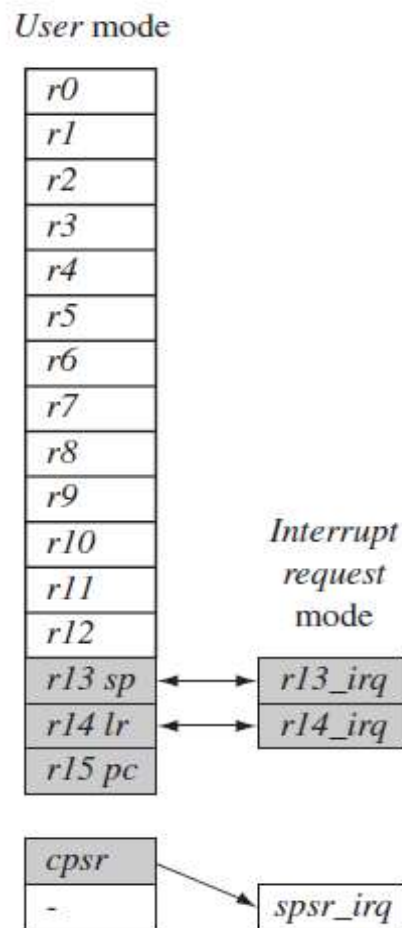


Figure: Changing Mode on an Exception

- ✓ The Figure shows the core changing from user mode to interrupt request mode, which happens when an interrupt request occurs due to an external device raising an interrupt to the processor core.
- ✓ This change causes user registers *r13* and *r14* to be banked. The user registers are replaced with registers *r13_irq* and *r14_irq*, respectively.
 - Note *r14_irq* contains the return address and *r13_irq* contains the stack pointer for interrupt request mode.

- ✓ The above Figure also shows a new register appearing in interrupt request mode: the *saved program status register (spsr)*, which stores the previous mode's *cpsr*. The *cpsr* being copied into *spsr_irq*.
- ✓ To return back to user mode, a special return instruction is used that instructs the core to restore the original *cpsr* from the *spsr_irq* and bank in the user registers *r13* and *r14*.
- ✓ Note that, the *spsr* can only be modified and read in a privileged mode. There is no *spsr* available in user mode.
- ✓ Another important feature to note is that the *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*. The saving of the *cpsr* only occurs when an exception or interrupt is raised.
- ✓ When power is applied to the core, it starts in supervisor mode, which is privileged. Starting in a privileged mode is useful since initialization code can use full access to the *cpsr* to set up the stacks for each of the other modes.
- ✓ The following Table lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr*.

Table: Processor Mode

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

State and Instruction Sets:

- ✓ The state of the core determines which instruction set is being executed. There are three instruction sets:
 - ARM
 - Thumb
 - Jazelle.
- ✓ The **ARM instruction set** is only active when the processor is in ARM state.
- ✓ The **Thumb instruction set** is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions.

- ✓ You cannot inter-mingle sequential ARM, Thumb, and Jazelle instructions.
- ✓ The Jazelle *J* and Thumb *T* bits in the *cpsr* reflect the state of the processor.
 - When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor.
 - When the *T* bit is 1, then the processor is in Thumb state.
- ✓ To change states the core executes a specialized branch instruction.

The following Table compares the ARM and Thumb instruction set features.

Table: ARM and Thumb Instruction Set Features

-	ARM (<i>cpsr T</i> = 0)	Thumb (<i>cpsr T</i> = 1)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +pc	8 general-purpose registers +7 high registers +pc

- ✓ The ARM designers introduced a third instruction set called Jazelle. **Jazelle** executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java byte-codes.
- ✓ To execute Java byte-codes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.

The following Table gives the Jazelle instruction set features.

Table: Jazelle instruction set features

-	Jazelle (<i>cpsr T</i> = 0, <i>J</i> = 1)
Instruction size	8-bit
Core Instructions	Over 60% of the Java byte-codes are implemented in hardware; the rest of the codes are implemented in software

Interrupt Masks:

- ✓ *Interrupt masks* are used to stop specific interrupt requests from interrupting the processor.
- ✓ There are two interrupt request levels available on the ARM processor core—
 - interrupt request (IRQ)
 - fast interrupt request (FIQ).

- ✓ The *cpsr* has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ, respectively.
- ✓ The *I* bit masks IRQ when set to binary 1; and similarly, the *F* bit masks FIQ when set to binary 1.

Condition Flags:

- ✓ Condition flags are updated by comparisons and the result of ALU operations that specify the **S** instruction suffix.
 - For example, if a SUBS subtract instruction results in a register value of zero, then the *Z* Flag in the *cpsr* is set. This particular subtract instruction specifically updates the *cpsr*.
- ✓ With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction. The flag is “sticky” in the sense that the hardware only sets this flag. To clear the flag you need to write to the *cpsr* directly.
- ✓ In Jazelle-enabled processors, the *J* bit reflects the state of the core; if it is set, the core is in Jazelle state. The *J* bit is not generally usable and is only available on some processor cores. To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.
- ✓ Most ARM instructions can be executed conditionally on the value of the condition flags.

The following Table lists the condition flags and a short description on what causes them to be set.

Table: Condition Flags

Flag	Flag Name	Set When
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero
N	Negative	bit 31 of the result is a binary 1

These flags are located in the most significant bits in the *cpsr*. These bits are used for conditional execution. The following Figure shows a typical value for the *cpsr* with both DSP extensions and Jazelle.



Figure: Example: *cpsr* = nzCvqjiFt_SVC

- ✓ For the condition flags a capital letter shows that the flag has been set. For interrupts a capital letter shows that an interrupt is disabled.
- ✓ In the *cpsr* example shown in above Figure, the *C* flag is the only condition flag set. The rest *nzvq* flags are all clear.
- ✓ The processor is in ARM state because neither the Jazelle *j* nor Thumb *t* bits are set. The IRQ interrupts are enabled, and FIQ interrupts are disabled.
- ✓ Finally, you can see from the Figure, the processor is in *supervisor (SVC) mode*, since the *mode[4:0]* is equal to binary 10011.

Conditional Execution:

- ✓ Conditional execution controls whether or not the core will execute an instruction.
- ✓ Prior to execution, the processor compares the condition attribute with the condition flags in the *cpsr*. If they match, then the instruction is executed; otherwise the instruction is ignored.
- ✓ The condition attribute is post-fixed to the instruction mnemonic, which is encoded into the instruction.
- ✓ The following Table lists the conditional execution code mnemonics. When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute.

Table: Condition Mnemonics

Mnemonic	Name	Condition flags
EQ	equal	<i>Z</i>
NE	not equal	<i>z</i>
CS HS	carry set/unsigned higher or same	<i>C</i>
CC LO	carry clear/unsigned lower	<i>c</i>
MI	minus/negative	<i>N</i>
PL	plus/positive or zero	<i>n</i>
VS	overflow	<i>V</i>
VC	no overflow	<i>v</i>
HI	unsigned higher	<i>zC</i>
LS	unsigned lower or same	<i>Z</i> or <i>c</i>
GE	signed greater than or equal	<i>NV</i> or <i>nv</i>
LT	signed less than	<i>Nv</i> or <i>nV</i>
GT	signed greater than	<i>NzV</i> or <i>nzv</i>
LE	signed less than or equal	<i>Z</i> or <i>Nv</i> or <i>nV</i>
AL	always (unconditional)	ignored

PIPELINE:

- ✓ A *pipeline* is the mechanism in a RISC processor, which is used to execute instructions.
- ✓ Pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.

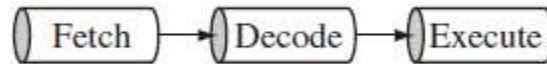


Figure: ARM7 Three-stage Pipeline

The above Figure shows a three-stage pipeline:

- *Fetch* loads an instruction from memory.
- *Decode* identifies the instruction to be executed.
- *Execute* processes the instruction and writes the result back to a register.

The following Figure illustrates pipeline using a simple example.

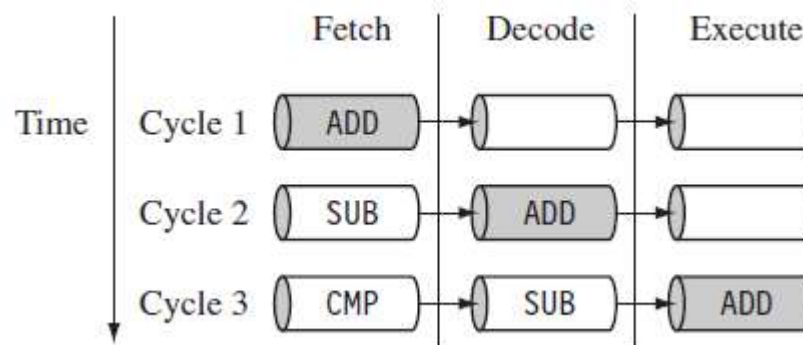


Figure: Pipelined Instruction Sequence

- ✓ The Figure shows a sequence of three instructions being fetched, decoded, and executed by the processor.
 - The three instructions are placed into the pipeline sequentially.
 - In the first cycle, the core fetches the ADD instruction from memory.
 - In the second cycle, the core fetches the SUB instruction and decodes the ADD instruction.
 - In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched.
- ✓ This procedure is called ***filling the pipeline***.
- ✓ The pipeline allows the core to execute an instruction every cycle.
 - As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn *increases the performance*.
 - The increased pipeline length also means increased *system latency* and there can be *data dependency* between certain stages.
- The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure.

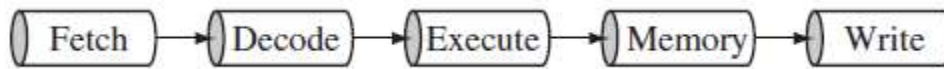


Figure: ARM9 Five-stage Pipeline

- The ARM9 adds a memory and writeback stage, which allows the ARM9 to –
 - process on average 1.1 Dhrystone MIPS per MHz
 - increase the instruction throughput in ARM9 by around 13% compared with an ARM7.
- The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in the following Figure.

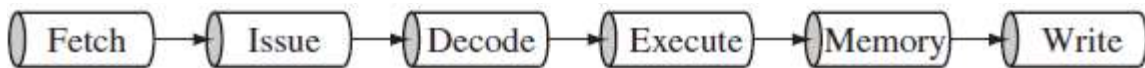


Figure: ARM10 Six-stage Pipeline

- The ARM10 –
 - can process on average 1.3 Dhrystone MIPS per MHz
 - have about 34% more throughput than an ARM7 processor core
 - but again at a higher latency cost.

NOTE: Even though the ARM9 and ARM10 pipelines are different, they still use the same *pipeline executing characteristics* as an ARM7. Hence, code written for the ARM7 will execute on an ARM9 or ARM10.

Pipeline Executing Characteristics:

- ✓ The ARM pipeline will not process an instruction, until it passes completely through the execute stage.
 - For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.

The following Figure shows an instruction sequence on an ARM7 pipeline.

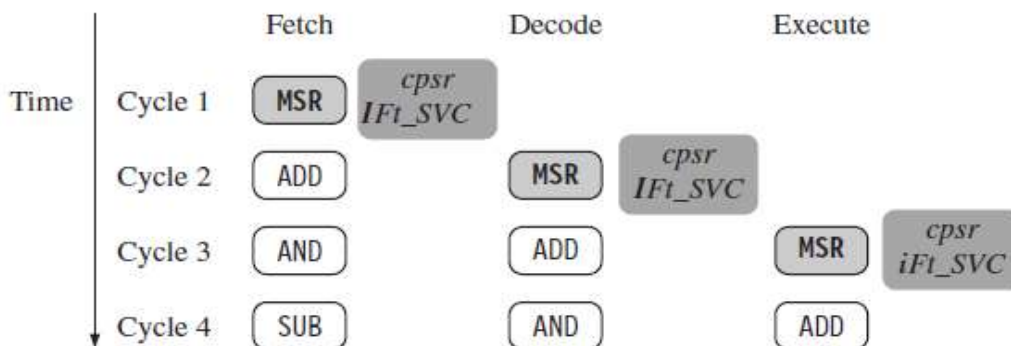


Figure: ARM Instruction Sequence

- ✓ The MSR instruction is used to enable IRQ interrupts, which only occurs once the MSR instruction completes the execute stage of the pipeline. It clears the *I* bit in the *cpsr* to enable the IRQ interrupts.
- ✓ Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.

The following Figure illustrates the use of the pipeline and the program counter *pc*.

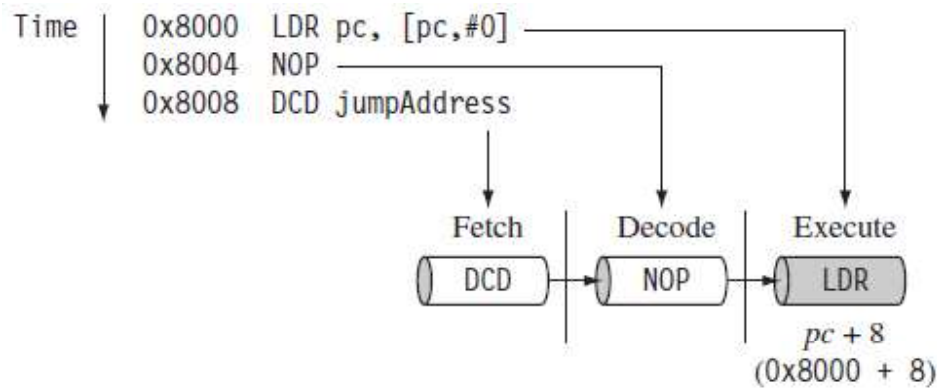


Figure: Example: $pc = \text{address} + 8$

- ✓ In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes. In other words, the *pc* always points to the address of the instruction being executed plus two instructions ahead.
- ✓ Note when the processor is in Thumb state the *pc* is the instruction address plus 4.
- ✓ There are three other characteristics of the pipeline.
 - First, the execution of a branch instruction or branching by the direct modification of the *pc* causes the ARM core to flush its pipeline.
 - Second, ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.
 - Third, an instruction in the execute stage will complete even though an interrupt has been raised. Other instructions in the pipeline will be abandoned, and the processor will start filling the pipeline.

EXCEPTIONS, INTERRUPTS AND THE VECTOR TABLE:

- ✓ When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the *vector table*.
 - The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.
 - The memory map address 0x00000000 (or in some processors starting at the offset 0xffff0000) is reserved for the vector table, a set of 32-bit words.

- ✓ When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see the following Table).

Table: The Vector Table

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

- ✓ Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:
 - **Reset** vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
 - **Undefined** instruction vector is used when the processor cannot decode an instruction.
 - **Software interrupt** vector is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
 - **Prefetch abort** vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
 - **Data abort** vector is similar to a prefetch abort, but is raised when an instruction attempts to access data memory without the correct access permissions.
 - **Interrupt request** vector is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.
 - **Fast interrupt request** vector is similar to the interrupt request, but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the *cpsr*.

CORE EXTENSIONS:

- ✓ *Core extensions* are the standard hardware components placed next to the ARM core.
- ✓ They improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications.

Each ARM family has different extensions available. There are *three hardware extensions*: cache and tightly coupled memory, memory management, and the coprocessor interface.

Cache and Tightly Coupled Memory:

- ✓ The cache is a block of fast memory placed between main memory and the core. It allows for more efficient fetches from some memory types. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.
- ✓ Most ARM-based embedded systems use a single-level cache internal to the processor.
- ✓ ARM has *two forms of cache*. The first is found attached to the Von Neumann-style cores. It combines both data and instruction into a single unified cache, as shown in the following Figure.

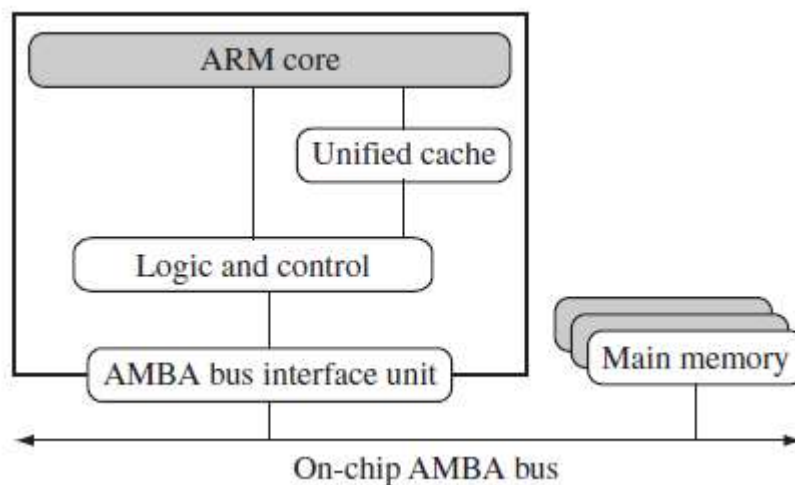


Figure: Von Neumann Architecture with Cache

- ✓ The second form, attached to the Harvard-style cores, has separate caches for data and instruction, as shown in the following Figure.

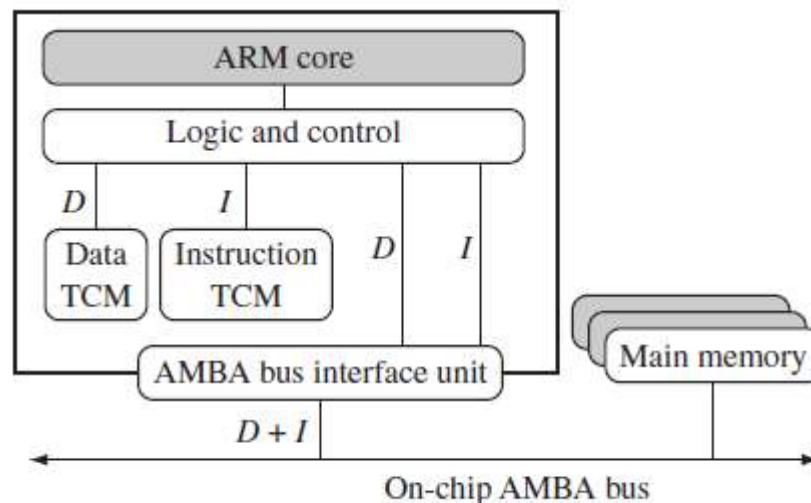


Figure: Harvard Architecture with TCMs

- ✓ A cache provides an overall increase in performance, but at the expense of predictable execution. But the real-time systems require the code execution to be deterministic— the time taken for loading and storing instructions or data must be predictable.
- ✓ This is achieved using a form of memory called *tightly coupled memory (TCM)*. TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data.
- ✓ TCMs appear as memory in the address map and can be accessed as fast memory.

By combining both technologies, ARM processors can have both improved performance and predictable real-time response. The following Figure shows an example core with a combination of caches and TCMs.

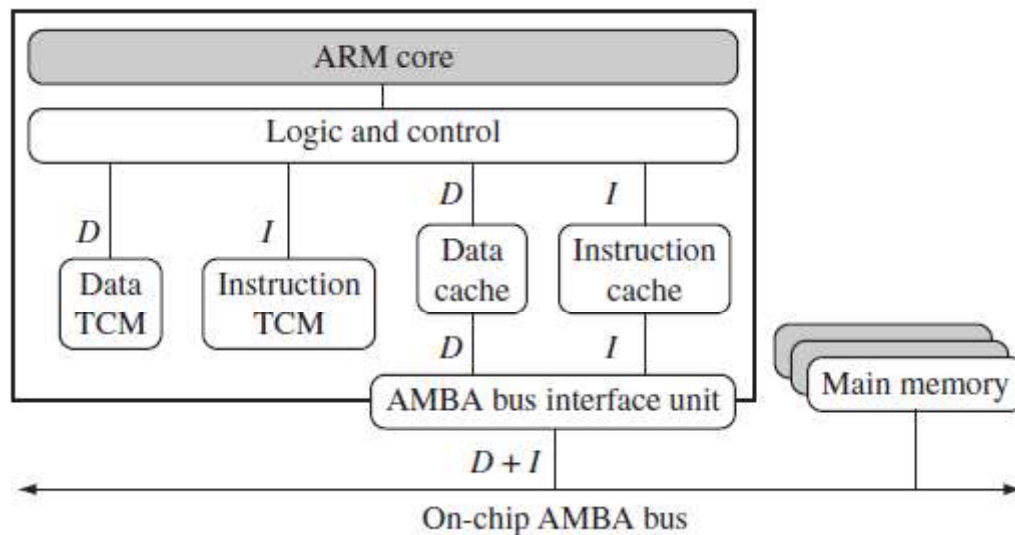
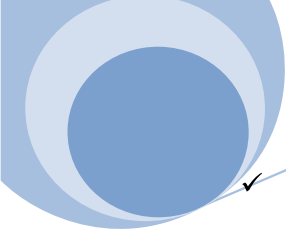


Figure: Harvard Architecture with Caches and TCMs

Memory Management:

- ✓ Embedded systems often use multiple memory devices. It is usually necessary to have a method to organize these devices and protect the system from applications trying to make inappropriate accesses to hardware. This is achieved with the assistance of memory management hardware.
- ✓ ARM cores have *three different types of memory management hardware*—
 - no extensions providing no protection
 - a memory protection unit (MPU) providing limited protection
 - a memory management unit (MMU) providing full protection
- ✓ **Non protected memory** is fixed and provides very little flexibility. It is normally used for small, simple embedded systems that require no protection from rogue applications.

- 
- ✓ **MPUs** employ a simple system that uses a limited number of memory regions. These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permissions. This type of memory management is used for systems that require memory protection but don't have a complex memory map.
 - ✓ **MMUs** are the most comprehensive memory management hardware available on the ARM. The MMU uses a set of translation tables to provide fine-grained control over memory. These tables are stored in main memory and provide a virtual-to-physical address map as well as access permissions. MMUs are designed for more sophisticated platform operating systems that support multitasking.

Coprocessors:

- ✓ Coprocessors can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers. More than one coprocessor can be added to the ARM core via the coprocessor interface.
- ✓ The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface.
 - For example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.
- ✓ The coprocessor can also extend the instruction set by providing a specialized group of new instructions.
 - For example, there are a set of specialized instructions that can be added to the standard ARM instruction set to process vector floating-point (VFP) operations.
- ✓ These new instructions are processed in the decode stage of the ARM pipeline.
 - If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor.
 - If the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software.

