ARM Microcontroller and Embedded Systems

Unit - 1

ARM DESIGN PHILOSOPHY

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

- Portable embedded systems require some form of battery power. The ARM processor has been specifically designed to be small to reduce power consumption and extend battery operation. This is essential for applications such as mobile phones and Personal Digital Assistants (PDAs).

- High code density is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions. High code density is important in mobile phones that contain limited amount of memory.

- Embedded systems are price sensitive and use slow and low-cost memory devices. Usage of low cost memory devices, produces substantial savings in high-volume applications like digital cameras etc.

ARM DESIGN PHILOSOPHY contd...

- Another important requirement is to reduce the area of the die taken up by the embedded processor. Smaller the area used by the embedded processor, more available space for specialized peripherals. This reduces the design and manufacturing cost.

- ARM has incorporated hardware debug technology within the processor so that software engineers can view what is happening while the processor is executing code. With greater visibility, software engineers can resolve issues faster and this reduces overall development costs.

ARM DESIGN PHILOSOPHY contd...

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. The transfer can occur on sequential memory addresses, which increases performance. Code density is also improved since multiple register transfers are common operations at the start and end of functions.

- ***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. This expands the capability of many instructions to improve core performance and code density.

ARM DESIGN PHILOSOPHY contd...

INSTRUCTION SET FOR EMBEDDED SYSTEMS

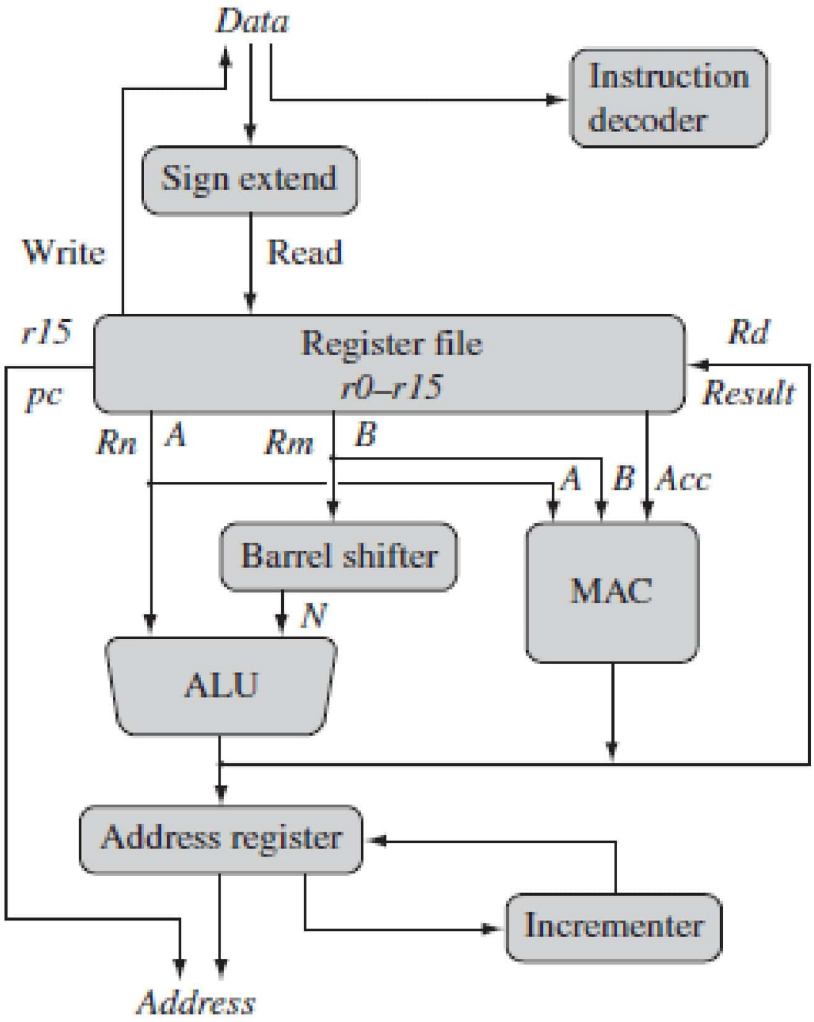
- ***Thumb 16-bit instruction set***—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions.

- ***Conditional execution***—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.

- ***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. These instructions are useful for signal processing applications.

"These additional features have made the ARM processor core one of the most commonly used 32-bit embedded processor cores."

ARM PROCESSOR CORE DATA FLOW MODEL



ARM PROCESSOR CORE DATA FLOW MODEL contd...

- A programmer can think of an ARM core as functional units connected by data buses as shown in figure where, the arrows represent the flow of data, lines represent 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 decodes the instructions before they are executed.

- 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
  - store instructions copy data from registers to memory.
- The data processing is carried out only in registers.

ARM PROCESSOR CORE DATA FLOW MODEL contd...

- 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.
- 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 which is directly stored in *Rd using result 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.

ARM PROCESSOR CORE DATA FLOW MODEL contd...

- Load and store instructions use the ALU to generate an address to be held in the address register and sent on the *Address* bus.

- For load and store instructions the incrementer unit updates the address register before the core reads or writes the next register value from or to the next sequential memory location.

REGISTERS

- General-purpose registers hold either data or an address.
- Registers are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4.*
- Below figure shows the active registers available in ***user*** **mode**—a protected mode normally used when executing applications.

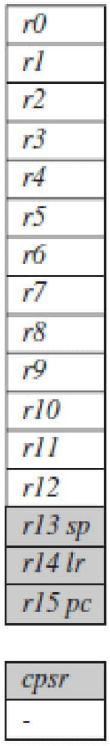


Figure: ***Registers available in user mode.***

REGISTERS contd...

- All the registers shown are 32 bits in size.
- There are up to 18 active registers:
  - 16 data registers which are visible to the programmer as *r0* to *r15*
  - 2 processor status registers.
- 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.
- 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 *r13* and *r14* can also be used as general-purpose registers. 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.

REGISTERS contd...

- 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* (Current Program Status Register)
  - *spsr* (Saved Program Status Register)

- 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.

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.
- Below Figure shows the basic layout of a generic program status register.

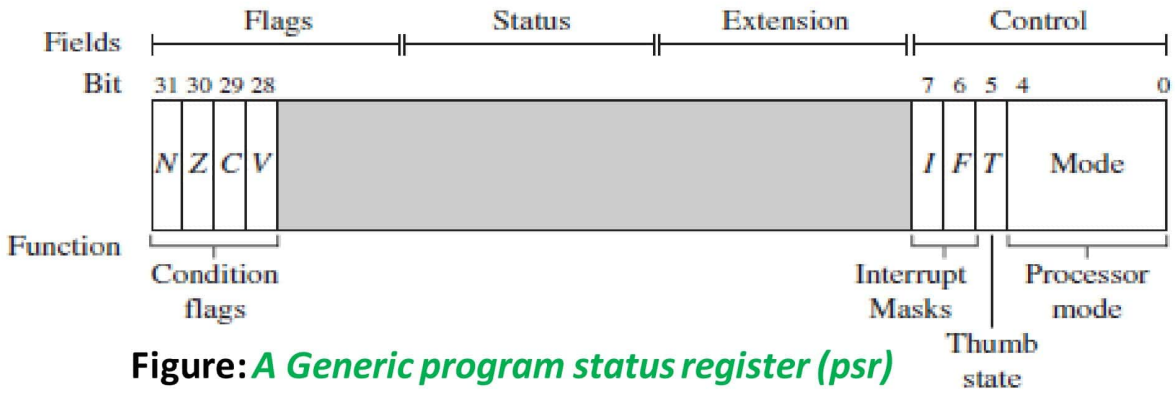


Figure: *A Generic program status register (psr)*

- 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.

ARM Microcontroller and Embedded Systems

Lecture-6

Processor Modes

- The processor mode determine 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 cpsr.
- A non-privileged mode only allows read access to the control field in 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
    - Undefined
  - **One nonprivileged mode**
    - User

Processor Modes contd...

- ***Abort Mode:*** The processor enters *abort* mode when there is a failed attempt to access memory.
- ***Fast interrupt request and interrupt request modes:*** These modes correspond to the two interrupt levels available on the ARM processor.
- ***Supervisor* mode:** It is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
- ***System* mode:** It is a special version of *user* mode that allows full read-write access to the *cpsr*.
- ***Undefined mode:*** This mode is used when the processor encounters an instruction that is undefined or not supported by the implementation.
- ***User mode:*** This mode is used for programs and applications.

Processor Modes contd...

- Below table lists various modes and associated binary patterns that represent each of the processor mode in cpsr.

Mode	Abbreviation	Privileged	Mode [4:0] in cpsr	
			Binary	Hex
User	usr	no	10000	0x10
Fast Interrupt Request	fiq	yes	10001	0x11
Interrupt Request	irq	yes	10010	0x12
Supervisor	svc	yes	10011	0x13
Abort	abt	yes	10111	0x17
undefined	und	yes	11011	0x1B
System	sys	yes	11111	0x1F

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 (bit 7) masks (disables) IRQ** when set to binary 1.

- The ***F* bit (bit 6) masks (disables) FIQ** when set to binary 1.

State and Instruction Set

- 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**.
- The Thumb instruction set is only active when the processor is in **Thumb state**.
- 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 32-bit 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** and executes **16-bit instructions**.
- To change states the core executes a **specialized branch instruction**.

State and Instruction Set contd...

- The ARM designers introduced a third instruction set called ***Jazelle***.
- *When T=0 and J=1, the processor is in Jazelle state and* executes **8-bit instructions.**
- Below table compares the 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*

ARM Microcontroller and Embedded Systems

Lecture-7 and Lecture-8

Condition Flags

- The condition flags are located in the **most significant bits in the *cpsr***.
- Condition flags are updated by **comparisons** and the **result of ALU operations** that specify the **S instruction suffix**.
- **Example**: If a **SUBS** subtract instruction results in a register value of zero, then the ***Z* flag in the *cpsr* is set to 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 Q flag is "sticky" and will remain set until explicitly cleared. 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**.
- Conditional flag bits are used for **conditional execution**.

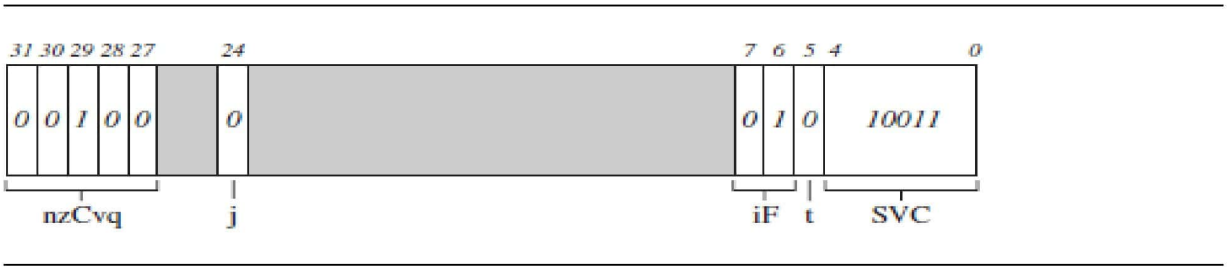
Condition Flags contd...

- Below table lists the **condition flags** and a short description on what causes them to be **set**.

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

Condition Flags contd...

- Below Figure shows a typical value for the *cpsr* with both DSP extensions and Jazelle.



Example: *cpsr* = *nzCvqjiFt_SVC*.

- When a bit is a binary 1 we use a capital letter; when a bit is a binary 0, we use a lowercase letter.
- 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 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* or Thumb *t* bits are set.
- The IRQ interrupts are enabled, and FIQ interrupts are disabled.
- 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.

- 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**.

Conditional Execution contd...

- Below table lists the **conditional execution code mnemonics** (condition attribute).

Code		Mnemonic	Name	Condition flags	
0000	0	EQ	equal	*Z*	
0001	1	NE	not equal	*z*	
0010	2	CS HS	carry set/unsigned higher or same	*C*	
0011	3	CC LO	carry clear/unsigned lower	*c*	
0100	4	MI	minus/negative	*N*	
0101	5	PL	plus/positive or zero	*n*	
0110	6	VS	overflow	*V*	
0111	7	VC	no overflow	*v*	
1000	8	HI	unsigned higher	*zC*	
1001	9	LS	unsigned lower or same	*Z* or *c*	
1010	A	GE	signed greater than or equal	*NV* or *nv*	**(N==V)**
1011	B	LT	signed less than	*Nv* or *nV*	**(N!=V)**
1100	C	GT	signed greater than	*NzV* or *nzv*	
1101	D	LE	signed less than or equal	*Z* or *Nv* or *nV*	**(Z or N!=V)**
1110	E	AL	always (unconditional)	ignored	

Conditional Execution contd...

- **Example1:**

```
MOV R0,#10

CMP R0,#5

MOVEQ R1,#0X10

MOVNE  R1,#0X20
```

Conditional Execution contd...

- **Example2:**

```
MOV R0,#10

CMP R0,#5

MOVLE R1,#0

MOVGT  R1,#1
```

ARM Microcontroller and Embedded Systems

Lecture-10

Banked Registers

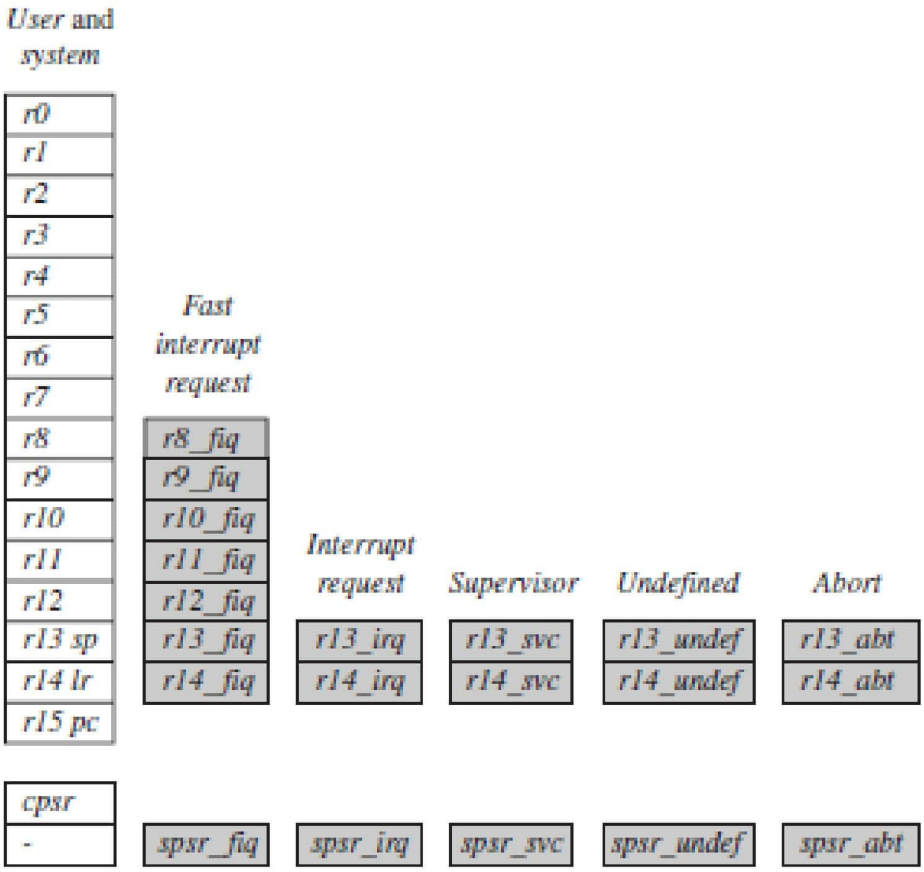


Figure: ***Complete ARM register set*****.**

Banked Registers contd...

- Figure shows all **37 registers** in the register file.
- Out of these 37 registers, **20 registers are hidden** from a program at different times. These registers are called ***banked registers***.
- Banked registers are available only when the processor is in a **particular mode**.

  Example: ***abort mode*** *has banked registers r13_abt, r14_abt and spsr_abt.*
- 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.

Banked Registers contd...

- A banked register maps one-to-one onto a *user mode register. If a processor mode is changed, a* ***banked register from the*** **new mode will replace an existing register.**

  Example: when the processor is in the ***interrupt request mode****, the instructions we* 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.

- The processor mode can be **changed by a program that writes directly to the *cpsr*** (privileged mode of processor) 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.

Banked Registers contd...

- Below figure illustrates what happens when an interrupt forces a mode change.

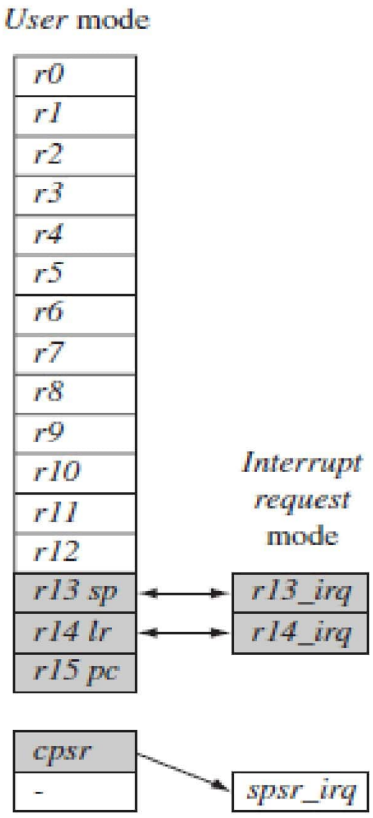


Figure: ***Changing mode on an exception.***

Banked Registers contd...

- 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.
- *r14_irq* contains the **return address** and *r13_irq* contains the **stack pointer for *interrupt request* mode**.
- Figure also shows a new register appearing in *interrupt request* mode: the **saved program status register *(spsr)*, which stores the previous mode's *cpsr*.**
- 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*.

Banked Registers contd...

- 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**.

- 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.

Note: The only privileged mode that is not entered by an exception is system mode.

ARM Microcontroller and Embedded Systems

Lecture-11

Pipeline

- 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.

"The pipeline design for each ARM family differs"

Pipeline contd...

Figure: ***ARM7 Three-stage pipeline.***

- 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.

Pipeline contd...

- Figure 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.

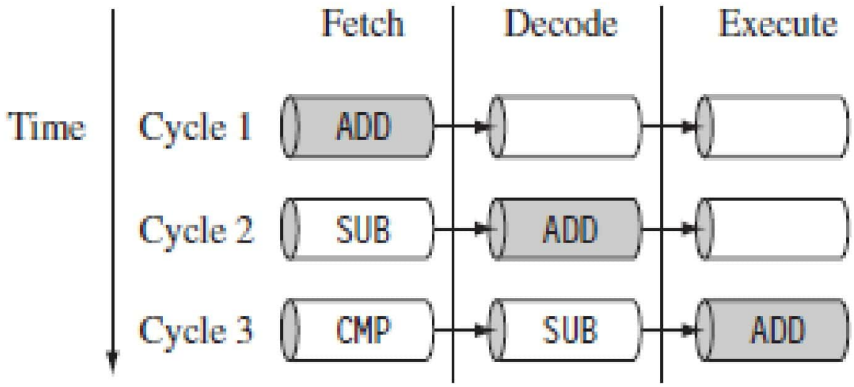


Figure: ***Pipelined Instruction Sequence.***

Pipeline contd...

- 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.

Pipeline contd...

- 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, because of which there is an increase in instruction throughput by around 13% compared with an ARM7.

- The maximum core frequency attainable using ARM9 is also higher.

Pipeline contd...

- The **ARM10** increases the pipeline length still further by adding a **sixth stage**, as shown in Figure.

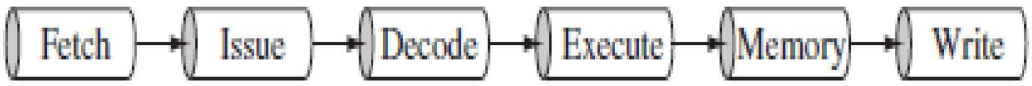


Figure: ***ARM10 six-stage pipeline.***

- There is an increase in instruction throughput by around 34% compared with ARM7 processor core, but again at a higher latency cost.

Pipeline Executing Characteristics contd...

- The ARM pipeline doesn't process an instruction until it passes completely through the execute stage.
- **Example:** An ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.

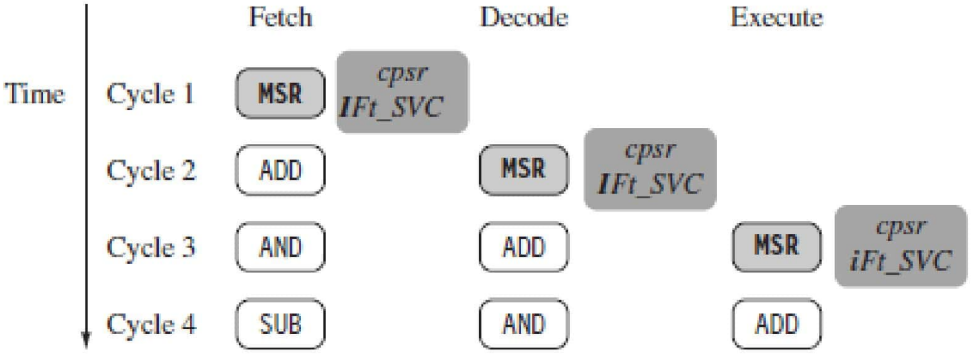


Figure: *ARM instruction sequence.*

- Figure 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.

Pipeline Executing Characteristics contd...

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

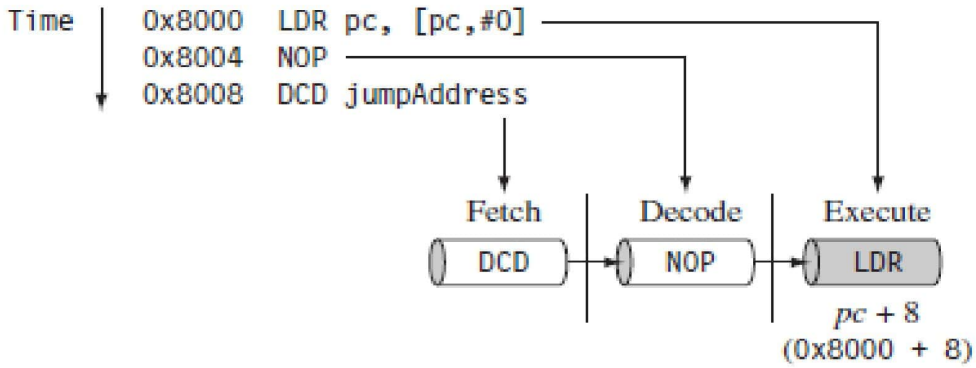


Figure: ***Example pc = address + 8.***

- In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes. i.e., *pc* always points to the address of the instruction being executed plus two instructions ahead.
  - This is important when the *pc* is used for calculating a relative offset and is an architectural characteristic across all the pipelines. Note when the processor is in Thumb state the *pc* is the instruction address plus 4.

Pipeline Executing Characteristics contd...

- There are three other characteristics of the pipeline worth mentioning.
  - **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.

Exceptions, Interrupts and Vector table

- 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:

  - ***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.

Exceptions, Interrupts and Vector table contd...

- ***Prefetch abort vector*** is used when the processor attempts to fetch an instruction from an address without the correct access permissions.

- ***Data abort vector*** *is used* 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*** *used* for hardware requiring faster response times. It can only be raised if FIQs are not masked in the *cpsr*.

Exceptions, Interrupts and Vector table contd...

- 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**).

Table: ***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

Exceptions, Interrupts and Vector table contd...

When an exception or interrupt occurs, following event occurs:

- cpsr is copied to spsr.
- Mode bits in cpsr is changed and mode change takes place.
- New registers (banked registers) get activated.
- T bit is cleared to zero.
- Interrupts are disabled.
- PC content(return address) is saved in R14 register(LR) of that particular mode.
- PC is loaded with the vector address corresponding to the exception or interrupt.

After completion of ISR, following events occur:

- spsr content is copied to cpsr.
- R14 register (LR) content is transferred to PC.
- User mode registers are reactivated.

ARM Microcontroller and Embedded Systems

Lecture-12

Unit-3

ARM INSTRUCTION SET

ARM INSTRUCTION SET

- ARM instructions process the data held in **registers** and access **memory** only with **load and store** instructions.
- ARM instructions commonly take **two** or **three** operands.

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

- **Example:** The **ADD** instruction above adds the two values stored in registers *r1* and *r2* (the source registers). It writes the result to register *r3* (the destination register).

ARM INSTRUCTION SET contd...

- The examples follow this format:

```
PRE     <pre-conditions>
        <instruction/s>
POST    <post-conditions>
```

- In the pre- and post-conditions, memory is denoted as

```
mem<data_size>[address]
```

- This refers to *data_size* bits of memory starting at the given byte *address*.
- Example: *mem32*[1024] is the 32-bit value starting at address 1024.

ARM INSTRUCTION SET contd...

- The **instruction classes** of ARM are as follows:
  - **Data processing instructions**
  - **Branch instructions**
  - **Load-store instructions**
  - **Software interrupt instruction**
  - **Program status register instructions**

DATA PROCESSING INSTRUCTIONS

- The **data processing instructions** manipulate data within registers.
- The Data processing instructions are as follows:
  - **Move instructions**
  - **Arithmetic instructions**
  - **Logical instructions**
  - **Comparison instructions**
  - **Multiply instructions**

- Most data processing instructions can process one of their operands using the **barrel shifter**.
- If you use the **S suffix** in a data processing instruction, then it updates the flags in the cpsr.

DATA PROCESSING INSTRUCTIONS contd...

- **Move** and **logical** operations update the carry flag *C*, negative flag *N*, and zero flag *Z*.
  - The ***Carry* flag C** is set from the result of the barrel shift as the last bit shifted out.
  - The ***Negative* flag N** is set to bit 31 of the result.
  - The ***Zero* flag Z** is set if the result is zero.

DATA PROCESSING INSTRUCTIONS contd...

Move Instructions

- Move is the simplest ARM instruction.

- This instruction is useful for setting initial values and transferring data between registers.

- **Syntax:**

  <instruction>{<cond>}{S} Rd, N

- It copies *N* into a destination register *Rd*, where *N* is a register or immediate value.

DATA PROCESSING INSTRUCTIONS contd...

Move Instructions

MOV Rd,N

- It copies N into the destination register Rd, where N (second operand) is a register R*m* or an immediate value preceded by #.

MVN Rd,N

- It copies NOT of N into the destination register Rd, where N (second operand) is a register R*m* or an immediate value preceded by #.

Note:

- **Only Destination operand is affected. Source operand is unaffected.**

DATA PROCESSING INSTRUCTIONS contd...

Examples:

1. Copy the contents of R5 register to R7 register.

PRE

R5 = 0x00000005

R7 = 0x00000008

MOV R7, R5

POST

R5 = 0x00000005

R7 = 0x00000005

DATA PROCESSING INSTRUCTIONS contd...

Examples:

2. Copy the NOT of contents of R5 register to R7 register.

PRE

R5 = 0x00000005

R7 = 0x00000008

MVN R7, R5

POST

R5 = 0x00000005

R7 = 0xFFFFFFFA

DATA PROCESSING INSTRUCTIONS contd...

Examples:

3. Give the status of C, Z and N flags after execution of the following instructions:

a) PRE

R1 = 0x00000002

R2 = 0x80000000

flags = **nzc**

MOVS R1, R2

POST

R1 = 0x80000000

R2 = 0x80000000

flags = Nzc

DATA PROCESSING INSTRUCTIONS contd...

Examples:

3. Give the status of C, Z and N flags after execution of the following instructions:

b) PRE

R0 = 0x00000050

flags = **nzc**

MOVS R0, #0

POST

R0 = 0x00000000

flags = nZc

DATA PROCESSING INSTRUCTIONS contd...

Examples:

3. Give the status of C, Z and N flags after execution of the following instructions:

c) PRE

R0 = 0x00000003

flags = **nzc**

MOVNES R0, #0

POST

R0 = 0x00000000

flags = nZc

DATA PROCESSING INSTRUCTIONS contd...

Examples:

3. Give the status of C, Z and N flags after execution of the following instructions:

d) PRE

R0 = 0x00000000

R1 = 0x90000034

flags = **nZc**

MOVEQS R0, R1

POST

R0 = 0x90000034

R1 = 0x90000034

flags = Nzc

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter

- The second operand N in MOV instruction (data processing) can be more than just a register or immediate value.

- It can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.

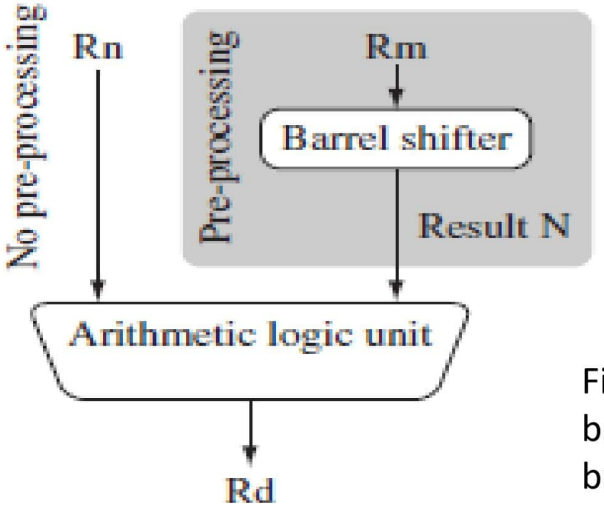


Figure shows the data flow between the ALU and the barrel shifter.

Figure: ***Barrel shifter and ALU.***

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter

- Data processing instructions are processed within the arithmetic logic unit (ALU).

- A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
  - This shift increases the power and flexibility of many data processing operations.

- Pre-processing or shift occurs within cycle time of the instructions.
  - This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter

- There are data processing instructions that do not use the barrel shift.
- **Example:**

  MUL (Multiply)

  CLZ (Count Leading Zeros)

  QADD (signed saturated 32-bit add)

ARM Microcontroller and Embedded Systems

Lecture-13

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter

- The five different shift operations that you can use within the barrel shifter are summarized in below table.

Table: ***Barrel Shifter Operations.***

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

Note: *x* represents the register being shifted and *y* represents the shift amount.

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter

- The syntax to perform Barrel shift operation in data processing instructions is shown in below table.

Table: ***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`

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter operation: LSL – Logical Shift Left

Example:

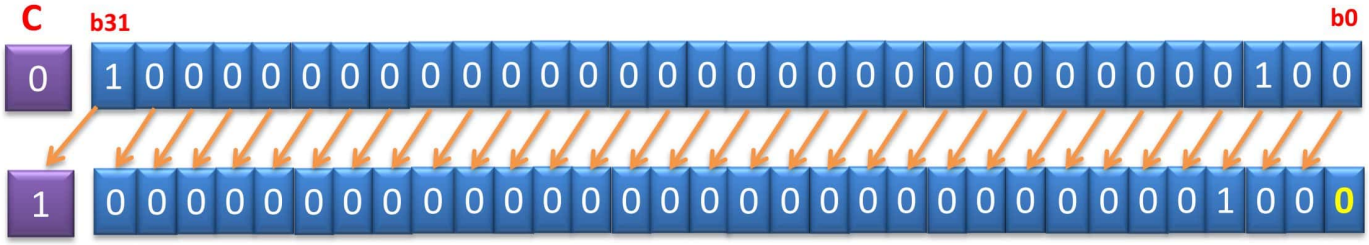
PRE

R0 = 0x00000000
R1 = 0x80000004
flags = nzcv
MOVS R0, R1, LSL #1

POST

R0 = 0x00000008
R1 = 0x80000004
flags = nzCv

Below figure illustrates a Logical Shift Left by one.



Here, **b31** is shifted to **C** flag, **b30** is shifted to **b31** and so on. **b0** is cleared to zero.
C flag holds the status of bit **(32-y)** of original value, where **y** is shift amount.

DATA PROCESSING INSTRUCTIONS contd...

Problems:

1. ***Multiply the contents of R5 register by 16 and transfer the result to R7 register.***

PRE

R5 = 0x00000005

R7 = 0x00000008

MOV R7, R5,LSL #4

POST

R5 = 0x00000005

R7 = 0x00000050

DATA PROCESSING INSTRUCTIONS contd...

Problems:

2. ***Check whether the data held in R0 register is positive or negative. If positive, then store 10H in R1 register. Otherwise, store 20H in R1 register.***

PRE

R0 = 0xC4005612

R1 = 0x00000000

R2 = 0x00000001

flags = nzc

```
MOVS R2, R0,LSL #1
MOVCS R1,#0X20     ;R0 contains negative number
MOVCC R1,#0X10     ;R0 contains positive number
```

POST

R0 = 0xC4005612

R1 = 0x00000020

R2 = 0X8800AC24

flags = NzC

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter operation: LSR – Logical Shift Right

Example:

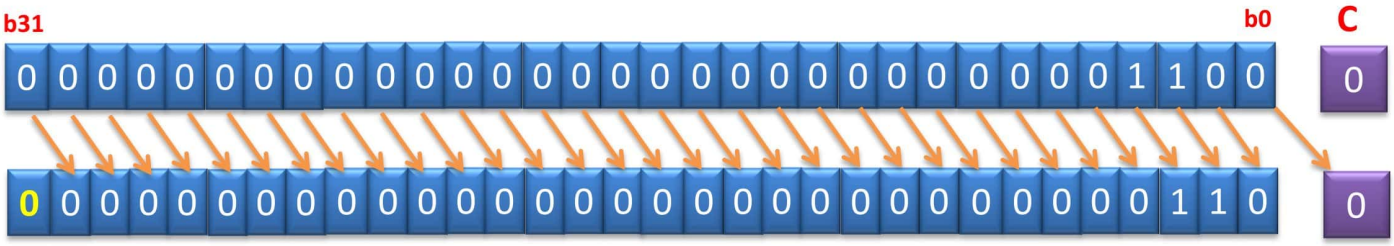
PRE

R0 = 0x00000000
R1 = 0x0000000C
flags = nzcv
MOVS R0, R1, LSR #1

POST

R0 = 0x00000006
R1 = 0x0000000C
flags = nzcv

Below figure illustrates a Logical Shift Right by one.



Here, **b0** is shifted to **C** flag, **b1** is shifted to **b0** and so on. **b31** is cleared to zero. **C** flag holds the status of bit **(y-1)** of original value, where **y** is shift amount.

DATA PROCESSING INSTRUCTIONS contd...

Problems:

1. ***Divide the contents of R1 register by 16 and transfer the result to R2 register.***

PRE

R1 = 0x00000041

R2 = 0x00000000

MOV R2, R1,LSR #4

POST

R1 = 0x00000041

R2 = 0x00000004

DATA PROCESSING INSTRUCTIONS contd...

Problems:

2. Check whether the data held in R0 register is odd or even. If odd, then store 10H in R1 register. Otherwise, store 20H in R1 register.

PRE

R0 = 0x00000029

R1 = 0x00000000

R2 = 0x00000000

flags = nzc

```
MOVS R2, R0,LSR #1
MOVCS R1,#0X10      ;R0 contains odd number
MOVCC R1,#0X20      ;R0 contains even number
```

POST

R0 = 0x00000029

R1 = 0x00000010

R2 = 0X00000014

flags = nzC

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter operation: ASR – Arithmetic Shift Right

Example1:

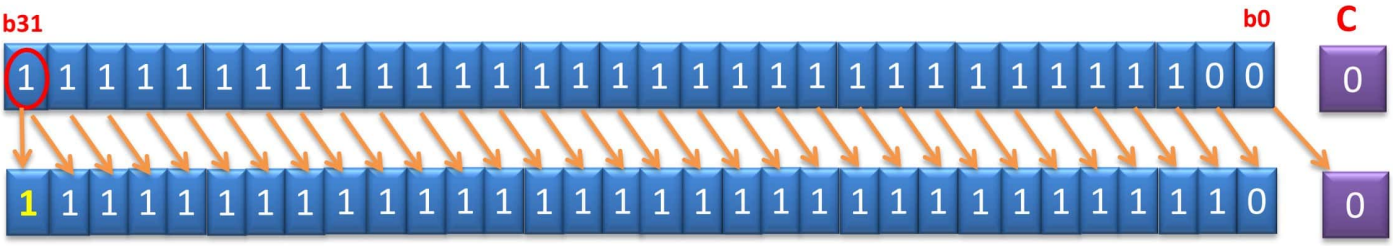
PRE

R0 = 0xFFFFFFFC
R1 = 0x00000000
flags = nzc
MOVS R1, R0, ASR #1

POST

R0 = 0xFFFFFFFC
R1 = 0xFFFFFFFE
flags = Nzc

Below figure illustrates a Arithmetic Shift Right by one.



- Here, **b0** is shifted to **C** flag, **b1** is shifted to **b0** and so on. **b31** contains the sign bit of original value.
- **ASR can be used to divide the signed value held in a register by power of 2.**

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter operation: ASR – Arithmetic Shift Right

Example:

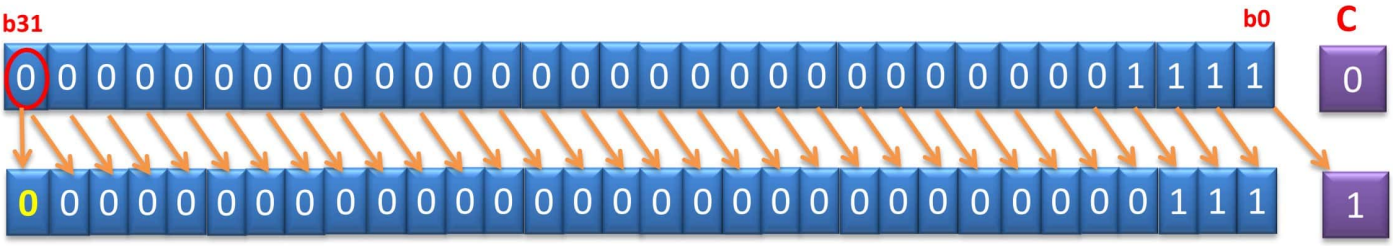
PRE

R0 = 0x0000000F
R1 = 0x00000000
flags = nzc
MOVS R1, R0, ASR #1

POST

R0 = 0x0000000F
R1 = 0x00000007
flags = nzC

Below figure illustrates a Arithmetic Shift Right by one.



ARM Microcontroller and Embedded Systems

Lecture-14

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter operation: ROR – Rotate Right

Example:

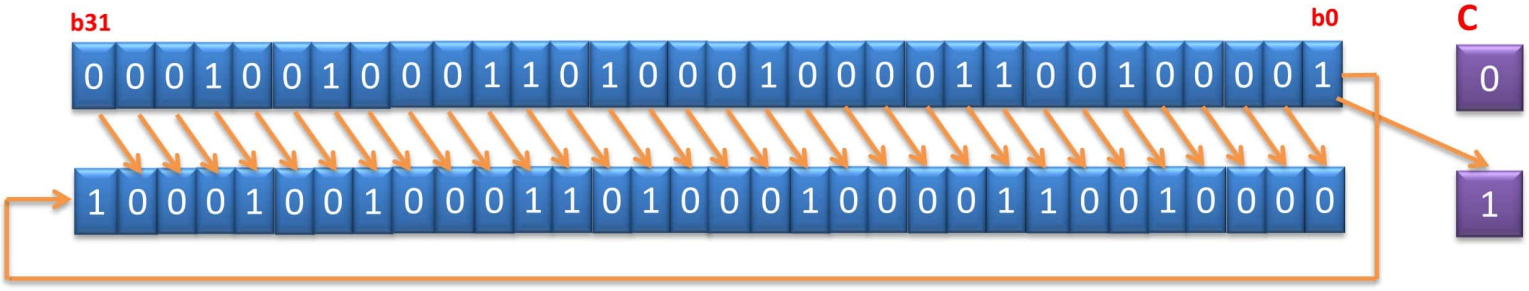
PRE

R0 = 0x12344321
R1 = 0x00000000
flags = nzc
MOVS R1, R0, ROR #1

POST

R0 = 0x12344321
R1 = 0x891A2190
flags = NzC

Below figure illustrates a Rotate Right by one.



- Here, **b0** is shifted to **b31**, **b31** is shifted to **b30** and so on. **b0** is also copied to **C** flag.

DATA PROCESSING INSTRUCTIONS contd...

Problems:

1. ***Swap the lower order 16 bit data with higher order 16 bit data held in R0 register and store the result in R1 register.***

PRE

R0 = 0xABCD1234

R1 = 0x00000000

MOV R1, R0,ROR #16 ; (x>>y)|(x<<(32-y))

POST

R0 = 0xABCD1234

R1 = 0x1234ABCD

DATA PROCESSING INSTRUCTIONS contd...

Problems:

2. ***Rotate left the data held in R0 register by 4 and store the result in R1 register.***

PRE

R0 = 0xABCDABCD

R1 = 0x00000000

MOV R1, R0,ROR #28 ;32-4=28 (ROL by 4 is same as ROR by 28)

POST

R0 = 0xABCDABCD

R1 = 0xBCDABCDA

DATA PROCESSING INSTRUCTIONS contd...

Barrel Shifter operation: RRX – Rotate Right eXtended

Example:

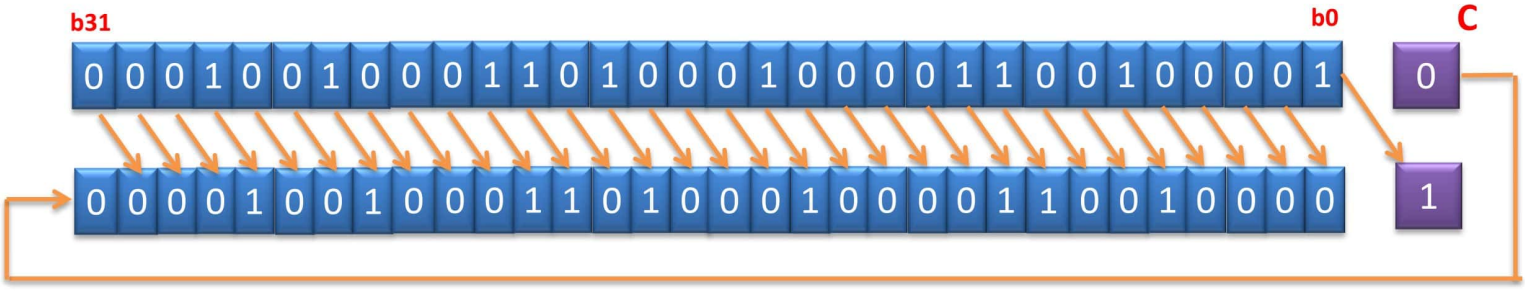
PRE

R0 = 0x12344321
R1 = 0x00000000
flags = nzc
MOVS R1, R0, RRX ;(C<<31) | (x>>1)

POST

R0 = 0x12344321
R1 = 0x091A2190
flags = nzC

Below figure illustrates a Rotate Right eXtended.



- Here, **b0** is shifted to **C** flag, **C** flag content is shifted to **b31, b31** is shifted to **b30** and so on.

DATA PROCESSING INSTRUCTIONS contd...

Problems:

1. ***Check whether the data held in R1 register is odd or even. If odd, then store 10H in R2 register. Otherwise, store 20H in R2 register.***

PRE

R0 = 0x00000000
R1 = 0x00000005
R2 = 0x00000000
flags = nzc

```
MOVS R0, R1,RRX
MOVCS R2,#0X10      ;R1 contains odd number
MOVCC R2,#0X20      ;R1 contains even number
```

POST

R0 = 0x00000002
R1 = 0x00000005
R2 = 0X00000010
flags = nzC

DATA PROCESSING INSTRUCTIONS contd...

Problems:

2. ***Treat R1 and R0 together as a 64-bit register and rotate the bits one place to the right. i.e., b0 of R1 must become b31 of R0, b0 of R0 must become b31 of R1.***

PRE

R0 = 0xFFFF0002
R1 = 0xFFFF0005
R2 = 0x00000000

```
MOVS R2, R0,RRX   ;b0 of R0 in C flag
MOVS R1,R1,RRX    ;C flag in b31 of R1 and b0 of R1 in C flag
MOVS R0,R0,RRX    ;C flag in b31 of R0 and b0 of R0 in C flag
```

POST

R0 = 0xFFFF8001
R1 = 0x7FFF8002
R2 = 0x7FFF8001

R1 Register	R0 Register
1111 1111 1111 1111 0000 0000 0000 0101	1111 1111 1111 1111 0000 0000 0000 0010
0111 1111 1111 1111 1000 0000 0000 0010	1111 1111 1111 1111 1000 0000 0000 0001

DATA PROCESSING INSTRUCTIONS contd...

R1 = 0xFFFF0005 R0 = 0xFFFF0002

1111 1111 1111 1111 0000 0000 0000 0101 1111 1111 1111 1111 0000 0000 0000 0010 0

MOVS R2, R0,RRX ;b0 of R0 in C flag

1111 1111 1111 1111 0000 0000 0000 0010 0

0111 1111 1111 1111 1000 0000 0000 0001 0

R2 = 0x7FFF8001

MOVS R1,R1,RRX ;C flag in b31 of R1 and b0 of R1 in C flag

1111 1111 1111 1111 0000 0000 0000 0101 0

0111 1111 1111 1111 1000 0000 0000 0010 1

R1 = 0X7FFF8002

MOVS R0,R0,RRX ;C flag in b31 of R0 and b0 of R0 in C flag

1111 1111 1111 1111 0000 0000 0000 0010 1

1111 1111 1111 1111 1000 0000 0000 0001 0

R0 = 0XFFFF8001

ARM Microcontroller and Embedded Systems

Lecture-15

DATA PROCESSING INSTRUCTIONS contd...

ARITHMETIC INSTRUCTIONS

- These instructions are used for addition and subtraction of 32-bit signed and unsigned values.
- **Syntax:**

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

where,

Rn is the source register1

N can be an immediate data, source register2 ***Rm*** or result of barrel shifter operation

Rd is the destination register

DATA PROCESSING INSTRUCTIONS contd...

ARITHMETIC INSTRUCTIONS

- The Arithmetic instructions supported are as follows:

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

Note:

- **SUBS** instruction is useful for decrementing loop counters.
- **ADC, SBC, RSC** instructions are used while dealing with multiword numbers.

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

1. ***Add the contents of R1 register with R2 and store the result in R3.***

PRE

R1 = 0x7FFFFFFF

R2 = 0x00000001

R3 = 0x00000000

flags = **nzcv**

ADDS R3, R1,R2

POST

R1 = 0x7FFFFFFF

R2 = 0x00000001

R3 = 0X80000000

flags = **NzcV**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

2. Add with carry the contents of R1 register and R2 and store the result in R3.

PRE

R1 = 0x7FFFFFFF

R2 = 0x00000000

R3 = 0x00000000

flags = **nzCv**

ADCS R3, R1,R2

POST

R1 = 0x7FFFFFFF

R2 = 0x00000000

R3 = 0X80000000

flags = **NzcV**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

3. Multiply the contents of R1 register by 3 without using MUL and store the result in R0.

PRE

R0 = 0x00000000

R1 = 0x00000005

ADD R0, R1,R1,LSL #1

POST

R0 = 0x0000000F

R1 = 0x00000005

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

4. Subtract the value stored in R2 register from the value stored in R1 and store the result in R0.

(a)

PRE

R0 = 0x00000002

R1 = 0x0000000A

R2 = 0x00000006

flags = **nzcv**

SUBS R0, R1,R2 ;R0 = R1 – R2

POST

R0 = 0x00000004

R1 = 0x0000000A

R2 = 0X00000006

flags = **nzCv**

Note: Complement of carry is stored in C flag.

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

4. Subtract the value stored in R2 register from the value stored in R1 and store the result in R0.

(b)

PRE

R0 = 0x00000002

R1 = 0x00000006

R2 = 0x0000000A

flags = **nzcv**

SUBS R0, R1,R2 ;R0 = R1 – R2

POST

R0 = 0xFFFFFFFC

R1 = 0x00000006

R2 = 0X0000000A

flags = **Nzcv**

Note: Complement of carry is stored in C flag.

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

4. Subtract the value stored in R2 register from the value stored in R1 and store the result in R0.

(c)

PRE

R0 = 0x00000002

R1 = 0x80000000

R2 = 0x00000001

flags = nzcv

SUBS R0, R1,R2 ;R0 = R1 – R2

POST

R0 = 0x7FFFFFFF

R1 = 0x80000000

R2 = 0X00000001

flags = nzCV

Note: Complement of carry is stored in C flag.

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

5. ***Subtract the contents of R0 register from R1 along with carry and store the result in R2.***

PRE

R0 = 0x00000002

R1 = 0x00000003

R2 = 0x00000000

flags = **nzcv**

SBCS R2, R1,R0 ;R2 = R1 – R0 - !C

POST

R0 = 0x00000002

R1 = 0x00000003

R2 = 0X00000000

flags = **nZCv**

Note: Complement of carry is stored in C flag.

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

6. Negate the value stored in R0 register and store the result in R0.

PRE

R0 = 0x00000004

```
MVN R0,R0        ; RSB R0,#0
ADD R0,#1        ; RSBS R0,#0
                 Flags: Nzcv
```

POST

R0 = 0xFFFFFFFC

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

7. ***Subtract the contents of R1 register from R0 along with carry RSC instruction and store the result in R2.***

PRE

R0 = 0x00000002

R1 = 0x00000003

R2 = 0x00000000

flags = **nzcv**

RSCS R2, R1,R0 ;R2 = R0 – R1 - !C

POST

R0 = 0x00000002

R1 = 0x00000003

R2 = 0XFFFFFFFE

flags = **Nzcv**

Note: Complement of carry is stored in C flag.

DATA PROCESSING INSTRUCTIONS contd...

Problems on Arithmetic Instructions:

8. Negate a 64 bit value held in R0 and R1 and store the result in R3 and R4.

PRE

R0 = 0x00000002

R1 = 0x00000000

RSBS R3,R0,#0

RSC R4,R1,#0

POST

R0 = 0x00000002

R1 = 0x00000000

R3 = 0xFFFFFFFE

R4 = 0xFFFFFFFF

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010

1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101

+ 1

1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110

ARM Microcontroller and Embedded Systems

Lecture-16

DATA PROCESSING INSTRUCTIONS contd...

LOGICAL INSTRUCTIONS

- These instructions are used to perform bitwise logical operations on the two source operands.
- **Syntax:**

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

where,

Rn is the source register1

N can be an immediate data, source register2 ***Rm*** or result of barrel shifter operation

Rd is the destination register

DATA PROCESSING INSTRUCTIONS contd...

LOGICAL INSTRUCTIONS

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Note:

- **BIC is useful when clearing status bits and is frequently used to change interrupt masks in cpsr.**
- **Logical instructions affect N, Z and C flags only if S suffix is present in the mnemonic. V flag is not affected.**
- **Logical instructions can use barrel-shifted second operand in the same way as the arithmetic instructions.**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Logical Instructions:

1. ***Perform logical AND operation between R1 and R2 and store the result in R0.***

PRE

R0 = 0x00000000

R1 = 0xABCD1234

R2 = 0x1234ABCD

flags = **nzcv**

ANDS R0, R1,R2

POST

R0 = 0x02040204

R1 = 0xABCD1234

R2 = 0X1234ABCD

flags = **nzcv**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Logical Instructions:

2. Clear bits at odd positions without affecting bits at even positions in R2 register and store the result in R0.

PRE

R0 = 0x00000000

R1 = 0xAAAAAAAA

R2 = 0xFFFFFFFF

BIC R0, R2,R1 ;R1 = 0x55555555

;AND R0,R2,R1

POST

R0 = 0x55555555

R1 = 0xAAAAAAAA

R2 = 0XFFFFFFFF

1111 1111 1111 1111 1111 1111 1111 1111

1010 1010 1010 1010 1010 1010 1010 1010

0101 0101 0101 0101 0101 0101 0101 0101

DATA PROCESSING INSTRUCTIONS contd...

Problems on Logical Instructions:

3. Perform A ⊕B . (Assume R1 contains A and R2 contains B)

A ⊕ B = $\overline{A} . B + A . \overline{B}$

(a)
```
MVN   R3,R1   ; complement of A
AND R3,R2     ; complement of A is ANDed with B
MVN R4,R2     ; complement of B
AND R4,R1     ;complement of B is ANDed with A
ORR R3,R4
```

(b)
```
BIC R3,R2,R1   ;complement of A  is ANDed with B
BIC R4,R1,R2   ;complement of B is ANDed with A
ORR R3,R4
```

(c)
```
EOR R3,R1,R2
```

DATA PROCESSING INSTRUCTIONS contd...

COMPARISON INSTRUCTIONS

- These instructions are used to compare or test a register with a 32-bit value.
- They update cpsr condition flag bits according to the result, but do not affect the registers used.
- The flag status can then be used to change the program flow by using conditional execution.
- The S suffix need not be applied for comparison instructions to update the flags.

DATA PROCESSING INSTRUCTIONS contd...

COMPARISON INSTRUCTIONS

- **Syntax:**

  <instruction>{<cond>} Rn, N

where,

Rn is the source register1

N can be an immediate data, source register2 ***Rm*** or result of barrel shifter operation

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

DATA PROCESSING INSTRUCTIONS contd...

Note:

- CMP and CMN instructions affect N, Z, C, V flags.
- TEQ and TST instructions affect N, Z and C flags.
- CMP is a subtraction instruction, SUBS with result discarded.
- CMN is an addition instruction, ADDS with result discarded.
- TST is a Logical instruction, ANDS with result discarded.
- TEQ is a Logical instruction, EORS with result discarded.

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

1. *Check whether the number in R0 register is positive or negative. If positive then store 10H in R2. Otherwise store 20H in R2.*

```
LDR R0,=0XF3425618
LDR R1,=0X80000000
TST R0,R1
MOVEQ R2,#0X10      ; Number stored in R0 is positive
MOVNE R2,#0X20      ; Number stored in R0 is negative
```

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

2. Check whether the number in R0 register is odd or even. If odd then store 10H in R2. Otherwise store 20H in R2.

```
MOV R0,#0X10
TST R0,#1
MOVEQ R2,#0X20      ;Number stored in R0 is even
MOVNE R2,#0X10      ;Number stored in R0 is odd
```

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

3. Check whether the contents of R1 and R2 registers are same or not. If same, then store 10H in R3. Otherwise store 20H in R3.

(a)

```
MOV R1,#2
MOV R2,#2
TEQ R1,R2
MOVEQ R3,#0X10      ;Contents of R1 and R2 are same
MOVNE R3,#0X20      ;Contents of R1 and R2 are different
```

(b)

```
CMP R1,R2
MOVEQ R3,#0X10
MOVNE R3,#0X20
```

(c)

```
RSB R3,R2,#0
ADDS R3,R1
MOVEQ R3,0X10
MOVNE R3,#0X20
```

ARM Microcontroller and Embedded Systems

Lecture-17

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(a) PRE

R0 = 0xF0000000

R1 = 0x00000001

flags = nzcv

CMN R0, R1

POST

R0 = 0xF0000000

R1 = 0x00000001

flags = Nzcv

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(b) PRE

R0 = 0x7FFFFFFF

R1 = 0x00000001

flags = nzcv

CMN R0, R1

POST

R0 = 0x7FFFFFFF

R1 = 0x00000001

flags = NzcV

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(c) PRE

R0 = 0xFFFFFFFF

R1 = 0x00000001

flags = **nzcv**

CMN R0, R1

POST

R0 = 0xFFFFFFFF

R1 = 0x00000001

flags = **nZCv**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(d) PRE

R0 = 0x00000004

R1 = 0x00000001

flags = **nzcv**

TST R0, R1

POST

R0 = 0x00000004

R1 = 0x00000001

flags = **nZcv**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(e) PRE

R0 = 0x00000004

R1 = 0x00000001

flags = nzcv

TST R0, R1,LSR #1

POST

R0 = 0x00000004

R1 = 0x00000001

flags = nZCv

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(f) PRE

R0 = 0x00000005

R1 = 0x00000005

flags = **nzcv**

TEQ R0, R1

POST

R0 = 0x00000005

R1 = 0x00000005

flags = **nZcv**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(g) PRE

R0 = 0x90125364

R1 = 0x03428152

flags = **nzcv**

TEQ R0, R1

POST

R0 = 0x90125364

R1 = 0x03428152

flags = **Nzcv**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(h) PRE

R0 = 0x00000004

R1 = 0x00000005

flags = nzcv

TEQ R0, R1,LSR #1

POST

R0 = 0x00000004

R1 = 0x00000005

flags = nzCv

DATA PROCESSING INSTRUCTIONS contd...

Problems on Comparison Instructions:

4. Give the status of the flags after execution of the following instructions.

(i) PRE

R0 = 0x00000004

R1 = 0x00000005

flags = **nzcv**

TEQ R0, R1,ROR #1

POST

R0 = 0x00000004

R1 = 0x00000005

flags = **NzCv**

DATA PROCESSING INSTRUCTIONS contd...

MULTIPLY INSTRUCTIONS

- The multiply instructions multiply the contents of two registers and depending upon the instruction, accumulate the result in another register.
- The long multiply instructions accumulate the result in two registers to represent a 64-bit value.
- **Syntax:**

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

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

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

Note:

- **Rd** and **Rm** registers cannot be same.
- Least significant 32-bits of the result is placed in **Rd**.
- Only **N** and **Z** flags are affected if S suffix is used. **C** and **V** flags are not affected in V5 and later.

DATA PROCESSING INSTRUCTIONS contd...

Problems on Multiply Instructions:

1. ***Multiply the contents of R1 and R2 and place the result in R0 register.***

PRE

R0 = 0x00000000

R1 = 0x00000005

R2 = 0x00000004

MUL R0, R1,R2 ; R0 = R1 * R2

POST

R0 = 0x00000014

R1 = 0x00000005

R2 = 0X00000004

DATA PROCESSING INSTRUCTIONS contd...

Problems on Multiply Instructions:

2. Multiply the contents of R1 and R2, add the result with R3 and place the final result in R4 register.

PRE

R1 = 0x00000004

R2 = 0x00000005

R3 = 0x00000007

R4 = 0x00000000

MLA R4, R1,R2,R3 ; R4 = R1 * R2 + R3

POST

R1 = 0x00000004

R2 = 0x00000005

R3 = 0X00000007

R4 = 0x0000001B

DATA PROCESSING INSTRUCTIONS contd...

Problems on Multiply Instructions:

3. Compute the sum of squares of first n natural numbers.

```
        MOV R0,#3
        MOV R1,#0
LOOP    MLA R1,R0,R0,R1
        SUBS R0,#1
        BNE LOOP
```

DATA PROCESSING INSTRUCTIONS contd...

Problems on Multiply Instructions:

4. Give the status of the flags after execution of the following instructions.

(a) PRE

R1 = 0x00000003

R2 = 0x00000000

flags = **nzcv**

MULNES R1, R2,R1

POST

R1 = 0x00000000

R2 = 0x00000000

flags = **nZcv**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Multiply Instructions:

4. Give the status of the flags after execution of the following instructions.

(b) PRE

R1 = 0x00000001

R2 = 0xFFFF0000

flags = **nzcv**

MULNES R1, R2,R1

POST

R1 = 0xFFFF0000

R2 = 0xFFFF0000

flags = **Nzcv**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Multiply Instructions:

4. Give the status of the flags after execution of the following instructions.

(C) PRE

R1 = 0x00000003

R2 = 0xFFFFFFF0

flags = **nzcv**

MULNES R1, R2,R1

POST

R1 = 0xFFFFFFD0

R2 = 0xFFFFFFF0

flags = **Nzcv**

DATA PROCESSING INSTRUCTIONS contd...

Problems on Multiply Instructions:

4. Give the status of the flags after execution of the following instructions.

(d) PRE

R1 = 0x00000004

R2 = 0x00000000

R3 = 0x00000000

flags = **nzcv**

MLAVCS R3, R1,R2,R3

POST

R1 = 0x00000004

R2 = 0x00000000

R3 = 0X00000000

flags = **nZcv**

DATA PROCESSING INSTRUCTIONS contd...

LONG MULTIPLY INSTRUCTIONS

- The long multiply instructions **(SMLAL, SMULL, UMLAL, UMULL)** produce a 64-bit result which is too large to fit a single 32-bit register.

- So, the 64-bit result is placed in two registers labeled ***RdLo*** and ***RdHi***. ***RdLo*** holds the lower 32 bits of the 64-bit result, and ***RdHi*** holds the higher 32 bits of the 64-bit result.

DATA PROCESSING INSTRUCTIONS contd...

LONG MULTIPLY INSTRUCTIONS

- **Syntax for Long Multiply Instructions:**

  <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	$[RdHi, RdLo] = Rm * Rs$

DATA PROCESSING INSTRUCTIONS contd...

LONG MULTIPLY INSTRUCTIONS

UMULL

UMULL{<cond>}{S} RdLo, RdHi, Rm, Rs

- UMULL interprets values in Rm and Rs as unsigned integers.
- It multiplies the integers in **Rm** and **Rs** and places the least significant 32 bits of the result in **RdLo** and most significant 32 bits of the result in **RdHi**

- **Example:**

```
PRE
        R0 = 0xF0000002
        R1 = 0x00000002
        R2 = 0x00000000
        R3 = 0x00000000
        UMULL R2, R3,R0,R1    ; [R3,R2] = R0 * R1
POST
        R0 = 0xF0000002
        R1 = 0x00000002
        R2 = 0xE0000004   ;RdLo
        R3 = 0x00000001   ;RdHi
```

DATA PROCESSING INSTRUCTIONS contd...

LONG MULTIPLY INSTRUCTIONS

SMULL

SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs

- SMULL interprets values in Rm and Rs as two's complement signed integers.
- It multiplies the integers in **Rm** and **Rs** and places the least significant 32 bits of the result in **RdLo** and most significant 32 bits of the result in **RdHi**
- **Example:**

```
PRE
        R0 = 0xFFFFFFFF
        R1 = 0x00000002
        R2 = 0x00000000
        R3 = 0x00000000
        flags = nzcv
        SMULLS R2, R3,R0,R1    ; [R3,R2] = R0 * R1
POST
        R0 = 0xFFFFFFFF
        R1 = 0x00000002
        R2 = 0xFFFFFFFE    ;RdLo
        R3 = 0xFFFFFFFF    ;RdHi
        flags = Nzcv
```

DATA PROCESSING INSTRUCTIONS contd...

LONG MULTIPLY INSTRUCTIONS

UMLAL

UMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs

- UMLAL interprets values in Rm and Rs as unsigned integers.
- It multiplies the integers in **Rm** and **Rs** and adds the 64-bit result to the 64-bit unsigned integer held in **RdHi** and **RdLo**. Then, it places the least significant 32 bits of the final result in **RdLo** and most significant 32 bits of the final result in **RdHi.**
- **Example:**

```
PRE
        R0 = 0x00000002
        R1 = 0x00000008
        R2 = 0xF0000000
        R3 = 0xF0000000
        flags = nzcv
        UMLALS R2, R3,R0,R1    ; [R3,R2] = R0 * R1 + [R3,R2]
POST
        R0 = 0x00000002
        R1 = 0x00000008
        R2 = 0xF0000010     ;RdLo
        R3 = 0xF0000000     ;RdHi
        flags = Nzcv
```

DATA PROCESSING INSTRUCTIONS contd...

LONG MULTIPLY INSTRUCTIONS

SMLAL

SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs

- SMLAL interprets values in Rm and Rs as two's complement signed integers.
- It multiplies the integers in **Rm** and **Rs** and adds the 64-bit result to the 64-bit unsigned integer held in **RdHi** and **RdLo**. Then, it places the least significant 32 bits of the final result in **RdLo** and most significant 32 bits of the final result in **RdHi.**
- **Example:**

```
PRE
        R0 = 0xFFFFFFFE
        R1 = 0x00000002
        R2 = 0xFFFFFFFE
        R3 = 0xFFFFFFFF
        flags = nzcv
        SMLALS R2, R3,R0,R1    ; [R3,R2] = R0 * R1 + [R3,R2]
POST
        R0 = 0xFFFFFFFE
        R1 = 0x00000002
        R2 = 0xFFFFFFFA      ;RdLo
        R3 = 0xFFFFFFFF      ;RdHi
        flags = Nzcv
```

ARM Microcontroller and Embedded Systems

Lecture-18

BRANCH INSTRUCTIONS

- A branch instruction is used to change the flow of execution or to call a routine.
- Branch instructions allows programs to have subroutines, *if-then-else* structures, and loops.
- The change of execution flow forces the **program counter *pc*** to point to a new address.
- The ARMv5E instruction set includes four different branch instructions: **B, BL, BX and BLX**

BRANCH INSTRUCTIONS contd...

- **Syntax:**

  B{<cond>} label

  BL{<cond>} label

  BX{<cond>} Rm

  BLX{<cond>} Rm

  BLX label

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

BRANCH INSTRUCTIONS contd...

- The branch instruction, **B label** causes a branch to label.
- The **BL label** instruction copies the address of next instruction after **BL** (return address) into **lr (r14)** and causes a branch to label.
- The address ***label*** is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction.
- ***T*** refers to the Thumb bit in the *cpsr*.
- If bit0 of **Rm** is set, the instructions **BX** & **BLX** sets T flag in cpsr and ARM switches to Thumb state i.e., the code at the destination is interpreted as Thumb code.
- **BX** instruction causes a branch to the address held in **Rm** and changes instruction set to Thumb if bit0 of **Rm** is set.

BRANCH INSTRUCTIONS contd...

- ***The BLX instruction has two alternative forms:***
  - **BLX label:** An unconditional branch with link to label address. This instruction copies the address of next instruction after **BLX** into **lr** and causes a branch to label. It also changes the state to Thumb.

  - **BLX {<cond>} Rm**: A conditional branch with link to the address held in a register. This instruction copies address of next instruction after **BLX** into **lr** and causes branch to the address held in Rm. It also changes the state to thumb if bit0 of **Rm** is set.

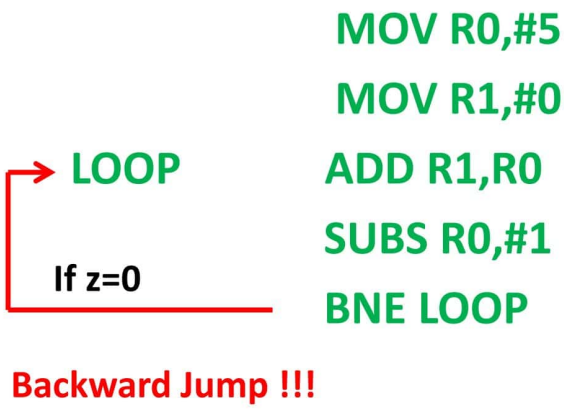
- The **BLX** instruction
  - Copies the address of next instruction into ***lr***.
  - Causes branch to label, or to the address held in ***Rm***
  - Changes instruction set to Thumb if either
    - bit0 of ***Rm*** is set
    - **BLX label** form is used

Note: BX & BLX causes a switch between ARM and Thumb state while branching to a subroutine.

BRANCH INSTRUCTIONS contd...

- Backward jumps and Forward jumps are possible using branch instructions.
  - In **Backward jump**, the label appears before branch instruction.
  - In **Forward jump**, the label appears after branch instruction.

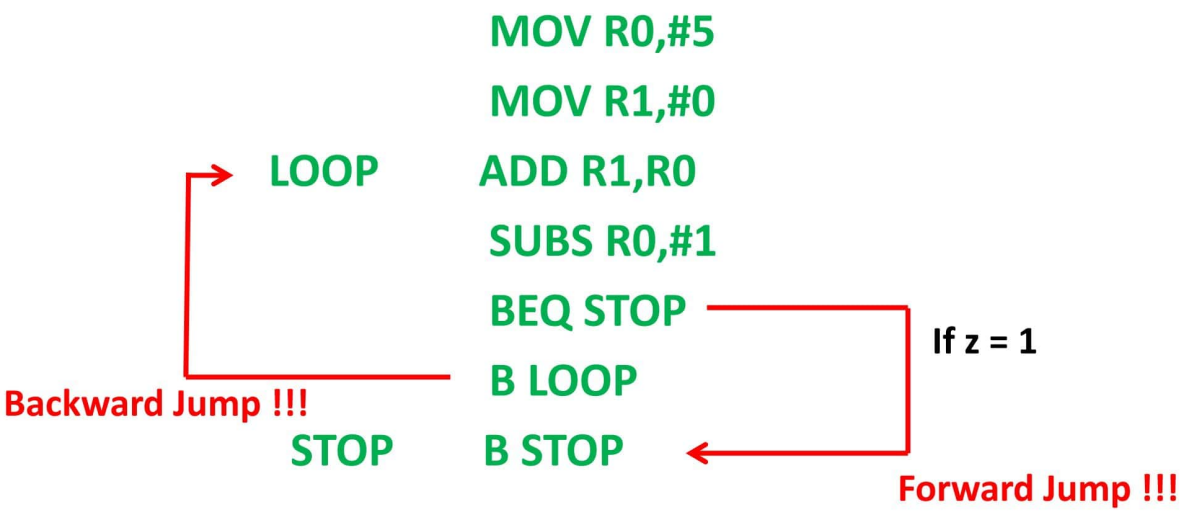
- **Example to compute sum of first *n* natural numbers.**



Backward Jump !!!

BRANCH INSTRUCTIONS contd...

- Example to compute sum of first *n* natural numbers.



BRANCH INSTRUCTIONS contd...

- Develop an ALP to compare two numbers held in R0 and R1. Design subroutine to compare two numbers and store 10H in R2 if they are equal and store 20H if they are not equal.

```
                AREA EX1,CODE,READONLY
ENTRY
                MOV R0,#5
                MOV R1,#10
                BL COMPARE
STOP            B STOP

COMPARE   CMP R0,R1
                MOVEQ R2,#0X10
                MOVNE R2,0X20
                BX LR
                END
```