Computer Organization and Architecture

Subject Code: B18CS4040

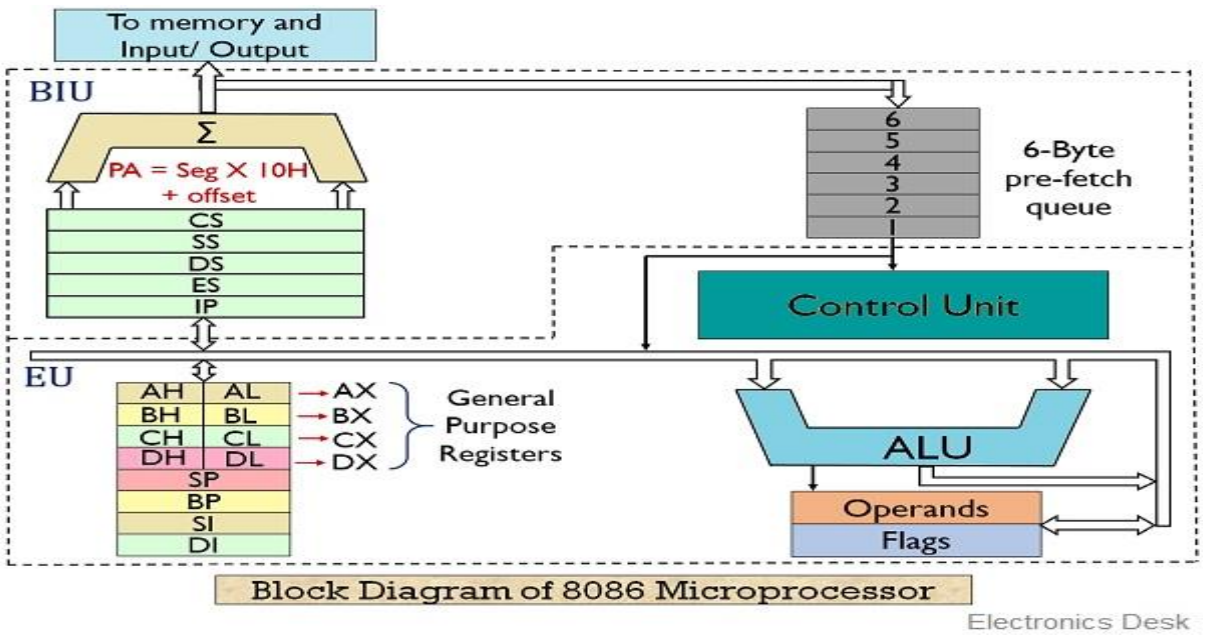
UNIT-1

What is a Microprocessor?

- Computer's Central Processing Unit (CPU) built on a **single Integrated Circuit (IC)** is called a **microprocessor**.
- A microprocessor consists of an ALU, control unit and register array. Where **ALU** performs arithmetic and logical operations on the data received from an input device or memory.
- Control unit controls the instructions and flow of data within the computer. And, **register array** consists of registers identified by letters like B, C, D, E, H, L, and accumulator.

Working of Microprocessor:

- The microprocessor follows a sequence to execute the instruction: Fetch, Decode, and then Execute.
- Initially, the instructions are stored in the storage memory of the Computer in sequential order.
- The microprocessor fetches those instructions from the stored area (memory), then decodes it and executes those instructions till STOP instruction is met.
- Then, it sends the result in binary form to the output port.
- Between these processes, the register stores the temporary data and ALU (Arithmetic and Logic Unit) performs the computing functions.

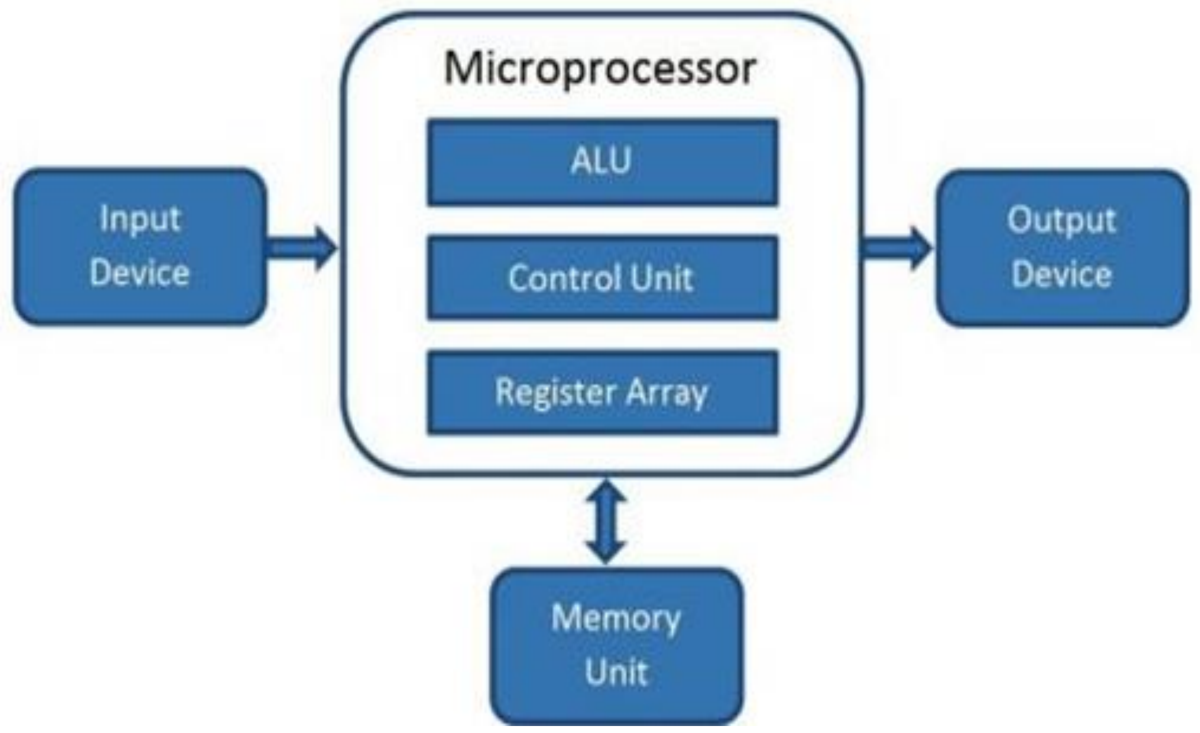


Block Diagram of 8086 Microprocessor

Microcomputer:

- A digital computer with one microprocessor which acts as a CPU is called microcomputer.
- It is a programmable, multipurpose, clock -driven, register-based electronic device that reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions and provides results as output.

Block Diagram of a Microcomputer:

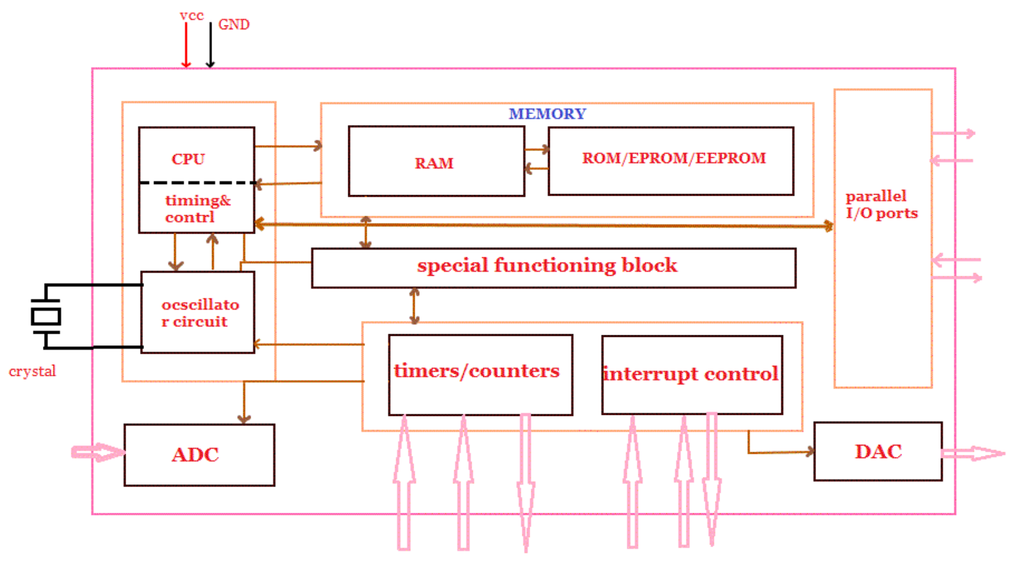


What is a Microcontroller?

- A microcontroller is a single chip microcomputer made through VLSI fabrication.
- A microcontroller also called an embedded controller because the microcontroller and its support circuits are often built into, or embedded in, the devices they control.
- A microcontroller is available in different word lengths like microprocessors (4bit,8bit,16bit,32bit,64bit and 128-bit microcontrollers are available today).
- A microcontroller basically contains one or more following components:
- Central processing unit(CPU)
- Random Access Memory)(RAM)
- Read Only Memory(ROM)
- Input/output ports
- Timers and Counters

- Interrupt Controls
- Analog to digital converters
- Digital analog converters
- Serial interfacing ports
- Oscillatory circuits

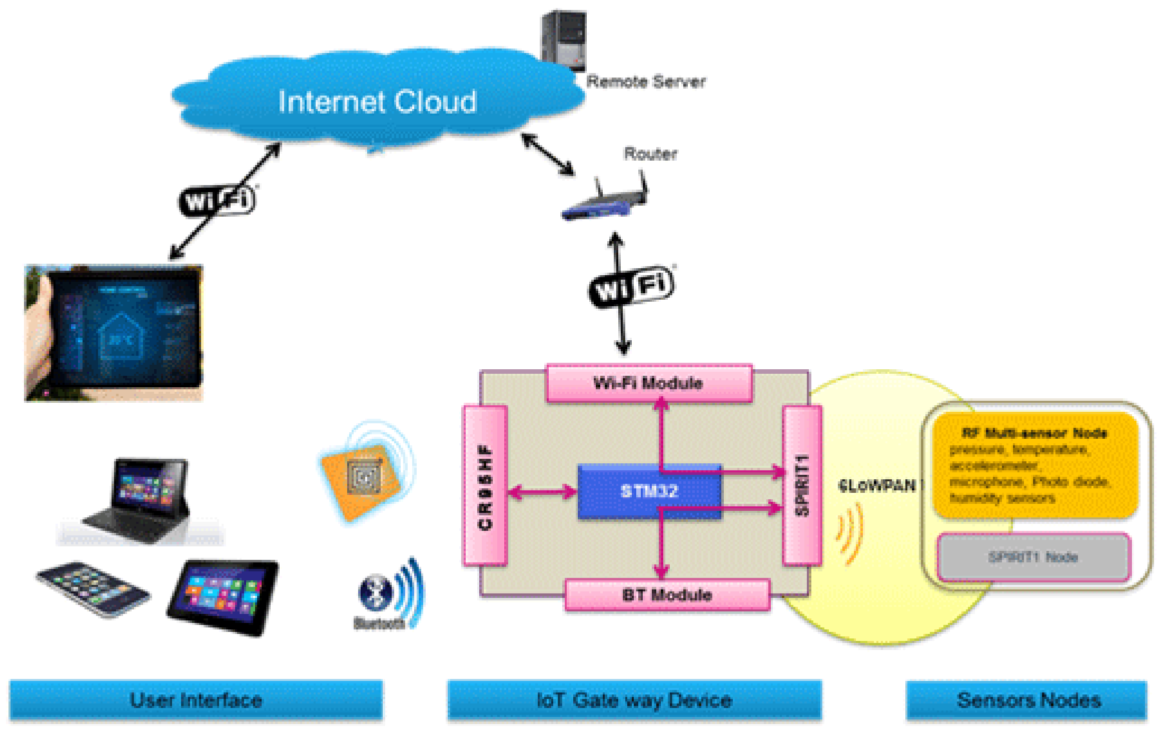
Microcontroller structure:



Internet of Things:

- Internet of things is a network of devices which can sense, accumulate and transfer data over the internet without human intervention.
- The Internet of Things (IoT) is about interconnecting embedded systems, bringing together two evolving technologies: wireless connectivity and sensors.
- These connected embedded systems are independent microcontroller-based computers that use sensors to collect data.
- These IoT systems are networked together usually by a wireless protocol such as WiFi, Bluetooth, 802.11.4, or a custom communication system.
- The networking protocol is selected based on the distribution of nodes and the amount of data to be collected.
- This data is sent over the network to the main hub or computer.

- This main computer collects and analyzes the data, storing it in memory and even making system decisions based on the results of the analysis.



ARM Processor:

- Advanced RISC Machines(ARM)Limited has designed a family of RISC-style processors.
- ARM licenses these designs to other companies for chip fabrication, together with software tools for system development and simulation.
- The main use for ARM processors is in low-power and low-cost embedded applications such as mobile telephones, communication modems, and automotive engine management systems.
- All ARM processors share a basic machine instruction set.
- The ISA version used here is called version 4 by ARM

ARM Characteristics:

- ARM word length is 32 bits, memory is byte-addressable using 32-bit addresses, and the processor registers are 32 bits long.
- Three operand lengths are used in moving data between the memory and the processor registers: byte (8 bits), half word (16 bits), and word (32bits).
- Both little-endian and big-endian memory addressing schemes are supported.
- In most respects, the ARM ISA reflects a RISC-style architecture, but it has some CISC-style features.

RISC-style Aspects

- All instructions have a fixed length of 32 bits.
- Only Load and Store instructions access memory.
- All arithmetic and logic instructions operate on operands in processor registers.

CISC-style Aspects

- Auto increment, Auto decrement, and PC-relative addressing modes are provided.
- Condition codes (N, Z, V, and C) are used for branching and for conditional execution of instructions.
- Multiple registers can be loaded from a block of consecutive memory words, or stored in a block, using a single instruction.

Unusual Aspects of the ARM Architecture:

The ARM architecture has a number of features not generally found in modern processors.

Conditional Execution of Instructions

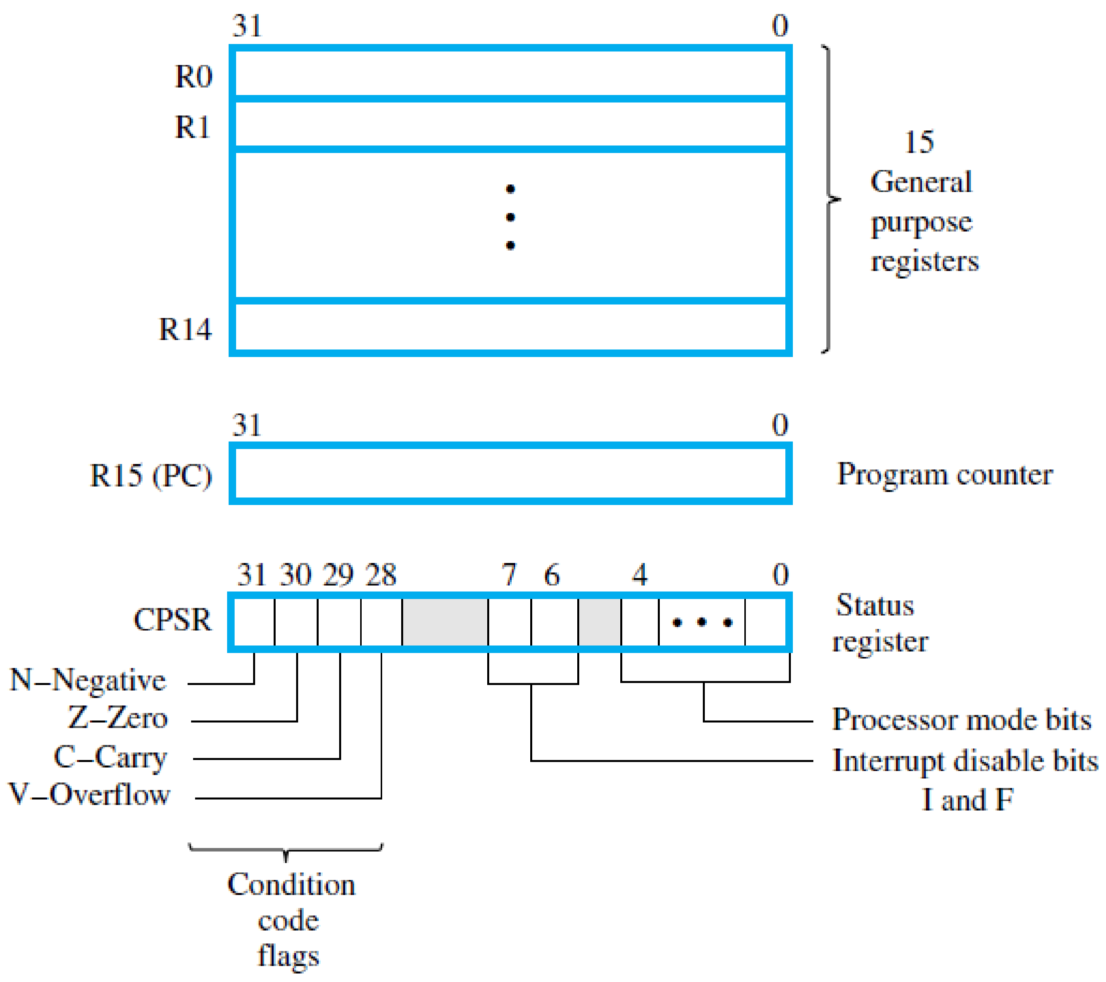
- An unusual feature of ARM processors is that all instructions are conditionally executed. An instruction is executed only if the current values of the condition code flags satisfy the condition specified in a 4-bit field of the instruction. Otherwise, the processor proceeds to the next instruction. One of the possible conditions specifies that the instruction is always executed.

No Shift or Divide Instructions

- Shift instructions are not provided explicitly. However, an immediate value or one of the register operands in arithmetic, logic, and move instructions can be shifted by a prescribed amount before being used in an operation. This feature can also be used to implement shift instructions implicitly.
- There are a number of different multiply instructions, with many of the variations intended for use in signal-processing applications. But there are no hardware Divide instructions. Division must be implemented in software

Register Structure:

- There are sixteen 32-bit processor registers for user application programs, labeled R0 through R15.
- They comprise fifteen general-purpose registers (R0 through R14) and the Program Counter (PC), which is register R15.
- The general purpose registers can hold either memory addresses or data operands.
- Registers R13 and R14 have dedicated uses related to the management of the processor stack and subroutines.
- The Current Program Status Register (CPSR), or simply the Status register holds the condition code flags (N, Z, C, V), interrupt-disable bits, and processor mode bits.



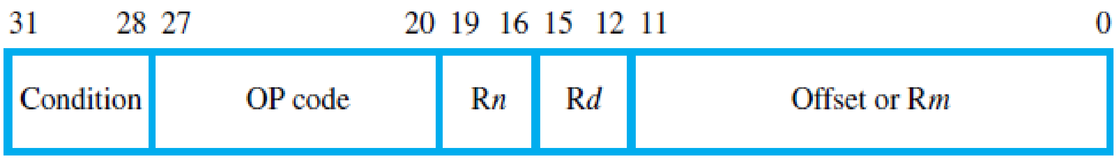
- There are a total of seven processor modes of operation.
- Application programs run in User mode. The other six modes are used to handle I/O device interrupts, processor powerup/reset, software interrupts, and memory access violations.
- We will assume that the processor is in User mode, executing an application program.
- There are a number of additional general-purpose registers called *banked registers.*
- They are duplicates of some of the R0 to R14 registers. Different banked registers are used when the processor switches from User mode into other modes of operation.
- The use of banked registers avoids the need to save and restore some of the User-mode register contents on mode switches.

Addressing Modes:

- Basic Indexed Addressing Mode (Pre indexed)
- Relative Addressing Mode
- Index Modes with Write back
- Register
- Immediate

Basic Indexed Addressing Mode:

- Basic method for addressing memory operands
- Pre-Index Addressing mode: The effective address of the operand is the sum of the contents of a base register, Rn, and a signed offset. In the Pre-indexed addressing mode, the original contents of register R*n are not changed in* the process of generating the effective address of the operand.
- Format for Load and Store instructions is



LDR Rd, [Rn, #offset]

Specifies the offset (expressed as a signed number) in the immediate mode and performs the operation

Rd← [[Rn] + offset]

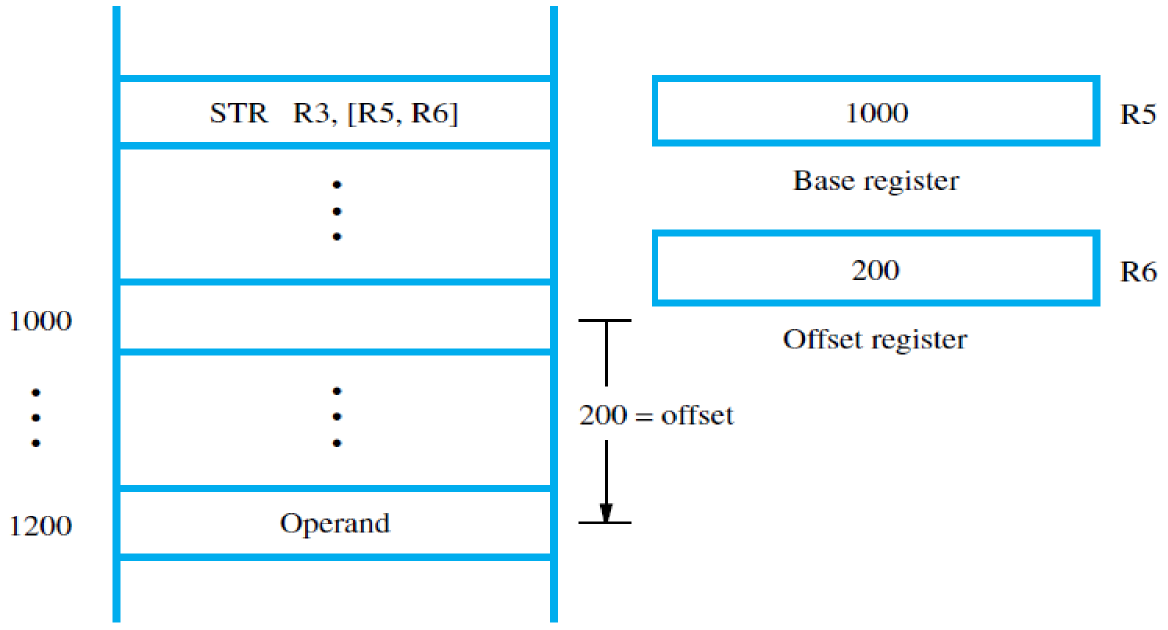
The instruction

LDR Rd, [Rn, Rm]

Performs the operation

Rd← [[Rn] + [Rm]]

Example:



(b) Pre-indexed addressing mode

Relative Addressing Mode:

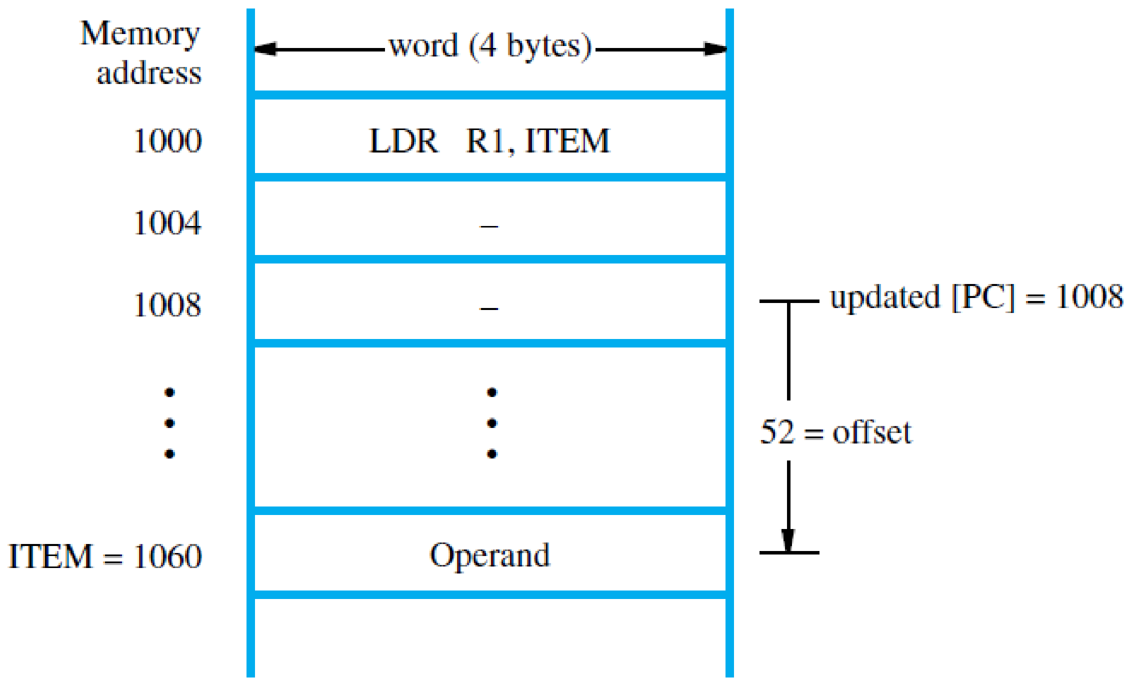
- The Program Counter, PC, may be used as the Base register Rn, along with an immediate offset
- The programmer simply places the desired address label in the operand field to indicate this mode

LDR R1, ITEM

loads the contents of memory location ITEM into register R1

The assembler determines the immediate offset as the difference between the address of the operand and the contents of the updated PC. When the effective address is calculated at instruction execution time, the contents of the PC will have been updated to the address two words (8 bytes) forward from the instruction containing the Relative addressing mode. The reason for this is that the ARM processor will have already fetched the next instruction. This is due to pipelined instruction execution.

Example:



(a) Relative addressing mode

Index Modes with Write back:

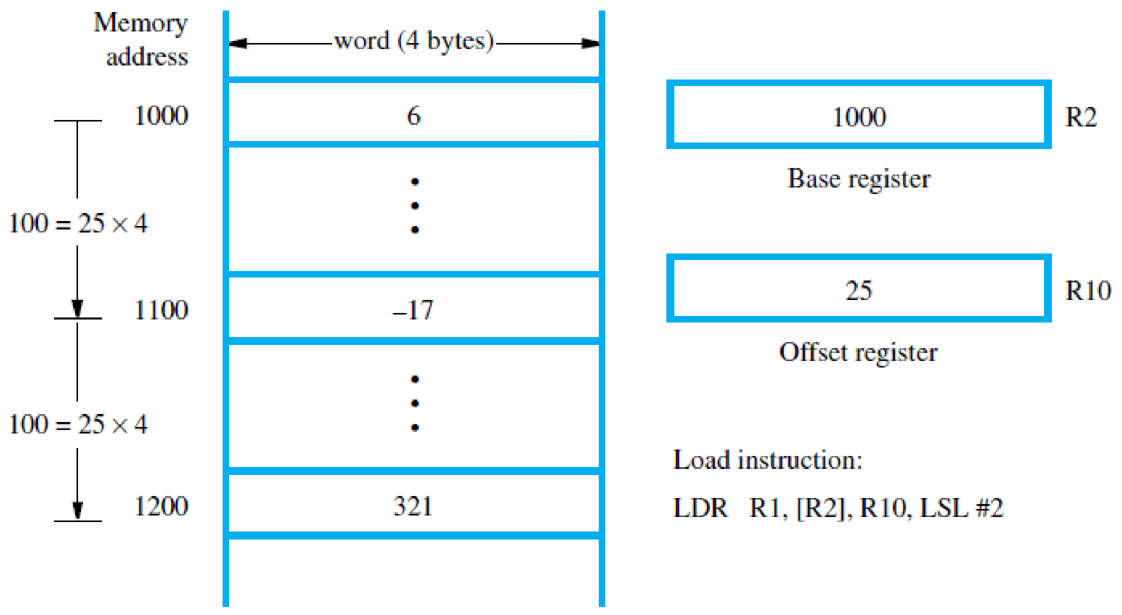
- In the Pre-indexed addressing mode, the original contents of register R*n are not changed in* the process of generating the effective address of the operand.
- There is a variation of this mode, called

Pre-indexed with write back mode—The effective address of the operand is generated in the same way as in the Pre-indexed mode, then the effective address is written back into Rn.

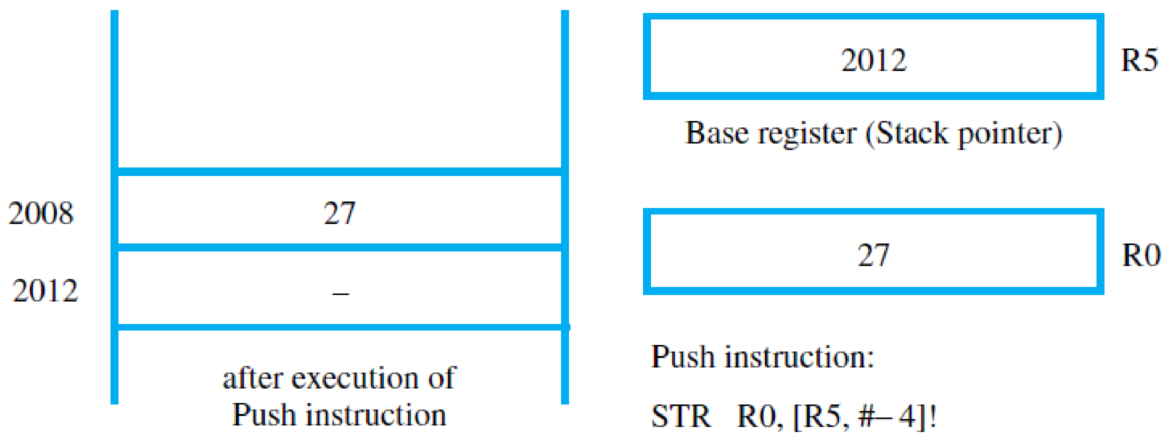
Post-indexed mode—The effective address of the operand is the contents of Rn. The offset is then added to this address and the result is written back into Rn.

Example:

Memory addressing modes involving writeback.



(a) Post-indexed addressing


2012
R5
Base register (Stack pointer)
2008
27
2012
–
27
R0
after execution of Push instruction
Push instruction:
STR R0, [R5, #– 4]!

(b) Pre-indexed addressing with writeback

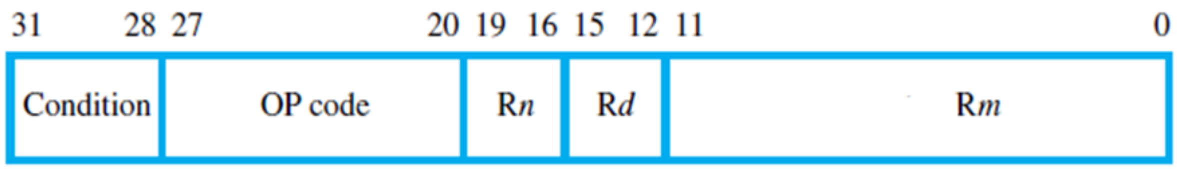
Table: ARM Indexed Addressing modes.

Name	Assembler syntax	Addressing function
With immediate offset:		
Pre-indexed	[R*n*, #offset]	EA = [R*n*] + offset
Pre-indexed with writeback	[R*n*, #offset]!	EA = [R*n*] + offset; R*n* ← [R*n*] + offset
Post-indexed	[R*n*], #offset	EA = [R*n*]; R*n* ← [R*n*] + offset
With offset magnitude in R*m*:		
Pre-indexed	[R*n*, ± R*m*, shift]	EA = [R*n*] ± [R*m*] shifted
Pre-indexed with writeback	[R*n*, ± R*m*, shift]!	EA = [R*n*] ± [R*m*] shifted; R*n* ← [R*n*] ± [R*m*] shifted
Post-indexed	[R*n*], ± R*m*, shift	EA = [R*n*]; R*n* ← [R*n*] ± [R*m*] shifted
Relative (Pre-indexed with immediate offset)	Location	EA = Location = [PC] + offset

EA = effective address
offset = a signed number contained in the instruction
shift = direction #integer
where direction is LSL for left shift or LSR for right shift; and
integer is a 5-bit unsigned number specifying the shift amount
±R*m* = the offset magnitude in register R*m* can be added to or subtracted from the contents of base register R*n*

Register addressing mode:

- It is the main way for accessing operands in arithmetic and logic instructions
- Format for arithmetic/ Logical instructions is



The basic assembly-language format for arithmetic instructions is

OPCODE R*d, Rn, Rm*

Example:

MOV R1,R2

MOV R3,R6

Immediate addressing mode:

- The second source operand is specified in the immediate mode
- The immediate operand is an 8-bit value contained in bits *b7−0 of the encoded machine* instruction.
- It is an unsigned number in the range 0 to 255. The assembly language allows negative values to be used as immediate operands.

Examples:

ADD R0, R3, #17

ADD R0, R3, #−17

Instructions Types:

- Load and Store Instructions
- Arithmetic Instructions
- Move Instructions
- Logic and Test Instructions
- Compare Instructions
- Setting Condition Code Flags
- Branch Instructions
- Subroutine Linkage Instructions

Load and Store Instructions:

- Move single word operands between the memory and registers.
- The Opcode mnemonics LDR and STR are used for these instructions.
- Byte and half-word values can also be transferred between memory and registers.
- If the operand is a byte, it is located in the low-order byte position of the register.
- If the operand is a half word, it is located in the low-order half of the register.
- For Load instructions, byte and half-word values are zero-extended to the 32-bit register length by using the instruction mnemonics LDRB and LDRH or are sign-extended by using LDRSB and LDRSH. The byte and half-word Store instructions have the mnemonics STRB and STRH
- Examples:

LDR Rd, [Rn] ;   Performs the operation   Rd←[[Rn]]

STR R0, [R5, #−4]! ;	offset −4 is added to the contents of R5 and the new value is written back into R5. The contents of register R0 are then stored at this location.

Loading and Storing Multiple Operands:

- There are two instructions for loading and storing multiple operands.
- They are called Block Transfer instructions.
- Any subset of the general-purpose registers can be loaded or stored.
- Only word operands are allowed. The OP codes used are LDM (Load Multiple) and STM (Store Multiple).
- The memory operands must be in successive word locations.
- All of the forms of pre- and post-indexing with and without write back are available.
- They operate on a base address register R*n* .
- The offset magnitude is always 4 in these instructions, so it does not have to be specified explicitly in the instruction.
- The list of registers must appear in increasing order in the assembly-language representation of the instruction, but they do not need to be contiguous.
- Example: LDMIA—Load multiple Increment After, STMIA—Store multiple Increment After.

LDMIA R10!, {R0, R1, R6, R7} ;	transfers the words from locations 1000, 1004, 1008, and 1012 into registers R0, R1, R6, and R7, leaving the address value 1016 in R10 after the last transfer, because write back is indicated by the exclamation character.

Arithmetic Instructions:

The basic assembly-language format for arithmetic instructions is

OPCODE R*d, Rn, Rm*

Examples:

ADD R0, R2, R4 ;	performs the operation R0←[R2] + [R4]
SUB R0, R6, R5 ;	performs the operation R0←[R6] − [R5]
ADD R0, R3, #17 ;	performs the operation R0←[R3] + 17

The immediate operand is an 8-bit value

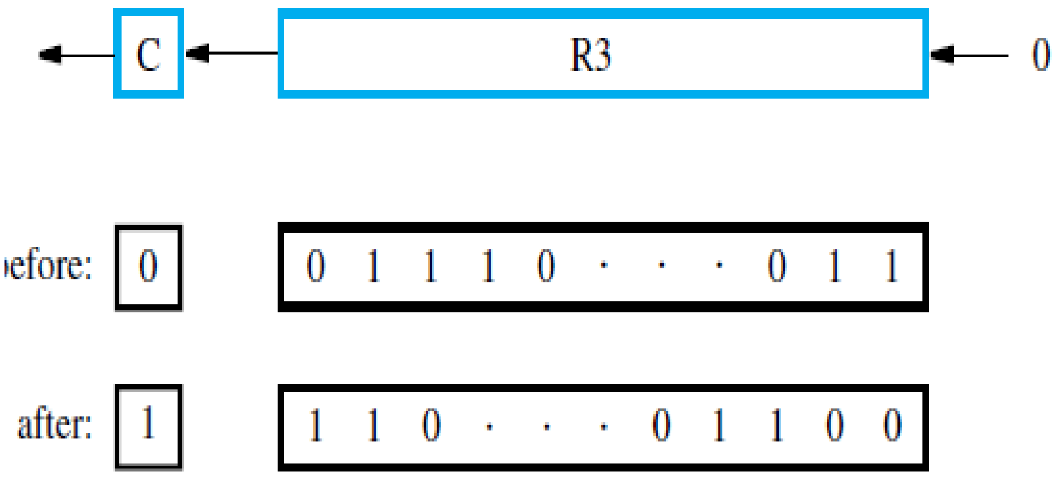
Shifting or Rotation of the Second Source Operand:

- When the second source operand is specified as the contents of a register, they can be shifted or rotated before being used in the operation. Logical shift left (LSL), logical shift right (LSR), arithmetic shift right (ASR), and rotate right (ROR) are available.
- The carry bit, C, is not involved in these operations. Shifting or rotation is specified after the register name for the second source operand.

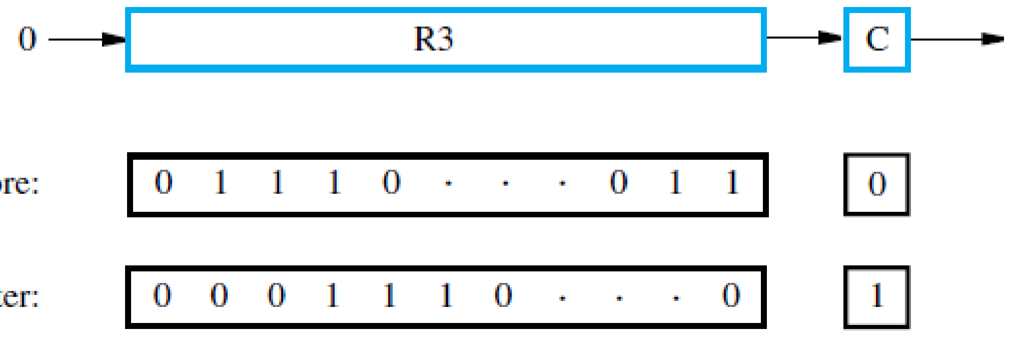
Examples:

ADD R0, R1, R5, LSL #4

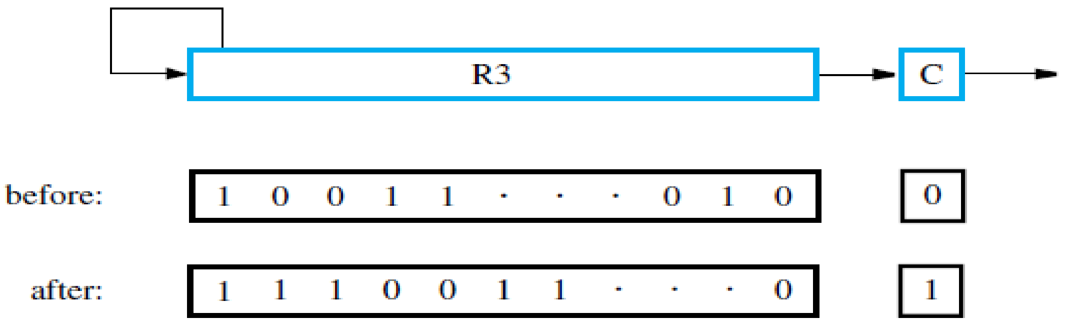
The second source operand, which is contained in register R5, is shifted left 4 bit positions (equivalent to [R5] × 16), then added to the contents of register R1. The sum is placed in register R0.



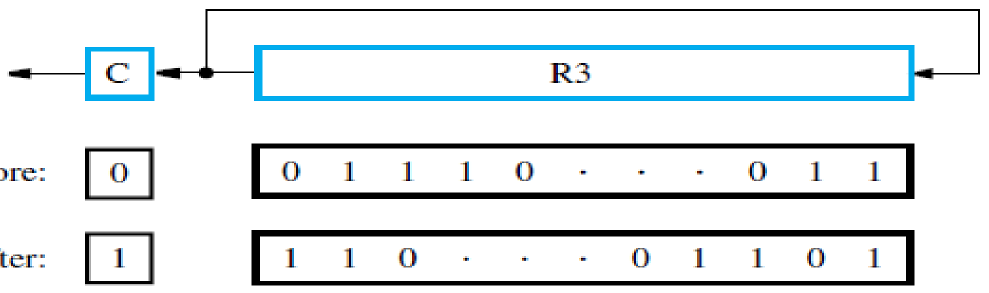
(a) Logical shift left LShiftL R3, R3, #2



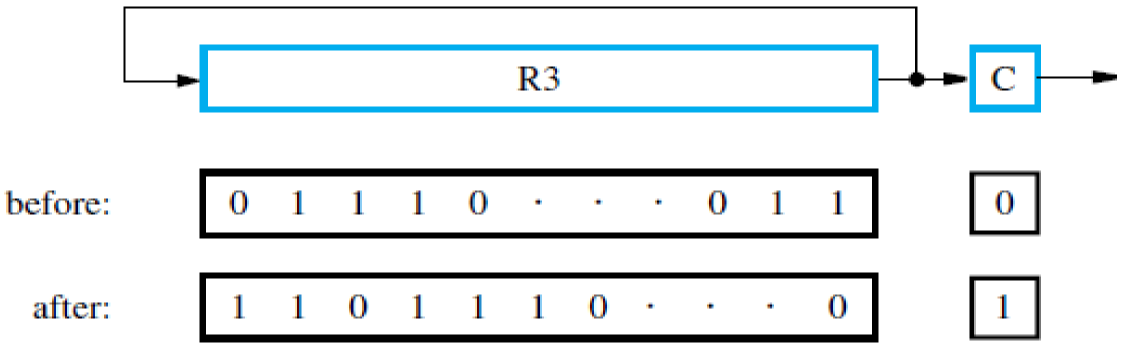
(b) Logical shift right LShiftR R3, R3, #2



(c) Arithmetic shift right AShiftR R3, R3, #2



(a) Rotate left without carry RotateL R3, R3, #2



(c) Rotate right without carry RotateR R3, R3, #2

Multiple-Word Operands:

- The carry flag, C, can be used to facilitate addition and subtraction operations that involve multiple-word numbers.
- Separate instructions are available for this purpose.
- Their assembly language mnemonics are ADC (Add with carry) and SBC (Subtract with carry).

Example:

ADDS R6, R2, R4

followed by the instruction

ADC R7, R3, R5

producing the 64-bit sum in the register pair R7, R6

Multiplication:

- The first version multiplies the contents of two registers and places the low-order 32-bits of the product in a third register. The high-order bits of the product are discarded.

Examples:

MUL R0, R1, R2 ; performs the operation R0←[R1] × [R2]

MLA R0, R1, R2, R3 ; performs the operation R0←([R1] × [R2]) + [R3]

This is called a Multiply-Accumulate operation.

Move Instructions:

- Used to copy the contents of one register into another or to load an immediate value into a register.

  MOV R*d, Rm*

  copies the contents of register R*m into register Rd.*

  MOV R*d, #value*

  loads an 8-bit immediate value into the destination register.

- A second version of the Move instruction, called Move Negative, with the OP-code mnemonic MVN, forms the bit-complement of the source operand before it is placed in the destination register. Recall that an 8-bit immediate value is an unsigned number in the range 0 to 255. The MVN instruction can be used to load negative values in 2's-complement representation as follows. Suppose we wish to load −5 into register R0.
- The instruction

  MVN R0, #4

  achieves the desired result because the bit-complement of 4 is the 2's-complement representation for −5. In general, to load −*c* into a register, the MVN instruction can be used with an immediate source operand value of *c* − 1. For the convenience of the programmer, the assembler program accepts an instruction such as

  MOV R0, #−5 (Pseudo instruction)

  and replaces it with the instruction

  MVN R0, #4

- ARM instructions consist of a single 32-bit word, so the address cannot be represented by an immediate value in a Move instruction.
- The assembler accepts an instruction of the form

  LDR R*i*, =ADDRESS

  to load the address value ADDRESS into register R*i*.

Examples:

MVN R0, #4

MOV R0, #−5

MOV Ri, Rj, LSL #4

Logic and Test Instructions:

- The logic operations AND, OR, XOR, and Bit-Clear are implemented by instructions with the OP codes AND, ORR, EOR, and BIC, respectively.

Examples:

```
                                ; R0 = 02FA62CA, R1=0000FFFF
AND R0, R0, R1 ;                R0= 000062CA (performs bitwise AND)
                                ; R0 = 0000FFFF, R1=8FFF1111
ORR R0,R0,R1    ;               R0=8FFFFFFF
                                ; R0 = 11110000, R1=11111111
EOR R0,R0,R1 ;                  R0=00001111
                                ; R0 = 02FA62CA, R1=0000FFFF
BIC R0, R0, R1 ;                R0 = 02FA0000
```

Complements each bit in R1 before ANDing them with the bits in register R0.

Test Instructions:

- Instructions called Test (TST) and Test Equivalence (TEQ) perform the logical AND and XOR operations, respectively, on their word operands, then set condition code flags based on the result.
- They do not store the result in a register. These instructions can be used to test how an unknown bit pattern matches up against a known bit pattern, and can then be followed by a Branch instruction that is conditioned on the result.

Examples:

TST Rn, #1; performs AND operation to test whether the low-order bit of register R*n is equal to 1.* if it *is* 1, then Z bit is cleared to zero.

TEQ Rn,#5 ; performs XOR operation to test whether R*n is contains 5.* If it *is equal*, then Z bit is set to 1.

Compare Instructions:

CMP Rn, Rm

performs the operation [Rn] − [Rm] internally, and sets the condition code flags based on the result of the subtraction operation. Z will be set if result of subtraction is zero.

Second operand can be an immediate value instead of the contents of a register.

Setting Condition Code Flags:

The arithmetic, logic, and Move instructions affect the condition code flags only if explicitly specified to do so by a bit in the OP-code field. This is indicated by appending the suffix S to the assembly language OP-code mnemonic.

For example, the instruction

ADDS R0, R1, R2

sets the condition code flags,

but

ADD R0, R1, R2

does not.

Branch Instructions:

Unconditional branch instruction – Control transferred to target unconditionally

Example:

```
B target ;        Branch to target
```

Conditional branch instructions - branch is executed only if the current state of the condition code flags corresponds to the condition specified in the Condition field of the instruction.

Example:

```
CMP R0,R1
BEQ target ;      Branch to target only if Z=1 else ignore
```

Condition field encoding in ARM instructions.

Condition suffix	Name	Condition code test
EQ	Equal (zero)	$Z = 1$
NE	Not equal (nonzero)	$Z = 0$
CS/HS	Carry set/Unsigned higher or same	$C = 1$
CC/LO	Carry clear/Unsigned lower	$C = 0$
MI	Minus (negative)	$N = 1$
PL	Plus (positive or zero)	$N = 0$
VS	Overflow	$V = 1$
VC	No overflow	$V = 0$
HI	Unsigned higher	$\overline{C} \vee Z = 0$
LS	Unsigned lower or same	$\overline{C} \vee Z = 1$
GE	Signed greater than or equal	$N \oplus V = 0$
LT	Signed less than	$N \oplus V = 1$
GT	Signed greater than	$Z \vee (N \oplus V) = 0$
LE	Signed less than or equal	$Z \vee (N \oplus V) = 1$
AL	Always	

Subroutine Linkage Instructions:

- The Branch and Link (BL) instruction is used to call a subroutine.
- The return address, which is the address of the next instruction after the BL instruction, is loaded into register R14, which acts as the link register.
- Since subroutines may be nested, the contents of the link register must be saved on the processor stack before a nested call to another subroutine is made.
- Register R13 is used as the processor stack pointer.

Calling program

```
          LDR     R1, N
          LDR     R2, =NUM1
          BL      LISTADD
          STR     R0, SUM
          ⋮
```

Subroutine

```
LISTADD   STMFD   R13!, {R3, R14}   Save R3 and return address in R14 on
                                       stack, using R13 as the stack pointe
          MOV     R0, #0
LOOP      LDR     R3, [R2], #4
          ADD     R0, R0, R3
          SUBS    R1, R1, #1
          BGT     LOOP
          LDMFD   R13!, {R3, R15}   Restore R3 and load return address
                                       into PC (R15).
```

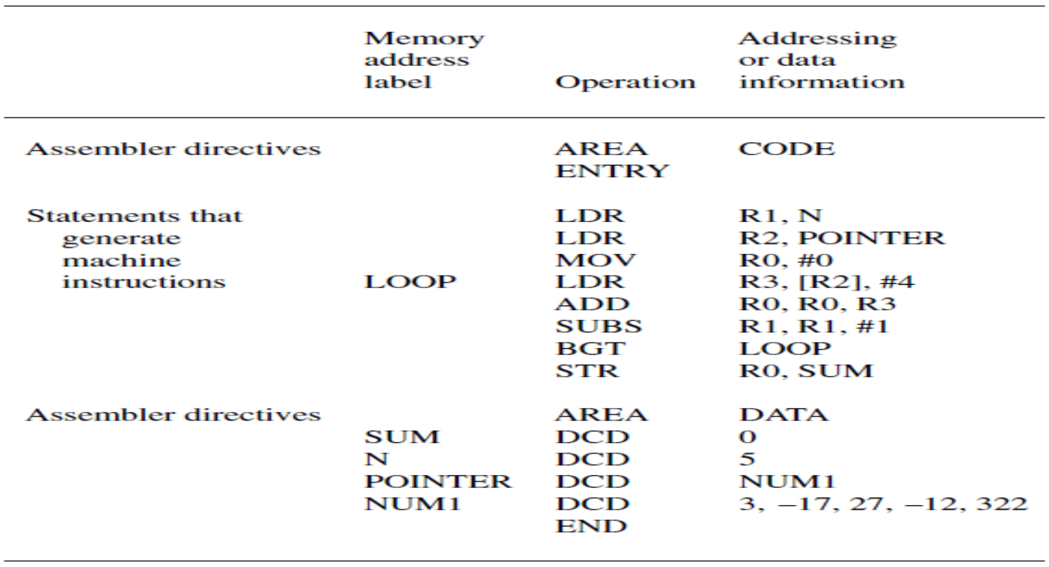
Parameters are passed through registers. The calling program passes the size of the number list and the address of the first number to the subroutine in registers R1 and R2. The subroutine passes the sum back to the calling program in register R0. The subroutine also uses register R3. Therefore, its contents, along with the contents of the link register R14, are saved on the stack by the STMFD instruction. The suffix FD in this instruction specifies that the stack grows toward lower addresses and that the stack pointer R13 is to be pre decremented before pushing words onto the stack. The LDMFD instruction restores the contents of register R3 and pops the saved return address into the PC (R15), performing the return operation automatically.

Assembler directives:

A special codes placed in assembly language program to instruct the assembler to perform a particular task of function is called Assembler directives.

The ARM assembly language has assembler directives to reserve storage space, assign numerical values to address labels and constant symbols, define where program and data blocks are to be placed in memory, and specify the end of the source program text.

Directive	**Description**
AREA	which uses the argument CODE or DATA, indicates the beginning of a block of memory that contains either program instructions or data.
ENTRY	specifies that program execution is to begin at the instruction that follows that it.
DCD	used to label and initialize the data operands.
EQU	declare symbolic names for constants
RN	to use symbolic names for registers, relating to their usage



	Memory address label	Operation	Addressing or data information
Assembler directives		AREA	CODE
		ENTRY	
Statements that generate machine instructions		LDR	R1, N
		LDR	R2, POINTER
		MOV	R0, #0
	LOOP	LDR	R3, [R2], #4
		ADD	R0, R0, R3
		SUBS	R1, R1, #1
		BGT	LOOP
		STR	R0, SUM
Assembler directives		AREA	DATA
	SUM	DCD	0
	N	DCD	5
	POINTER	DCD	NUM1
	NUM1	DCD	3, −17, 27, −12, 322
		END	

COUNTER RN 3

establishes the name COUNTER for register R3. The register names R0 to R15, PC (for R15), and LR (for R14) are predefined by the assembler.

Pseudoinstructions:

- A pseudoinstruction is an assembly-language instruction that performs some desired operation but does not correspond directly to an actual machine instruction.
- The assembler accepts such an instruction and replaces it with an actual machine instruction that performs the desired operation.
- Pseudoinstructions are provided for the convenience of the programmer.

Examples:

LDR R3, =127 ;	Loads a const value of 127 in R3
LDR R3, =&A123B456 ;	Loads address of the memory location of the number stored, in R3
ADR R3, LOCATION ;	loads the 32-bit address represented by LOCATION into R3.

The equal sign in front of the value distinguishes this instruction from an actual Load instruction.

Operating Modes and Exceptions:

The ARM processor has seven operating modes.

- User mode
- Five exception modes
- System mode.

User mode:

- Application programs run in User mode.
- The normal sixteen processor registers are in use.

Five exception modes:

- **Fast interrupt (FIQ) mode** is entered when an external device raises a fast-interrupt request to obtain urgent service. The FIQ interrupt is intended for one device or a small number of devices that require rapid response. The banked registers for the FIQ processor mode include five general-purpose registers R8_fiq through R12_fiq in addition to the stack pointer register R13_fiq and the link register R14_fiq.
- **Ordinary interrupt (IRQ) mode** is entered when an external device raises a normal interrupt request. All other I/O devices use the IRQ interrupt line to request service.
- **Supervisor (SVC) mode** is entered on powerup or reset, or when a user program executes a Software Interrupt instruction (SWI) to call for an operating system routine to be executed. This is the highest priority exception. It places the processor into a known initial state so that operating system software can begin or restart operation properly. Any program executing when this exception occurs is abandoned.
- **Memory access violation (Abort) mode** is entered when an attempt by the current program to fetch an instruction or a data operand causes a memory access violation. Processor implementations may include a memory management unit that restricts programs to valid areas of the address space for their instructions and data. If the processor issues an address for an instruction fetch or data operand access outside these areas, an exception occurs and the Abort mode is entered.
- **Unimplemented instruction (Undefined) mode** is entered when the current program attempts to execute an unimplemented instruction. If the processor tries to execute an instruction that is not implemented in hardware, an exception is raised and the Undefined mode is entered.
- For example, a floating-point arithmetic operation that can be supported by special hardware may not be implemented in the current processor. In this case, the exception can cause a software implementation of the operation to be executed.

System Mode:

- The System mode is a privileged mode that uses the same registers as those used in the User mode.
- It can only be entered from another exception mode. Its purpose is to facilitate linkage to subroutines during exception handling without overwriting the link register R14_mode.
- When in System mode, subroutine Call instructions use the normal link register R14. After returning from all subroutine calls, the original exception mode is reentered, regaining access to the link register R14_mode.

When an exception occurs, the following actions are taken on the switch from User mode to the appropriate exception mode:

1. The contents of the Program Counter (R15) are loaded into the banked Link register (R14_mode) of the exception mode.
2. The contents of the Status register (CPSR) are loaded into the banked Saved Status register (SPSR_mode).
3. The mode bits of CPSR are changed to represent the appropriate exception mode, and the interrupt-disable bits I and F are set appropriately.
4. The Program Counter (R15) is loaded with the dedicated vector address for the exception, and the instruction at that address is fetched and executed to begin the exception-service routine.

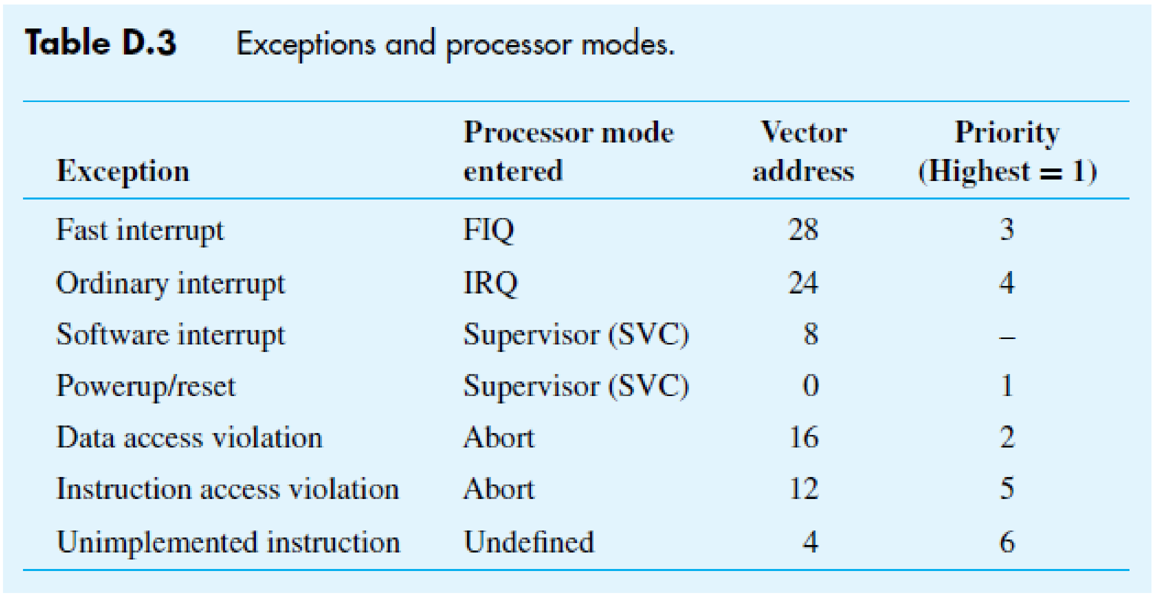


Table D.3 Exceptions and processor modes.

Exception	Processor mode entered	Vector address	Priority (Highest = 1)
Fast interrupt	FIQ	28	3
Ordinary interrupt	IRQ	24	4
Software interrupt	Supervisor (SVC)	8	–
Powerup/reset	Supervisor (SVC)	0	1
Data access violation	Abort	16	2
Instruction access violation	Abort	12	5
Unimplemented instruction	Undefined	4	6

Banked Registers:

When the processor is operating in either the User or System mode, the normal sixteen processor registers are in use. When an exception occurs and a switch is made from User mode to one of the five exception modes, some of these sixteen registers are replaced by an equal number of *banked registers.* The contents of the replaced registers are left unchanged. There is a different set of banked registers for each of the five exception modes,

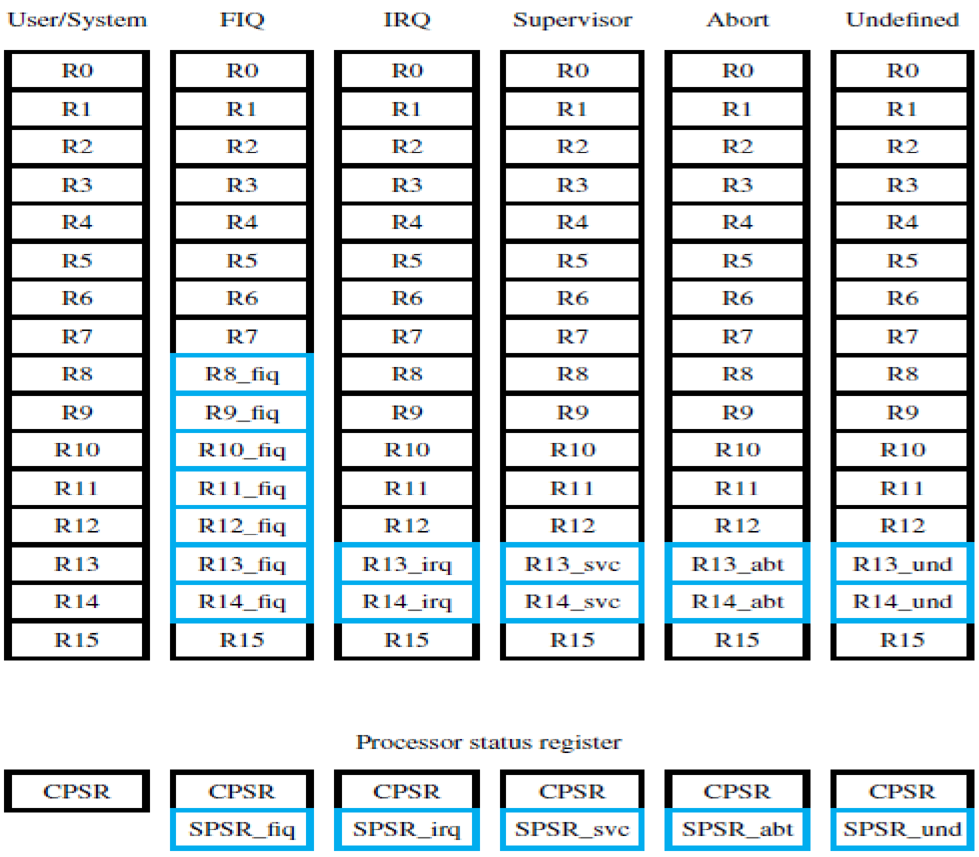


Figure D.16 Accessible registers in different modes of the ARM processor.

Conditional Execution of Instructions:

The conditional execution of all ARM instructions permits shorter routines to be written in place of routines written for conventional RISC machines where there are a number of branch instructions.

```
LOOP      Branch_if_[R2]=[R3]   NEXT
          Branch_if_[R2]>[R3]   REDUCE
          Subtract              R3, R3, R2
          Branch                LOOP
REDUCE    Subtract              R2, R2, R3
          Branch                LOOP
NEXT      next instruction
```

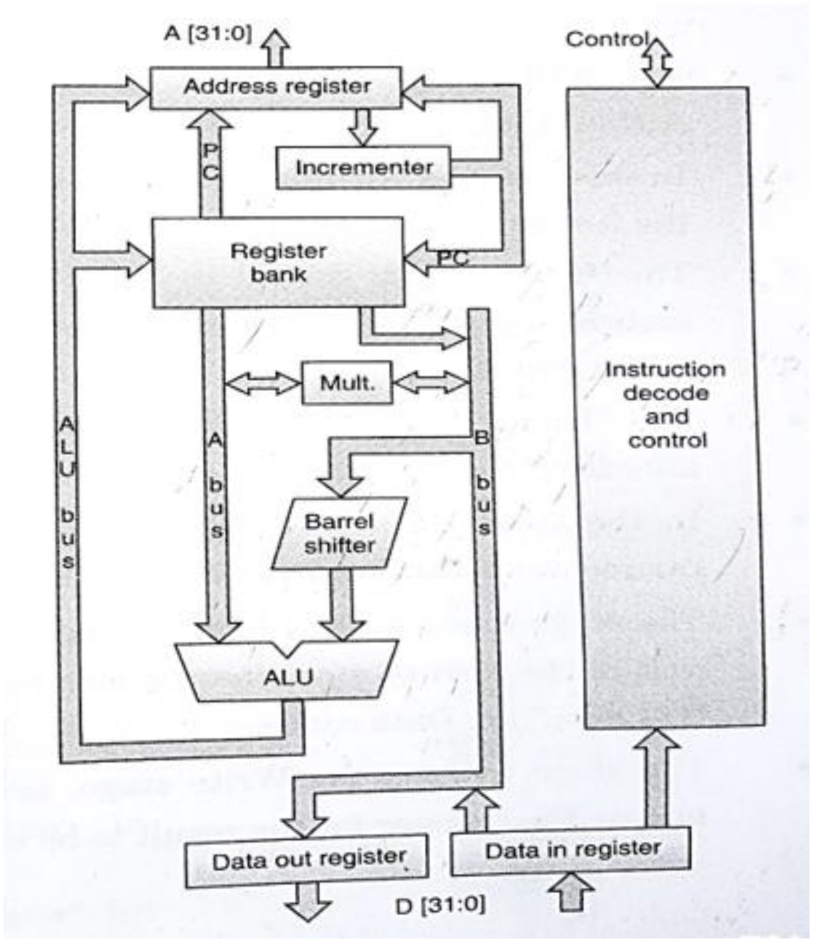
(a) GCD algorithm using RISC-style instructions

```
LOOP    CMP      R2, R3
        SUBGT    R2, R2, R3
        SUBLT    R3, R3, R2
        BNE      LOOP
NEXT    next instruction
```

(b) GCD algorithm using ARM instructions

Organization of an ARM Processor:

The organization of an ARM Processor with three stage pipeline consists of the following: Register bank: This includes various registers as seen in the programmer's model of ARM.



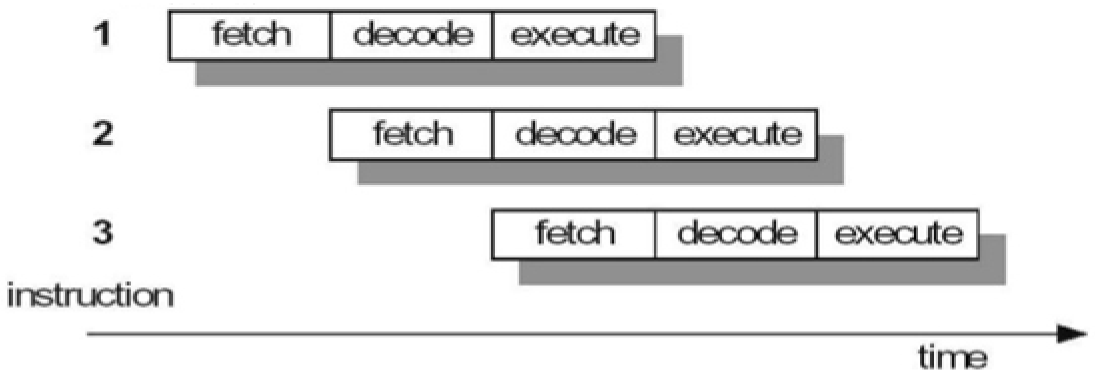
Barrel Shifter: Shift or rotate the operand by any number of bits.

The ALU: This unit performs the various arithmetic and logical operations

Address register and increment: The register stores the address and the incrementer increments the same so as to point to the next instruction. 5. Data register: It is used as a buffer to store the data when written to the memory or read from the memory.

Instruction Decoder: As the name says, it decodes the instruction and issues the control signals accordingly. Hence, the Instruction decoder is associated with the control logic that issues the control signals. Fig shows the implementation of the three stage pipelining in ARM. In the figure, the flow of the instruction shows the three stages of pipelining. The instruction is first fetched by the PC (at the top) giving address to the address register and is incremented using the incrementer, so that the PC points to the next instruction. The instruction is fetched through the data bus (at the bottom) and is given to the Instruction Decode and Control unit. This unit decodes the instruction, which is the second stage of the pipeline and then the instruction is executed by the ALU, multiplier and barrel shifter using the registers from the register bank.

Fetch – The instruction is fetched from memory and placed in the instruction pipeline



Decode – The instruction is decoded and the datapath control signals prepared for the next cycle

Execute – The register bank is read, an operand shifted, the ALU result generated and written back into destination register

Example Programs: Assembly Language Program(ALU)

Count the number of 1s in a 32 - bit Binary Number

```
        AREA  RESET,CODE
        ENTRY
        LDR R0,=0xAAAAAABB              ;Load the Number in R0
        MOV R1,#0                       ;Clear R1 to Zero
LOOP
        MOVS R0, R0, LSR #1  ;Right Shift R0 by 1 and move with status new value in R0
        ADDCS R1, R1, #1            ;Add 1 to R1 if carry flag was set
        CMP R0, #0                  ;Check if R0 has become zero
        BNE LOOP                    ;If R0 not Zero branch back to LOOP
EXIT    B  EXIT
        END
```

;Check if the given number is odd or even. Store FFh in R4 if odd, else 80h in R4

```
        AREA  RESET,CODE
        ENTRY
        MOV R0, #25
        AND R0, #1
        CMP R0, #0
        BEQ EVEN
        MOV R4, #255
        B STOP
EVEN    MOV R4, #128
STOP    B STOP
         END
```

;Check if the given number is odd or even. Store FFh in R4 if odd, else 80h in R4

```
        AREA  RESET,CODE
        ENTRY
        MOV R0, #25
        MOVS R0, R0, LSR #1
        MOVCS R4, #255
        MOVCC R4, #128
STOP    B STOP
         END
```

;Find the smallest number in a given array and Store the result in memory location ;0x40000000

```
        AREA RESET,CODE
        ENTRY
        LDR R0,=DATA1
        LDR R3,=0X40000000
        LDR R4,=0X05            ;//N- number of elements
        LDR R1,[R0],#04
        SUB R4,R4,#01
BACK
        LDR R2,[R0]
        CMP R1,R2
        BLS LESS ;// BRANCH ON LOW
        MOV R1,R2
LESS
        ADD R0,R0,#04
        SUB R4,R4,#01
        CMP R4,#00
        BNE BACK
        STR R1,[R3]             ;// SMALLEST VALUE STORED IN MEMORY LOCATION
STOP B STOP
DATA1 DCD 64,05,96,10,65
        END
```