ARM PROCESSOR

UNIT-4

Contents

- *Fundamentals*
- *Registers*
- *Current program status register*
- *Pipeline concept*
- *Interrupt and the vector table*

Fundamentals

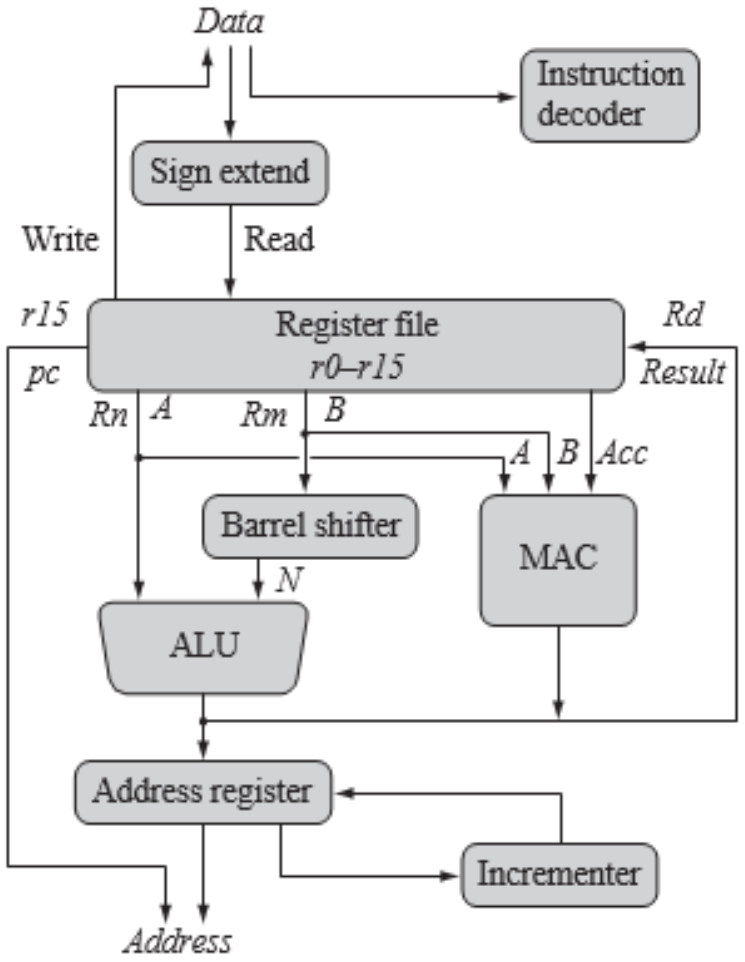
- ARM is **Advanced RISC Machine**, earlier known as Acorn RISC Machine.
- It was built by **Acorn Computers** along with VLSI technology in **1990.**
- ARM is a **32-bit microprocessor** . The ARM processor core is a key component of many successful 32-bit embedded systems.
- ARM's designers have come a long way from the first **ARM1 prototype in 1985**. Over one billion ARM processors had been shipped worldwide by the end of 2001. The **ARM core uses a RISC architecture**.
- RISC processors are designed to perform a smaller number of computer instructions so that they can operate at a higher speed, performing more millions of instructions per second (MIPS) as compared to CISC (Complex instruction set Computer) processors.
- ARM is one of the most licensed and thus **widespread processor cores** in the world.
- ARM cores are widely used in **mobile phones, handheld organizers, and a multitude of other everyday portable consumer device**s due to low power consumption and reasonable performance (MIPS/watt).
- **ARM does not manufacture its own VLSI devices**. It licenses out its core to many companies such as TI, Philips, Intel etc.

Fundamentals

- Because of their **reduced instruction set, they require fewer transistors(*This is because the simpler instructions require less decoding and processing, allowing for a more streamlined execution path.*)**, which enables a smaller die size for the integrated circuitry (IC).
- The ARM processor's smaller size, reduced complexity and lower power consumption makes them suitable for increasingly miniaturized devices.
- ARM7 and older versions support **Von Neumann Architecture.**
- **ARM9 and newer versions support Harvard Architecture**.
- In Von Neumann implementation data items and instructions share same bus.
- In Harvard implementation two different buses –ARM high performance bus (AHB) and ARM Peripheral Bus (APB) for high performance and low performance peripherals respectively.

Fundamentals

- ARM core is a collection of functional units connected by data buses, as shown in Figure below , where, the arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area. The figure shows not only the flow of data but also the abstract components that make up an ARM core.



Eg of data processing operation involving the use of barrel shifter.

Pre:r5=5

r7=8

MOV r7,r5,LSL #2

Post r5=5

r7=20

Fundamentals

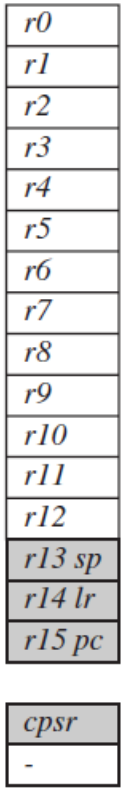
- Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item. Figure in the previous slide shows a **Von Neumann implementation of the ARM—data items and instructions share the same bus. I**n contrast, Harvard implementations of the ARM use two different 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 a load-store architecture. This 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, and conversely the 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 solely 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, *Rn* and *Rm*,** and a single result or destination register, *Rd*. Source operands are read from the register file using the internal buses *A* and *B*, respectively.

Fundamentals

- The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values *Rn* and *Rm* from the *A* and *B* buses and computes a result. Data processing instructions write the result in *Rd* 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 *Rm* 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 *Rd* 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. Figure below shows the active registers available in user mode—a protected mode normally used when executing applications. **The processor can operate in seven different modes**.



Registers

- 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 are visible to the programmer as r0 to r15.
- The ARM processor has three registers assigned to a particular task or special function: r13, r14, and r15. They are frequently given different labels to differentiate them from the other registers.
- In the Figure in the previous slide , the shaded registers identify the assigned special-purpose 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.

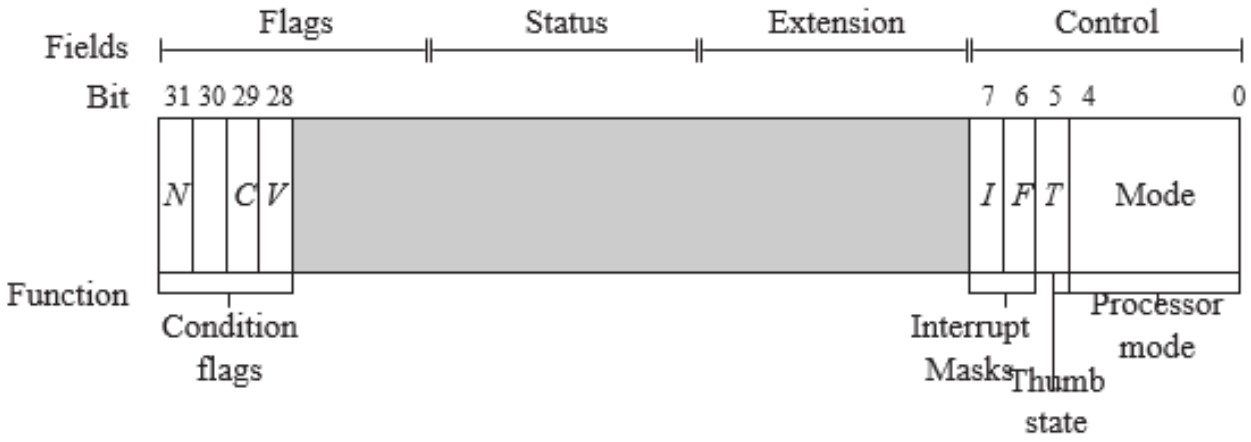
■ Register r15 is the program counter (pc) and contains the address of the next instruction to be fetched by the processor.

Registers

- **Depending upon the context**, **registers *r13* and *r14* can also be used as general-purpose registers**, which can be particularly useful since these registers are banked during a processor mode change. However, it is dangerous to use *r13* as a general register when the processor is running any form of operating system because operating systems often assume that *r13* always points to a valid stack frame.
- 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. However, **there are instructions that treat r14 and r15 in a special way.**
- In addition to the 16 data registers, there are two program status registers: cpsr and spsr(the current and saved program status registers, respectively).
- The register file contains all the registers available to a programmer. **Which registers are visible to the programmer depend upon the current mode of the processor.**

CPSR(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.
- Figure below shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.



A generic program status register (*psr*).

CPSR(Current Program Status Register)

- 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.**

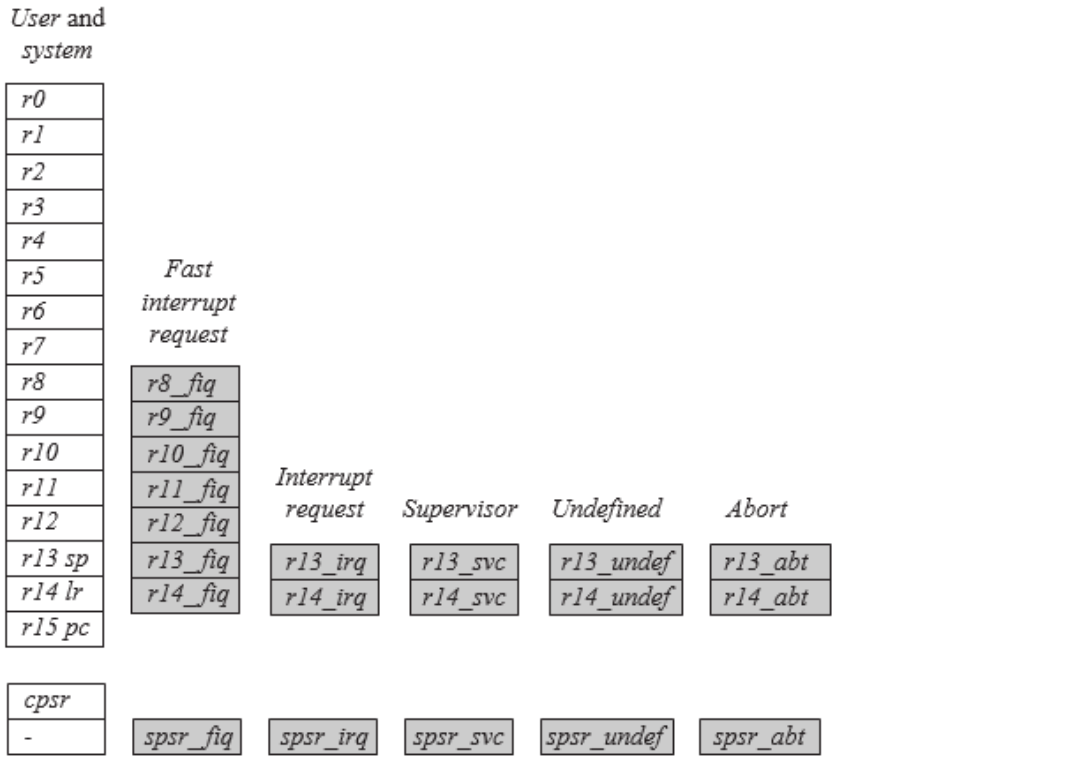
CPSR(Current Program Status Register)

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 nonprivileged: **A privileged mode allows full read-write access to the cpsr.**
- Conversely, a nonprivileged 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) and one nonprivileged mode (user).
- 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.
- User mode is used for **programs and applications.**

CPSR(Current Program Status Register)

- ***BANKED REGISTERS:***Figure shows all 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.



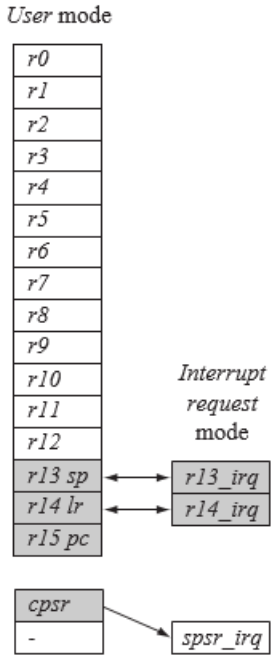
Complete ARM register set.

CPSR(Current Program Status Register)

- 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.
- 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.

CPSR(Current Program Status Register)

- 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.
- Figure below illustrates what happens when an interrupt forces a mode change. **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.



CPSR(Current Program Status Register)

- The Figure in the previous slide also shows a new register appearing in *interrupt request* mode: the saved program status register *(spsr)*, which stores the previous mode's *cpsr.* You can see in the diagram 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.**
- The **saving of the cpsr only occurs when an exception or interrupt is raised**.
- The current active processor mode occupies the five least significant bits of the cpsr.
- 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**.

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

CPSR(Current Program Status Register)

STATE AND INSTRUCTION SETS

- The **state of the core determines which instruction set is being executed.** There are three instruction sets: ARM, Thumb, and Jazelle.
- The ARM instruction set is only active when the processor is in ARM state. Similarly 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 intermingle 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 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 bytecodes**.
- To execute Java bytecodes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.

Table 2.2 ARM and Thumb instruction set features.

	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	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*

CPSR(Current Program Status Register)

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) and 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.
- The carry flag is set from the result of barrel shifter as the last bit is shifted out. The N bit is set to bit 31 of the result. The Z flag is set if the result is zero.

CPSR(Current Program Status Register)

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, frequently used to indicate equality
N	Negative	bit 31 of the result is a binary 1

- 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 above Table lists the condition flags and a short description on what causes them to be set. These flags are located in the most significant bits in the cpsr. These bits are used for conditional execution.

CPSR(Current Program Status Register)

CONDITIONAL EXECUTION

- **Conditional execution controls whether or not the core will execute an instruction. Most instructions have a condition attribute that determines if the core will execute it based on the setting of the condition flags**. 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 postfixed to the instruction mnemonic, which is encoded into the instruction. When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute.

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

Table: Conditional Mnemonics

Pipelining

- Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.
- A **pipeline can be visualized as a collection of processing segments** through which binary information flows.
- Each segment performs partial processing dictated by the way the task is partitioned.
- The **result obtained from the computation in each segment is transferred to the next segment in the pipeline**. The **final result is obtained after the data have passed through all segments**.

Pipelining

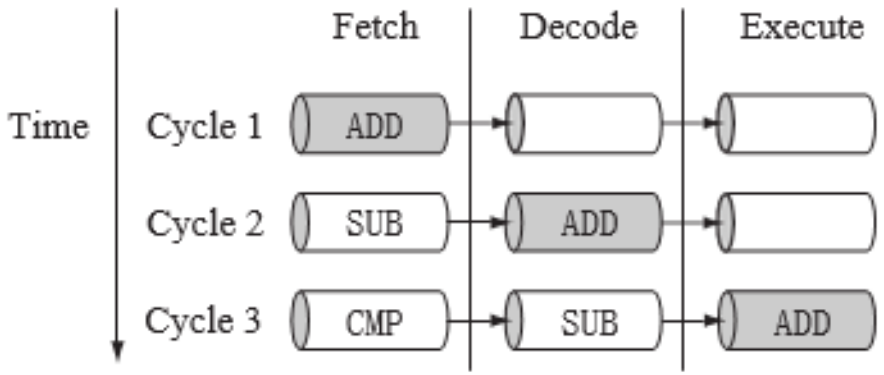
- A **pipeline is the mechanism a RISC processor uses to execute instructions**.
- Using a pipeline **speeds up execution by fetching the next instruction** while other instructions are being decoded and executed. One way to view the pipeline is to think of it as an automobile assembly line, with each stage carrying out a particular task to manufacture the vehicle.

ARM7 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.*

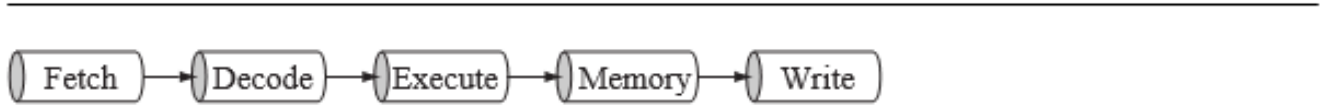
Pipelining

- Figure in the next slide illustrates the pipeline using a simple example. It shows a sequence of three instructions being fetched, decoded, and executed by the processor. Each instruction takes a single cycle to complete after the pipeline is filled.
- 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 system latency also increases because it takes more cycles to fill the pipeline before the core can execute an instruction. The increased pipeline length also means there can be data dependency between certain stages. You can write code to reduce this dependency by using instruction scheduling

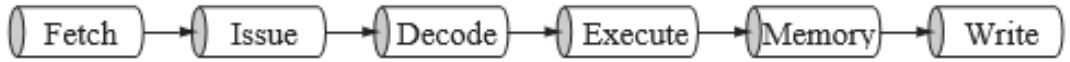


Pipelined instruction sequence.

Pipelining



ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

- The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages. The ARM9 adds a memory and writeback stage, which allows the ARM9 to process on average 1.1 Dhrystone MIPS per MHz.

Pipelining

The five stages of pipeline are:

- **Fetch** – The instruction is fetched from the memory and stored in the instruction register.
- **Decode** – The instruction is moved to the decoder which decodes the instruction. It activates the appropriate control signals and takes the necessary steps for the the next execution stage.
- **Execute** – An operand is shifted and the ALU result generated. If the instruction is a load or store, the memory address is computed in the ALU.
- **Memory**– Data memory is accessed if required. Otherwise the ALU result is simply buffered for one cycle.
- **Write back** – The result generated by the instruction are written back to the register file, including any data loaded from memory.

Pipelining

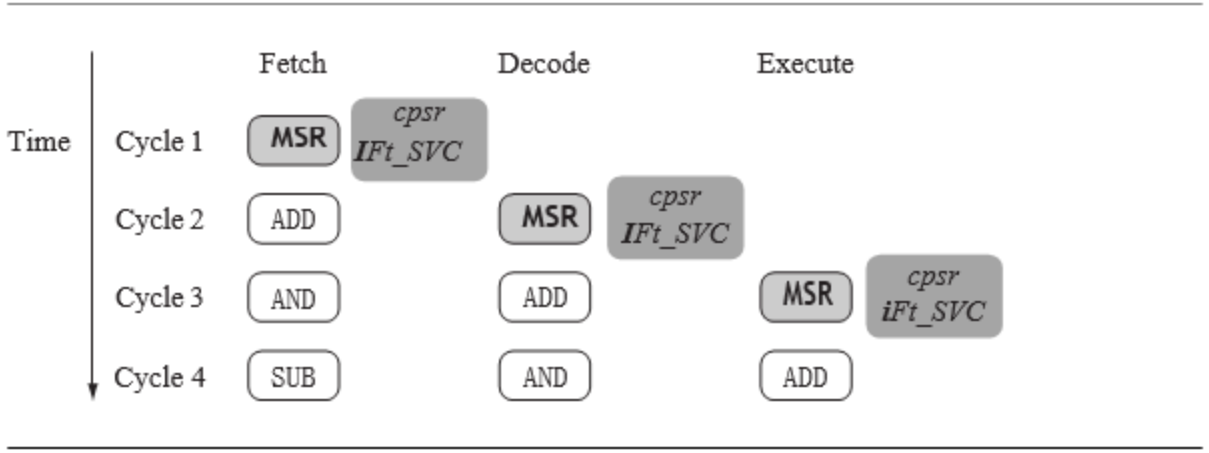
- The ARM10 increases the pipeline length still further by adding a sixth stage. The ARM10 can process on average 1.3 Dhrystone MIPS per MHz.
- During the issue stage, the processor checks if the instruction has any dependencies on previously executed instructions and if required are the rescources available.

PIPELINE EXECUTING CHARACTERISTICS

The ARM pipeline has not processed 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.

Pipelining

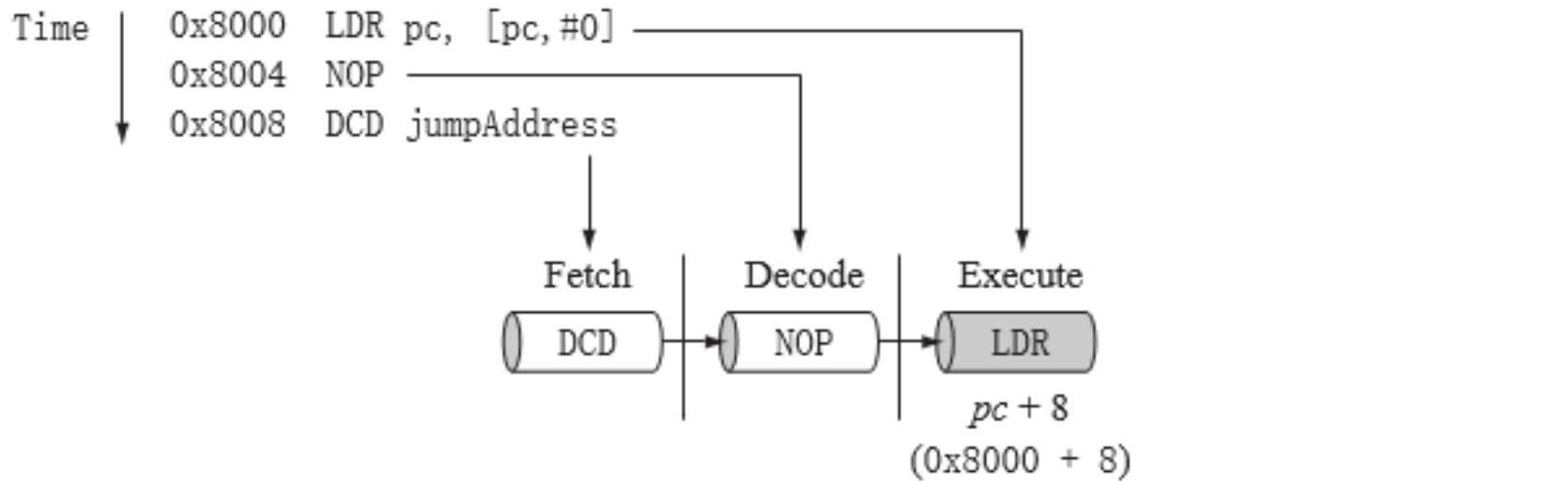
- Figure below shows an instruction sequence on an ARM7 pipeline. The MSR instruction is used to enable IRQ interrupts, which only occurs once the MSR instruction completes the execute stage of the pipeline. It clears the *I* bit in the *cpsr* to enable the IRQ interrupts. Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.



ARM instruction sequence.

Pipelining

- Figure in the next illustrates the use of the pipeline and the program counter pc. 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 from the appropriate entry in the vector table.



Example: *pc* = address + 8.

DRAWBACKS/ HAZARDS OF PIPELINING

1) DATA HAZARD/ DATA DEPENDENCY HAZARD

- Data Hazard is caused when **the result (destination) of one instruction becomes the operand (source) of the next instruction**.
- For eg: Consider two instructions I1 and I2 (I1 being the first).

Assume I1: LDR r0,[r1]

- Assume I2: ADD r2,**r0**,r3

Clearly in I2, r0 should get the value of location pointed by r1.

- But this can only happen once I1 has completely finished execution. In a multistage pipeline, I2 may reach execution stage before I1 has finished getting the value at location [r0], and hence effect the execution of L2 . This is called **data dependency hazard**

2) CONTROL HAZARD/ CODE HAZARD

- Pipelining assumes that the program will always flow in a sequential manner. Hence, it performs various stages of the forthcoming instructions before-hand, while the current instruction is still being executed. While programs are sequential most of the times, it is not true always. Sometimes, branches do occur in programs.

DRAWBACKS/ HAZARDS OF PIPELINING

Eg: B Down

ADD r0,r1,r2

MOV r5,r7 LSL #2

…

…

…

Down: SUB r0,r1,r2

- "B Down" is a branch instruction. After this instruction, program should jump to the location "Down" and continue with SUB r0,r1,r2 instruction.
- But, in a multistage pipeline processor, the sequentially next instructions after "B Down" have already been fetched and decoded. These instructions will now have to be discarded and fetching will begin all over again from SUB r0,r1,r2 . This will keep several units of the architecture idle for some time. This is called a pipeline bubble. The **problem of branching is solved** in higher processors by a method called "**Branch Prediction Algorithm**"

DRAWBACKS/ HAZARDS OF PIPELINING

3) STRUCTURAL HAZARD

- Structural hazards are caused by **physical constraints in the architecture like the buses**. Even in the most basic form of pipelining, we want to execute one instruction and fetch the next one. Now as long as execution only involves registers, pipelining is possible. But **if execution requires to read/ write data from the memory, then it will make use of the buses, which means fetching cannot take place at the same time**. So the fetching unit will have to wait and hence a pipeline bubble is caused. This problem is solved in complex Harvard architecture processors, which use separate memories and separate buses for programs and data. This means fetching and execution can actually happen at the same time without any interference with each other.

Interrupt 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 is reserved for the vector table, a set of 32-bit words.
- On some processors the vector table can be optionally located at a higher address in memory (starting at the offset 0xffff0000). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.
- When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table .
- Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:

Interrupt and the vector table

- 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.

Interrupt and the vector table

- *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*.

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

Interrupt and the vector table

- Note: Fast Interrupt Request has higher priority and faster response time than Interrupt requests.