



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – I - MICROPROCESSOR, MICRO CONTROLLER AND EMBEDDED SYSTEM– SPHA5204

UNIT 1

8086 MICROPROCESSORS (16 BIT)

8086 microprocessor - Internal architecture, signals-addressing modes- instruction formats-instruction set, Programming-addition, subtraction, multiplication and division, Interfacing-traffic light controller, stepper motor control.

INTRODUCTION

The microprocessor is the central unit of a computer system that performs arithmetic and logic operations, which generally include adding, subtracting, transferring numbers from one area to another, and comparing two numbers. It's often known simply as a processor, a central processing unit, or as a logic chip. It's essentially the engine or the brain of the computer that goes into motion when the computer is switched on. It's a programmable, multipurpose device that incorporates the functions of a CPU (central processing unit) on a single IC (integrated circuit). A microprocessor accepts binary data as input, processes that data, and then provides output based on the instructions stored in the memory. The data is processed using the microprocessor's ALU (arithmetical and logical unit), control unit, and a register array. The register array processes the data via a number of registers that act as temporary fast access memory locations. The flow of instructions and data through the system is managed by the control unit.

Benefits of a Microprocessor

Less cost - Due to their use of IC technology, microprocessors don't cost much to produce. This means that the use of microprocessors can greatly reduce the cost of the system it's used in.

Fast - The technology used to produce modern microprocessors has allowed them to operate at incredibly high speeds--today's microprocessors can execute millions of instructions per second.

Power consumption - Power consumption is much lower than other types of processors since microprocessors are manufactured using metal oxide semiconductor technology. This makes devices equipped with microprocessors much more energy efficient.

Portable - Due to how small microprocessors are and that they don't consume a lot of power, devices using microprocessors can be designed to be portable (like smartphones).

Reliable - Because semiconductor technology is used in the production of microprocessors, their failure rate is extremely low.

They are versatile - The same microprocessor chip can be used for numerous applications as long as the programming is changed, making it incredibly versatile.

Common Terms used – in Microprocessors

Word Length: Word length refers to the number of bits in the processor's internal data bus--or the number of bits that a processor can process at any given time. For example, an 8-bit processor will have 8-bit registers, an 8-bit data bus, and will perform 8-bit processing at a time.

Instruction Set: The instruction set is the series of commands that a microprocessor can understand. Essentially, it's the interface between the hardware and the software.

Cache Memory: The cache memory is used to store data or instructions that the software or program frequently references during operation. Basically, it helps to increase the operation's overall speed by allowing the processor to access data more quickly than from a regular RAM.

Clock Speed: The clock speed is the speed at which a microprocessor is able to execute instructions. It's typically measured in Hertz and expressed in measurements like MHz (megahertz) and GHz (gigahertz).

Bus: A bus is the term used to describe the set of conductors that transmit data or that address or control information to the microprocessor's different elements. Most microprocessors consist of three different buses, which include the data bus, the address bus, and the control bus.

Categories of Microprocessors

Based on Word Length

Microprocessors can be based on the number of bits the processor's internal data bus or the number of bits that it can process at a time (which is known as the word length). Based on its word length, a microprocessor can be classified as 8-bit, 16-bit, 32-bit, and 64-bit.

RISC - Reduced Instruction Set Computer

RISC microprocessors are more general use than those that have a more specific set of instructions. The execution of instructions in a processor requires a special circuit to load and process data. Because RISC microprocessors have fewer instructions, they have simpler circuits, which means they operate faster. Additionally, RISC microprocessors have more registers, use more RAM, and use a fixed number of clock cycles to execute one instruction.

CISC - Complex Instruction Set Computer

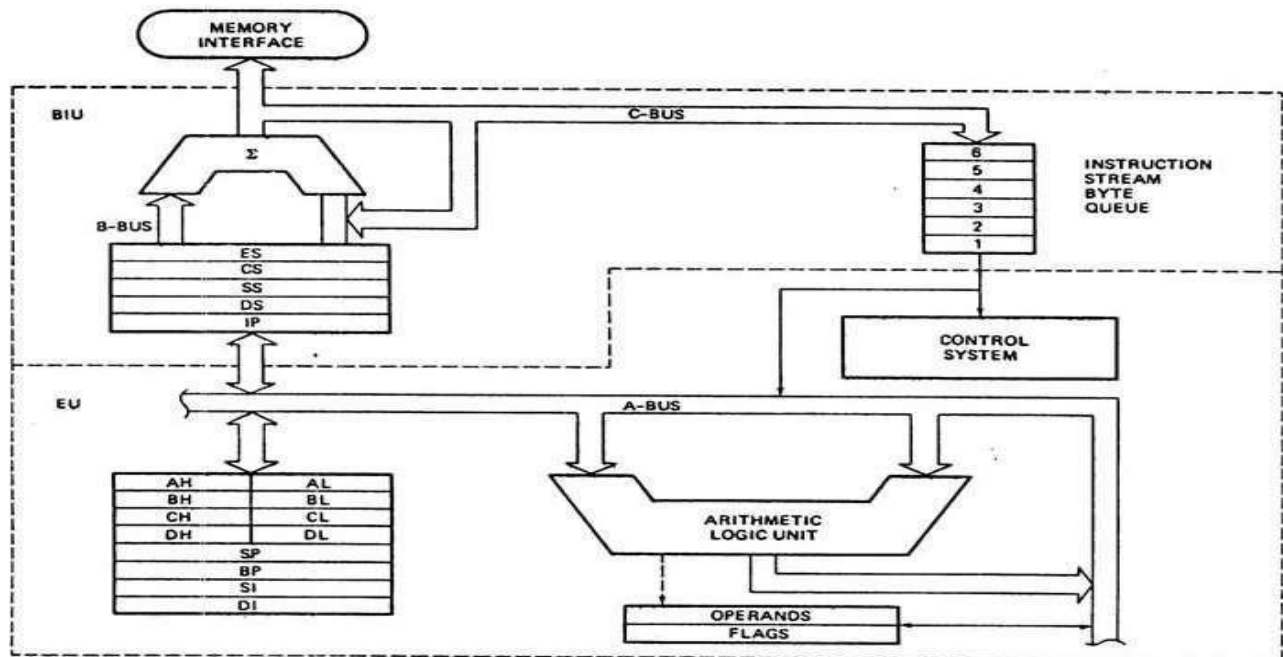
CISC microprocessors are the opposite of RISC microprocessors. Their purpose is to reduce the number of instructions for each program. The number of cycles per instruction is ignored. Because complex instructions are made directly into the hardware, CISC microprocessors are more complex and slower.

CISC microprocessors use little RAM, have more transistors, have fewer registers, have numerous clock cycles for each instruction, and have a variety of addressing modes.

Special Purpose Processors

Some microprocessors are built to perform specific functions. For example, coprocessors are used in combination with a main processor, while a transputer is a transistor computer: a microprocessor that has its own local memory.

ARCHITECTURE OF 8086



It is Pipelined architecture. The 8086 CPU is divided into two functional units: Bus Interface Unit (BIU) , Execution Unit (EU)

The Bus Interface Unit (BIU)

The BIU handles all data and addresses on the buses for the execution unit such as it sends out addresses, fetches instructions from memory, reads/writes data from ports and memory. It contains the bus interface logic, Segment register, stack pointer, base pointer, index pointer, memory address logic and 6 byte Instruction queue (FIFO). BIU fetches the instruction code from memory and stores in the queue.

Instruction Queue

To increase the execution speed, BIU fetches as many as six instruction bytes ahead to time from

memory. The prefetched instruction bytes are held for the EU in a first in first out group of registers called a **instruction queue**. When the EU is ready for its next instruction, it simply reads the instruction from this instruction queue. This is much faster than sending out an address to the system memory and to send back the next instruction byte. Fetching the next instruction while the current instruction executes is called **pipelining**.

Execution unit (EU)

It consists of ALU, General purpose register, Flag register, Instruction decoder, Pointer and index register and the control unit which are required to execute an instruction. EU gets the opcode of the instruction from instruction queue. Then it will be decoded and executed. BIU and EU are independent. The overlapping operation of BIU and EU functional unit of a microprocessor is called pipelining.

Register

It has fourteen 16 bit register. All the register subdivide in to

1. Data Register
2. Segment Register
3. Pointer Register
4. Index register (program counter)
5. Flag Register

Data Register

It has four 16 bit general purpose register (AX, BX, CX, DX)

16 bit Register	8 bit High-order Register	8 bit lower order Register
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

AX register:

- It serves as accumulator.
- It performs I/O operation and process data through AX, AH, AL.
- Result stored in accumulator.
- For 16 bit multiplication/division AX contain one word operand.
- For 32 bit multiplication/division AX contain lower order word operand.

BX register:

- It acts as a index register for MOVE operation.
- Base register for data memory address.

CX register:

- CX register can be used as a count register for string operation.
- It will have the count .
- It has large number of iteration.
- CX holds desired number of repetition and it automatically decremented by one after the execution of instruction.
- CX become zero , the execution instruction is terminated.

DX register:

- DX can be used as a port address for IN and OUT instruction.
- The DX can be used in I/O instruction, Divide, Multiple Instruction.
- In 32 bit multiplication/division instruction DX is used to hold the higher order word operand.

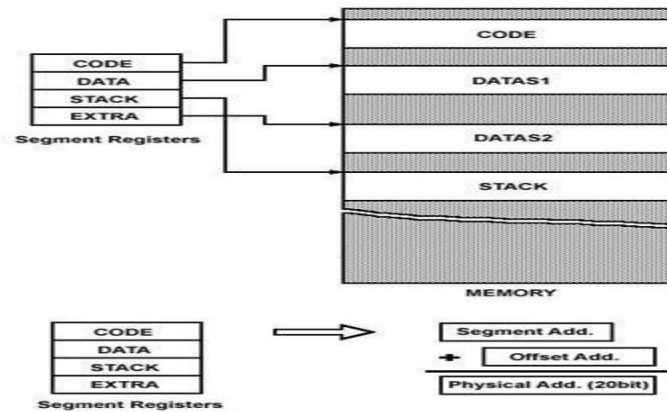
Segment Registers:

The BIU contains four 16-bit segment registers, one MB memory. Each divided in to 16 parts which are called segments. Each segment occupies the 64KB memory space.

They are:

- Code segment (CS) register
- Data segment (DS) register
- Stack segment (SS) register
- Extra segment (ES) register

These segment registers are used to hold the upper 16 bits of the starting address for each of the segments. The part of a segment starting address stored in a segment register is often called the **segment base**.



1. Code Segment (CS)

The CS register is used for addressing a memory location in the Code Segment of the memory, where the executable program is stored.

2. Data Segment (DS)

The data segment register points to the data segment of the memory, where data is stored

3. Stack Segment (SS)

It is used to addressing stack segment of the memory in which data is stored.

4. Extra Segment (ES):

It can be used as another data segment of the memory.

Pointer and Index register

1. Stack Pointer(SP)
2. Base Pointer(BP)
3. Source Index(SI)
4. Destination Index(DI)
5. Instruction Pointer(IP)

Flag register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

S- Sign Flag : This flag is set, when the result of any computation is negative.

Z- Zero Flag: This flag is set, if the result of the computation or comparison performed by the previous instruction is zero.

P- Parity Flag: This flag is set to 1, if the lower byte of the result contains even number of 1's.

C- Carry Flag: This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

T- Tarp Flag: If this flag is set, the processor enters the single step execution mode.

Interrupt Flag: If this flag is set, the maskable interrupt are recognized by the CPU, otherwise they are ignored.

D- Direction Flag: This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

AC-Auxiliary Carry Flag: This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

O- Over flow Flag: This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit signed operations, then the overflow will be set.

Addressing modes of 8086

The method of specifying the data to be operated.

Immediate Addressing mode:

The 8 bit or 16 bit data is provided in the instruction itself.

EX: MOV BX, 5000H-Load 5000H in to BX register.

MOV CX, 4500H-Store 4500H in to CX register.

Register Addressing mode:

The data in a register or in a register pair specified by the instruction.

EX: MOV AL, BL-The content present in BL register is copied to AL register.

MOV AX, BX-The content present in BX register is copied to AX register.

Memory addressing mode:

Memory Addressing requires determination of physical address. The physical address can be computed from the content of segment register and an effective address. The segment address identifies the starting location of the segment in the memory and the effective address represents the offset of the operand from the beginning of this segment of memory. The 20 bit effective

address can be made up of base, index and displacement.

16- bit effective address (EA)=Base+ Displacement.

20 bit physical address (PA) = Segment*10+Base+Index+Displacement.

TYPES:

1. Direct addressing mode
2. Register indirect addressing mode
3. Based addressing mode
4. Indexed addressing mode
5. Based Indexed addressing mode
6. Based Indexed with displacement addressing mode
7. String addressing mode
8. Branch addressing mode

Direct addressing mode:

The instruction or operand specifies the memory address where data is located. EX:

MOV AX, [5000H]-copies the 2 byte of data starting from memory location DS*10+5000H to AX register. LSB-Data from DS*10+5000H, MSB- Data from DS*10+5001H

MOV AX, SS: [2000H]: Similarly to access the memory from Stack segment.

Register indirect addressing mode:

The instruction specifies the register containing the address where the data is located. Where data is located. This addressing mode works with SI, DI, BX and BP registers.

EX: MOV AL, [BX]-The Data present in BX register pointed to memory location (which default indicates the data segment) is moved to AL register.

MOV AL, CS:[SI]- The Data present in SI register pointed to memory location which indicates the code segment is moved to AL register.

Based addressing mode:

The 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP), the resulting value is a pointer to location where data resides.

EX: MOV AL, [BX+8 Bit DISP]-The content of memory location pointed by physical address (PA) of the base register BX with displacement is copied to AL register.

MOV AH, [BP+8 Bit DISP]- The content of memory location pointed by physical address (PA) of the base register BP with displacement is copied to AH register.

Indexed addressing mode:

The 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.

EX: MOV AL, CS: [SI+DISP]- The content of memory location pointed by physical address (PA) of the index register SI with displacement which is present in code segment is copied to AL register.

MOV AL, SS: [DI+DISP]- The content of memory location pointed by physical address (PA) of the index register DI with displacement which is present in stack segment is copied to AL register.

Based Indexed addressing mode:

The contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.

EX: MOV AL, [BX+DI]- The content of memory location pointed by physical address(PA) of the summation base register BX and index register DI is copied to AL register.

MOV AL, [BP+SI]- The content of memory location pointed by physical address (PA) of the summation base register BP and index register SI is copied to AL register.

Based Indexed with displacement

The 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI) with displacement value, the resulting value is a pointer to location where data resides.

EX: MOV AL, [BX+DI+DISP]-The 16-bit operand is added to the contents of a BX and DI with displacement ,the resulting value is a pointer to location of AL.

MOV AL, [BP+SI+DISP]- The 16-bit operand is added to the contents of a BP and SI with displacement ,the resulting value is a pointer to location of AL.

String Addressing Mode:

String is a byte or word s which are stored in memory.

EX: MOV SB-The memory source address is the SI register in the data segment. The memory of destination address is the DI register in the extra segment.

Branch Addressing Mode:

- Intra segment mode- Intra segment Direct, Intra segment Indirect
- Inter segment mode- Inter segment Direct, Inter segment Indirect

EX: JNC START If CY=0, then PC is loaded with current PC contents plus 8 bit signed value of START, otherwise the next instruction is executed.

Direct IO port addressing:

It is used to access data from standard IO-mapped devices or ports. In the direct port addressing mode, an 8-bit port address is directly specified in the instruction.

EX: IN AL, [09H]-The content of the port with address 09H is moved to the AL register.

Indirect IO port addressing:

It is used to access data from standard IO mapped devices or ports. In the indirect port addressing mode, the instruction will specify the name of the register which holds the port address.

EX: OUT [DX],AX-The content of the AX is moved to the port whose address is specified by the DX register.

Relative Addressing mode:

The effective address of a program instruction is specified relative to instruction pointer(IP) by an 8 bit signed displacement.

EX: JZ 0AH-Jump on Zero

Implied Addressing mode:

The instruction itself will specify the data to be operated by the instruction.

EX: CLC-clear flag

CMC-complement carry flag

INSTRUCTION SET OF 8086

The command applied to the microprocessor to perform a specific function.

TYPES:

1. Data transfer Instruction
2. Arithmetic and Logical Instruction
3. Branch Instruction
4. Loop Instruction
5. Process control Instruction
6. Flag Manipulation Instruction
7. Shift and Rotate Instruction.
8. String Manipulation Instruction

Data transfer Instruction:

The data is copied from source to Destination without any change. i.e register to register

Memory to register; register to memory, Data given immediately to register or memory.[MOV,LDS,XCHG,PUSH,POP]

EX: MOV AX, SI-The content of SI is moved to the AX register.

XCHG DH, CL-The content of CL register is exchanged with content of DH register.

Arithmetic and Logical Instruction:

It performs the arithmetic, logical, increment, decrement, compare and scan instruction.[ADD,SUB,MUL,DIV,INC,CMP,DAS,AND,OR,NOT,TEST,XOR]

EX: ADC BH,CL-The content of BH register ,the AL register and the carry flag are added. The result is stored in the BH- register.

XOR BX,DX-The content of the BX and DX registers are Exclusive ORed .This result is stored in BX register.

Control Transfer Instruction:

- Branch Instruction
- Loop Instruction

Branch Instruction:

The instruction will transfer the control of execution to the specified address. The JUMP, CALL, interrupt and Return belongs to the Branch instruction.

EX: JNC 4500H-Jump if no carry to the memory location 4500H

CALL -This Instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALL: Near and Far.

JCXZ Instruction - Jump if the CX register is zero

Loop Instruction:

These instructions have REP prefix with CX used as count register, and they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ, LOOPZ instruction belong to this Loop instruction. It is used to implement different delay loops.

EX: LOOP Instruction - Loop to specified label until CX = 0

LOOPE / LOOPZ - loop while CX \neq 0 and ZF = 1

Process control Instruction:

The instruction control the machine status CLC, CMC, CLI, STD, STI, NOP, HLT, WAIT and LOCK instruction.

EX:CLC-Carry flag is reset to zero.

CMC-Carry flag is complemented.

HLT-Halt the program execution.

WAIT-Wait for test line active.

Flag Manipulation Instruction:

All the instruction which directly affect the flag registers come under this group of instruction. Instructions like CLD, STD, CLI, and STI belong to this category.

EX:STD: Set direction Flag

STC: Set Carry Flag

Shift and Rotate Instruction: The instruction involves in the bitwise shifting OR Rotation in either direction with or without a count in CX. The example instructions are RCL, RCR, ROL, ROR, SAL, SHL, SAR and SHR.

EX: ROR AX, CL- Rotate word or byte operand right by CL times. So CF and OF flag gets affected.

SAL AX, CL-Shift word or byte operand CL times.

Note: Shift and Rotate examples comes under Logical

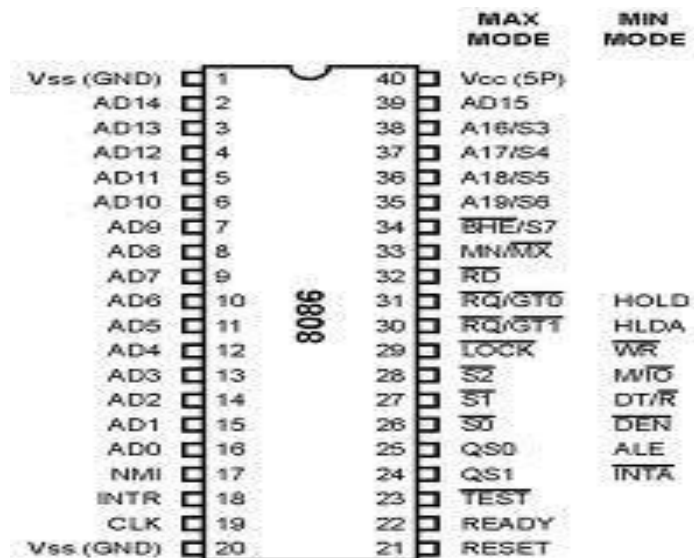
String Instruction:

These instructions involve various string manipulation operations like load, move, scan, compare, store .EX: MOVS, LODS and STOS.

EX: LODSW Instruction - Load string byte into AL or Load string word into AX

MOVSW Instruction - Move string byte or string word.

PIN DIAGRAM



- AD7-AD0 : address/data bus(multiplexed)
 - memory address or I/O port no : whenever ALE = 1
 - data : whenever ALE = 0
 - high-impedance state : during a hold acknowledge
- A15-A8 : address bus
 - high-impedance state : during a hold acknowledge
- AD15-AD8 : address/data bus(multiplexed)
 - memory address bits A15-A8 : whenever ALE = 1
 - data bits D15-D8 : whenever ALE = 0

- high-impedance state : during a hold acknowledge
- AD15-AD0-Address/Data Bus

- A19-A16-Address/Status
- A19/S6-A16/S3 : address/status bus(multiplexed)
 - memory address A19-A16, status bits S6-S3
 - high-impedance state : during a hold acknowledge
 - S6 : always remain a logic 0
 - S5 : indicate condition of IF flag bits
 - S4, S3 : show which segment is accessed during current bus cycle
 - S4, S3 : can used to address four separate 1M byte memory banks by decoding them as A21, A20
- TEST'(BUSY') : tested by the WAIT instruction
 - WAIT instruction function as a NOP : if TEST'= 0
 - WAIT instruction wait for TEST' to become 0:if TEST'=1
- INTR: Interrupt request
 - It is a level triggered input which is sampled during the last clock cycle of each instruction to determine the processor should enter in to interrupt vector look up table located in the system memory.
 - It is internally masked by software
- NMI : Non-maskable interrupt
 - similar to INTR except that no check IF flag bit
 - if NMI is activated : use interrupt vector 2
- RESET :
 - Reset signal active high for minimum of four clock cycles.
 - The signals terminate its present activity and the system is reset.
- CLK(CLOCK) : provide basic timing to μ
 - duty cycle of 33%

- VCC(power supply) : +5.0V, $\pm 10\%$
- GND(Ground) : two pins labeled GND
- MN/MX' : select either minimum or maximum mode
 - Low—Maximum
 - High--Minimum
- BHE'/S7 : bus high enable
 - Enable the most significant data bus bits(D15-D8) during read or write operation
 - Status of S7 : always a logic 1
- Minimum Mode Pins: MN = 1(directly to +5.0V) next p
- IO/M'(8088) or M/IO'(8086) : select memory or I/O
 - address bus : whether memory or I/O port address
- WR: write signal(high impedance state during hold ack.)
 - strobe that indicate that output data to memory or I/O
 - during WR'=0 : data bus contains valid data for M or I/O
- RD': Control signal read operation(it is an active low signal)
 - Depends upon the status of S2 pin it will read the memory of I/O or memory.

SIMPLE PROGRAMS

ADDITION OF TWO 16 BIT NUMBER

Mov AX,[2000]

ADD AX,[2002]

MOV [2004],AX

HLT

ADDITION OF TWO 32 BIT NUMBER

```
MOV    BL,00
MOV AX,[1700]
ADD  AX,[1704]
MOV[1800],AX
MOV AX,[1702]
ADC  AX,[1706]
JNC L1
INC BL
MOV [1802],AX
MOV[1804],BL
HLT
```

SUBTRACTION OF TWO 16 BIT NUMBER

```
MOV AX,[1500]
SUB  AX,[1502]
MOV [1800],AX
HLT
```

SUBTRACTION OF TWO 32 BIT NUMBER

```
MOV    BL,00
MOV AX,[1600]
SUB  AX,[1604]
MOV[1800],AX
MOV AX,[1602]
SBB  AX,[1606]
JNC L1
```

INC BL

MOV [1802],AX

MOV[1804],BL

HLT

MULTIPLICATION TWO 16 BIT NUMBER

MOV AX, [1200]

MUL AX,[1202]

MOV [1300],AX

MOV[1302],DX

HLT

DIVISION OF 32BIT NUMBER

MOV AX,[1600]

MOV DX,[1602]

MOV BX,[1604]

DIV BX

MOV [1500],AX

MOV[1502],DX

HLT

INTERFACING TRAFFIC LIGHT PROGRAM

MOV BX,1100

MOV CX,no of data

MOV AL,[BX]

OUT [PPICNT],AL

INC BX


```
MOV    AL,[BX]
```

```
OUT[PPIAPRT],AL
```

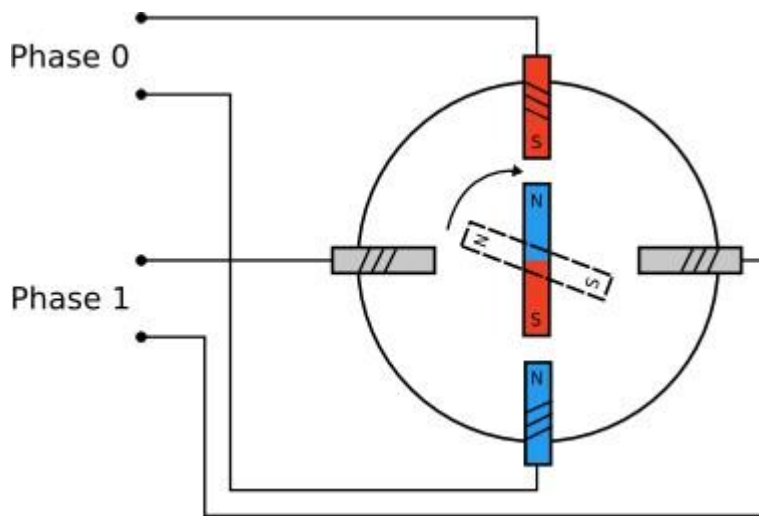
```
INC BX
```

```
MOV AL,[BX] OUT
```

```
[PPIBPRT],AL HLT
```

STEPPER MOTOR PROGRAM

Diagram:



Truth Table:

A ₁	A ₂	B ₁	B ₂	Hex Value
1	0	1	0	0A
0	1	1	0	06
0	1	0	1	05
1	0	0	1	09

```
1002 OUT 26,AL /*Move 80(Hex) to PPI, 26 is the CW of PPI*/
```

```
1004 MOV AL,0A /*Energizing the poles*/
```

```
1006 OUT 20,AL
```

```
1008 CALL 2000 /*Give some delay to rotate
```

```
Motor*/ 1010 MOV AL,06
```

```
1012 OUT 20,AL
```

```
1014 CALL 2000
```

```
1016 MOV AL,05
```

```
1018 OUT 20,AL
```

```
1020 CALL 2000
```

```
1022 MOV AL,09
```

```
1024 OUT 20,AL
```

```
1026 CALL 2000
```

```
1028 JMP 1004 /*Repeat the steps*/
```

```
1030 HLT
```

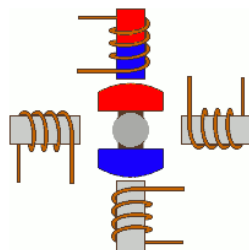
Stepper Motor

A Stepper Motor or a step motor is a brushless, synchronous motor which divides a full rotation into a number of steps. Unlike a DC motor which rotates continuously when a fixed DC voltage is applied to it, a step motor rotates in discrete step angles. This means, that it converts electrical power into mechanical power. The stepper motors also differs in the way they are powered. Instead of an AC or a DC voltage, they are driven (usually) with pulses. Each pulse is translated into a degree of rotation. The Stepper Motors therefore are manufactured with steps per revolution of 12, 24, 72, 144, 180, and 200, resulting in stepping angles of 30, 15, 5, 2.5, 2, and 1.8 degrees per step. The stepper motor can be controlled with or without feedback.

Working of Stepper motor

- There are 4 coils with 90° angle between each other fixed on the stator. The way that the coils are interconnected, will finally characterize the type of stepper motor connection, the coils are not connected together. The above motor has 90° rotation step. The coils are activated in a cyclic order, one by one. The rotation direction of the shaft is determined by the order that the coils are activated.

The coils are energized in series, with about 1sec interval. The shaft rotates 90° each time the next coil is activated. Figure below shows the conduction of stepper motors.



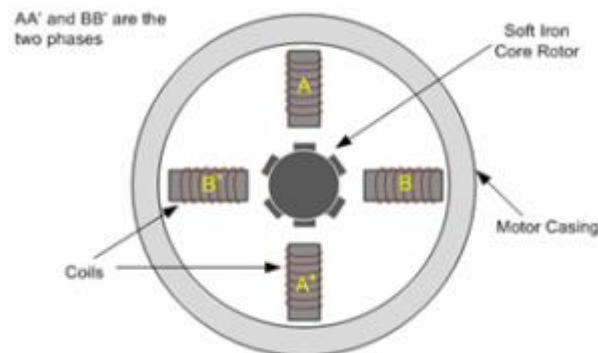
Types of stepper motors

1. Permanent magnet stepper
2. Hybrid synchronous stepper

3. Variable reluctance stepper

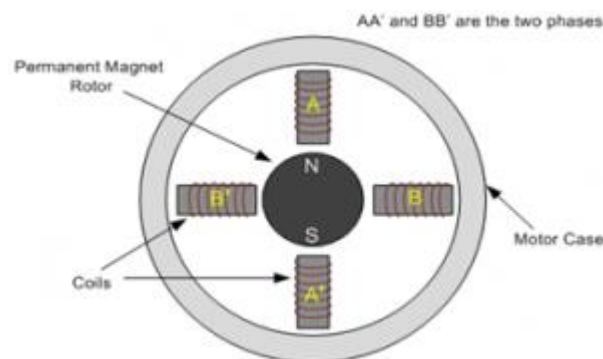
Variable Reluctance Motors

- The variable reluctance motor in the illustration has four “stator pole sets” (A, B, A', B'), set 15 degrees apart. Current applied to pole A through the motor winding causes a magnetic attraction that aligns the rotor (tooth) to pole A. Energizing stator pole B causes the rotor to rotate 15 degrees in alignment with pole B. This process will continue with pole A' and back to A in a clockwise direction. Reversing the procedure (B' to A) would result in a counter clockwise rotation. Figure below shows the Variable Reluctance Motors.



Permanent Magnet Motors

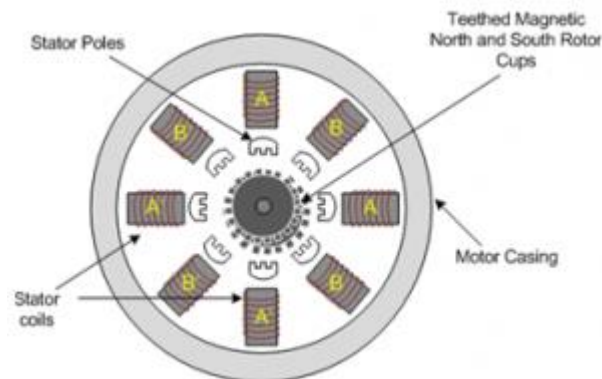
- The rotor and stator poles of a permanent magnet stepper are not teathed. Instead the rotor have alternative north and south poles parallel to the axis of the rotor shaft. When a stator is energized, it develops electromagnetic poles. The magnetic rotor aligns along the magnetic field of the stator. The other stator is then energized in the sequence so that the rotor moves and aligns itself to the new magnetic field. This way energizing the stators in a fixed sequence rotates the stepper motor by fixed angles. Figure below shows the Permanent Magnet Motors.



Hybrid Motors

- They are constructed with multi-toothed stator poles and a permanent magnet rotor. Standard hybrid motors have 200 rotor teeth and rotate at 1.80 step angles. Other hybrid motors are available in 0.9° and 3.6° step angle configurations. Because they exhibit high static and dynamic torque and run at very high step rates, hybrid motors are used in a wide variety of industrial applications. A hybrid stepper is a combination of both permanent magnet and the variable reluctance. It has a magnetic teathed rotor which better guides magnetic flux to preferred location in the air gap. The magnetic

rotor has two cups. One for north poles and second for the south poles. The rotor cups are designed so that the north and south poles arrange in alternative manner. Figure below shows the Hybrid Motors.



A stepper motor is an electromechanical device which converts electrical pulses into discrete mechanical movements. The shaft or spindle of a stepper motor rotates in discrete step increments when electrical command pulses are applied to it in the proper sequence. The motors rotation has several direct relationships to these applied input pulses. The sequence of the applied pulses is directly related to the direction of motor shafts rotation. The speed of the motor shafts rotation is directly related to the frequency of the input pulses and the length of rotation is directly related to the number of input pulses applied.

Advantages

- The rotation angle of the motor is proportional to the input pulse.
- The motor has full torque at stand- still (if the windings are energized)
- Precise positioning and repeat- ability of movement since good stepper motors have an accuracy of 3 – 5% of a step and this error is noncumulative from one step to the next.
- Excellent response to starting/ stopping/reversing.
- Very reliable since there are no con-tact brushes in the motor. Therefore the life of the motor is simply dependant on the life of the bearing.
- The motors response to digital input pulses provides open-loop control, making the motor simpler and less costly to control.
- It is possible to achieve very low speed synchronous rotation with a load that is directly coupled to the shaft.
- A wide range of rotational speeds can be realized as the speed is proportional to the frequency of the input pulses

Disadvantages

- Resonances can occur if not properly controlled.
- Not easy to operate at extremely high speeds.

Open Loop Operation

One of the most significant advantages of a stepper motor is its ability to be accurately controlled in an open loop system. Open loop control means no feedback information about position is needed. This type of control eliminates the need for expensive sensing and feedback devices such as optical encoders. Your position is known simply by keeping track of the input step pulses.

Stepper Motor Modes

Stepper motors are driven by waveforms which approximate sinusoidal waveforms. There are three excitation modes commonly used with stepper motors: full-step, half-step and micro stepping.

Stepper Motor – Full-Step (Two Phases are on)

In full-step operation, the stepper motor steps through the normal step angle, e.g. with a 200 step/revolution the motor rotates 1.8° per full step, while in half-step operation the motor rotates 0.9° per full step. There are two kinds of full-step modes which are single-phase full-step excitation and dual-phase full-step excitation. In single-phase full-step excitation, the stepper motor operates with only one phase energized at a time. This mode is typically used in applications where torque and speed performances are less important, wherein the motor operates at a fixed speed and load conditions are well defined. Typically, stepper motors are used in full-step mode as replacements in existing motion systems, and not used in new developments. Problems with resonance can prohibit operation at some speeds. This mode requires the least amount of power from the drive power supply of any of the excitation modes. In dual-phase full-step excitation, the stepper motor operates with two phases energized at a time. This mode provides excellent torque and speed performance with minimal resonance problems.

Stepper Motor – Half-Step

Stepper motor half-step excitation mode alternates between single and dual-phase operations resulting in steps that are half the normal step size. Therefore, this mode provides twice the resolution. While the motor torque output varies on alternate steps, this is more than offset by the need to step through only half the angle. This mode had become the predominately used mode by Anaheim Automation beginning in the 1970's, because it offers almost complete freedom from resonance issues. The stepper motor can operate over a wide range of speeds and drive almost any load commonly encountered. Although half-step drivers are still a popular and affordable choice, many newer micro stepping drivers are cost-effective alternatives. Anaheim Automation's BLD75 series is a popular half-step driver and is suitable for a wide range of stepper motors. With this driver, the customer only needs a transformer, as the other power supply components are built into the driver itself.

Stepper Motor – Micro stepping

In the stepper motor micro stepping mode, a stepper motor's natural step angle can be partitioned into smaller angles. For example: a conventional 1.8° motor has 200 steps per revolution. If the motor is micro stepped with a 'divide-by-10,' then each micro step moves the motor 0.18° , which becomes 2,000 steps per revolution. The micro steps are produced by proportioning the current in the two windings according to sine and cosine functions. This mode is widely used in applications requiring smoother motion or higher resolution. Typical micro step modes range from 'divide-by-10' to 'divide-by-256' (51,200 steps per revolution for a 1.8° motor). Some micro step drivers have a fixed divisor, while the more expensive micro step drivers provide for selectable divisors. For cost-effective micro step drivers.

Applications of Stepper motors

Aircraft – In the aircraft industry, stepper motors are used in aircraft instrumentations, antenna and sensing applications, and equipment scanning

- Automotive – The automotive industry implements stepper motors for applications concerning cruise control, sensing devices, and cameras. The military also utilizes stepper motors in their application of positioning antennas
- Chemical – The chemical industry makes use of stepper motors for mixing and sampling of materials. They also utilize stepper motor controllers with single and multi-axis stepper motors for equipment testing
- Consumer Electronics and Office Equipment – In the consumer electronics industry, stepper motors are widely used in digital cameras for focus and zoom functionality features. In office equipment, stepper motors are implemented in PC-based scanning equipment, data storage drives, optical disk drive driving mechanisms, printers, and scanners
- Gaming – In the gaming industry, stepper motors are widely used in applications like slot and lottery machines, wheel spinners, and even card shufflers
- Industrial – In the industrial industry, stepper motors are used in automotive gauges, machine tooling with single and multi-axis stepper motor controllers, and retrofit kits which make use of stepper motor controllers as well. Stepper motors can also be found in CNC machine control
- Medical – In the medical industry, stepper motors are utilized in medical scanners, microscopic or nanoscopic motion control of automated devices, dispensing pumps, and chromatograph auto-injectors. Stepper motors are also found inside digital dental photography (X-RAY), fluid pumps, respirators, and blood analysis machinery, centrifuge
- Scientific Instruments –Scientific equipment implement stepper motors in the positioning of an observatory telescope, spectrographs, and centrifuge
- Surveillance Systems – Stepper motors are used in camera surveillance.

PART A

1. Construct the format of Flag Register in 8086 Microprocessor with bit position
2. Compare the Minimum and Maximum mode signal in 8086.
3. Differentiate with examples (i)PUSH (ii) POP
4. Explain the following instructions:
i) MUL (ii) ADD (iii) ADC (iv) SUB (v) DIV
5. Define the following (i) JC & JNC instructions (ii) CALL and RET?
6. Grade the 8085 and 8086 Microprocessor.
7. Discriminate the Program counter and Stack Pointer in 8086 Microprocessor
8. Discuss an ALP for Addition of two 16 bits 8086 Microprocessor
9. Demonstrate an ALP for subtraction of two 16 bits in 8086.
10. Examine an ALP for Multiplication of two 16 bits in 8086
11. Evaluate ALP Division of two 16 bit using 8086.

PART B

1. Elaborate the organization of 8086 architecture with neat diagram.
2. Discuss the classification of 8086 instruction with examples.
3. Classify the various type of addressing modes used in 8086 with examples.
4. Construct an ALP for Stepper Motor Controller in 8085.
5. Create an ALP for Traffic Light controller in 8085.
6. Distinguish the branching instruction of 8086 microprocessor with examples.
7. Compare the 8086 data transfer instruction with adequate examples.
8. Interpret the pin details of 8086 and briefly explain its functions.



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – II – Advanced Microprocessors – SPHA5204

UNIT II ADVANCED PROCESSORS

Review of processor and its types-80286-80386-80486-Introduction to Pentium family-MMX architecture and instruction set- Multi core processor architecture.

ARCHITECTURE & FEATURES 80286

The 80286 is the first family of advanced microprocessors with memory management and protection mode. It is 16 bit processor with 134000 transistors. The CPU, with its 24-bit address bus is able to address up to 16 Mbytes of physical memory. It has 12.5 MHz, 10 MHz and 8 MHz clock frequencies. Average speed is 0.21 instructions per clock cycle. It compatible with 8086 in terms of instruction set. It has two operating modes namely real address mode and virtual address mode. In real address mode, the 80286 can address up to 1Mb of physical memory address. In virtual address mode, it can address up to 16 Mb of physical memory address space and 1 GB of virtual memory address space. The instruction set of 80286 includes the instructions of 8086 and 80186. It has some extra instructions to support operating system and memory management. In real address mode, the 80286 is object code compatible with 8086. In protected virtual address mode, it is source code compatible with 8086. It is five times faster than the standard 8086.

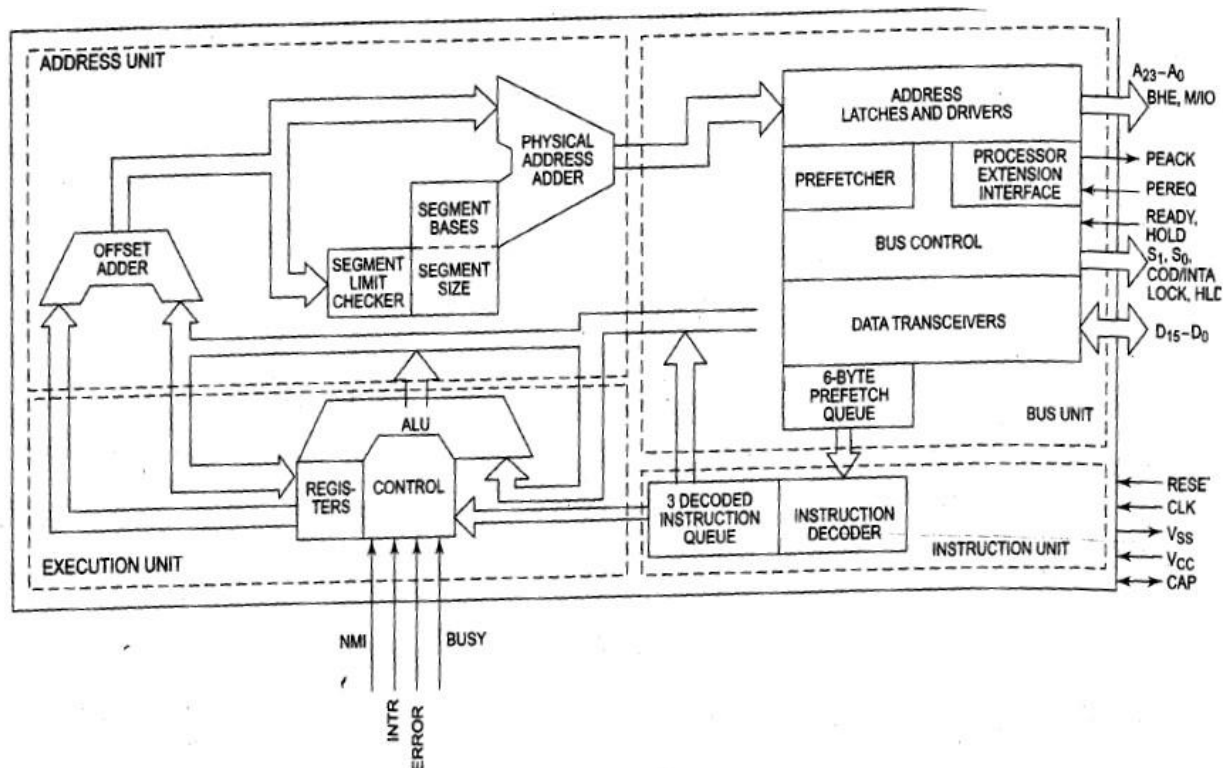


Fig:1.1: Architecture of 80286

Register of 80286

The 80286 contains almost the same set of registers, as in 8086, namely

1. Eight 16-bit general purpose registers
2. Four 16-bit segment registers
3. Status and control registers
4. Instruction Pointer

Functional blocks of 80286

1. Address Unit (AU)
2. Bus Unit (BU)
3. Instruction Unit (IU)
4. Execution Unit (EU)

Address Unit

The address unit is responsible for calculating the physical address (PA or MA) of instructions and data accessed. PA is 29 bit, offset is 16 bit. The address lines derived by this unit may be used to address different peripherals. The PA is handed over to the bus unit (BU) of the CPU.

Bus Unit

The bus unit has 16-bit data bus, 24-bit address bus and control bus. It performs all external operations. Latches and drivers of address bus transmit address from A19-A0 for read and write operations. It will fetch instruction bytes from the memory. Instructions are fetched in advance and stored in a queue to enable faster execution of the instructions. This concept is called instruction pipelining. It controls the prefetcher module. These prefetched instructions are arranged in a 6-byte instructions queue.

Instruction Unit

The 6-byte prefetch queue forwards the instructions arranged in it to the instruction unit (IU). The instruction unit accepts instructions from the prefetch queue and an instruction decoder decodes them one by one. The output of the decoding circuit drives a control circuit in the execution unit, which is responsible for executing the instructions received from the decoded instruction queue.

Execution Unit

The decoded instruction queue sends the data part of the instruction over the data bus. The EU contains the register bank used for storing the data as scratch pad, or used as special purpose registers. The ALU, the heart of the EU, carries out all the arithmetic and logical operations and sends the results over the data bus or back to the register bank.

Register

The architecture has fifteen registers. It is grouped in to four categories are listed below.

1. General purpose registers
2. Segment registers
3. Base and Index registers
4. Status and control registers

General purpose Registers

It is used to store arithmetic and logical operations. They are AX, BX, CX , DX can be used either as 16 bit words or split into pair of 8 bit registers.

Segment Registers

It is used to select the segment of memory that is immediately addressable for code, stack and data.

Base and Index Registers

General purpose register can also used to determine offset address of operand in memory. It holds base address or indexes to particular location within a segment.

Status and control registers:

It is a 16 bit special purpose register used for record and control of the 80286 processor. The instruction pointer contains the offset address of the next sequential instruction to be executed.

Flag Word Description

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
-----	-----	-----	-----	-----	-----	----	----	----	----	----	----	----	----	----	----

X	NT	IOP L	IOP L	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF
---	----	----------	----------	----	----	----	----	----	----	---	----	---	----	---	----

D3 2	D3 1	D3 0	D2 9	D2 8	D2 7	D2 6	D2 5	D2 4	D2 3	D2 2	D2 1	D2 0	D1 9	D1 8	D1 7	D1 6
X	X	X	X	X	X	X	X	X	X	X	X	X	TS	EM	MP	PE

Status Flag:

1. Carry Flag (CF) □ D0 set on carry/borrow produced ,when adding/subtracting bits.
2. Parity Flag (PF) □ D2 set on even number of ones.
3. Auxiliary carry Flag (AF) □ D4 set on carry or borrow to LSB to MSB otherwise cleared.
4. Zero Flag (ZF) □ Set on D6 is zero
5. Sign Flag (SF) □ Set on D7 is one
6. Overflow Flag (OF) □ Too large or small numbers.

Control Flag

1. Trap Flag (TF) □ Set, a single interrupts occurs after the next instruction executes.TF is cleared by the single step interrupt.
2. Interrupt Flag (IF) □ Set, Maskable interrupt will cause the CPU to transfer control to an interrupt vector specified location.
3. Direction Flag (DF) □ Causes string instruction to auto decrement the appropriate index registers when set. Clearing DF causes auto increment.

Machine Status Word

1. Task switch □ TS is set, This flag indicates the next instruction using extension will generate exception7, permitting the emulation of processor extension is for the current task.
2. Processor Extension (EM) □ The emulate processor extension flag causes a processor extension absent exception and permits the emulation of processor extension by CPU.
3. Monitor Processor Extension (MP) □ The monitor process extension flag allows WAIT instruction to generate a processor extension not present in the extension.
4. Protection enable (PE) □ Protection enable flag places the 80286 in protected mode, PE is set. This can only be cleared by resetting the CPU.

Other Flag

1. Nested Flag

2. I/O privilege level

Interrupt

The Interrupts of 80286 may be divided into three categories,

1. External or hardware interrupts
2. INT instruction or software interrupts
3. Interrupts generated internally by exceptions

While executing an instruction, the CPU may sometimes be confronted with a special situation because of which further execution is not permitted. While trying to execute a divide by zero instruction, the CPU detects a major error and stops further execution. In this case, we say that an exception has been generated. In other words, an instruction exception is an unusual situation encountered during execution of an instruction that stops further execution. The return address from an exception, in most of the cases, points to the instruction that caused the exception. As in the case of 8086, the interrupt vector table of 80286 requires 1Kbytes of space for storing 256, four-byte pointers to point to the corresponding 256 interrupt service routines (ISR). Each pointer contains a 16-bit offset followed by a 16-bit segment selector to point to a particular ISR. The calculation of vector pointer address in the interrupt vector table from the (8-bit) INT type is exactly similar to 8086. Like 8086, the 80286 supports the software interrupts of type 0 (INT 00) to type FFH (INT FFH).

Maskable Interrupt INTR

This is a maskable interrupt input pin of which the INT type is to be provided by an external circuit like an interrupt controller. The other functional details of this interrupt pin are exactly similar to the INTR input of 8086.

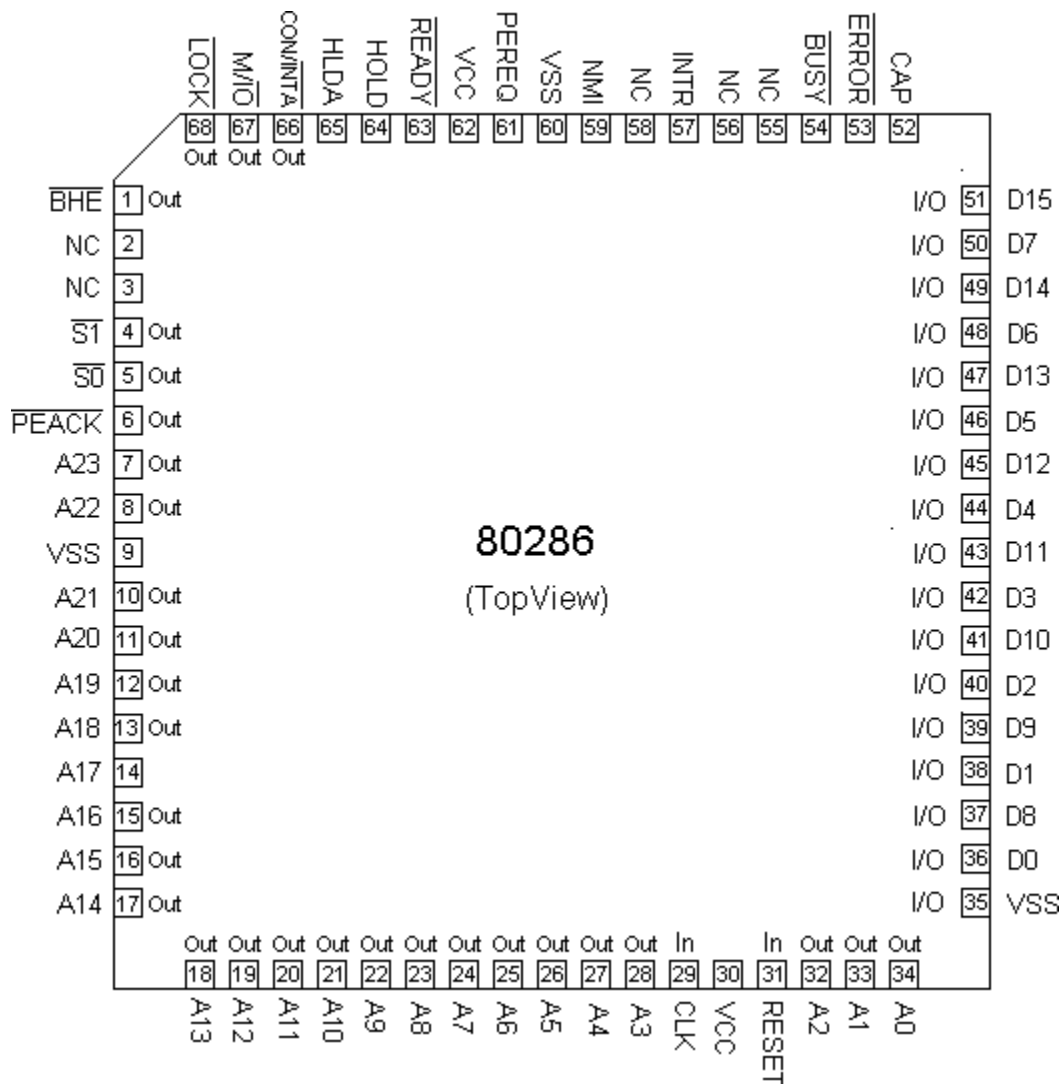
Non-Maskable Interrupt NMI

It has higher priority than the INTR interrupt. Whenever this interrupt is received, a vector value of 02 is supplied internally to calculate the pointer to the interrupt vector table. Once the CPU responds to a NMI request, it does not serve any other interrupt request (including NMI). Further it does not serve the processor extension (coprocessor) segment overrun interrupt, till either it executes IRET or it is reset. To start with, this clears the IF flag which is set again with the execution of IRET, i.e. return from interrupt.

Single Step Interrupt

As in 8086, this is an internal interrupt that comes into action, if trap flag (TF) of 80286 is set. The CPU stops the execution after each instruction cycle so that the register contents (including flag register), the program status word and memory, etc. may be examined at the end of each instruction execution. This interrupt is useful for troubleshooting the software. An interrupt vector type 01 is reserved for this interrupt.

Signal Description of 80286



CLK

This is the system clock input pin. The clock frequency applied at this pin is divided by two internally and is used for deriving fundamental timings for basic operations of the circuit. The clock is generated using 8284 clock generator.

D15-D0

These are sixteen bidirectional data bus lines.

A23-A0

These are the physical address output lines used to address memory or I/O devices. The address lines A23 - A16 are zero during I/O transfers

$\overline{\text{BHE}}$

This output signal, as in 8086, indicates that there is a transfer on the higher byte of the data bus (D15 – D8).

$\overline{\text{S}}_1, \overline{\text{S}}_0$

These are the active-low status output signals which indicate initiation of a bus cycle and with M/IO and COD/INTA, they define the type of the bus cycle.

$\text{M}/\overline{\text{IO}}$

This output line differentiates memory operations from I/O operations. If this signal is it “0” indicates that an I/O cycle or INTA cycle is in process and if it is “1” it indicates that a memory or a HALT cycle is in progress.

$\text{COD}/\overline{\text{INTA}}$

This output signal, in combination with M/ IO signal and S1 , S0 distinguishes different memory, I/O and INTA cycles.

$\overline{\text{LOCK}}$

This active-low output pin is used to prevent the other masters from gaining the control of the bus for the current and the following bus cycles. This pin is activated by a "LOCK" instruction prefix, or automatically by hardware during XCHG, interrupt acknowledge or descriptor table access

80C286 Bus Cycle Status Definition				
$\text{COD}/\overline{\text{INTA}}$	$\text{M}/\overline{\text{IO}}$	S_1	S_0	Bus Cycle
0(low)	0	0	0	Interrupt acknowledge
0	0	0	1	Will not occur
0	0	1	0	Will not occur
0	0	1	1	None; not a status cycle

0	1	0	0	IF A1 = 1 then halt; else shutdown
0	1	0	1	Memory data read
0	1	1	0	Memory data write
0	1	1	1	None; not a status cycle

80C286 Bus Cycle Status Definition				
COD/INTA	M/I/O	S1	S0	Bus Cycle
1(High)	0	0	0	Will not occur
1	0	0	1	I/O read
1	0	1	0	I/O write
1	0	1	1	None; not a status cycle
1	1	0	0	Will not occur
1	1	0	1	Will not occur Memory instruction read
1	1	1	0	Will not occur
1	1	1	1	None; not a status cycle

READY

This active-low input pin is used to insert wait states in a bus cycle, for interfacing low speed peripherals. This signal is neglected during HLDA cycle.

HOLD and HLDA

This pair of pins is used by external bus masters to request for the control of the system bus (HOLD) and to check whether the main processor has granted the control (HLDA) or not, in the same way as it was in 8086.

INTR :

Through this active high input, an external device requests 80286 to suspend the current instruction execution and serve the interrupt request. Its function is exactly similar to that of INTR pin of 8086.

NMI :

The Non-Maskable Interrupt request is an active-high, edge-triggered input that is equivalent to an INTR signal of type 2. No acknowledge cycles are needed to be carried out.

PEREQ and $\overline{\text{PEACK}}$ (Processor Extension Request and Acknowledgement)

Processor extension refers to coprocessor (80287 in case of 80286 CPU). This pair of pins extends the memory management and protection capabilities of 80286 to the processor extension 80287. The PEREQ input requests the 80286 to perform a data operand transfer for a processor extension. The PEACK active-low output indicates to the processor extension that the requested operand is being transferred.

 $\overline{\text{BUSY}}$ and $\overline{\text{ERROR}}$:

Processor extension both are active-low input signals indicate the operating conditions of a processor extension to 80286. The BUSY goes low, indicating 80286 to suspend the execution and wait until the BUSY become inactive. In this duration, the processor extension is busy with its allotted job. Once the job is completed the processor extension drives the $\overline{\text{BUSY}}$ input high indicating 80286 to continue with the program execution. An active $\overline{\text{ERROR}}$ signal causes the 80286 to perform the processor extension interrupt while executing the WAIT and ESC instructions. The active $\overline{\text{ERROR}}$ signal indicates to 80286 that the processor extension has committed a mistake and hence it is reactivating the processor extension .

CAP :

A 0.047 μf , 12V capacitor must be connected between this input pin and ground to filter the output of the internal substrate bias generator. For correct operation of 80286 the capacitor must be charged to its operating voltage. Till this capacitor charges to its full capacity, the 80286 may be

kept stuck to reset to avoid any spurious activity.

V_{ss} :

This pin is a system ground pin of 80286.

V_{cc} :

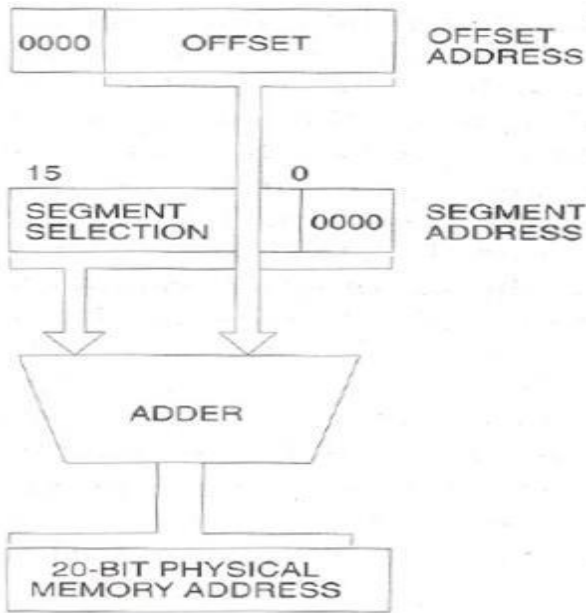
This pin is used to apply +5V power supply voltage to the internal circuit of 80286. RESET the active-high RESET input clears the internal logic of 80286, and reinitializes it

Reset

The active-high reset input pulse width should be at least 16 clock cycles. The 80286 requires at least 38 clock cycles after the trailing edge of the RESET input signal, before it makes the first opcode fetch cycle.

Real Address Mode

It act as a fast 8086. Instruction set is upwardly compatible. It address only 1 M byte of physical memory using A0-A19. In real addressing mode of operation of 80286, it just acts as a fast 8086. The instruction set is upward compatible with that of 8086. The 80286 addresses only 1Mbytes of physical memory using A0- A19. The lines A20-A23 are not used by the internal circuit of 80286 in this mode. In real address mode, while addressing the physical memory, the 80286 uses BHE along with A0- A19. The 20-bit physical address is again formed in the same way as that in 8086. The contents of segment registers are used as segment base addresses. The other registers, depending upon the addressing mode, contain the offset addresses. Because of extra pipelining and other circuit level improvements, in real address mode also, the 80286 operates at a much faster rate than 8086, although functionally they work in an identical fashion. As in 8086, the physical memory is organized in terms of segments of 64Kbyte. An exception is generated, if the segment size limit is exceeded by the instruction or the data. The overlapping of physical memory segments is allowed to minimize the memory requirements for a task. The 80286 reserves two fixed areas of physical memory for system initialization and interrupt vector table. In the real mode the first 1Kbyte of memory starting from address 0000H to 003FFH is reserved for interrupt vector table. Also the addresses from FFFF0H to FFFFFH are reserved for system initialization. The program execution starts from FFFFH after reset and initialization. The interrupt vector table of 80286 is organized in the same way as that of 8086. Some of the interrupt types are reserved for exceptions, single-stepping and processor extension segment. When the 80286 is reset, it always starts the execution in real address mode. In real address mode, it performs the following functions: it initializes the IP and other registers of 80286, it prepares for entering the protected virtual address mode.



Real Address Mode Address Calculation

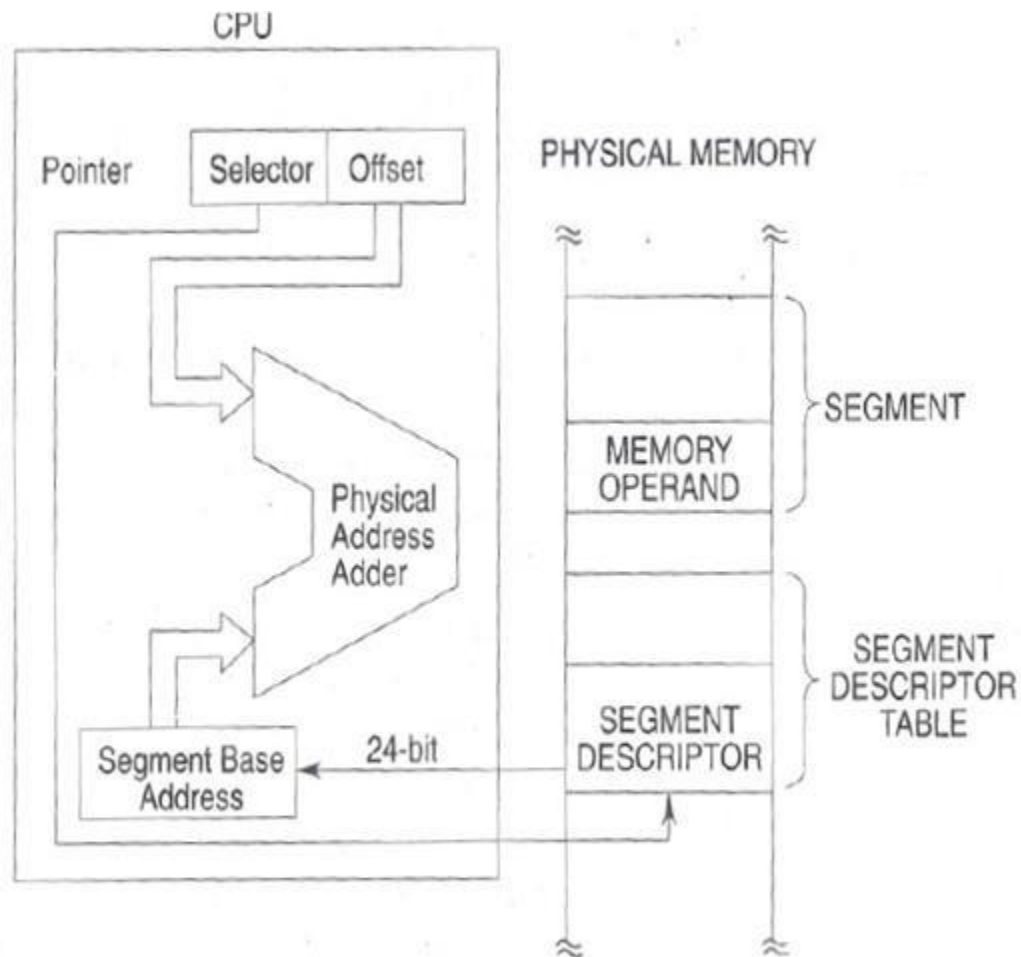
Protected virtual address mode (PVAM)

80286 is the first processor to support the concepts of virtual memory and memory management. The virtual memory does not exist physically it still appears to be available within the system. The concept of VM is implemented using Physical memory that the CPU can directly access and secondary memory that is used as storage for data and program, which are stored in secondary memory initially. The Segment of the program or data required for actual execution at that instant, is fetched from the secondary memory into physical memory. After the execution of this fetched segment, the next segment required for further execution is again fetched from the secondary memory, while the results of the executed segment are stored back into the secondary memory for further references. This continues till the complete program is executed. During the execution the partial results of the previously executed portions are again fetched into the physical memory, if required for further execution. The procedure of fetching the chosen program segments or data from the secondary storage into physical memory is called swapping. The procedure of storing back the partial results or data back on the secondary storage is called un swapping. The virtual memory is allotted per task. The 80286 is able to address 1 G byte of virtual memory per task. The complete virtual memory is mapped on to the 16Mbyte physical memory.

If a program is larger than 16Mbyte is stored on the hard disk and is to be executed, if it is fetched in terms of data or program segments of less than 16Mbyte in size into the program memory by swapping sequentially as per sequence of execution. Whenever the portion of a program is required for execution by the CPU, it is fetched from the secondary memory and placed in the physical memory is called swapping in of the program. A portion of the program or important partial results required for further execution , may be saved back on secondary storage to make the PM free for

further execution of another required portion of the program is called

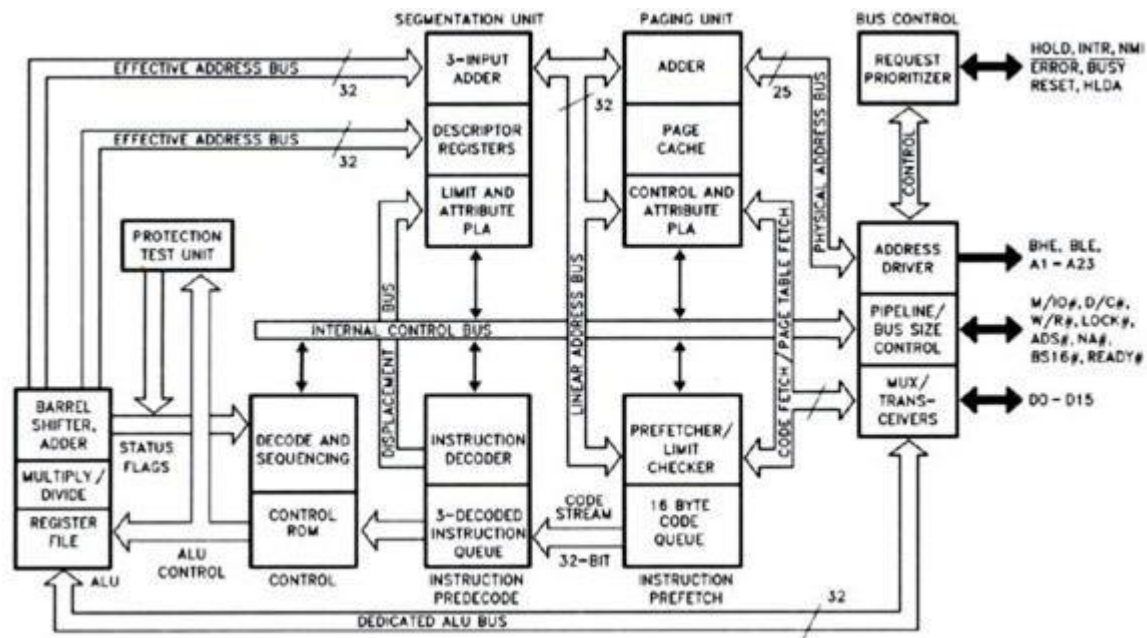
Swapping out of the executable program.



Physical Address Calculation in PVAM

80286 uses the 16-bit content of a segment register as a selector to address a descriptor stored in the physical memory. The descriptor is a block of contiguous memory locations containing information of a segment, like segment base address, segment limit, segment type, privilege level, segment availability in physical memory, descriptor type and segment use another task.

ARCHITECTURE OF 80386



Features of 80386:

This 80386 is a 32bit processor that supports 8bit/32bit data operands. The 80386-instruction set is upward compatible with all its predecessors. The 80386 can run 8086 applications under protected mode in its virtual 8086 mode of operation. With the 32 bit address bus, the 80386 can address up to 4Gbytes of physical memory. The physical memory is organized in terms of segments of 4Gbytes at maximum. The 80386 CPU supports 16K number of segments and thus the total virtual space of 4Gbytes * 16K = 64 Terabytes. The memory management section of 80386 supports the virtual memory, paging and four levels of protection, maintaining full compatibility with 80286. The 80386 offers a set of 8 debug registers DR 0-DR 7 for hardware debugging and control. The 80386 has on-chip address translation cache. The concept of paging is introduced in 80386 that enables it to organize the available physical memory in terms of pages of size 4Kbytes each, under the segmented memory. The 80386 can be supported by 80387 for mathematical data processing.

Architecture:

The Internal Architecture of 80386 is divided into 3 sections.

- Central processing unit –Two types=> Execution unit and Instruction unit
- Memory management unit
- Bus interface unit

Execution unit :

It has 8 General purpose and 8 Special purpose registers which are either used for handling data or calculating offset addresses.

Instruction unit:

It decodes the opcode bytes received from the 16-byte instruction code queue and arranges them in a 3- instruction decoded instruction queue. After decoding them pass it to the control section for deriving the necessary control signals. The barrel shifter increases the speed of all shift and rotate operations. The multiply / divide logic implements the bit-shift-rotate algorithms to complete the operations in minimum time. Even 32- bit multiplications can be executed within one microsecond by the multiply / divide logic.

Memory management unit :

The Memory management unit consists of

- Segmentation unit
- Paging unit.

Segmentation unit:

It allows the use of two address components, viz. segment and offset for relocability and sharing of code and data. The segments of size 4Gbytes. The Segmentation unit provides a 4 level protection mechanism for protecting and isolating the system code and data from those of the application program

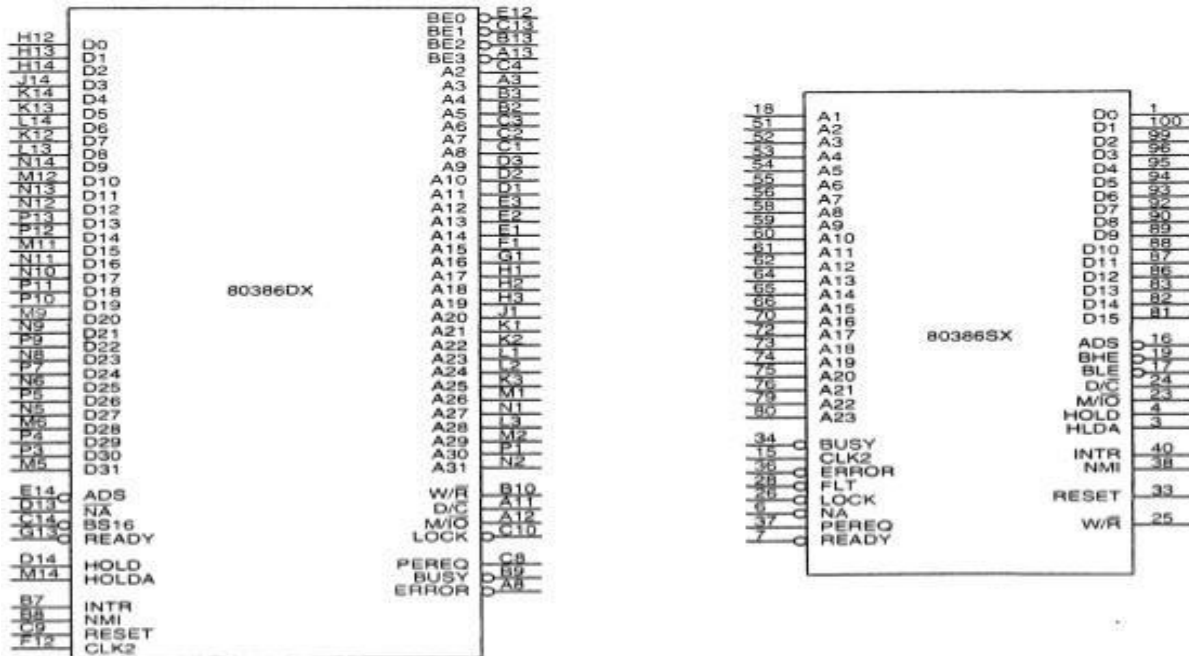
The Paging unit :

It organizes the physical memory in terms of pages of 4kbytes size each. It works under the control of the segmentation unit, i.e. each segment is further divided into pages. The virtual memory is also organizes in terms of segments and pages by the memory management unit. It converts linear addresses into physical addresses. The control and attribute PLA checks the privileges at the page level. Each of the pages maintains the paging information of the task. The limit and attribute PLA checks segment limits and attributes at segment level to avoid invalid accesses to code and data in the memory segments.

Bus control unit

It has a prioritize to resolve the priority of the various bus requests. which controls the access of the bus. The address driver drives the bus enable and address signal A0 – A31. The pipeline and dynamic bus sizing unit handle the related control signals. The data buffers interface the internal data bus with the system bus.

Pin Diagram:



Pin Description:

CLK2: The input pin provides the basic system clock timing for the operation of 80386.

D 0 – D31: These 32 lines act as bidirectional data bus during different access cycles

A31 – A 2: These are upper 30 bit of the 32- bit address bus.

BE0 to BE 3: The 32- bit data bus supported by 80386 and the memory system of 80386 can be viewed as a 4- byte wide memory access mechanism. The 4 byte enable lines BE 0 to BE 3, may be used for enabling these 4 blanks. Using these 4 enable signal lines, the CPU may transfer 1 byte / 2 / 3 / 4 byte of data simultaneously.

W/R#: The write / read output distinguish the write and read cycles from one another.

D/C#: This data / control output pin distinguishes between a data transfer cycle from a machine control cycle like interrupt acknowledge.

M/I/O#: This output pin differentiates between the memory and I/O cycles.

LOCK#: The LOCK# output pin enables the CPU to prevent the other bus masters from gaining

the control of the system bus.

NA#: The next address input pin, if activated, allows address pipelining, during 80386 bus cycles.

ADS#: The address status output pin indicates that the address bus and bus cycle definition pins (W/R#, D/C#, M/IO#, BE 0# to BE 3#) are carrying the respective valid signals. The 80383 does not have any ALE signals and so this signals may be used for latching the address to external latches.

READY#: The ready signals indicate to the CPU that the previous bus cycle has been terminated and the bus is ready for the next cycle. The signal is used to insert WAIT states in a bus cycle and is useful for interfacing of slow devices with CPU.

VCC: These are system power supply lines.

VSS: These return lines for the power supply

BS16#: The bus size – 16 input pin allows the interfacing of 16 bit devices with the 32 bit wide 80386 data bus. Successive 16 bit bus cycles may be executed to read a 32 bit data from a peripheral.

HOLD: The bus hold input pin enables the other bus masters to gain control of the system bus if it is asserted.

HLDA: The bus hold acknowledge output indicates that a valid bus hold request has been received and the bus has been relinquished by the CPU.

BUSY#: The busy input signal indicates to the CPU that the coprocessor is busy with the allocated task.

ERROR#: The error input pin indicates to the CPU that the coprocessor has encountered an error while executing its instruction.

PEREQ: The processor extension request output signal indicates to the CPU to fetch a data word for the coprocessor.

INTR: This interrupt pin is a maskable interrupt, that can be masked using the IF of the flag register.

NMI: A valid request signal at the non-maskable interrupt request input pin internally generates a non- maskable interrupt of type2.

RESET: A high at this input pin suspends the current operation and restart the execution from the starting location.

N / C : No connection pins are expected to be left open while connecting the 80386 in the circuit.

Register Organization:

The 80386 has eight 32 - bit general purpose registers which may be used as either 8 bit or 16 bit registers. 32 - bit register known as an extended register, is represented by the register name with prefix E. Example: A 32 bit register corresponding to AX is EAX, similarly BX is EBX etc. The 16 bit registers BP, SP, SI and DI in 8086 are now available with their extended size of 32 bit and are names as EBP, ESP, ESI and EDI. AX represents the lower 16 bit of the 32 bit register EAX. BP, SP, SI, DI represents the lower 16 bit of their 32 bit counterparts, and can be used as independent 16 bit registers. The six segment registers available in 80386 are CS, SS, DS, ES, FS and GS. The CS and SS are the code and the stack segment registers respectively, while DS, ES, FS, GS are 4 data segment registers. The 16 bit instruction pointer IP is available along with 32 bit counterpart EIP.

Flag Register of 80386: It is a 32 bit register. Out of the 32 bits, Intel has reserved bits D18 to D31, D 5 and D 3, while D1 is always set at 1. Two extra new flags are added to the 80286 flag to derive the flag register of 80386. They are VM and RF flags.

D15	D14	D13 D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16	D15
															VM	RF

D31-D17 reserved for Intel

VM - Virtual Mode Flag:

If this flag is set, the 80386 enters the virtual 8086 mode within the protection mode. This is to be set only when the 80386 is in protected mode. In this mode, if any privileged instruction is executed an exception 13 is generated. This bit can be set using IRET instruction or any task switch operation only in the protected mode.

RF- Resume Flag:

This flag is used with the debug register breakpoints. It is checked at the starting of every instruction cycle and if it is set, any debug fault is ignored during the instruction cycle. The RF is automatically reset after successful execution of every instruction, except for IRET and POPF instructions. Also, it is not automatically cleared after the successful execution of JMP, CALL and INT instruction causing a task switch. These instructions are used to set the RF to the value specified by the memory data available at the stack.

Segment Descriptor Registers:

This registers are not available for programmers, rather they are internally used to store the descriptor information, like attributes, limit and base addresses of segments. The six segment registers have corresponding six 73 bit descriptor registers. Each of them contains 32 bit base address, 32 bit base limit and 9 bit attributes. These are automatically loaded when the corresponding segments are loaded with selectors.

Control Registers:

The 80386 has three 32 bit control registers (CR), CR 2 and CR 3 to hold global machine status independent of the executed task. Load and store instructions are available to access these registers. System Address Registers: Four special registers are defined to refer to the descriptor tables supported by 80386. The 80386 supports four types of descriptor table, viz. global descriptor table (GDT), interrupt descriptor table (IDT), local descriptor table (LDT) and task state segment descriptor (TSS).

Debug and Test Registers: Intel has provide a set of 8 debug registers for hardware debugging. Out of these eight registers DR 0 to DR 7, two registers DR 4 and DR 5 are Intel reserved. The initial four registers DR 0 to DR 3 store four program controllable breakpoint addresses, while DR 6 and DR 7 respectively hold breakpoint status and breakpoint control information. Two more test register are provided by 80386 for page caching namely test control and test status register.

ADDRESSING MODES:

The 80386 supports overall eleven addressing modes to facilitate efficient execution of higher level language programs. In case of all those modes, the 80386 can now have 32-bit immediate or 32- bit register operands or displacements. The 80386 has a family of scaled modes. In case of scaled modes, any of the index register values can be multiplied by a valid scale factor to obtain the displacement. The valid scale factor are 1, 2, 4 and 8.

The different scaled modes are as follows. Scaled Indexed Mode: Contents of the index register are multiplied by a scale factor that may be added further to get the operand offset.

Based Scaled Indexed Mode:

Contents of the index register are multiplied by a scale factor and then added to base register to obtain the offset. Based Scaled Indexed Mode with Displacement: The Contents of the index register are multiplied by a scaling factor and the result is added to a base register and a

displacement to get the offset of an operand. After reset, the 80386 starts from memory location FFFFFFF0H under the real address mode. In the real mode, 80386 works as a fast 8086 with 32-bit registers and data types. In real mode, the default operand size is 16 bit but 32-bit operands and addressing modes may be used with the help of override prefixes. The segment size in real mode is 64k, hence the 32-bit effective addressing must be less than 0000FFFFH. The real mode initializes the 80386 and prepares it for protected mode.

Memory Addressing in Real Mode: In the real mode, the 80386 can address at the most 1Mbytes of physical memory using address lines A 0-A19. Paging unit is disabled in real addressing mode, and hence the real addresses are the same as the physical addresses. To form a physical memory address, appropriate segment registers contents (16-bits) are shifted left by four positions and then added to the 16-bit offset address formed using one of the addressing modes, in the same way as in the 80386 real address modes. The segment in 80386 real mode can be read, write or executed, i.e. no protection is available. Any fetch or access past the end of the segment limit generates exception 13 in real address mode. The segments in 80386 real mode may be overlapped or no overlapped. The interrupt vector table of 80386 has been allocated 1Kbyte space starting from 00000H to 003FFH.

Protected Mode of 80386

All the capabilities of 80386 are available for utilization in its protected mode of operation. The 80386 in protected mode support all the software written for 80286 and 8086 to be executed under the control of memory management and protection abilities of 80386. The protected mode allows the use of additional instruction, addressing modes and capabilities of 80386.

Addressing in protected mode:

In this mode, the contents of segment registers are used as selectors to address descriptors which contain the segment limit, base address and access rights byte of the segment. The effective address (offset) is added with segment base address to calculate linear address. This linear address is further used as physical address, if the paging unit is disabled. Otherwise the paging unit converts the linear address into physical address. The paging unit is a memory management unit enabled only in protected mode. The paging mechanism allows handling of large segments of memory in terms of pages of 4Kbyte size. The paging unit operates under the control of segmentation unit. The paging unit if enabled converts linear addresses into physical address, in protected mode.

Segmentation

Descriptor tables: These descriptor tables and registers are manipulated by the operating system to ensure the correct operation of the processor, and hence the correct execution of the program. Three types of the 80386 descriptor tables are listed as follows:

- Global Descriptor Table (GDT)
- Local Descriptor Table (LDT)
- Interrupt Descriptor Table (IDT)

Descriptors:

The 80386 descriptors have a 20-bit segment limit and 32-bit segment address. The descriptor of 80386 are 8-byte quantities access right or attribute bits along with the base and limit of the segments. Descriptor Attribute Bits: The A (accessed) attributed bit indicates whether the segment has been accessed by the CPU or not. The TYPE field decides the descriptor type and hence the segment type. The S bit decides whether it is a system descriptor (S=0) or code/data segment descriptor (S=1).The DPL field specifies the descriptor privilege level. The D bit specifies the code segment operation size. If D=1, the segment is a 32-bit operand segment, else, it is a 16-bit operand segment. The P bit (present) signifies whether the segment is present in the physical memory or not. If P=1, the segment is present in the physical memory. The G (granularity) bit indicates whether the segment is page addressable. The zero bit must remain zero for compatibility with future process.

The AVL (available) field specifies whether the descriptor is for user or for operating system. • The 80386 has five types of descriptors listed as follows:

1. Code or Data Segment Descriptors.
2. System Descriptors.
3. Local descriptors.
4. TSS (Task State Segment) Descriptors.
5. GATE Descriptors.

The 80386 provides a four level protection mechanism exactly in the same way as the 80286 does.

Paging operation:

Paging is one of the memory management techniques used for virtual memory multitasking operating system. The segmentation scheme may divide the physical memory into a variable size segments but the paging divides the memory into a fixed size pages. The segments are supposed to be the logical segments of the program, but the pages do not have any logical relation with the program. The pages are just fixed size portions of the program module or data. The advantage of paging scheme is that the complete segment of a task need not be in the physical memory at any time. Only a few pages of the segments, which are required currently for the execution, need to be available in the physical memory. Thus the memory requirement of the task is substantially reduced, relinquishing the available memory for other tasks. Whenever the other pages of task are

required for execution, they may be fetched from the secondary storage. The previous pages which are executed need not be available in the memory, and hence the space occupied by them may be relinquished for other tasks.

Paging Descriptor Base Register:

The control register CR2 is used to store the 32-bit linear address at which the previous page fault was detected. The CR 3 is used as page directory physical base address register, to store the physical starting address of the page directory. The lower 12 bit of the CR3 are always zero to ensure the page size aligned directory. A move operation to CR 3 automatically loads the page table entry caches and a task switch operation, to load CR 0 suitably.

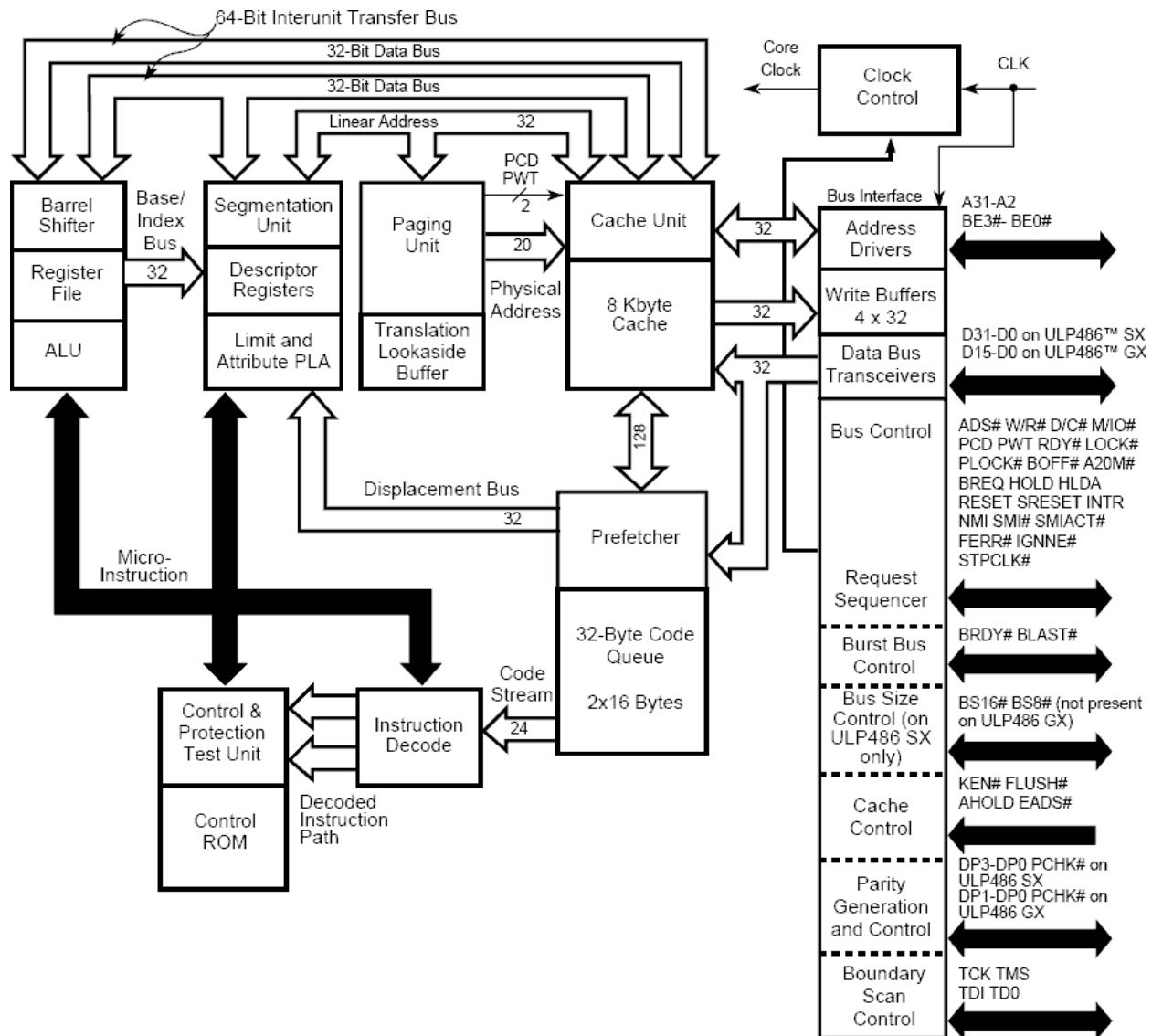
Page Directory:

This is at the most 4Kbytes in size. Each directory entry is of 4 bytes, thus a total of 1024 entries are allowed in a directory. The upper 10 bits of the linear address are used as an index to the corresponding page directory entry. The page directory entries point to page tables.

Page Tables:

Each page table is of 4Kbytes in size and many contain a maximum of 1024 entries. The page table entries contain the starting address of the page and the statistical information about the page. The upper 20 bit page frame address is combined with the lower 12 bit of the linear address. The address bits A12- A21 are used to select the 1024 page table entries. The page table can be shared between the tasks. The P bit of the above entries indicates, if the entry can be used in address translation. If P=1, the entry can be used in address translation, otherwise it cannot be used. The P bit of the currently executed page is always high. The accessed bit A is set by 80386 before any access to the page. If A=1, the page is accessed, else unaccessed. The D bit is set before a write operation to the page is carried out. The D-bit is undefined for page director entries. The OS reserved bits are defined by the operating system software. The User / Supervisor (U/S) bit and read/write bit are used to provide protection. These bits are decoded to provide protection under the 4 level protection models. The level 0 is supposed to have the highest privilege, while the level 3 is supposed to have the least privilege. This protection provide by the paging unit is transparent to the segmentation unit.

ARCHITECTURE OF 80486



It is 32 bit processor. One of the most obvious feature included in a 80486 is a built in math coprocessor. This coprocessor is essentially the same as the 80387 processor used with a 80386, but being integrated on the chip allows it to execute math instructions about three times as fast as a 80386/387 combination. 80486 is an 8Kbyte code and data cache. To make room for the additional signals, the 80486 is packaged in a 168 pin, pin grid array package instead of the 132 pin PGA used for the 80386.

Flag Register of 80486:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF	

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16	D15
															VM	RF

31 18 17 16 Reserved for INTEL E F L A G Flags

CF: Carry Flag

AF: Auxiliary carry

ZF: Zero Flag

SF : Sign Flag

TF : Trap Flag

IE : Interrupt Enable

DF : Direct Flag

OF : Over Flow

IOPL : I/O Privilege Level

NT : Nested Task Flag

RF : Resume Flag

VM : Virtual Mode

AC : Alignment Check

The memory system for the 486 is identical to 386 microprocessor. The 486 contains 4G bytes of memory beginning at location 00000000H and ending at FFFFFFFFH. The major change to the memory system is internal to 486 in the form of 8K byte cache memory, which speeds the execution of instructions and the acquisition of data. Another addition is the parity checker/generator built into the 80486 microprocessor. Parity Checker / Generator : Parity is often used to determine if data are correctly read from a memory location. INTEL has incorporated an internal parity generator / decoder.

Parity is generated by the 80486 during each write cycle. Parity is generated as even parity and a parity bit is provided for each byte of memory. The parity check bits appear on pins DP0-DP3, which are also parity inputs as well as parity outputs. These are typically stored in memory during each write cycle and read from memory during each read cycle. On a read, the microprocessor checks parity and generates a parity check error, if it occurs on the PCHK# pin. A parity error causes no change in processing unless the user applies the PCHK signal to an interrupt input.

Interrupts are often used to signal a parity error in DS-based computer systems. This is same as 80386, except the parity bit storage. If parity is not used, Intel recommends that the DP0 – DP3 pins be pulled up to +5v.

Cache memory: The cache memory system stores data used by a program and also the instructions of the program. The cache is organized as a 4 way set associative cache with each location containing 16 bytes or 4 double words of data.

Control register CR0: It is used to control the cache with two new control bits not present in the 80386 microprocessor. The CD (cache disable) , NW (non-cache write through) bits are new to the 80486 and are used to control the 8K byte cache. If the CD bit is a logic 1, all cache operations are inhibited. This setting is only used for debugging software and normally remains cleared. The NW bit is used to inhibit cache write through operation. As with CD, cache write through is inhibited only for testing. For normal operations CD = 0 and NW = 0. Because the cache is new to 80486 microprocessor and the cache is filled using burst cycle not present on the 386.

The 80486 contains the same memory-management system as the 80386. This includes a paging unit to allow any 4K byte block of physical memory to be assigned to any 4K byte block of linear memory. The only difference between 80386 and 80486 memory-management system is paging. The 80486 paging system can disabled caching for section of translation memory pages, while the 80386 could not. If these are compared with 80386 entries, the addition of two new control bits is observed (PWT and PCD). The page write through and page cache disable bits control caching.

The PWT controls how the cache functions for a write operation of the external cache memory. It does not control writing to the internal cache. The logic level of this bit is found on the PWT pin of the 80486 microprocessor. Externally, it can be used to dictate the write through policy of the external caching. The PCD bit controls the on-chip cache. If the PCD = 0, the on-chip cache is enabled for the current page of memory. Note that 80386 page table entries place a logic 0 in the PCD bit position, enabling caching. If PCD = 1, the on-chip cache is disable. Caching is disable regard less of condition of KEN#, CD, and NW.

Test registers : TR3,TR5

Cache data register (TR3):

It is used to access either the cache fill buffer for a write test operation or the cache read buffer for a cache read test operation. In order to fill or read a cache line (128 bits wide), TR3 must be written or read four times. The contents of the set select field in TR5 determine which internal cache line is written or read through TR3. The 7 bit test field selects one of the 128 different 16 byte wide cache lines. The entry select bits of TR5 select an entry in the set or the 32 bit location in the read buffer.

Control bits in TR5:

It is used to enable the

- Fill buffer or read buffer operation (00).
- Perform a cache write (01),
- Perform a cache read (10)
- Flush the cache (11).

The cache status register (TR4) hold the cache tag, LRU bits and a valid bit. This register is loaded with the tag and valid bit before a cache a cache write operation and contains the tag, valid bit, LRU bits, and 4 valid bits on a cache test read. Cache is tested each time that the microprocessor is reset if the AHOLD pin is high for 2 clocks prior to the RESET pin going low. This causes the 486 to completely test itself with a built in self test or BIST. The BIST uses TR3, TR4, TR5 to completely test the internal cache. Its outcome is reported in register EAX. If EAX is a zero, the microprocessor, the coprocessor and cache have passed the self test. The value of EAX can be tested after reset to determine if an error is detected. In most of the cases we do not directly access the test register unless we wish to perform our own tests on the cache or TLB.

Pin Definitions :

A 31-A2 : Address outputs A31-A2 provide the memory and I/O with the address during normal operation. During a cache line invalidation A31-A4 are used to drive the microprocessor.

A20 : The address bit 20 mask causes the 80486 to wrap its address around from location 000FFFFFFH to 00000000H as in 8086. This provides a memory system that functions like the 1M byte real memory system in the 8086 processors.

ADS: The address data strobe becomes logic zero to indicate that the address bus contains a valid memory address.

AHOLD: The address hold input causes the microprocessor to place its address bus connections at their high-impedance state, with the remainder of the buses staying active. It is often used by another bus master to gain access for a cache invalidation cycle.

BREQ: This bus request output indicates that the 486 has generated an internal bus request. BE 3-BE 0 : Byte enable outputs select a bank of the memory system when information is transferred between the microprocessor and its memory and I/O. The BE 3 signal enables D31 – D24 , BE2 enables D23-D16, BE1 enables D15 – D8 and BE 0 enables D 7-D 0

BLAST: The burst last output shows that the burst bus cycle is complete on the next activation of BRDY# signal

BOFF : The Back-off input causes the microprocessor to place its buses at their high impedance state during the next cycle. The microprocessor remains in the bus hold state until the BOFF# pin is placed at a logic 1 level.

NMI : The non-maskable interrupt input requests a type 2 interrupt.

BRDY : The burst ready input is used to signal the microprocessor that a burst cycle is complete.

KEN : The cache enable input causes the current bus to be stored in the internal.

LOCK : The lock output becomes a logic 0 for any instruction that is prefixed with the lock prefix.

W / R : current bus cycle is either a read or a write.

IGNNE : The ignore numeric error input causes the coprocessor to ignore floating point error and to continue processing data. The signal does not affect the state of the FERR pin.

FLUSH : The cache flush input forces the microprocessor to erase the contents of its 8K byte internal cache

EADS: The external address strobe input is used with AHOLD to signal that an external address is used to perform a cache invalidation cycle.

FERR : The floating point error output indicates that the floating point coprocessor has detected an error condition. It is used to maintain compatibility with DOS software.

BS 8 : The bus size 8, input causes the 80486 to structure itself with an 8-bit data bus to access byte-wide memory and I/O components.

BS16: The bus size 16, input causes the 80486 to structure itself with an 16-bit data bus to access word-wide memory and I/O components.

PCHK : The parity check output indicates that a parity error was detected during a read operation on the DP 3 – DP 0 pin.

PLOCK : The pseudo-lock output indicates that current operation requires more than one bus cycle to perform. This signal becomes a logic 0 for arithmetic coprocessor operations that access

64 or 80 bit memory data.

PWT: The page write through output indicates the state of the PWT attribute bit in the page table entry or the page directory entry.

RDY : The ready input indicates that a non-burst bus cycle is complete. The RDY signal must be returned or the microprocessor places wait states into its timing until RDY is asserted.

M / IO : Memory / IO defines whether the address bus contains a memory address or an I/O port number. It is also combined with the W/ R signal to generate memory and I/O read and write control signals.

The 80486 data bus, address bus, byte enable, ADS#, RDY#, INTR, RESET, NMI, M/IO#, D/C#, W/R#, LOCK#, HOLD, HLDA and BS16# signals function as we described for 80386. • The 80486 requires 1 clock instead of 2 clock required by 80386. A new signal group on the 486 is the PARITY group DP 0-DP 3 and PCHK#. These signals allow the 80486 to implement parity detection / generation for memory reads and memory writes. During a memory write operation, the 80486 generates an even parity bit for each byte and outputs these bits on the DP 0-DP3 lines.

A normal 80486 memory read operation to read a line into the cache requires 2 clock cycles. However, if a series of reads is being done from successive memory locations, the reads can be done in burst mode with only 1 clock cycle per read. To start the process the 80486 sends out the first address and asserts the BLAST# signal high. When the external DRAM controller has the first data bus, it asserts the BRDY# signal. The 80486 reads the data word and outputs the next address. Since the data words are at successive addresses, only the lower address bits need to be changed. If the DRAM controller is operating in the page or the static column modes then it will only have to output a new column address to the DRAM.

In this mode the DRAM will be able to output the new data word within 1 clock cycle. When the processor has read the required number of data words, it asserts the BLAST# signal low to terminate the burst mode. The final signal we want to discuss here are the bus request output signal BREQ, the back-off input signal BOFF#, the HOLD signal and the hold-acknowledge signal HLDA. These signals are used to control sharing the local 486 bus by multiple processors (bus master). When a master on the bus need to use the bus, it asserts its BERQ signal .An external parity circuit will evaluate requests to use the bus and grant bus use to the highest – priority master. To ask the 486 to release the bus, the bus controller asserts the 486 HOLD input or BOFF# input. If the HOLD input is asserted, the 486 will finish the current bus cycle, float its buses and assert the HLDA signal. To prevent another master from taking over the bus during a critical operation, the 486 can assert its LOCK# or PLOCK# signal.

The extended flag register EFLAG is illustrated in the figure. The only new flag bit is the AC alignment check, used to indicate that the microprocessor has accessed a word at an odd address or a double word boundary. Efficient software and execution require that data be stored at word or double word boundaries.

MMX instruction set:

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
Unpack	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		

MMX Instructions

MMX is widely supported: Intel Pentium MMX, Pentium 2+, Celeron, AMD K6, Athlon, Duron, and higher ([PENT,MMX] in NASM docs).

Extended MMX instructions are supported by newer processors: Pentium 3+, Celeron 2+, Athlon and Duron ([KATMAI,MMX] in NASM docs).

- 8 64-bit registers, MM0 to MM7.
- Registers represent arrays of 8 bytes, 4 words, or 2 dwords.
- Used for high speed, low precision integer vector operations (such as for image and signal processing).
- Registers overlap FPU registers ST(0) to ST(7)

Instruction Set

```

                                PADDUSW mm1, mm2
                                ^ ^ ^ ^ ^ ^
                                | | | | |
P = packed (all MMX instructions) -----+ | | | |
Instruction (add) -----+ | | | |
S = saturation, US = unsigned saturation -----+ | | |
B, W, D, Q = 8 bytes, 4 words, 2 dword, or 1 qword ----+ | |
                                                         | |

```



```

Destination is an MMX register, mm0-mm7 -----+ |
Source is an MMX register or 64 bit memory location -----+

```

Saturation means that overflows are handled by replacing with the closest representable value, e.g. 0 to 255 for unsigned byte saturation, -32768 to 32767 for signed word, etc.

Unless noted, all instructions take two operands, a and b.
a is mm0-mm7, b is mm0-mm7 or a 64 bit memory location.
a[0], b[0] means the low order byte, word, or dword of a or b.
x means B W D or Q indicating 8, 16, 32, or 64 bit size of input elements.
r means eax, ebx, ecx, edx, esi, edi, esp, ebp.
mm means one of mm0-mm7.
mem means memory in any addressing mode (e.g. qword ptr [eax+ebx*8+offset])
i means an immediate constant (0, 1, 2...).
* means extended MMX, not supported on some older processors.

Instruction	Input Sizes	Notes

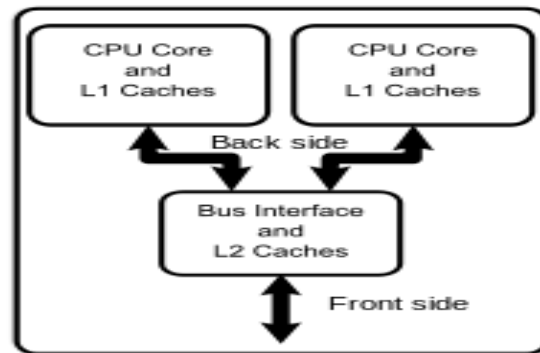
; Required FPU reset after any MMX instruction before FPU instructions		
EMMS		No operands
; Move		
MOVQ	Q	a = b, a may be memory if b is mm0-7
MOVNTQ mem, mm	* Q	a = b, fast (non temporal) store
MOVD a, mm	D	a = mm[0], a is a 32 bit register or memory
MOVD mm, b	D	to Q, mm = {b, 0}, b is a 32 bit register or memory
; Parallel arithmetic		
PADDx	B W D	a += b, discard carry
PADDsx	B W	a += b, signed saturation
PADDUSx	B W	a += b, unsigned saturation
PSUBx	B W D	a -= b, discard borrow
PSUBsx	B W	a -= b, signed saturation
PSUBUSx	B W	a -= b, unsigned saturation
PMULLW	W	a = (a * b) & 0xffff
PMULHW	W	a = (a * b) >> 16, signed
PMULHUW	* W	a = (a * b) >> 16, unsigned
; Complex multiply or vector product		
PMADDWD	W	to D, a = {a[0]*b[0]+a[1]*b[1], a[2]*b[2]+a[3]*b[3]}, signed
; Parallel compare, conditional store		
PCMPEQx	B W D	a = -(a == b)
PCMPGTx	B W D	a = -(a > b), signed
PMOVMskB r, mm	* B	r = (mm < 0), 8 bits, zero extended
MASKMOVQ	* B	if (b[i] < 0) [edi+i] = a[i], b must be mm0-7
; Logical (element boundaries are irrelevant)		
PAND	Q	a &= b
PANDN	Q	a = ~a & b
POR	Q	a = b
PXOR	Q	a ^= b
; Shift		

PSLLx	W D Q	a <=> b[0], b may also be 0..63
PSRLx	W D Q	a >=> b[0], unsigned, b may also be 0..63
PSRax	W D	a >=> b[0], signed, b may also be 0..31
; Pack to smaller type		
PACKSSWB	W	to B, a = {a[0]..a[3],b[0]..b[3]}, signed saturation
PACKUSWB	W	to B, a = {a[0]..a[3],b[0]..b[3]}, unsigned saturation
PACKSSDW	D	to W, a = {a[0], a[1],b[0], b[1]}, signed saturation
; Unpack		
PUNPCKLBW	B	a = {a[0],b[0]...a[3],b[3]}
PUNPCKHBW	B	a = {a[4],b[4]...a[7],b[7]}
PUNPCKLWD	W	a = {a[0],b[0],a[1],b[1]}
PUNPCKHWD	W	a = {a[2],b[2],a[3],b[3]}
PUNPCKLDQ	D	a = {a[0],b[0]}
PUNPCKHDQ	D	a = {a[1],b[1]}
; Parallel min, max, average		
PMINUB	* B	a = min(a, b), unsigned
PMAXUB	* B	a = max(a, b), unsigned
PMINSW	* W	a = min(a, b), signed
PMAXSW	* W	a = max(a, b), signed
PAVGx	* B W	a = (a + b + 1) >> 1
; Reorder elements		
PSHUFW a, b, i	* W	a = {b[i], b[i>>2], b[i>>4], b[i>>6]}
; Extract single element		
PEXTRW r, mm, i	* W	r = mm[i], i=0..3, zero extend
PINSRW mm, b, i	* W	mm[i] = x, i=0..3, b is 16/32 bit reg or memory
; Vector sum of absolute differences		
PSADBW	* B	to W, a[0] = sum(abs(a-b)), unsigned, zero extended

MOVNTQ and MASKMOVQ store non-temporal data - data which will not be reloaded for awhile and therefore should not be stored in cache. This frees cache memory.

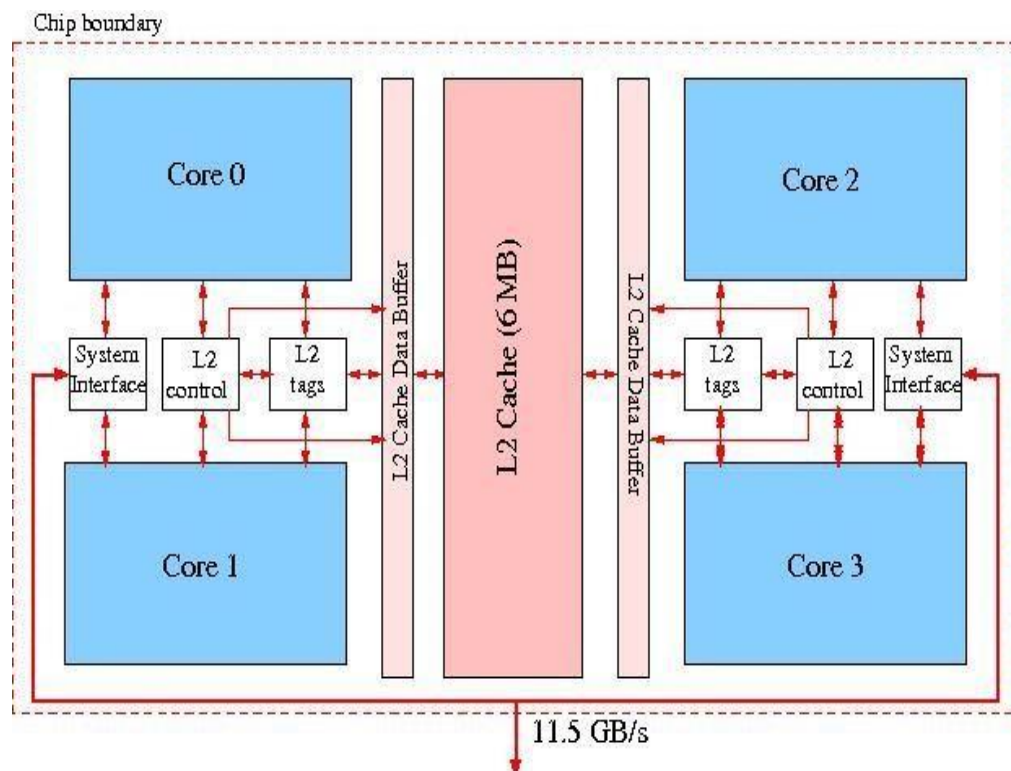
Multi core Processor - Types

1.Core duo Processor:



Execution time will be decreased when two processor connected together.

Core 2 quad processor:



Execution time will be decreased when four processor connected together.

PART A

1. Discuss the register in 80286 Microprocessor
2. Compare 8086 & 80286 Microprocessor
3. Describe the Registers in 80386 Microprocessor
4. Sketch the Flag Register in 80286 Microprocessor
5. List the Flag Register in 80386
6. Inspect the Register in 80486 Microprocessor
7. Demonstrate the any two types of MMX instruction set.
8. Develop the floating point in 80486 Microprocessor
9. Explain the Flag Register in 80486. Microprocessor
10. Classify the features of 80286 Microprocessor
11. Measure the features of 80386 Microprocessor
12. Illustrate the features of 80486 Microprocessor

PART B

1. Construct Architecture of Intel 80286 Microprocessor.
2. Discuss the Architecture of Intel 80386 Microprocessor.
3. Produce the Construction architecture of Intel 80486 Microprocessor.
4. Discuss in detail about the MMX Architecture.
5. Explain the MMX instruction set.
6. Explain the Multi core Processor.
7. Distinguish between 80286, 80386 and 80486 microprocessors
8. Analyze the 80286 and 80386 microprocessors
9. Compare 80386 and 80486 microprocessors

TEXT / REFERENCE BOOKS

1. A.K Ray and K M Bhurchandi, Advanced Microprocessors and Peripherals, 3RD edition, TMH, 2017.
2. Joseph Yiu, The Definitive Guide to the ARM Cortex-M3, 2nd Edition, Newnes, 2015.
3. Dr. Mark Fisher, ARM Cortex M4 Cookbook, Packt, 2016.
4. David Hanes, "IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things", Cisco press, 2017.
5. Olivier Hersent, David Boswarthick, Omar Elloumi, "The Internet of Things: Key Applications and Protocols", 2nd Edition, Wiley, 2012.
6. Rajkamal, "Embedded system-Architecture, Programming, Design", TMH, 2011.
7. Jonathan W. Valvano, "Embedded Microcomputer Systems, Real Time Interfacing", Cengage Learning, 3rd Edition, 2012.



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT III – ARM CONTROLLER– SPHA5204

Unit –III

ARM PROCESSOR ARCHITECTURE

CISC and RISC ARCHITECTURE

Central Processing Unit Architecture operates the capacity to work from “Instruction Set Architecture” to where it was designed. The architectural designs of CPU are RISC (Reduced instruction set computing) and CISC (Complex instruction set computing). CISC has the ability to execute addressing modes or multi-step operations within one instruction set. It is the design of the CPU where one instruction performs many low-level operations. For example, memory storage, an arithmetic operation and loading from memory. RISC is a CPU design strategy based on the insight that simplified instruction set gives higher performance when combined with a microprocessor architecture which has the ability to execute the instructions by using some microprocessor cycles per instruction.

What is RISC and CISC ARCHITECTURES?

Hardware designers invent numerous technologies & tools to implement the desired architecture in order to fulfill these needs. Hardware architecture may be implemented to be either hardware specific or software specific, but according to the application both are used in the required quantity. As far as the processor hardware is concerned, there are 2 types of concepts to implement the processor hardware architecture. First one is RISC and other is CISC.

CISC Architecture

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. Computers based on the CISC architecture are designed to decrease the memory cost. Because, the large programs need more storage, thus increasing the memory cost and large memory becomes more expensive. To solve these problems, the number of instructions per program can be reduced by embedding the number of operations in a single instruction, thereby making the instructions more complex.

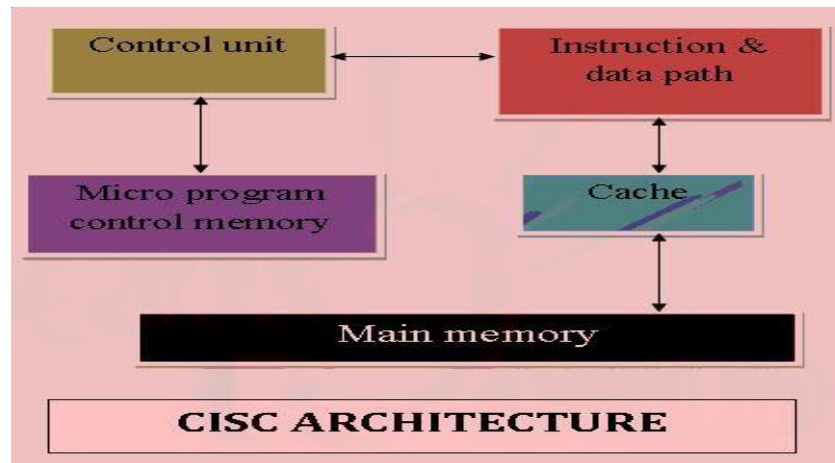


Figure 3.1 CISC Architecture

- MUL loads two values from the memory into separate registers in CISC.
- CISC uses minimum possible instructions by implementing hardware and executes operations.
- Instruction Set Architecture is a medium to permit communication between the programmer and the hardware. Data execution part, copying of data, deleting or editing is the user commands used in the microprocessor and with this microprocessor the Instruction set architecture is operated.
- The main keywords used in the above Instruction Set Architecture are as below

Instruction Set: Group of instructions given to execute the program and they direct the computer by manipulating the data. Instructions are in the form – Opcode (operational code) and Operand. Where, opcode is the instruction applied to load and store data, etc. The operand is a memory register where instruction applied.

Addressing Modes: Addressing modes are the manner in the data is accessed. Depending upon the type of instruction applied, addressing modes are of various types such as direct mode where straight data is accessed or indirect mode where the location of the data is accessed. Processors having identical ISA may be very different in organization. Processors with identical ISA and nearly identical organization is still not nearly identical.

CPU performance is given by the fundamental law

$$CPU\ Time = \frac{Seconds}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instructions} \times \frac{Seconds}{Cycle} \dots\dots\dots(3.1)$$

Thus, CPU performance is dependent upon Instruction Count, CPI (Cycles per instruction) and Clock cycle time. And all three are affected by the instruction set architecture.

	Instruction Count	CPI	Clock
Program	X		
Compiler	X	X	
Instruction Set Architecture	X	X	X
Microarchitecture		X	X
Physical Design			X

Figure 3.2 Instruction Count of the CPU

This underlines the importance of the instruction set architecture. There are two prevalent instruction set architectures

Examples of CISC PROCESSORS

IBM 370/168 – It was introduced in the year 1970. CISC design is a 32 bit processor and four 64-bit floating point registers.

VAX 11/780 – CISC design is a 32-bit processor and it supports many numbers of addressing modes and machine instructions which is from Digital Equipment Corporation.

Intel 80486 – It was launched in the year 1989 and it is a CISC processor, which has instructions varying lengths from 1 to 11 and it will have 235 instructions.

Characteristics of CISC Architecture

- Instruction-decoding logic will be Complex.
- One instruction is required to support multiple addressing modes.
- Less chip space is enough for general purpose registers for the instructions that are Operated directly on memory.
- Various CISC designs are set up two special registers for the stack pointer, handling interrupts, etc.

- ▯ MUL is referred to as a “complex instruction” and requires the programmer for storing functions.

RISC Architecture

RISC (Reduced Instruction Set Computer) is used in portable devices due to its power efficiency. For Example, Apple iPod and Nintendo DS. RISC is a type of microprocessor architecture that uses highly-optimized set of instructions. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program. Pipelining is one of the unique feature of RISC. It is performed by overlapping the execution of several instructions in a pipeline fashion. It has a high performance advantage over CISC.

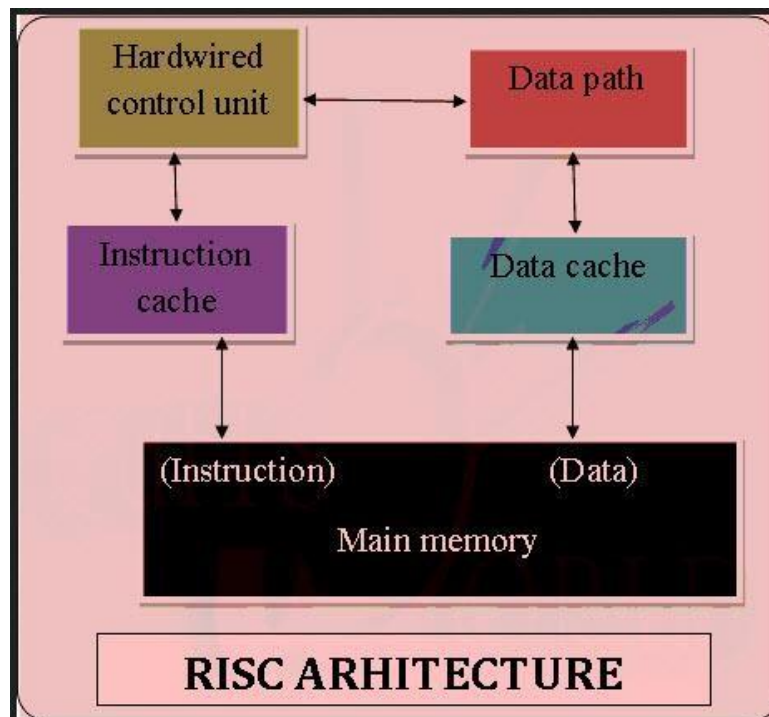


Fig 3.3 RISC Architecture

RISC processors take simple instructions and are executed within a clock cycle

Characteristics of RISC Architecture

- Simple Instructions are used in RISC architecture.
- RISC helps and supports few simple data types and synthesize complex data types.
- RISC utilizes simple addressing modes and fixed length instructions for pipelining.
- RISC permits any register to use in any context.

- One Cycle Execution Time
- The amount of work that a computer can perform is reduced by separating “LOAD” and “STORE” instructions.
- RISC contains Large Number of Registers in order to prevent various number of interactions with memory.
- In RISC, Pipelining is easy as the execution of all instructions will be done in a uniform interval of time i.e. one clock.
- In RISC, more RAM is required to store assembly level instructions.
- Reduced instructions need a less number of transistors in RISC.
- RISC uses Harvard memory model means it is Harvard Architecture.
- A compiler is used to perform the conversion operation means to convert a high-level language statement into the code of its form.

RISC & CISC Comparison

CISC	RISC
It is prominent on Hardware	It is prominent on the Software
It has high cycles per second	It has low cycles per second
It has transistors used for storing Instructions which are complex	More transistors are used for storing memory
LOAD and STORE memory-to-memory is induced in instructions	LOAD and STORE register-register are independent
It has multi-clock	It has a single - clock

Table 3.1 Comparison between CISC & RISC

MUL instruction is divided into three instructions

“LOAD” – moves data from the memory bank to a register

“PROD” – finds product of two operands located within the registers

“STORE” – moves data from a register to the memory banks

The main difference between RISC and CISC is the number of instructions and its complexity.

History of ARM Processors

ARM machines have a history of living up to the expectations of their developers, right from the very first ARM machine ever developed. It all began in the 1980s when Acorn Computers Ltd., spurred by the success of their platform BBC Micro wished to move on from simple CMOS processors to something more powerful, something that could stand strong against the IBM machines launched in 1981. The solutions available in the market like the Motorola 68000 were not powerful enough to handle graphics and GUIs leaving only one option with the company, make their own processor.

Inspired by the making of 32 bit processors by some undergraduates at Berkeley and a one man design centre Western Design Centre, Phoenix, Steve Ferber and Sophie Wilson of Acorn Ltd. set out to make their own processors. Sophie developed the instruction set and simulated it on the BBC Basic which convinced many in the company that it was not just anything half-hearted shot aimed in darkness. With the support and permission of the then CEO Hermann Hauser, the ARM project formally took off in 1983 with VLSI Technology as their silicon partner, to produce an ARM processor with latencies as low as that of the 6502. The first ARM core dubbed as ARM1 was delivered by VLSI Technology in 1985. This processor used in conjunction with the BBC Micro helped in the development of the next generation called ARM2. 1987 saw the release of ARM Archimedes.

Acorn floated a new company Advanced RISC Machines Ltd. solely dedicated for ARM core development. In 1992, Acorn won the Queen's Award for Technology for the ARM. Apple and ARM collaborated to develop the ARM6 cores on which the Apple Newton PDAs were based. Later, the technology was also transferred to Intel over a settlement of lawsuit. Intel further modified it and developed its own high performance line XScale, now sold to Marvell. ARM Inc. is involved with developing cores primarily while its licensees make microcontroller and processors, the most popular being the ARM7TDMI machines. Some prominent licensees of ARM machines are Alcatel Lucent, Apple, Atmel, Cirrus Logic, Freescale, DEC, Intel, LG, Marvell, Microsoft, Nvidia, Qualcomm, Samsung, Sharp, ST microelectronics, Symbios Logic, Texas

Instruments, VLSI Technology, Yamaha, Zilabs etc.

Architecture

History and Development

- ARM was developed at Acron Computers ltd of Cambridge, England between 1983 and 1985.
- RISC concept was introduced in 1980 at Stanford and Berkley.
- ARM ltd was found in 1990.
- ARM cores are licensed to partners so as to develop and fabricate new microcontrollers around same processor cores.

ARM Architecture

- Architecture of ARM is Enhanced RISC Architecture.
- It has large uniform Register file.
- Employs Load Store Architecture- Here operations operate on registers and not in memory locations.
- Architecture is of uniform and fixed length.
- 32 bit processor. It also has 16 bit variant i.e. it can be used as 32 bit and as 16 bit processor.

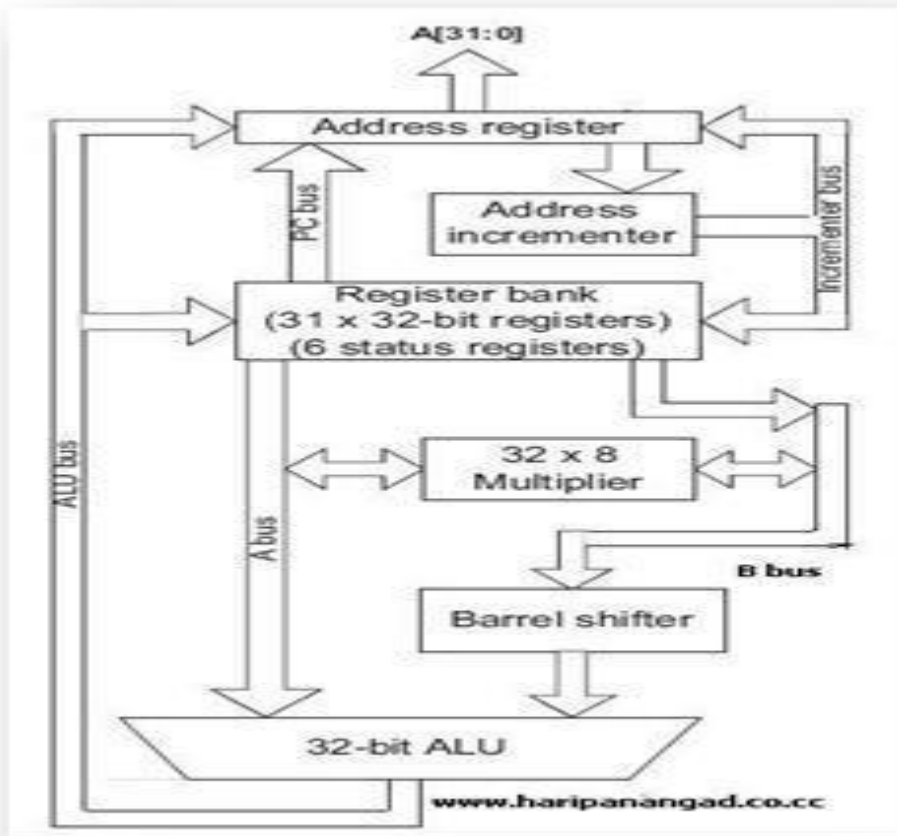


Figure 3.4 ARM Architecture

Core Data path

- Architecture is characterized by Data path and control path.
- Data path is organized in such a way that, operands are not fetched directly from memory locations. Data items are placed in register files. No data processing takes place in memory locations.
- Instructions typically use 3 registers. 2 source registers and 1 destination register.
- Barrel Shifter pre-processes data, before it enters ALU.

- Barrel Shifter is basically a combinational logic circuit, which can shift data to left or right by arbitrary number of position in same cycle.

- Increment or Decrement logic can update register content for sequential access.

ARM Organization

Register Bank is connected to ALU via two data paths.

- A bus
- B bus

B bus goes via Barrel Shifter. It pre-processes data from source register by shifting left or right or even rotating. The Program Counter is that part of register Bank that generate address.

Registers in register bank are symmetric i.e., they can have both data and address. Program counter generates address for next function. Address Incrementer block, increments or decrements register value independent of ALU. There is an Instruction Decode and control block that provides control signals. (Not in figure)

Pipeline

- In ARM 7, a 3 stage pipeline is used. A 3 stage pipeline is the simplest form of pipeline that does not suffer from the problems such as read before write.
- In a pipeline, when one instruction is executed, second instruction is decoded and third instruction will be fetched.
- This is executed in a single cycle.

Register Bank

- ARM 7 uses load and store Architecture.
- Data has to be moved from memory location to a central set of registers.
- Data processing is done and is stored back into memory.
- Register bank contains, general purpose registers to hold either data or address.
- It is a bank of 16 user registers R0-R15 and 2 status registers.
- Each of these registers is 32 bit wide.

Data Registers- R0-R15

- R0-R12 - General Purpose Registers
- R13-R15 - Special function registers of which,

R13 - Stack Pointer, refers to entry pointer of Stack.

R14 - Link Register, Return address is put to this whenever a subroutine is called.

R15 - Program Counter

Depending upon application R13 and R14 can also be used as GPR. But not commonly used.

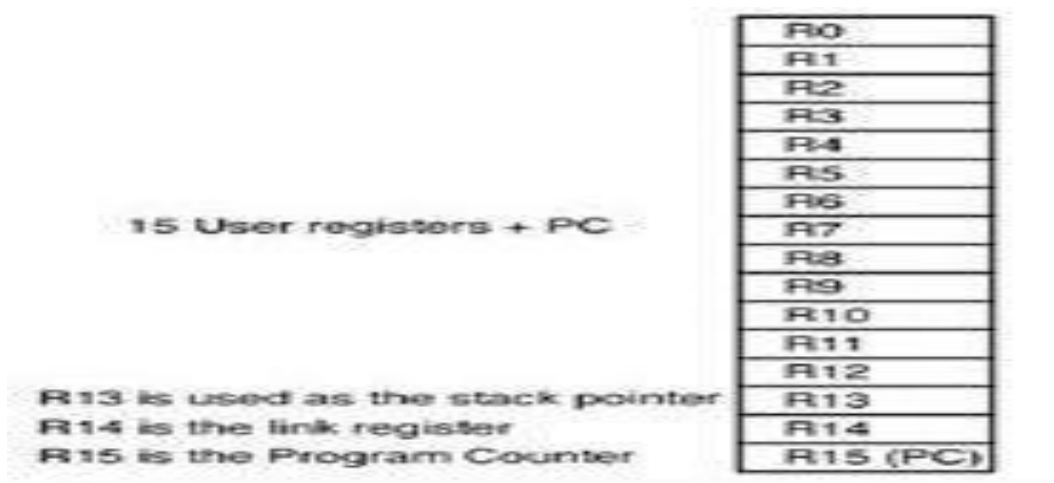


Fig 2.5 User Register

In addition there are 2 status registers

- CPSR - Current program status register, status of current execution is stored.
- SPSR - Saved program Status register, includes status of program as well as processor.

CPSR

CPSR contains a number of flags which report and control the operation of ARM7 CPU.



Fig 2.6 CPSR Register

Conditional Code Flags

N - Negative Result

from ALU Z - Zero

result from ALU

C - ALU operation

carried out V - ALU

operation overflowed

Interrupt Enable Bits

I - IRQ, Interrupt Disable

F - FIQ, Disable Fast

Interrupt T- Bit

If

T=0, Processor in ARM

Mode. T=1, Processor in

THUMB Mode Mode Bits

Specifies the processor Modes.

ARM features

- Barrel Shifter in data path that maximize the usage of hardware available on the chip.
- Auto increment and Auto decrement addressing modes to optimize program loop. This feature is not common in RISC architecture.
- Load and Store instruction to maximize data throughput.
- Conditional execution of instructions, to maximize execution throughput.

ARM INSTRUCTION SET

We know that the ARM provides by way of memory and registers, and the sort of instructions to manipulate them. All ARM instructions are 32 bits long. Here is a typical one:

10101011100101010010100111101011

Usually, mnemonics are followed by one or more operands which are used to completely describe the instruction. An example for mnemonic is **ADD**, for 'add two registers'. This alone doesn't tell the assembler which registers to add and where to put the result. If the left and right hand side of the addition are R1 and R2 respectively, and the result is to go in R0, the operand part would be written R0,R1,R2. Thus the complete add instruction, in assembler format, would be:

ADD R0, R1, R2 ;R0 = R1 + R2

Most ARM mnemonics consist of three letters, e.g. **SUB**, **MOV**, **STR**, **STM**. Certain 'optional extras' may be added to slightly alter the affect of the instruction, leading to mnemonics such as **ADCNES** and **SWINE**.

The mnemonics and operand formats for all of the ARM's instructions are described in detail in the sections below. At this stage, we don't explain how to create programs, assemble and run them. There are two main ways of assembling ARM programs - using the assembler built-in to BBC BASIC, or using a dedicated assembler. The former method is more convenient for testing short programs, the latter for developing large scale projects. Chapter Four covers the use of the BASIC assembler.

Condition codes

The property of conditional execution is common to all ARM instructions, so its representation in assembler is described before the syntax of the actual instructions. As mentioned before, there are four bits of condition encoded into an instruction word. This allows sixteen possible conditions. If the condition for the current instruction is true, the execution goes ahead. If the condition does not hold, the instruction is ignored and the next one executed.

The result flags are altered mainly by the data manipulation instructions. These instructions only affect the flags if you explicitly tell them to. For example, a **MOV** instruction which copies the contents of one register to another. No flags are affected. However, the **MOVS** (move with Set) instruction additionally causes the result flags to be set. The way in which each instruction affects the flags is described below.

To make an instruction conditional, a two-letter suffix is added to the mnemonic. The suffixes, and their meanings, are listed below.

AL Always

An instruction with this suffix is always executed. To save having to type '**AL**' after the majority of instructions which are unconditional, the suffix may be omitted in this case. Thus **ADDAL** and **ADD** mean the same thing: add unconditionally.

NV Never

All ARM conditions also have their inverse, so this is the inverse of always. Any instruction with this condition will be ignored. Such instructions might be used for 'padding' or perhaps to use up a (very) small amount of time in a program.

EQ Equal

This condition is true if the result flag Z (zero) is set. This might arise after a compare instruction where the operands were equal, or in any data instruction which received a zero result into the destination.

NE Not equal

This is clearly the opposite of **EQ**, and is true if the Z flag is cleared. If Z is set, and instruction with the **NE** condition will not be executed.

VS Overflow set

This condition is true if the result flag V (overflow) is set. Add, subtract and compare instructions affect the V flag.

VC

Overflow

clear The

opposite to

VS. **MI**

Minus

Instructions with this condition only execute if the N (negative) flag is set. Such a condition would occur when the last data operation gave a result which was negative. That is, the N

flag reflects the state of bit 31 of the result. (All data operations work on 32-bit numbers.)

PL Plus

This is the opposite to the **MI** condition and instructions with the **PL** condition will only execute if the N flag is cleared.

The next four conditions are often used after comparisons of two unsigned numbers. If the numbers being compared are $n1$ and $n2$, the conditions are $n1 \geq n2$, $n1 < n2$, $n1 > n2$ and $n1 \leq n2$, in the order presented.

CS Carry set

This condition is true if the result flag C (carry) is set. The carry flag is affected by arithmetic instructions such as **ADD**, **SUB** and **CMP**. It is also altered by operations involving the shifting or rotation of operands (data manipulation instructions).

When used after a compare instruction, **CS** may be interpreted as 'higher or same', where the operands are treated as unsigned 32-bit numbers. For example, if the left hand operand of **CMP** was 5 and the right hand operand was 2, the carry would be set. You can use **HS** instead of **CS** for this condition.

CC Carry clear

This is the inverse condition to **CS**. After a compare, the **CC** condition may be interpreted as meaning 'lower than', where the operands are again treated as unsigned numbers. An synonym for **CC** is **LO**.

HI Higher

This condition is true if the C flag is set and the Z flag is false. After a compare or subtract, this combination may be interpreted as the left hand operand being greater than the right hand one, where the operands are treated as unsigned.

LS Lower or same

This condition is true if the C flag is cleared or the Z flag is set. After a compare or subtract, this combination may be interpreted as the left hand operand being less than or equal to the right hand one, where the operands are treated as unsigned.

The next four conditions have similar interpretations to the previous four, but are used when signed numbers have been compared. The difference is that they take into account the state of the V (overflow) flag, whereas the unsigned ones don't.

Again, the relationships between the two numbers which would cause the condition to be true are $n1 \geq n2$, $n1 < n2$, $n1 > n2$, $n1 \leq n2$.

GE Greater than or equal

This is true if N is cleared and V is cleared, or N is set and V is set.

LT Less than

This is the opposite to **GE** and instructions with this condition are executed if N is set and V is cleared, or N is cleared and V is set.

GT Greater than

This is the same as **GE**, with the addition that the Z flag must be cleared too.

LE Less than or equal

This is the same as **LT**, and is also true whenever the Z flag is set.

Note that although the conditions refer to signed and unsigned numbers, the operations on the numbers are identical regardless of the type. The only things that change are the flags used to determine whether instructions are to be obeyed or not.

The flags may be set and cleared explicitly by performing operations directly on R15, where they are stored.

Group one - data manipulation

Assembler format

ADD has the following format:

ADD{**cond**}{**S**} <**dest**>, <**lhs**>, <**rhs**>

The parts in curly brackets are optional. **Cond** is one of the two-letter condition codes listed above. If it is omitted, the 'always' condition **AL** is assumed. The **S**, if present, causes the instruction to affect the result flags. If there is no **S**, none of the flags will be changed. For example, if an instruction **ADDS** *É* yields a result which is negative, then the N flag will be set. However, just **ADD** *É* will not alter N (or any other flag) regardless of the result.

After the mnemonic are the three operands. <**dest**> is the destination, and is the register number where the result of the **ADD** is to be stored. Although the assembler is

happy with actual numbers here, e.g. 0 for R0, it recognises R0, R1, R2 etc. to stand for the register numbers. In addition, you can define a name for a register and use that instead. For example, in BBC BASIC you could say:-

iac = 0

where **iac** stands for, say, integer accumulator. Then this can be used in an instruction:-

ADD iac, iac, #1

The second operand is the left hand side of the operation. In general, the group one instructions act on two values to provide the result. These are referred to as the left and right hand sides, implying that the operation determined by the mnemonic would be written between them in mathematics. For example, the instruction:

ADD R0, R1, R2

has R1 and R2 as its left and right hand sides, and R0 as the result. This is analogous to an assignment such as **R0=R1+R2** in BASIC, so the operands are sometimes said to be in 'assignment order'.

The **<lhs>** operand is always a register number, like the destination. The **<rhs>** may either be a register, or an immediate operand, or a shifted or rotated register. It is the versatile form that the right hand side may take which gives much of the power to these instructions.

If the **<rhs>** is a simple register number, we obtain instructions such as the first **ADD** example above. In this case, the contents of R1 and R2 are added (as signed, 32-bit numbers) and the result stored in R0. As there is no condition after the instruction, the **ADD** instruction will always be executed. Also, because there was no **S**, the result flags would not be affected.

The three examples below all perform the same **ADD** operation (if the condition is true):

ADDNE R0, R0, R2

ADDS R0, R0, R2

ADDNES R0, R0, R2

They all add R2 to R0. The first has a **NE** condition, so the instruction will only be executed if the Z flag is cleared. If Z is set when the instruction is encountered, it is ignored. The second one is unconditional, but has the **S** option. Thus the N, Z, V and C flags will be altered to reflect the result. The last example has the condition and the **S**, so if Z is cleared, the **ADD** will occur and the flags set accordingly. If Z is set, the **ADD** will be skipped and the flags remain unaltered.

Immediate operands

Immediate operands are written as a # followed by a number. For example, to increment R0, we would use:

ADD R0, R0, #1

Now, as we know, an ARM instruction has 32 bits in which to encode the instruction type, condition, operands etc. In group one instructions there are twelve bits available to encode immediate operands. Twelve bits of binary can represent numbers in the range 0..4095, or - 2048..+2047 if we treat them as signed.

Shifted operands

If the <rhs> operand is a register, it may be manipulated in various ways before it is used in the instruction. The contents of the register aren't altered, just the value given to the ALU, as applied to this operation (unless the same register is also used as the result, of course).

The particular operations that may be performed on the <rhs> are various types of shifting and rotation. The number of bits by which the register is shifted or rotated may be given as an immediate number, or specified in yet another register.

Shifts and rotates are specified as left or right, logical or arithmetic. A left shift is one where the bits, as written on the page, are moved by one or more bits to the left, i.e. towards the more significant end. Zero-valued bits are shifted in at the right and the bits at the left are lost, except for the final bit to be shifted out, which is stored in the carry flag. Left shifts by n bits effectively multiply the number by 2^n , assuming that no significant bits are 'lost' at the top end.

A right shift is in the opposite direction, the bits moving from the more significant end to the lower end, or from left to right on the page. Again the bits shifted out are lost, except for the last one which is put into the carry. If the right shift is logical then zeros are shifted into the left end. In arithmetic shifts, a copy of bit 31 (i.e. the sign bit) is shifted in.

Right arithmetic shifts by n bits effectively divide the number by 2^n , rounding towards minus infinity (like the BASIC **INT** function). A rotate is like a shift except that the bits shifted in to the left (right) end are those which are coming out of the right (left) end.

RM INSTRUCTION SET

Registers,
Memory Access, and Data Transfer
Arithmetic and Logic Instructions

Branch Instructions Assembly
Language/O Operations
Subroutines

Registers and Memory Access

In the ARM architecture Memory is byte addressable 32-bit addresses 32-bit processor registers Two operand lengths are used in moving d the memory and the processor registers Bytes (8 bits) and words (32 bits) Word addresses must be aligned, i.e., they multiple of 4 Both little-endian and big-endian memory address supported When a byte is loaded from memory into a register or stored from a register into the memory It always located in the low-order byte position of the register.

Register Structure

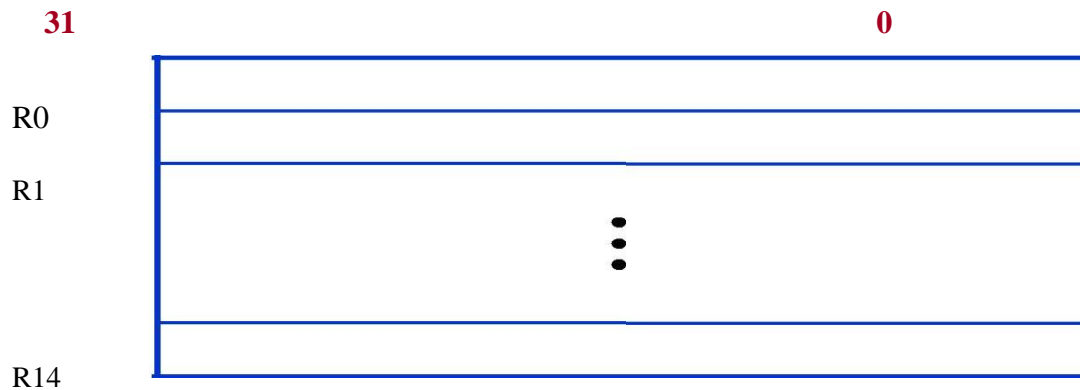


Figure 2.13 The register structure.

Register Structure

There are 15 additional general-purpose registers, the banked registers. They are duplicates of some of the R0 to R14 registers. They are used when the processor switches into Interrupt modes of operation. Saved copies of the Status register are also a Supervisor and Interrupt modes.

ARM Instruction Format

Each instruction is encoded into a 32-bit word. Access to memory is provided only by Load instructions. The basic encoding format for the instruction Load, Store, Move, Arithmetic, and Logic is shown below.

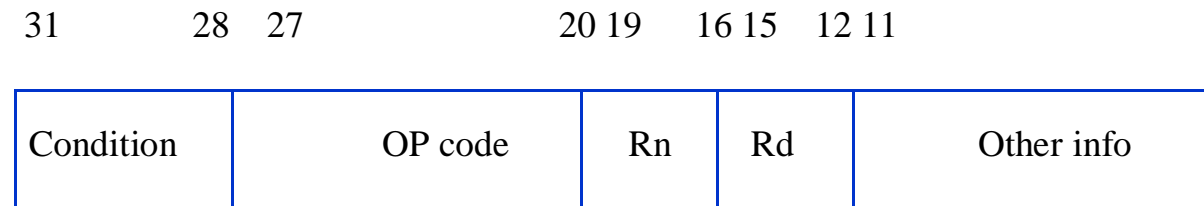


Figure 2.13 The ARM instruction format

An instruction specifies a conditional execution (Condition), the OP code, two or three registers (Rn and Rm), and some other information.

Memory Addressing Modes

Pre-indexed mode

The effective address of the operand is the sum of the base register Rn and an offset value.

Pre-indexed with write back mode

The effective address of the operand is generated as in the Pre-indexed mode, and then the effective address is written back into Rn.

Post-indexed mode

The effective address of the operand is the contents of the register. The offset is then added to this address and the result is stored into Rn.

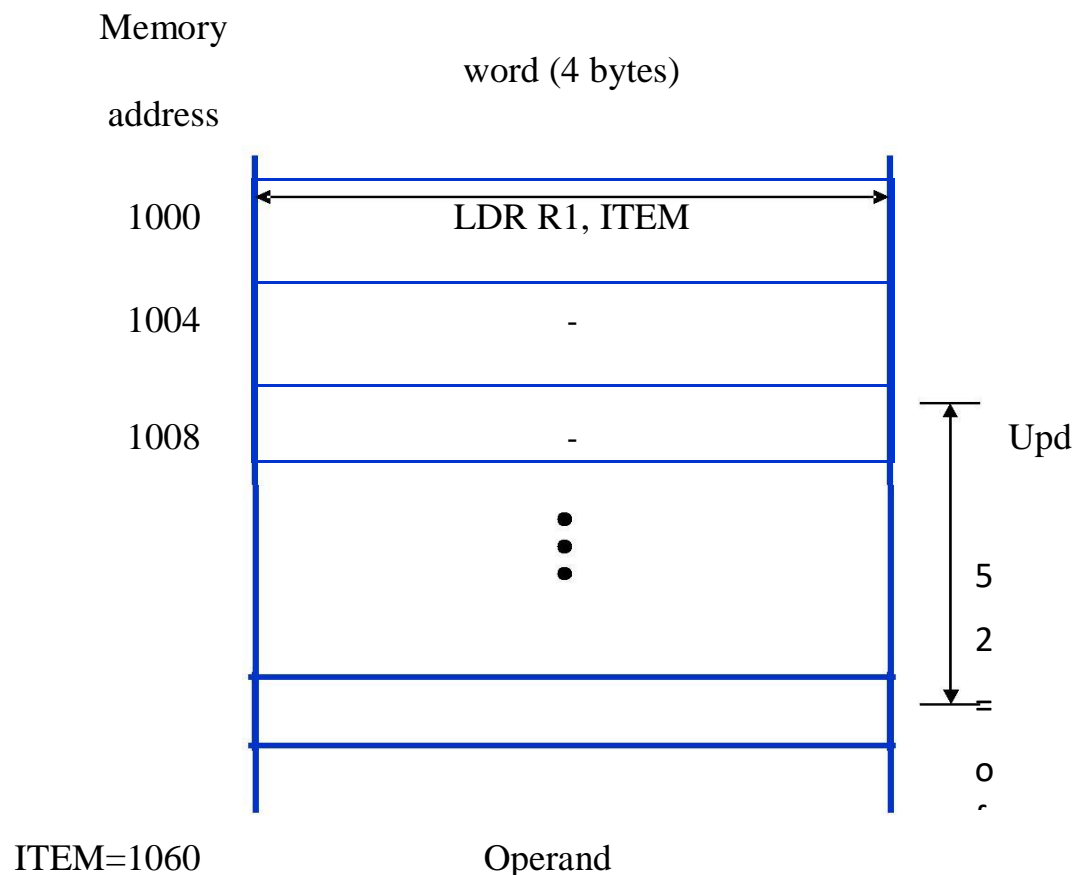
ARM Indexed Addressing

Table 2.2 ARM indexed addressing

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

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 format
 ± Rm=the offset magnitude in register Rm can be added to or subtracted from the contents of based register Rn

Relative Addressing Mode



The operand must be within the range of 4095 bytes forward or backward from the updated PC.

Pre-Indexed Addressing Mode

Post-Indexed

Addressing with Write Back

Pre-Indexed

Addressing with Write Back

Load/Store Multiple

Operation

In ARM processors, 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 stored. Only word operands are allowed, and the OP codes LDM (Load Multiple) and STM (Store Multiple). The memory operands must be in successive locations. All of the forms of pre- and post-indexing without write back are available. They operate on a Base register R_n specified in instruction and offset is always 4

LDMIA $R_{10}!, \{R_0, R_1, R_6, R_7\}$

IA: "Increment After" corresponding to post-indexing

Arithmetic Instructions

The basic expression for arithmetic

instruction OPcode $R_d, R_n,$

R_m

For example, ADD R_0, R_2, R_4

Performs the operation R_0

$[R_2] + [R_4]$ SUB R_0, R_6, R_5

Performs the operation R_0

$[R_6] - [R_5]$ Immediate mode: ADD

$R_0, R_3, \#17$

Performs the operation $R_0 = [R_3] + 17$

The second operand can be shifted or rotated used in the operation. For example, ADD $R_0, R_1, R_5, LSL \#4$ operates a second operand stored in R_5 is shifted left 4-bit position (equivalent to $[R_5] \times 16$), and its is then added to R_1 ; the sum is placed in R_0

Logic Instructions

The logic operations AND, OR, XOR, and NOT are implemented by instructions with the OP codes ORR, EOR, and BIC.

For example

AND R_0, R_0, R_1 : performs $R_0 = [R_0] \& [R_1]$

The Bit-Clear instruction (BIC) is closely related AND instruction. It complements each bit in operand R_m before AND with the bits in register R_n .

For example, BIC R0, R0, R1. Let R0=02FA62CA, Then the instruction results in the pattern 02FA0 in R0

The Move Negative instruction complement the source operand and places the result in For example, MVN R0, R3

Branch Instructions

Conditional branch instructions contain a signed 24bit offset that is added to the updated contents of the Program Counter to generate the branch target address The format for the branch instructions is shown below

31	28 27	24 23
Condition	OP code	offset

Offset is a signed 24-bit number. It is shifted left (all branch targets are aligned word addresses), s to 32 bits, and added to the updated PC to generate the branch target address. The updated points to the instruction that is two forward from the branch instruction.

An Example of Adding Numbers

	LDR	R1, N	Load count into R1
	LDR	R2, POINTER	Load address NUM1 into R2
	MOV	R0, #0	Clear accumulator R0
LOOP	LDR	R3, [R2], #4	Load next number into R3
	ADD	R0, R0, R3	Add number into R0
	SUBS	R1, R1, #1	Decrement loop counter R1
	BGT	LOOP	Branch back if not done
	STR	R0, SUM	Store sum

Assume that the memory location N, POINTER, and SUM are within the range Reachable by the offset relative to the PC

GT: signed greater than

BGT: Branch if Z=0 and N=0

THUMB INSTRUCTION SET

The Thumb instruction set is a subset of the most commonly used 32-bitARM instructions. Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model.

- To understand 16-bit Thumb mode operation of ARM Processor. To understand the features of Thumb mode operation and how Thumb instructions decompress

to ARM Mode.

- To know the technique of switching between ARM and Thumb mode of operations.

- To know the similarities and differences between ARM and Thumb mode of operation. To understand exception handling and branching in Thumb mode.
- To understand operation of data processing instructions and data transfer instructions in Thumb mode.

ARM7TDMI processor has two instruction sets:

- the standard 32-bit **ARM** instruction set
- a 16-bit **THUMB** instruction set.

ARM architecture versions v4T and above define a 16-bit instruction set called the Thumb instruction set. The functionality of the Thumb instruction set is a subset of the functionality of the 32-bit ARM instruction set. A processor that is executing Thumb instructions is operating in *Thumb state*. A processor that is executing ARM instructions is operating in *ARM state*. A processor in ARM state cannot execute Thumb instructions, and a processor in Thumb state cannot execute ARM instructions. You must ensure that the processor never receives instructions of the wrong instruction set for the current state. Each instruction set includes instructions to change processor state. The processor in Thumb mode uses same eight general-purpose integer registers that are available ARM mode. Some Thumb instructions also access the PC(ARM register 15),the Link Register(ARM register 14) and Stack Pointer(ARM register 13).When R15 is read, bit[0] is zero and bits[31:1]contain the PC. when R15 is written, bit[0] is IGNORED and bits[31:1] are written to the PC.

Thumb does not provide direct access to the CPSR or any SPSR.

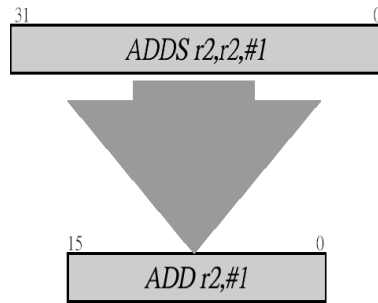
Thumb execution is flagged by the T bit(bit[5]) in the CPSR.

T==0 32-bit instructions are fetched(ARM instruction)

T==1 16-bit instructions are fetched(Thumb instruction)

use ARM code in 32-bit on-chip memory for small speed- critical routines

use Thumb code in 16-bit off-chip memory for large non-critical control routines



Switching between ARM and Thumb States of Execution Using BX Instruction

Thumb Programmers Model

- Registers r0 to r7 are accessible (Lo)
- Few instructions require r8 to r15 to be specified
- r13 is used as the stack pointer
- r14 is used as the link register
- r15 is used as the program counter

Thumb General registers and Program Counter

User / System	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
SP	SP_FIQ	SP_SVC	SP_ABT	SP_IRQ	SP_UND
LR	LR_FIQ	LR_SVC	LR_ABT	LR_IRQ	LR_UND
PC	PC_FIQ	PC_SVC	PC_ABT	PC_IRQ	PC_UND

Thumb Program Status Registers

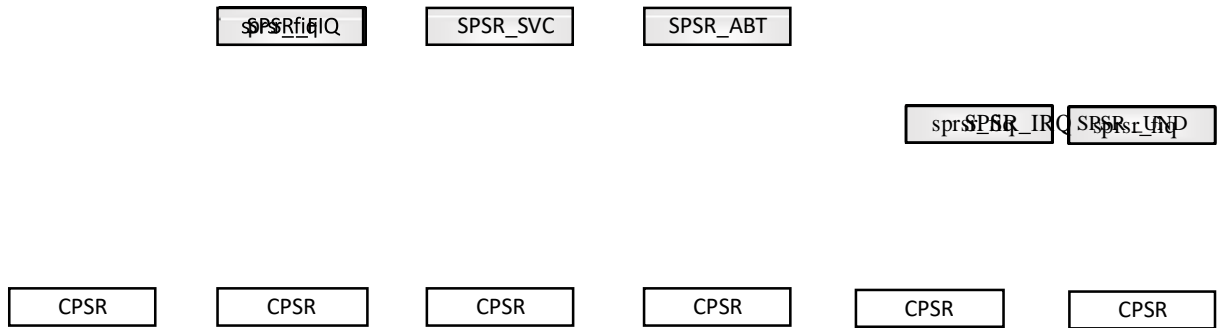


Figure 3.14 Thumb general register and thumb status reg

Branch Instruction Formats

Instruction formats

- `<op> Rd, Rn, Rm`
- `<op> Rd, Rn, # <imm3>`
- `<op> Rd|Rn, Rm|Rs`
- `<op> Rd, Rn, #<sh 5>`
- `<op> Rd, #<imm 8>`
- ❖ `MOV Rd, #<imm8>`
- ❖ `MVN Rd, Rm`
- ❖ `CMP Rn, #<imm8>`
- ❖ `CMP Rn, Rm`
- ❖ `CMN Rn, Rm`
- ❖ `TST Rn, Rm`
- ❖ `ADD Rd, Rn, #<imm3>`
- ❖ `ADD Rd, #<imm8>`
- ❖ `ADD Rd, Rn, Rm`
- ❖ `ADC Rd, Rm`
- ❖ `SUB Rd, Rn, #<imm3>`

- ❖ SUB Rd, #<imm8>
- ❖ SUB Rd, Rn, Rm
- ❖ SBC Rd, Rm
- ❖ NEG Rd, Rn
- ❖ LSL Rd, Rm, #<#sh>
- ❖ LSL Rd, Rs
- ❖ LSR Rd, Rm, #<#sh>
- ❖ LSR Rd, Rs
- ❖ ASR Rd, Rm, #<#sh>
- ❖ ASR Rd, Rs
- ❖ ROR Rd, Rs
- ❖ AND Rd, Rm
- ❖ EOR Rd, Rm
- ❖ ORR Rd, Rm
- ❖ BIC Rd, Rm
- ❖ MUL Rd, Rm
- ❖ **Data Transfer Instruction**
- ❖ LDR|STR Rd, [Rn, #off5]
- ❖ LDR|STR Rd, [Rn, Rm]
- ❖ LDRB|STRB Rd, [Rn, #off5]
- ❖ LDRB|STRB Rd, [Rn, Rm]
- ❖ LDRH|STRH Rd, [Rn, #off5]
- ❖ LDRH|STRH Rd, [Rn, Rm]
- ❖ Signed operands:
- ❖ LDR|STR {S} {H|B} Rd, [Rn, Rm]

ARM CORES

The ARM families are divided in three macro areas, see below.

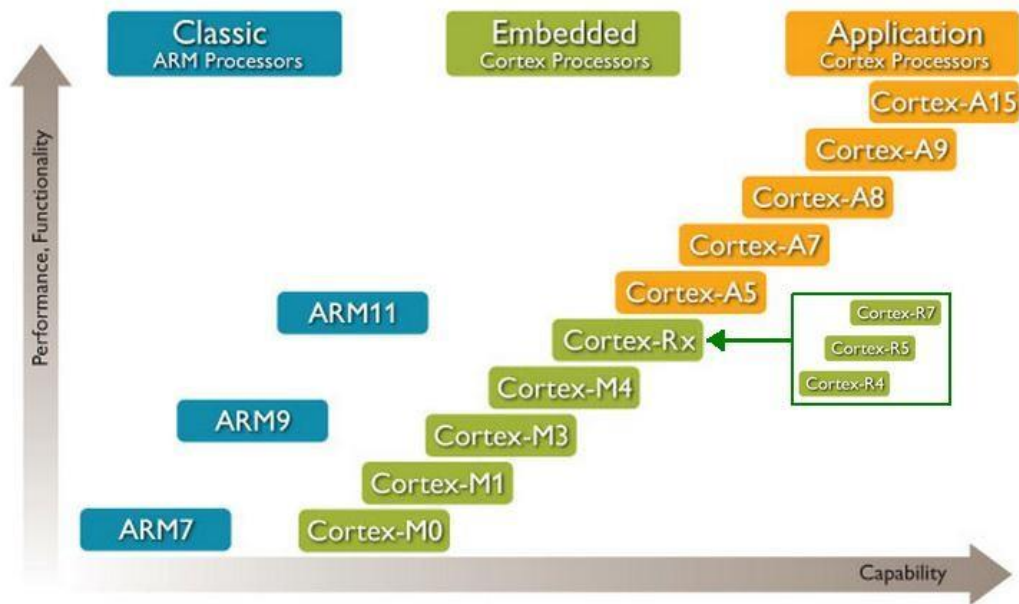


Figure 3.15 ARM cores

The ARM Cortex-A is a group of 32-bit and 64-bit RISC ARM processor cores licensed by ARM Holdings. The cores are intended for application use. The group consists of the 32-bit ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, and ARM Cortex-A17 MPCore,^[1] and the 64-bit ARM Cortex-A53, ARM Cortex-A57, and ARM Cortex-A72. The 64-bit ARM Cortex-A cores implement the ARMv8-A profile of the ARMv8 architecture. The 32-bit ARM Cortex-A cores implement the ARMv7-A profile of the ARMv7 architecture. The main distinguishing feature of the ARMv7-A profile, compared to the other two profiles, the ARMv7-R profile implemented by the ARM Cortex-R cores and the ARMv7-M profile implemented by most of the ARM Cortex-M cores, is that only the ARMv7-A profile includes a memory management unit (MMU).^[2] Many modern operating systems require a MMU to run.

ARM Cortex-M

The **ARM Cortex-M** is a group of 32-bit RISC ARM processor cores licensed by ARM Holdings. The cores are intended for microcontroller use, and consist of the Cortex-M0, M0+, M1, M3, M4, and M7

Arm Cortex-R

The Arm Cortex-R real time processor offers high performance computing solutions for Embedded systems where reliability, high availability, fault tolerance, maintainability and deterministic real time responses are essential. The Cortex-R series processors provide fast time-to-market through proven technology shipped in billions of products, and leverages the vast ARM ecosystem and global, local language, 24/7 support services to ensure rapid and low-risk development

Cortex-R series processors deliver fast and deterministic processing and high performance, while meeting challenging real-time constraints in a range of situations. They combine these features in a performance, power and area optimized package, making them the trusted choice in reliable systems demanding high error-resistance.

PART A

1. Identify the main Feature of ARM Microcontroller
2. Discuss about the application of ARM Microcontroller
3. Develop the debugging techniques
4. Explain the Harvard Architecture.
5. Compare the Harvard Architecture with Von- Neuman Architecture
6. Assess the memory management in ARM Microcontroller
7. Describe the need of Raspberry pi
8. Analyze the Von Neuman Architecture.

PART B

1. Briefly compare the properties of Cortex M0, M3, M4, M7 cores. Such as machine cycles and pipelines
2. Distinguish the various version of ARM cortex M series
3. Classify the Thumb instruction set of ARM Processor
4. Construct the Raspberry pi 3 development board with necessary Sensors, Bluetooth and Wi-fi devices.
5. Describe the traditional debugging features and memory management in ARM Microcontroller.

TEXT / REFERENCE BOOKS

1. A.K Ray and K M Bhurchandi, Advanced Microprocessors and Peripherals, 3RD edition, TMH, 2017.
2. Joseph Yiu, The Definitive Guide to the ARM Cortex-M3, 2nd Edition, Newnes, 2015.
3. Dr. Mark Fisher, ARM Cortex M4 Cookbook, Packt, 2016.
4. David Hanes, "IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things", Cisco press, 2017.
5. Olivier Hersent, David Boswarthick, Omar Elloumi, "The Internet of Things: Key Applications and Protocols", 2nd Edition, Wiley, 2012.
6. Rajkamal, "Embedded system-Architecture, Programming, Design", TMH, 2011.
7. Jonathan W. Valvano, "Embedded Microcomputer Systems, Real Time Interfacing", Cengage Learning, 3rd Edition, 2012.

ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee. Subsequently, additional companies, including Texas Instruments, NEC, Sharp, and ST Microelectronics, licensed the ARM processor designs, extending the applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems, and many other consumer products. The ARM microcontroller architecture comes with a few different versions such as ARMv1, ARMv2 etc and each one has its own advantage and disadvantages.

ARM ARCHITECTURE VERSIONS

The evolution of features and enhancements to the processors over time has led to successive versions of the ARM architecture. Note that architecture version numbers are independent from processor names. For example, the ARM7TDMI processor is based on the ARMv4T architecture (the T is for Thumb® instruction mode support).

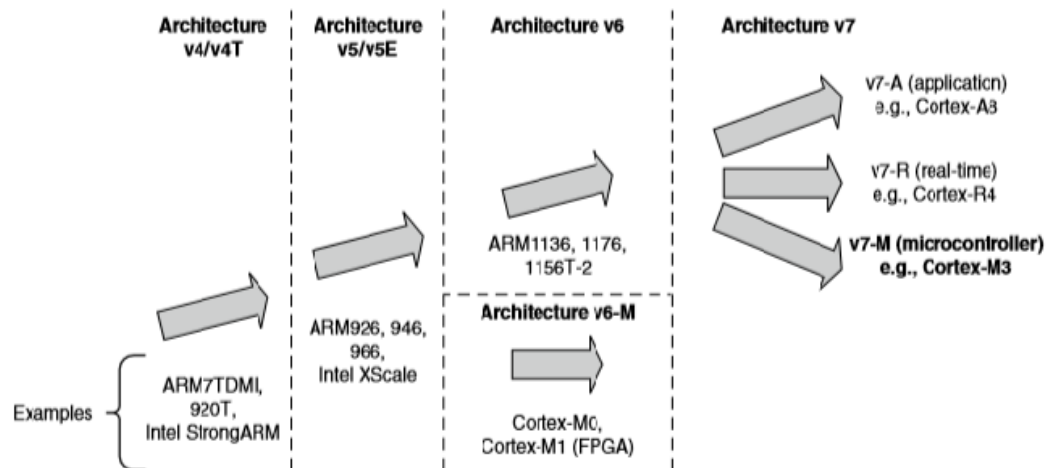


Figure 1: The Evolution of ARM Processor Architecture.

The ARMv5E architecture was introduced with the ARM9E processor families, including the ARM926E-S and ARM946E-S processors. This architecture added “Enhanced” Digital Signal Processing (DSP) instructions for multimedia applications. With the arrival of the ARM11 processor family, the architecture was extended to the ARMv6. New features in this architecture included memory system features and Single Instruction–Multiple Data (SIMD) instructions. Processors based on the ARMv6 architecture include the ARM1136J(F)-S, the ARM1156T2(F)-S, and the ARM1176JZ(F)-S.

Over the past several years, ARM extended its product portfolio by diversifying its CPU development, which resulted in the architecture version 7 or v7. In this version, the architecture design is divided into three profiles:

- **A-profile** is designed for high-performance open application platforms.

- **R-profile** is designed for high-end embedded systems in which real-time performance is needed.
- **M-profile** is designed for deeply embedded microcontroller-type systems.

The Cortex processor families are the first products developed on architecture v7, and the Cortex-M3 processor is based on one profile of the v7 architecture, called ARM v7-M, an architecture specification for microcontroller products.

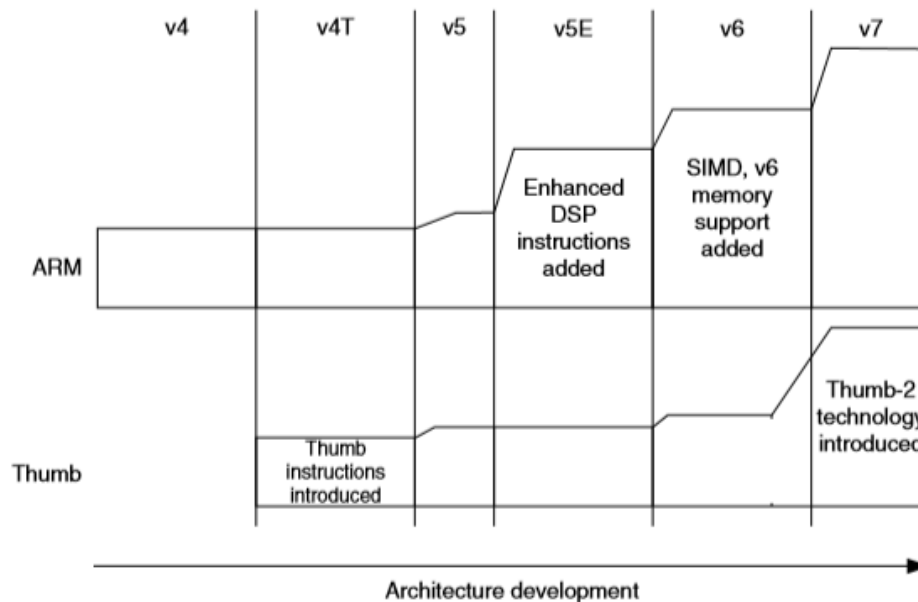


Figure : Instruction set Enhancement in ARM architectures.

Historically (since ARM7TDMI), two different instruction sets are supported on the ARM processor: the ARM instructions that are 32 bits and Thumb instructions that are 16 bits. During program execution, the processor can be dynamically switched between the ARM state and the Thumb state to use either one of the instruction sets. The Thumb instruction set provides only a subset of the ARM instructions, but it can provide higher code density. It is useful for products with tight memory requirements.

In 2003, ARM announced the Thumb-2 instruction set, which is a new superset of Thumb instructions that contains both 16-bit and 32-bit instructions. The extended instruction set in Thumb-2 is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions. It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between ARM state and Thumb state. Focused on small memory system devices such as microcontrollers and reducing the size of the processor, the Cortex-M3 supports only the Thumb-2 (and traditional Thumb) instruction set. Instead of using ARM instructions for some operations, as in traditional ARM processors, it uses the Thumb-2 instruction set for all operations. As a

result, the Cortex-M3 processor is not backward compatible with traditional ARM processors. That is, you cannot run a binary image for ARM7 processors on the Cortex-M3 processor. Nevertheless, the Cortex-M3 processor can execute almost all the 16-bit Thumb instructions, including all 16-bit Thumb instructions supported on ARM7 family processors, making application porting easy.

With support for both 16-bit and 32-bit instructions in the Thumb-2 instruction set, there is no need to switch the processor between Thumb state (16-bit instructions) and ARM state (32-bit instructions). For example, in ARM7 or ARM9 family processors, you might need to switch to ARM state if you want to carry out complex calculations or a large number of conditional operations and good performance is needed, whereas in the Cortex-M3 processor, you can mix 32-bit instructions with 16-bit instructions without switching state, getting high code density and high performance with no extra complexity. The Thumb-2 instruction set is a very important feature of the ARMv7 architecture. Compared with the instructions supported on ARM7 family processors (ARMv4T architecture), the Cortex-M3 processor instruction set has a large number of new features. For the first time, hardware divide instruction is available on an ARM processor, and a number of multiply instructions are also available on the Cortex-M3 processor to improve data-crunching performance. The Cortex-M3 processor also supports unaligned data accesses, a feature previously available only in high-end processors.

THE ARM CORTEX-M3 PROCESSOR

The ARM Cortex™-M3 processor, the first of the Cortex generation of processors released by ARM in 2006, was primarily designed to target the 32-bit microcontroller market. The Cortex-M3 processor provides excellent performance at low gate count and comes with many new features previously available only in high-end processors. The Cortex-M3 addresses the requirements for the 32-bit embedded processor market in the following ways:

- **Greater performance efficiency:** Allowing more work to be done without increasing the frequency or power requirements
- **Low power consumption:** Enabling longer battery life, especially critical in portable products including wireless networking applications
- **Enhanced determinism:** guaranteeing that critical tasks and interrupts are serviced as quickly as possible and in a known number of cycles
- **Improved code density:** ensuring that code fits in even the smallest memory footprints
- **Ease of use:** providing easier programmability and debugging for the growing number of 8-bit and 16-bit users migrating to 32 bits

- **Lower cost solutions:** reducing 32-bit-based system costs close to those of legacy 8-bit and 16-bit devices and enabling low-end, 32-bit microcontrollers to be priced at less than US\$1 for the first time
- **Wide choice of development tools:** from low-cost or free compilers to full-featured development suites from many development tool vendors

There are 3 subfamilies within the ARM cortex family

ARM Cortex-A family (v7-A):

Applications processors for full OS and 3rd party applications

ARM Cortex-R family (v7-R):

Embedded processors for real-time signal processing, control applications

ARM Cortex-M family (v7-M):

Microcontroller-oriented processors for MCU and SoC applications

CORTEX-M3 PROCESSOR APPLICATIONS

With its high performance and high code density and small silicon footprint, the Cortex-M3 processor is ideal for a wide variety of applications:

Low-cost microcontrollers: The Cortex-M3 processor is ideally suited for low-cost microcontrollers, which are commonly used in consumer products, from toys to electrical appliances. It is a highly competitive market due to the many well-known 8-bit and 16-bit microcontroller products on the market. Its lower power, high performance, and ease-of-use advantages enable embedded developers to migrate to 32-bit systems and develop products with the ARM architecture.

Automotive: Another ideal application for the Cortex-M3 processor is in the automotive industry. The Cortex-M3 processor has very high-performance efficiency and low interrupt latency, allowing it to be used in real-time systems. The Cortex-M3 processor supports up to 240 external vectored interrupts, with a built-in interrupt controller with nested interrupt supports and an optional MPU, making it ideal for highly integrated and cost-sensitive automotive applications.

Data communications: The processor's low power and high efficiency, coupled with instructions in Thumb-2 for bit-field manipulation, make the Cortex-M3 ideal for many communications applications, such as Bluetooth and ZigBee.

Industrial control: In industrial control applications, simplicity, fast response, and reliability are key factors. Again, the Cortex-M3 processor's interrupt feature, low interrupt latency, and enhanced fault-handling features make it a strong candidate in this area.

Consumer products: In many consumer products, a high-performance microprocessor (or several of them) is used. The Cortex-M3 processor, being a small processor, is highly efficient and low in power and supports an MPU enabling complex software to execute while providing robust memory protection. There are already many Cortex-M3 processor-based products on the market, including low-end products priced as low as US\$1, making the cost of ARM microcontrollers comparable to or lower than that of many 8-bit microcontrollers.

CORTEX-M3 PROCESSOR VERSUS CORTEX-M3-BASED MCUs

The Cortex-M3 processor is the central processing unit (CPU) of a microcontroller chip. In addition, a number of other components are required for the whole Cortex-M3 processor-based microcontroller. After chip manufacturers license the Cortex-M3 processor, they can put the Cortex-M3 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features.

Cortex-M3 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features. This book focuses on the architecture of the processor core.

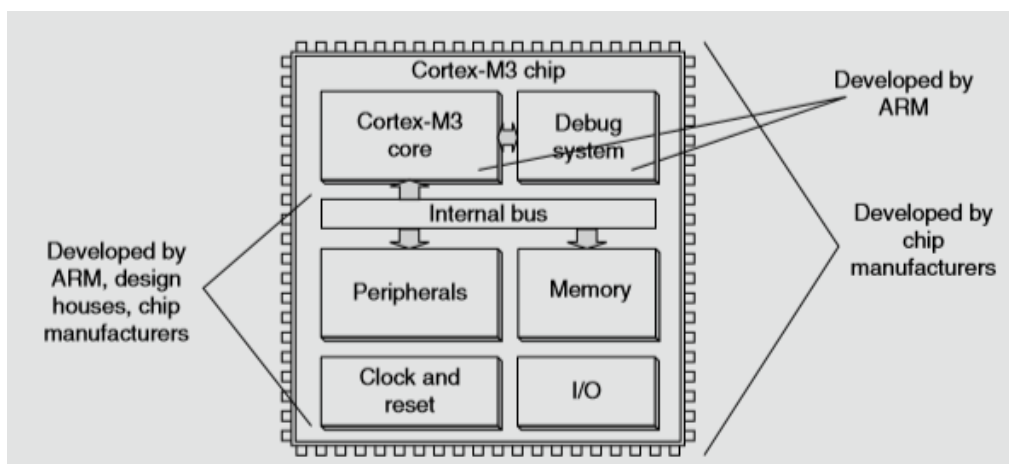


Figure 1: The Cortex-M3 Processor based MCU.

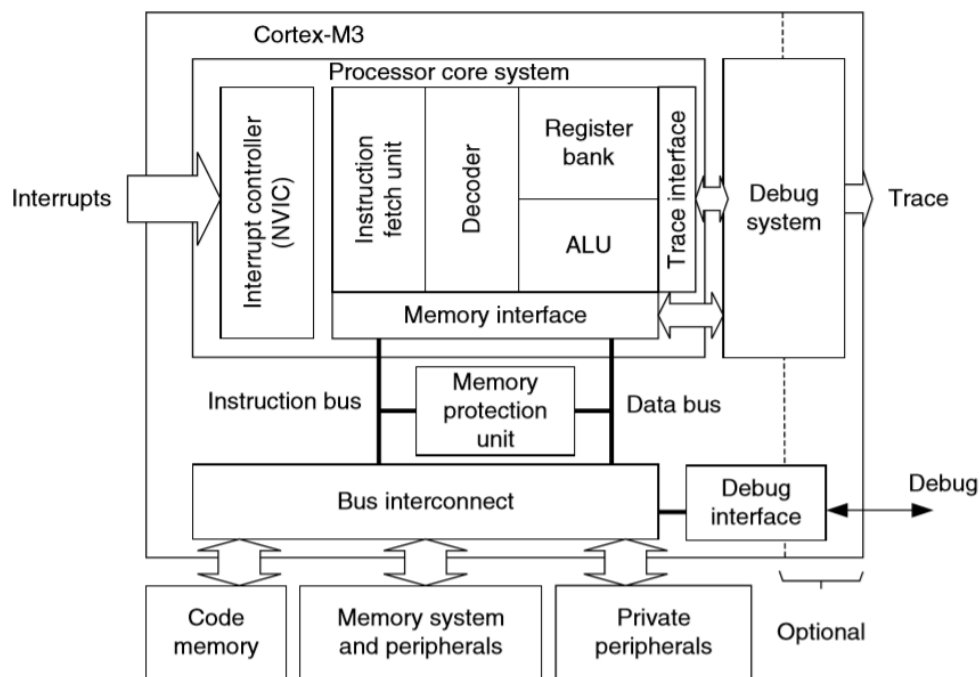
Cortex M3 Architecture

The Cortex™-M3 is a 32-bit microprocessor. It has a 32-bit data path, a 32-bit register bank, and 32-bit memory interfaces (see Figure 2.1). The processor has a Harvard architecture, which means that it has a separate instruction bus and data bus. This allows instructions and data accesses to take place at the same time, and as a result of this, the

performance of the processor increases because data accesses do not affect the instruction pipeline. This feature results in multiple bus interfaces on Cortex-M3, each with optimized usage and the ability to be used simultaneously.

However, the instruction and data buses share the same memory space (a unified memory system). In other words, you cannot get 8 GB of memory space just because you have separate bus interfaces. For complex applications that require more memory system features, the Cortex-M3 processor has an optional Memory Protection Unit (MPU), and it is possible to use an external cache if it's required. Both little endian and big endian memory systems are supported.

The Cortex-M3 processor includes a number of fixed internal debugging components. These components provide debugging operation supports and features, such as breakpoints and watchpoints.



REGISTERS

The Cortex-M3 processor has registers R0 through R15 (see Figure 2.2). R13 (the stack pointer) is banked, with only one copy of the R13 visible at a time.

R0–R12: General-Purpose Registers R0–R12 are 32-bit general-purpose registers for data operations. Some 16-bit Thumb® instructions can only access a subset of these registers (low registers, R0–R7).

R13: Stack Pointers

The Cortex-M3 contains two stack pointers (R13). They are banked so that only one is visible at a time. The two stack pointers are as follows:

- Main Stack Pointer (MSP): The default stack pointer, used by the operating system (OS) kernel and exception handlers

- Process Stack Pointer (PSP): Used by user application code The lowest 2 bits of the stack pointers are always 0, which means they are always word aligned.

R14: The Link Register When a subroutine is called, the return address is stored in the link register.

R15: The Program Counter The program counter is the current program address. This register can be written to control the program flow.

Name		Functions (and banked registers)
R0		General-purpose register
R1		General-purpose register
R2		General-purpose register
R3		General-purpose register
R4		General-purpose register
R5		General-purpose register
R6		General-purpose register
R7		General-purpose register
R8		General-purpose register
R9		General-purpose register
R10		General-purpose register
R11		General-purpose register
R12		General-purpose register
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP), Process Stack Pointer (PSP)
R14		Link Register (LR)
R15		Program Counter (PC)

Low registers

High registers

FIGURE 2.2

Registers in the Cortex-M3.

Special Registers: The Cortex-M3 processor also has a number of special registers. They are as follows:

- Program Status registers (PSRs)
- Interrupt Mask registers (PRIMASK, FAULTMASK, and BASEPRI)
- Control register (CONTROL)

These registers have special functions and can be accessed only by special instructions. They cannot be used for normal data processing

Table 2.1 Special Registers and Their Functions	
Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

CORTEX M3 CPU operating modes

The Cortex-M3 processor has two modes and two privilege levels. The operation modes (thread mode and handler mode) determine whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler (see Figure 2.4). The privilege levels (privileged level and user level) provide a mechanism for safeguarding memory accesses to critical regions as well as providing a basic security model.

When the processor is running a main program (thread mode), it can be either in a privileged state or a user state, but exception handlers can only be in a privileged state. When the processor exits reset, it is in thread mode, with privileged access rights. In the privileged state, a program has access to all memory ranges (except when prohibited by MPU settings) and can use all supported instructions. Software in the privileged access level can switch the program into the user access level using the control register. When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler.

A user program cannot change back to the privileged state by writing to the control register (see Figure 2.5). It has to go through an exception handler that programs the control register to switch the processor back into the privileged access level when returning to thread mode. The separation of privilege and user levels improves system reliability by preventing system configuration registers from being accessed or changed by some untrusted programs. If an MPU is available, it can be used in conjunction with privilege levels to protect critical memory locations, such as programs and data for OSs. For example, with privileged accesses, usually used by the OS kernel, all memory locations can be accessed (unless prohibited by MPU setup). When the OS launches a user application, it is likely to be executed in the user access level to protect the system from failing due to a crash of untrusted user programs.

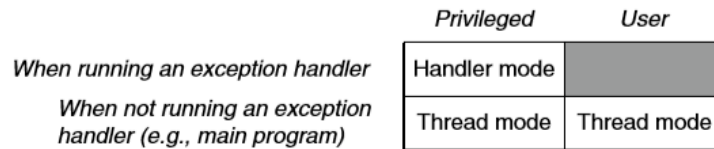


FIGURE 2.4

Operation Modes and Privilege Levels in Cortex-M3.

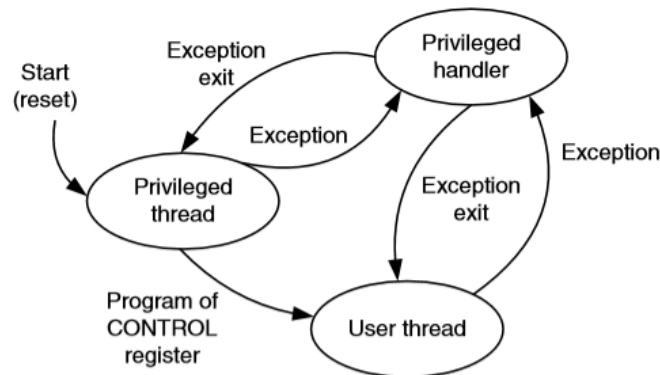


FIGURE 2.5

Allowed Operation Mode Transitions.

ARM MEMORY ORGANIZATION

The Cortex-M3 and Cortex-M4 have a predefined memory map. This allows the built-in peripherals, such as the interrupt controller and the debug components, to be accessed by simple memory access instructions.

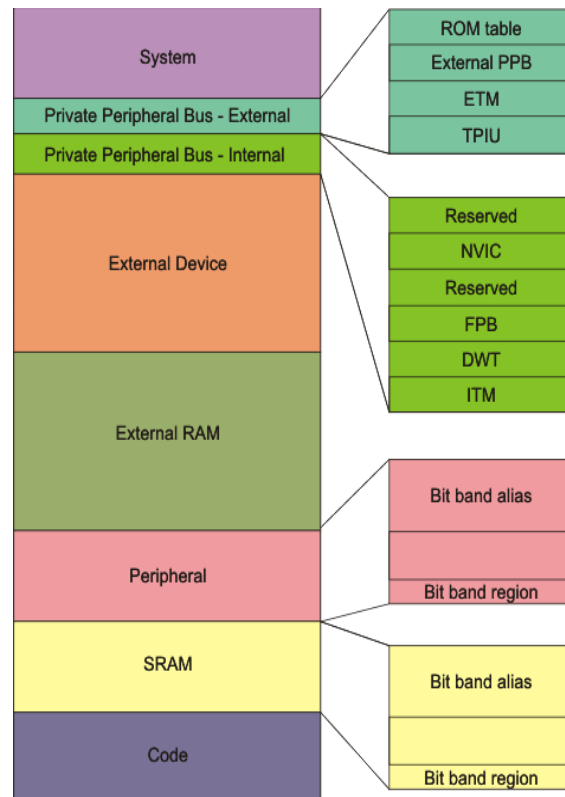
Thus, most system features are accessible in program code. The predefined memory map also allows the Cortex-M3 processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC) designs.

Overall, the 4 GB memory space can be divided into ranges as shown in picture below. The Cortex-M3 design has an internal bus infrastructure optimized for this memory usage.

The ARM Cortex-M3 memory is divided into following regions :

- **System** - .
- **Private Peripheral Bus - External** - Provides access to :
 - the Trace Port Interface Unit (TPIU),
 - the Embedded Trace Macrocell (ETM),
 - the ROM table,
 - implementation-specific areas of the PPB memory map.

A graphical representation of the ARM memory is shown in picture below :



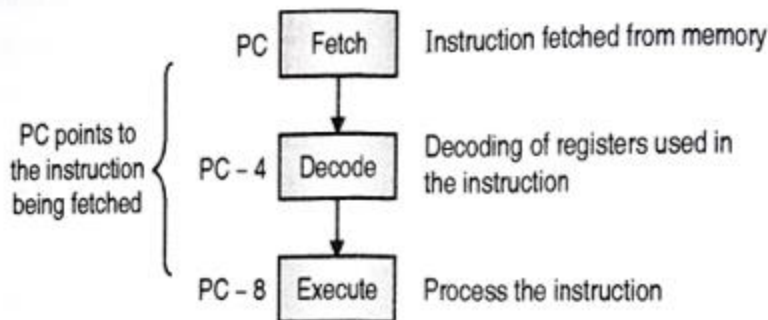
ARM Memory Map

- **Private Peripheral Bus - External** - Provides access to :
 - the Instrumentation Trace Macrocell (ITM),
 - the Data Watchpoint and Trace (DWT),
 - the Flashpatch and Breakpoint (FPB),
 - the System Control Space (SCS), including the MPU and the Nested Vectored Interrupt Controller (NVIC).
- **External Device** - This region is used for external device memory.
- **External RAM** - This region is used for data.
- **Peripheral** - This region includes bit band and bit band alias areas.
 - Peripheral Bit-band alias - Direct accesses to this memory range behave as peripheral memory accesses, but this region is also bit addressable through bit-band alias.
 - Peripheral bit-band region - Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write.
- **SRAM** - This executable region is for data storage. Code can also be stored here. This region includes bit band and bit band alias areas.
 - SRAM Bit-band alias - Direct accesses to this memory range behave as SRAM memory accesses, but this region is also bit addressable through bit-band alias.

- **SRAM bit-band region** - Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write.
- **Code** - This executable region is for program code. Data can also be stored here.

PIPELINING

The Process of fetching the next instruction while the current instruction is being executed is called as “pipelining”. Pipelining is supported by the processor to increase the speed of program execution. Increases throughput. Several operations take place simultaneously, rather than serially in pipelining. The Pipeline has three stages fetch, decode and execute as shown in Fig.

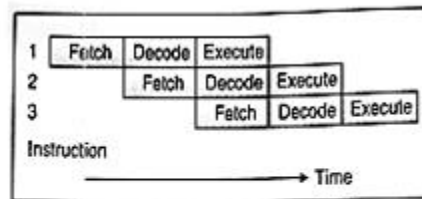


The three stages used in the pipeline are:

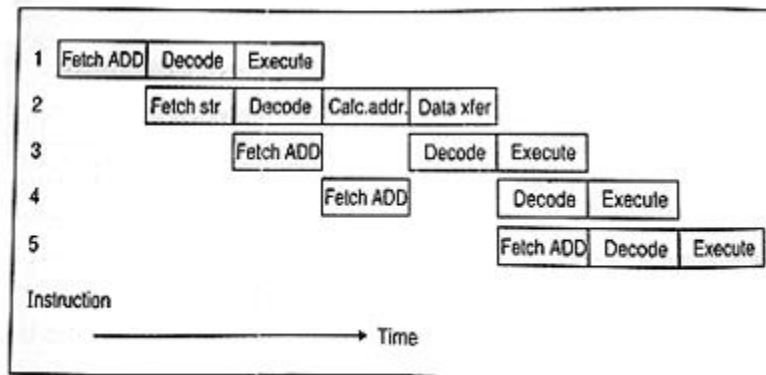
- Fetch** : In this stage the ARM processor fetches the instruction from the memory.
- Decode** : In this stage recognizes the instruction that is to be executed.
- Execute** : In this stage the processor processes the instruction and writes the result back to desired register.

If these three stages of execution are overlapped, we will achieve higher speed of execution. Such pipeline exists in version 7 of ARM processor. Once the pipeline is filled, each instructions require

s one cycle to complete execution. Below fig shows three staged pipelined instruction.



(a) Single cycle instruction execution for a 3-stage pipeline in ARM



In first cycle, the processor fetches instruction 1 from the memory. In the second cycle the processor fetches instruction 2 from the memory and decodes instruction 1. In the third cycle the processor fetches instruction 3 from memory, decodes instruction 2 and executes instruction 1. In the fourth cycle the processor fetches instruction 4, decodes instruction 3 and executes instruction 2. The pipeline thus executes an instruction in three cycles i.e. it delivers a throughput equal to one instruction per cycle.

In case of a multi-cycle instruction as shown in Fig. 9.10.2(b), instruction 2 (i.e. STR of the store instruction) requires 4 clock cycles and hence the pipeline stalls for one clock pulse. The first instruction completes execution in the third clock pulse, while the second instruction instead of completing execution in fourth clock pulse completes the same in fifth clock pulse. Thereafter every instruction completes execution in one clock pulse as seen in this figure.

The amount of work done at each stage can be reduced by increasing the number of stages in the pipeline. To improve the performance, the processor then can be operated at higher operating frequency.

As more number of cycles are required to fill the pipeline, the system latency also increases. The data dependency between the stages can also be increased as the stages of pipeline increase. So the instructions need to be scheduled while writing code to decrease data dependency.

PIPELINE HAZARDS

1. Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycles.
2. Any condition that causes a stall in the pipeline operations can be called a hazard.
3. There are primarily three types of hazards:
 - i. Data Hazards
 - ii. Control Hazards or instruction Hazards
 - iii. Structural Hazards.

i. Data Hazards:

A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result of which some operation has to be delayed and the pipeline stalls. Whenever there are two instructions one of which depends on the data obtained from the other.

$A = 3 + A$

$B = A * 4$

For the above sequence, the second instruction needs the value of 'A' computed in the first instruction.

Thus the second instruction is said to depend on the first.

If the execution is done in a pipelined processor, it is highly likely that the interleaving of these two instructions can lead to incorrect results due to data dependency between the instructions. Thus the pipeline needs to be stalled as and when necessary to avoid errors.

ii. Structural Hazards:

This situation arises mainly when two instructions require a given hardware resource at the same time and hence for one of the instructions the pipeline needs to be stalled.

The most common case is when memory is accessed at the same time by two instructions. One instruction may need to access the memory as part of the Execute or Write back phase while other instruction is being fetched. In this case if both the instructions and data reside in the same memory. Both the instructions can't proceed together and one of them needs to be stalled till the other is done with the memory access part. Thus in general sufficient hardware resources are needed for avoiding structural hazards.

iii. Control hazards:

The instruction fetch unit of the CPU is responsible for providing a stream of instructions to the execution unit. The instructions fetched by the fetch unit are in consecutive memory locations and they are executed.

However the problem arises when one of the instructions is a branching instruction to some other memory location. Thus all the instruction fetched in the pipeline from consecutive memory locations are invalid now and need to be removed (also called flushing of the pipeline). This induces a stall till new instructions are again fetched from the memory address specified in the branch instruction.

Thus the time lost as a result of this is called a branch penalty. Often dedicated hardware is incorporated in the fetch unit to identify branch instructions and compute branch addresses as soon as possible and reducing the resulting delay as a result.

INTERRUPT CONTROLLER or Handler in Cortex-M3

Cortex-M3 processor includes an interrupt controller called the Nested Vectored Interrupt Controller (NVIC). It is closely coupled to the processor core and provides a number of features as follows:

- Nested interrupt support
- Vectored interrupt support
- Dynamic priority changes support
- Reduction of interrupt latency
- Interrupt masking

Nested Interrupt Support The NVIC provides nested interrupt support. All the external interrupts and most of the system exceptions can be programmed to different priority levels. When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level. If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.

Vectored Interrupt Support The Cortex-M3 processor has vectored interrupt support. When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory. There is no need to use software to determine and branch to the starting address of the ISR. Thus, it takes less time to process the interrupt request.

Dynamic Priority Changes Support: Priority levels of interrupts can be changed by software during run time. Interrupts that are being serviced are blocked from further activation until the ISR is completed, so their priority can be changed without risk of accidental reentry.

Reduction of Interrupt Latency: The Cortex-M3 processor also includes a number of advanced features to lower the interrupt latency. These include automatic saving and restoring some register contents, reducing delay in switching from one ISR to another, and handling of late arrival interrupts.

Interrupt Masking: Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI,

PRIMASK, and FAULTMASK. They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

THE MEMORY MAP OF CORTEX-M3

The Cortex-M3 has a predefined memory map. This allows the built-in peripherals, such as the interrupt controller and the debug components, to be accessed by simple memory access instructions.

0xFFFFFFFF	System level	Private peripherals including build-in interrupt controller (NVIC), MPU control registers, and debug components
0xE0000000		
0xDFFFFFFF	External device	Mainly used as external peripherals
0xA0000000		
0x9FFFFFFF	External RAM	Mainly used as external memory
0x60000000		
0x5FFFFFFF	Peripherals	Mainly used as peripherals
0x40000000		
0x3FFFFFFF	SRAM	Mainly used as static RAM
0x20000000		
0x1FFFFFFF	CODE	Mainly used for program code. Also provides exception vector table after power up
0x00000000		

FIGURE 2.6

The Cortex-M3 Memory Map.

Thus, most system features are accessible in C program code. The predefined memory map also allows the Cortex-M3 processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC) designs.

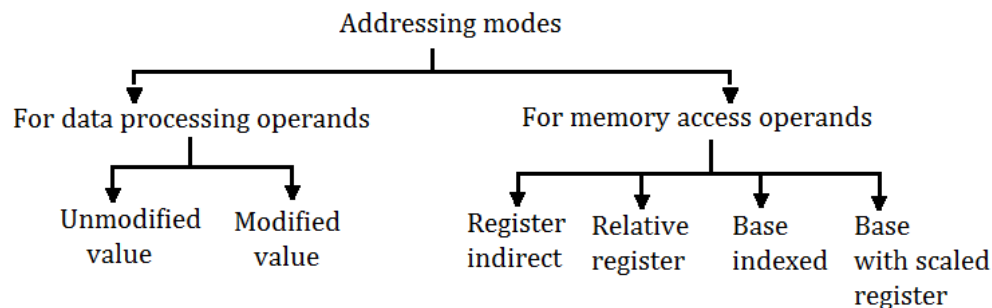
Overall, the 4 GB memory space can be divided into ranges as shown in Figure 2.6. The Cortex-M3 design has an internal bus infrastructure optimized for this memory usage. In addition, the design allows these regions to be used differently. For example, data memory can still be put into the CODE region, and program code can be executed from an external Random Access Memory (RAM) region. The system-level memory region contains the interrupt controller and the debug components. These devices have fixed addresses, detailed in Chapter 5. By having fixed addresses for these peripherals, you can port applications between different Cortex-M3 products much more easily.

The Cortex-M3 Instruction Set

- Memory access instructions
- General data processing instructions
- Multiply and divide instructions
- Saturating instructions
- Bitfield instructions
- Branch and control instructions
- Miscellaneous instructions.

ADDRESSING MODES OF ARM PROCESSOR

Addressing modes of ARM processor are classified as follows:



Addressing modes for Data Processing Operand (i.e op1):

These are two methods for addressing these operands

Unmodified value In this addressing mode, the register or a value is given unmodified i.e. without any shift or rotation e.g., (i) `MOV R0, # 1234 H` This instruction will move the immediate constant value 1234 into register R0.

Modified value In this addressing mode, the given value or register is shifted or rotated. These are different shift and rotate operations possible as listed below with examples.

(1) Logical shift left This will take the value of a register and shift the value towards most Significant bits, by n bits. e.g. `MOV R0, R1, LSL # 2`

After the execution of this instruction R0 will become the value of R1 shifted 2 bits.

(2) Logical shift right This will take the value of a register and shift the value towards right by n bits. e.g. `MOV R0, R1, LSR R2` After the execution of this instruction R0 will have the value of R1 shifted right by R2 times. R1 and R2 are not altered.

(3) Arithmetic shift right This is similar to logical shift right, except that the MSB is retained as well as shifted for arithmetic shift operation e.g. `MOV R0, R1, ASR #2` After the execution of this instruction R0 will have the value of R1 Arithmetic; shifted right by 2 bits.

(4) Rotate right This will take the value of a register and rotate it right by n bits e.g. MOV R0, R1, ROR R2 After the execution of this instruction R0 will have the value of R1 rotated right for R2 times.

(5) Rotate right extended This is similar to Rotate right by one bit, with the carry flag moved into the MSB, i.e. it is similar to rotate right through carry e. g. MOV R0, R1 RRX After the execution of this instruction R0 Will have the value of register R1 rotated right through carry by 1 bit.

Addressing Modes for Memory Access Operand

As already discussed load and store instructions are used to access memory. The different memory access addressing modes are

- (i) Register indirect addressing mode
- (ii) Relative register indirect addressing mode
- (iii) Base indexed indirect addressing mode
- (iv) Base with scale register addressing mode

Each of these addressing modes have offset addressing, Pre-index addressing and post-index addressing as explained in the examples for each addressing mode

(i) Register indirect addressing mode

In this addressing mode, a register is used to give the address of the memory location to be accessed. e. g. LDR R0, [R1] This instruction will load the register R0 with the 32-bit word at the memory address held in the register R1.

(ii) Relative register indirect addressing mode In this addressing mode the memory address is generated by an immediate value added to a register. Pre index and post index are supported in this addressing mode. e. g. (a) LDR R0, [R1, #4]

This instruction will load the register R0 with the word at the memory address calculated by adding the constant address contained in the R1 register value 4 to the memory address contained in R1 register e.g. (b) LDR R0, [R1, #4]!

This is a pre-index addressing. This instruction is same as that in e. g. (a) this instruction also places the new address in R1 i.e R1 (R1 + 4. e.g. (c) LDR, [R1], #4

This is post-index addressing. This instruction will load register R0 with the word at memory address given in register R1. It will then calculate the new address by adding 4 to R1 and place this new address in R1

(iii) Base indexed indirect addressing mode In this addressing mode the memory address is generated by adding the values of two registers. Pre-index and post-index are supported also in this addressing mode. e.g. (a) LDR R0, [R1, R2]

This instruction will load the register R0 with the word at memory address calculated by adding register R1 to register R2. e.g. (b) LDR R0, [R1, R2]!

This is pre-index addressing. This instruction is same as that in e.g. (a). This instruction also places the new address in R1 i. e. $R1 = R1 + R2$. e.g. (c) LDR R0, [R1], R2

This is a post-index addressing. This instruction will load register R0 with the word at memory address given in register R1. It will then calculate the new address by adding the value in register R2 to register R1 and Place this new address in R1.

(iv) Base with scaled register addressing mode In this addressing mode the memory address is generated by a register value added to another register shifted left. Pre-index and post-index are supported in this addressing mode. e.g. (a) LDR R0, [R1, R2, LSL #2]

This instruction will load the register R0 with the word at the memory address calculated by adding register R1 with register R2 shifted left by 2 bits. e.g. (b) LDR R0, [R1, R2, LSL #2]

This is a pre-indexed addressing. This instruction will load the register R0 with the word at the memory address calculated by adding register R1 with register R2 shifted left by 2 bits. The new address is placed in register R1.

i.e. $R1 = R1 + R2 \ll 2$.

e.g. (c) LDR R0, [R1], R2, LSL #2.

This is a post-indexed addressing. This instruction will load the register R0 with the word at memory address contained in register R1. It will then calculate the new address by adding register R1 with register R2 shifted left by two bits. The new address is placed in register.

I/O PROGRAMMING IN CORTEX-M3

GPIO in Cortex-M3 LPC1768 Microcontroller is the most basic peripheral. GPIO, General Purpose Input Output is what let's your microcontroller be something more than a weak auxiliary processor. With it you can interact with physical world, connecting up other devices and turning your microcontroller into something useful. GPIO has two fundamental operating modes, input and output. Input let's you read the voltage on a pin, to see whether it's held low(0V) or high(3V) and deal with that information programatically. Output let's you set the voltage on a pin, again either high or low.

Every pin on LPC1768 can be used as GPIO pin and can be independently set to act as input or output. In next tutorial we'll get you into how to achieve these goal. I mean reading the status of switch and making LED blink. But for now we only have to look at basics, which is very important to understand before we go and build application. Depending on LPC17xx version the pinout maybe different. Here we'll focus on 100 pin LPC1768 as an example. Please keep [LPC1768 User Manual](#) with you [Chapter: 9, Page No:129]. pins on LPC1768 are divided into 5 groups (PORTs) starting from 0 to 4. Pin naming convention: P0.0 (group 0, pin 0) or (port 0, pin 0). Each pin has 4 operating modes: GPIO(default), 1st alternate function, 2nd alternate function, 3rd alternate

function. Almost all GPIO pins are powered automatically so we don't need to turn them on always. Let's have a look at details about configuration of these GPIO port pins.

1. Pin Function Setting

The LPC_PINCON register controls operating mode of these pin.

LPC_PINCON → PINSEL0 [1:0] control PIN 0.0 operating mode.

[Page No: 102, Table: 74].

.....

LPC_PINCON → PINSEL0 [31:30] control PIN 0.15 operating mode.

LPC_PINCON → PINSEL1 [1:0] control PIN 0.16 operating mode.

.....

LPC_PINCON → PINSEL1 [29:28] control PIN 0.30 operating mode.

LPC_PINCON → PINSEL2 [1:0] control PIN 1.0 operating mode.

.....

LPC_PINCON → PINSEL2 [31:30] control PIN 1.15 operating mode

LPC_PINCON → PINSEL3 [1:0] control PIN 1.16 operating mode

.....

LPC_PINCON → PINSEL3 [31:30] control PIN 1.31 operating mode

.....

LPC_PINCON → PINSEL9 [25:24] control PIN 4.28 operating mode

LPC_PINCON → PINSEL9 [27:26] control PIN 4.29 operating mode

NOTE: some register bits are reserved and are not used to control a pin for example,

LPC_PINCON → PINSEL9 [23:0] are reserved.

LPC_PINCON → PINSEL9 [31:28] are reserved.

Bit Value	Function
00	GPIO Function
01	1 st alternate function
10	2 nd alternate function
11	3 rd alternate function

Example:

To set pin 0.3 as GPIO (set corresponding bit to 00)

LPC_PINCON → PINSEL0 &= ~ ((1<<7) | (1<<6));

To set pin 0.3 as ADC channel 0.6 (2nd alternate function, set corresponding bit to 10)

LPC_PINCON → PINSEL0 &= ((1<<7) | (0<<6)); // you may omit (0<<6)

For reference follow [Page No: 117, Table: 80]

2. Pin Direction Setting

Register **LPC_GPIO_n → FIODIR [31:0]** control the pin input/output, where ‘n’ stands for pin group (0-4). To set a pin as output, set the corresponding bit to ‘1’. To set a pin as input, set the corresponding bit to ‘0’, by default, all pins are set as input (all bits are 0).

Example:

To set 0.3 as output

LPC_GPIO → FIODIR |= (1<<3);

3. Pin is Set as Output

A pin digital high/low setting

LPC_GPIO_n → FIOSET is used to turn a pin to HIGH. Register LPC_GPIO_n → FIOCLR is used to turn a pin to low. To turn a pin to digital ‘1’ (high), set the corresponding bit of LPC_GPIO_n → FIOSET to 1. To turn a pin to digital ‘0’ (low), set the corresponding bit of LPC_GPIO_n → FIOCLR to 1.

Example

Turn Pin 0.3 to high

LPC_GPIO0 → FIOSET |= (1<<3);

If we set LPC_GPIO_n → FIOSET bit to ‘0’ there is no effect.

Turn Pin 0.3 to low

LPC_GPIO0 → FIOCLR |= (1<<3);

If we set LPC_GPIO_n → FIOCLR bit to ‘0’ there is no effect.

4. Pin is Set to Input

- **Read a Pin Value**

Register LPC_GPIO_n → FIOPIN stores the current pin state. The corresponding bit is ‘1’ indicates that the pin is driven high.

Example

To read current state of Pin 0.3

Value = ((LPC_GPIO0 → FIOPIN & (1<<3)) >> 3);

Note: write 1/0 to corresponding bit in LPC_GPIO_n → FIOPIN can change the output of the pin to 1/0 but it is not recommended. We should use LPC_GPIO_n → FIOSET and GPIO_n → FIOCLR instead.

- **Pin Internal Pull up Setting**

Register LPC_PINCON → PINMODE_n is used to set up a pin internal pull-up.

LPC_PINCON → PINMODE0 [1:0] control P0.0 internal pull-up

.....

LPC_PINCON → PINMODE0 [31:30] control P0.15 internal pull-up,

Please see LPC_PINCON → PINSEL_n for the full list [Page No: 114].

Bit Value	Pin Mode
00	On chip pull-up resistor enabled

01	Repeater Mode
10	Tri-State mode, (neither pull-up nor pull-down)
11	On chip pull-down resistor enabled

- **Example:**
By default all pins which are set as input has internal pull-up on (00).
- **To disable internal pull-up on pin 0.3**
LPC_PINCON → *PINSEL0* |= (1<<7);

+++++



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

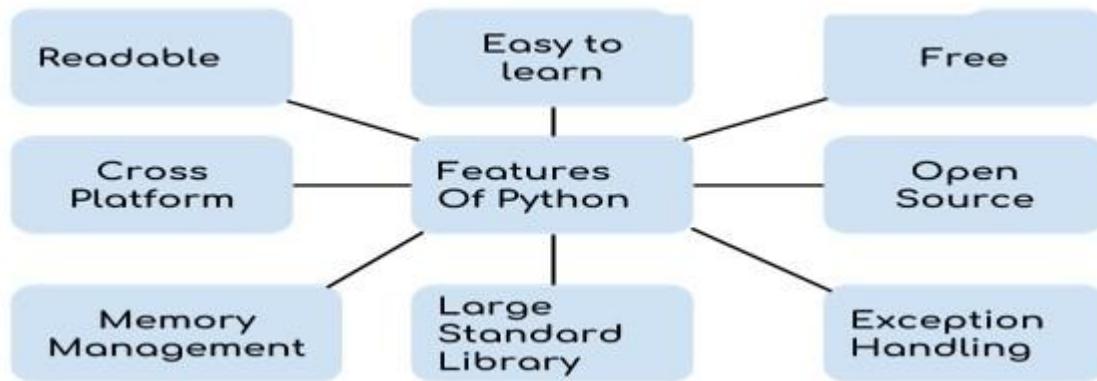
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – IV - PYTHON– SPHA5204

Introduction to Python

Python is developed by **Guido van Rossum**. Guido van Rossum started implementing Python in 1989. Python is a very simple programming language so even if you are new to programming, you can learn python without facing any issues.



Readable: Python is a very readable language.

Easy to Learn: Learning python is easy as this is a expressive and high level programming language, which means it is easy to understand the language and thus easy to learn.

Cross platform: Python is available and can run on various operating systems such as Mac, Windows, Linux, Unix etc. This makes it a cross platform and portable language.

Open Source: Python is a open source programming language.

Large standard library: Python comes with a large standard library that has some handy codes and functions which we can use while writing code in Python.

Free: Python is free to download and use. This means you can download it for free and use it in your application. See: Open Source Python License. Python is an example of a FLOSS (Free/Libre Open Source Software), which means you can freely distribute copies of this software, read its source code and modify it.

Supports exception handling: If you are new, you may wonder what is an exception? An exception is an event that can occur during program execution and can disrupt the normal flow of program. Python supports exception handling which means we can write less error prone code and can test various scenarios that can cause an exception later on.

Advanced features: Supports generators and list comprehensions. We will cover these features later.

Automatic memory management: Python supports automatic memory management which means the memory is cleared and freed automatically. You do not have to bother clearing the memory.

Applications:

Web development – Web framework like Django and Flask are based on Python. They help you write server side code which helps you manage database, write backend programming logic, mapping urls etc.

Machine learning – There are many machine learning applications written in Python. Machine learning is a way to write a logic so that a machine can learn and solve a particular problem on its own. For example, products recommendation in websites like Amazon, Flipkart, eBay etc. is a machine learning algorithm that recognises user's interest. Face recognition and Voice recognition in your phone is another example of machine learning.

Data Analysis – Data analysis and data visualisation in form of charts can also be developed using Python.

Scripting – Scripting is writing small programs to automate simple tasks such as sending automated response emails etc. Such type of applications can also be written in Python programming language.

Introduction to Different IoT Tools

IoT Tools stands for the Internet of Things Tools. It is a network or connection of devices, vehicles, equipment applying embedded electronics, home appliances, buildings and many more. This helps in collecting and exchanging different kinds of data. It also helps the user to control the devices remotely over a network.

Today in the internet-driven world, IoT has engulfed the IT industry and is the latest buzzword. It has opened many new horizons for companies and developers working on IoT. Many exceptional products have been developed due to IoT app development. Companies providing Internet of Things solution are creating hardware and software designs to help the IoT developers to create new and remarkable IoT devices and applications.

List of IoT Tools:

Some IoT tools that help developers in developing IoT applications and devices are discussed below:

1. Tessel 2

It is used to build basic IoT prototypes and applications. It helps through its numerous modules and sensors. Using Tessel 2 board, a developer can avail Ethernet connectivity, Wi-Fi

connectivity, two USB ports, a micro USB port, 32MB of Flash, 64MB of RAM. Additional modules can also be integrated like cameras, accelerometers, RFID, GPS, etc.

Tessel 2 can support Node.JS and can use libraries of Node.JS. It contains two processors, its hardware uses 48MHz Atmel SAMD21 and 580.

MHz MediaTek MT7620n coprocessor. One processor can help to run firmware applications at high speed and the other one helps in the efficient management of power and in exercising good input/output control.

2. Eclipse IoT

This tool or instrument allows the user to develop, adopt and promote open source IoT technologies. It is best suited to build IoT devices, Cloud platforms, and gateways. Eclipse supports various projects related to IoT. These projects include open-source implementations of IoT protocols, application frameworks and services, and tools for using Lua programming language which is promoted as the best-suited programming language for IoT.

3. Arduino

Arduino is an Italy based IT company that builds interactive objects and microcontroller boards. It is an open-source prototyping platform that offers both IoT hardware and software. Hardware specifications can be applied to interactive electronics and software includes Integrated Development Environment (IDE). It is the most preferable IDEs in all IoT development tools. This platform is easy and simple to use.

4. Platform IoT

It is a cross-platform IoT IDE. It comes with the integrated debugger. It is the best for mobile app development and developers can use a friendly IoT environment for development. A developer can port the IDE on Atom editor or it can install it as a plugin. It is compatible with more than 400 embedded boards and has more than 20 development frameworks and platforms. It offers a remarkable interface and is easy to use.

5. M2M Labs Mainspring

It is an IoT platform and an open source application framework. It is used to build a machine to machine applications (M2M) which can be used in fields of remote monitoring and fleet management. It supports much functionality like validation and normalization of data, device configuration, data retrieval processes and flexible modeling of devices. It is based on Apache, Cassandra, NoSQL database and Java.

6. Kinoma

It is a Marvell semiconductor hardware prototyping platform. It enables three different projects. To support these projects two products are available Kinoma Create and Element Board. Kinoma Create is a hardware kit for prototyping electronic and IoT enabled devices. Kit contains supporting essentials like Bluetooth Low Energy (BLE), integrated Wi-Fi, speaker, microphone and touch screen. Element Board is the smallest JavaScript-powered IoT product platform.

7. Device- Hive

It is based on Data Art's AllJoyn. It is a free open source M2M i.e. machine to machine communication framework. It was launched in 2012 and considered the most preferable IoT app development platform. It has cloud-based API which can be controlled remotely irrespective of network configuration. Its libraries, protocols, and management portal are controlled in a similar manner. It is best suited for applications related to smart home tech, security, automation, and sensors.

8. Kaax

It provides end to end support for IoT devices connected across the cloud. Due to its multi-purpose middleware, it allows users to build connected applications, IoT applications, and many smart products. Open source kit is described as 'hardware agnostic' by Kaax i.e. it can interface with any hardware like sensors, gateways, and other devices. It helps developers to distribute firmware updates remotely, and to enable cross-platform interoperability.

9. Home Assistant

It is an open source tool mostly used for functions based on the Python coding system and home automation. Desktop and mobile browsers help to control their IoT system. It is easy to set up and is famous for its smooth operations, privacy standards, and security. It can support systems running on Python 3.

10. Net

It is an integrated solution for developers of IoT. It offers services like cloud integration and business intelligence to provide both web technologies and hardware. Its development kit is delivered as a platform as a service i.e. PaaS which allows the developers to efficiently utilize its power for development purpose.

11. Raspbian

This IDE is created for Raspberry Pi board. It has more than 35000 packages and with the help of precompiled software, it allows rapid installation. It was not created by the parent organization but by the IoT tech enthusiasts. For working with Raspberry Pi, this is the most suitable IDE available.

Developing Application through IOT Tools:

Technology is on a constant innovative role and with that perspective has brought a significant change in our lives. With the passage of time connectivity has improved a lot better; thanks to the smart devices, the wireless communication, sensors, cloud based computing system and much more.

But the one platform that has already created the buzzword and hype is Internet of Things (IoT), enabling the organizations to take controls and manage the operations conveniently thereby undertaking the tougher projects to deal with.

The Three Pillars of IoT

It is important for the app developers to note that the entire structure of the Internet of Things, basically rests upon three major pillars. They include:

- Network
- Things in Themselves
- Cloud

The network usually performs the same function to what router does in connecting the network to the device. Here the devices are linked to the cloud. The information is received from the infrastructure stationed at data centers. The things provide the data stream and also manages it. On the other hand, the things are organized by software. The Things in themselves acts as an Internet Gateway is regarded as an important structure that helps in other device communication through a single or many communication protocol. The processor is not high powered and in most of the cases it gets directly linked up to Internet of Things. The Operating System is also embedded. The device that gets connected to the network normally does not have a screen. The Cloud is a server that serves as a security cover safeguarding your confidential data. During the critical juncture, the ordered data gets processed whereas the processing of a program occurs during the concluding stages. This program can be anything from a web app to a mobile app or even a software that users make use of.

Applications for IoT

So, now that you finally sit down for developing an application for the Internet of Things, there are a few factors taken into consideration. Let's have a quick glance at these:

1. Choose an Appropriate and Convenient Platform

The first and foremost step that the developer needs to ensure is selecting the appropriate platform for the development process. The platforms such as Ubidots, Xively or Thingworx are IoT proven and offer the scope to design the best in class apps. Apart from that if you are choosing an authenticated platform, you are also avoiding the unnecessary exposures. With the help of these platforms you don't have to start anything from the beginning.

2. Consider the Industry for IoT Application

Today Internet of Things does not have limited services but its scope has much widened and extended. So, it is essential to consider the industry. The industries are connected with the devices and network to offer solutions. As such there are a diverse set of industries that is optimally connected such as the healthcare, transportation, energy resources, sports, manufacturing etc.

For instance, it will become easier for the people to find transportation such as connecting buses or trains. Side by side you will also have to find ways to improve in connecting the things.

3. Segregating services from API Interface

While you are developing the apps for IoT, it becomes important to separate the services from API interface. But why? This is because you want your app to smoothly run on mobile and web desktop. Managing your IoT applications well will help to provide better opportunities.

4. IoT Data Must be Secured Strongly

It becomes the sheer responsibility of the application developer to offer a strong secured environment to IoT data especially from the physical attacks. The security becomes paramount in case of GPS networks or in case of banking apps.

5. The Different Levels of IoT Apps

Understanding the various levels of IoT applications is pivotal as it gives an idea to know the system and its function. There are four different layers; the devices, the ingestion tier, the analytics area and lastly the end-user. So, first consider the devices that you will be connecting. In the ingestion tier the infrastructure or the software receives data or organizes it. In the third

layer the data is mainly processed with the help of analytics area. The last is the end users for whom the app is getting developed.

6. Keep an Eye on IoT Device Firmware Security

The Internet of Things is always connected with things and keeps communicating with it. This is what distinguishes them from the traditional web and mobile apps. The hardware is always apprehended to have security based issues in the firmware and so it becomes essential keep on updating firmware. The firmware needs to be authenticated and signed before the update.

7. Not Compromise with Speed and Quality

As an app developer you need to keep in mind when creating applications for the IoT, you cannot compromise with the speed and quality at any cost. You need to focus on transforming the ideas into actuality and offer stable working prototype. Once you achieve, you can think of being successful.

8. Provide Scalability to the Applications

The Internet of Things based applications should be scalable. The IoT is still a new concept but it has already been predicted that it has immense potential and will become larger than ever with the time to arrive.

Scalability will allow your app to remain in light even after a long period of time. The Internet of Things is although new into the technological arena but isn't a foreign term any more. Gradually it is expanding and has reached to a different height where accessing information and getting connected has become easier and cost effective. The applications for IoT is a challenge for the developers because it is not based on the conventional methods unlike web or mobile apps.

SENSOR BASED APPLICATION THROUGH EMBEDDED SYSTEM

SENSORS

Sensors are synonymous to human sense organs. These are the eyes of modern electronic devices utilized to implement automation in different practical fields. Sensors form the basic components of many systems, in which a process is controlled based on the signals sensed by the sensors. Sensors are modern electronic devices used frequently to detect various signals generated as a response to various natural or artificial ambient factors. In effect, a sensor

converts a physical parameter into a signal, generally in terms of voltage or current. The physical parameter that a sensor responds to may be temperature, pressure, humidity, velocity, radiation, vibration, etc. Sensors are the main parts of many measurement devices like thermometer, barometer, accelerometer, etc. Sensors can be coupled in many ways to build measuring and the final process control device. Mechatronics is a branch of engineering where mechanical components are organized in such a way that their working is controlled by some electronic circuits. Here sensors play an important role in the controlling of the mechanical processes. Therefore sensors are termed as the electronic eyes of many automatic systems where a larger computer or microcontroller is generally used. Numerous sensors and their applications are available today. A comparative study of the performance of an individual sensor in a specific application gives a clear vision which sensor can be employed singly or in conjunction with others. In any sensor based application, the working system requires economy and simplicity for its real life implementation. For selecting a sensor for a particular application there are certain features that have to be considered. These are accuracy, ambient conditions like temperature or humidity, range of measurement, ease of calibration, resolutions, cost, etc.

SOME USEFUL SENSORS

Capacitive Sensors Capacitors are basically two conducting plates separated by some dielectric media. From its physical parameters the value of capacitance can be expressed as $C = \frac{\epsilon A}{d}$ where ϵ is the dielectric constant of the medium between the plates, A is the area of a plate projected on the other plate and d is the distance between the plates. A capacitor can be constructed in a number of ways out of which some common forms are: parallel plate capacitor, spherical capacitor, cylindrical capacitor etc. Some very special forms are used for some specific applications and one such form is the fringe field capacitor. By changing the value of different parameters, the value of a capacitor can be changed. Based on this simple proposition a capacitor can be utilized as a sensor [73]. Capacitive sensors can directly sense a variety of things like motion, chemical composition, electric field and also indirectly sense many other variables which can be converted into motion or dielectric constant, such as pressure, acceleration, fluid level and fluid composition. They are built using conductive sensing electrodes in a dielectric, with excitation voltages of the order of five volts and detection circuits which turn a capacitance variation into a voltage, frequency, or pulse width variation. Capacitive sensor based embedded system is also used to determine moisture content of tea leaves .

Light Sensors

Eye is the light sensor that every high level creature is equipped with. Human eye sense visible light of electro-magnetic spectrum whereas other creatures have different forms of organs capable of sensing wide range of spectrum. Light sensors are obtained by exploiting the variations of some physical property of an object when exposed to light. Some of these objects are light dependent resistance (LDR), photo diode, charged couple device (CCD), solar cell

etc. A very common use of LDR as a light sensor is in automatic street lighting system, where the street light glows on at night and glows off at day light. Light sensors are widely used for a number of applications [75] [76] [77][78]. Monitoring of moisture in a transformer oil using optical fiber as sensor has been discussed by Laskar and Bordoloi. Pressure Sensors A sensor capable of sensing pressure and its magnitude is termed as a pressure sensor. A diaphragm made by etching a single silicon crystal is widely used to form a miniature pressure sensor. Most of these sensors use piezo-resistive elements to quantify the state of the diaphragm on application of a pressure. Another well known approach for the quantification of the state of a diaphragm is to use a variable gap capacitor placed between the diaphragm and an attached chip, which directly measures the deflection of the diaphragm on application of pressure [80]. This arrangement is useful to measure weight as pressure exerted by an object is directly proportional to its weight. Hazarika and Pegu [81] have published a microcontroller based air pressure monitoring instrumentation system using optical fibres as sensor.

Chemical Sensors

A sensor is said to be a chemical sensor which contains an analytical device that can sense data about the chemical composition of a liquid or a gas sample. The sensor provides the information in the form of a measurable physical quantity that is correlated with the chemical composition of the given sample. In the process of sensing the two main steps involved are recognition and transduction.

In the recognition step, the molecules of the target composition interact selectively with receptor molecules included in the structure of the recognition element of the sensor. As a result, an attribute of a physical parameter varies and this variation is utilized to generate the output signal.

4.3.5. Biosensors If a chemical sensor is equipped with a biological material as a recognition element, the sensor is termed as a biosensor. Alternately, in biomedicine and biotechnology, sensors which detect biological objects such as cells, protein, nucleic acid or bio mimetic polymers, are called biosensors.

EMBEDDED SYSTEM

The word embedded implies that some entity is homogeneously integrated within a system. In the present day context, an embedded system is a hardware electronics arrangement within which a software is loaded in the memory element to drive the system to achieve its goal. The system may or may not be programmable depending on its application. An embedded system is a realization of a definition of some kind of automatic process guided by a set of rules. The hardware and their response to the real time working environment are bonded together by the embedded software. An embedded system can be visualized as a computational unit with essential software, though it is not a computer in the conventional way. It can be specified as

A stand alone system designed to perform a particular task without any human intervention.

A hardware and software system intended to perform a specific job in an efficient and cheaper way.

An intelligent system capable of taking input from the environment and produce appropriate response.

A system capable of taking inputs from the environment along with processing of the parameter to give a suitable output, all within a small time period.

The system which works conveniently in time critical situations where responses in a specific time frame are required, because of its capability of taking input processing-output actions very fast.

SCHEMATIC OF EMBEDDED SYSTEM

An embedded system is fast in response to its inputs, small in size, consumes low power and over all it is reliable in operations. These are generally designed for real time applications. Therefore their power requirement and size should be small enough to incorporate them in practical applications. To minimise their size and power requirement, the hardware is provided in the form of a microcontroller which provide computation engine, input/output facilities, communication ports, user interface, memory and display etc. A typical block diagram of an embedded system is shown in

The software embedded is written in assembly level or high level language and compiled to machine level code before it is burnt into the EPROM inbuilt with the microcontroller which acts as the computational engine of the system. The software is written in such a way that it will be compact enough to be burnt in a small memory area, consumes low power during use and compatible to processor speed to achieve efficiency.

TYPES OF EMBEDDED SYSTEM

Based on practical requirements, performance, size an embedded system can be classified into different types. A typical classification is shown below: According to performance and functional requirement, embedded systems have been classified into four categories, viz.

- Real time embedded systems
- Stand alone embedded system
- Networked embedded systems
- Mobile embedded systems

According to the scale of performance of the microcontroller, embedded systems have been classified into three categories, viz.

- Small scale embedded systems
- Medium scale embedded systems
- Sophisticated embedded systems

APPLICATIONS OF EMBEDDED SYSTEM

Embedded systems are used in many different applications like house hold electronics items, automobiles, aircraft, military weapons, telecommunications, smart cards, missiles, satellites, computer networking, surveillance systems, medical electronics, industrial control, digital consumer electronics etc. In house hold applications, embedded systems include washing machine, dishwashers, electric oven, home security systems, automatic door opening & closing systems, keyless entry system etc. Embedded systems in automobiles and in telecommunications cover applications in vehicle and cruise control system, body and engine safety, keyless starting system, entertainment devices in car like MP3 players etc, ecommerce and mobile phone, robotics in industrial assembly line, mobile computing, wireless communications, computer networking etc. Embedded systems in smart cards, missiles and satellites category includes applications in security systems, telephone and banking sectors, defence weaponry and aerospace devices, communication. Embedded systems in peripherals & computer networking are seen in blade servers, routers, wireless P. C. card, card swiping system, displays and monitors, image processing, network cards and printers etc. Pacemakers, various control systems in MRI, CT scan, X-ray machines are few examples of embedded systems in the area of medical electronics. Embedded systems in consumer electronics include digital camera, HD TV, DVD players, MP3 players, Set top box etc.

Implementing IOT Concepts with Python:

Python in IoT development

Python plays a significant role in developing internet of things, along with python we use different languages for developing IOT those are

- Assembly
- B#
- C
- C++
- Java
- Javascript
- Php
- Python
- Rust and many more

As of now java programming language is widely used for developing the IoT devices now python is coming into the field, with the following features of python most of developers prefer python programming language.

PART A

1. Describe the role of Python in IoT development
2. Develop the features of Python
3. Classify the IoT hardware processing
4. Evaluate the history of IoT techniques
5. Analyze the common challenges in IoT
6. Explain how sensors are interfaced with in Embedded system
7. Identify the classification of Embedded system
8. Evaluate the communication devices used in IoT

PART B

1. Analyze the different IoT Tools
2. Discriminate the developing applications through IoT Tools
3. Design an embedded system using processor, external memory and sensor
4. Discuss the IoT concepts using Python
5. Explain the functional layers and capabilities of an IoT with a neat diagram.

TEXT / REFERENCE BOOKS

1. A.K Ray and K M Bhurchandi, Advanced Microprocessors and Peripherals, 3RD edition, TMH, 2017.
2. Joseph Yiu, The Definitive Guide to the ARM Cortex-M3, 2nd Edition, Newnes, 2015.
3. Dr. Mark Fisher, ARM Cortex M4 Cookbook, Packt, 2016.
4. David Hanes, "IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things", Cisco press, 2017.
5. Olivier Hersent, David Boswarthick, Omar Elloumi, "The Internet of Things: Key Applications and Protocols", 2nd Edition, Wiley, 2012.
6. Rajkamal, "Embedded system-Architecture, Programming, Design", TMH, 2011.
7. Jonathan W. Valvano, "Embedded Microcomputer Systems, Real Time Interfacing", Cengage Learning, 3rd Edition, 2012.

1. INTRODUCTION TO PYTHON:

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.

It was created by Guido van Rossum during 1985- 1990.

Python got its name from “Monty Python’s flying circus”. Python was released in the year 2000.

- **Python is interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-Level programmers and supports the development of a wide range of applications.

Python Features:

- **Easy-to-learn:** Python is clearly defined and easily readable. The structure of the program is very simple. It uses few keywords.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Interpreted:** Python is processed at runtime by the interpreter. So, there is no need to compile a program before executing it. You can simply run the program.
- **Extensible:** Programmers can embed python within their C,C++,JavaScript , ActiveX, etc.
- **Free and Open Source:** Anyone can freely distribute it, read the source code, and edit it.
- **High Level Language:** When writing programs, programmers concentrate on solutions of the current problem, no need to worry about the low level details.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

Applications:

- ❖ Bit Torrent file sharing
- ❖ Google search engine, YouTube
- ❖ Intel, Cisco, HP,IBM
- ❖ i-Robot
- ❖ NASA
- ❖ Face book, Drop box

Python interpreter:

Interpreter: To execute a program in a high-level language by translating it one line at a time.

Compiler: To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

Compiler	Interpreter
Compiler Takes Entire program as input	Interpreter Takes Single instruction as input
Intermediate Object Code is Generated	No Intermediate is Object Code Generated
Conditional Control Statements are Executed faster	Conditional Control Statements are Executed slower
Memory Requirement is More (Since Object Code is Generated)	Memory Requirement is Less
Program need not be compiled every time	Every time higher level program is converted into lower level program
Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)
Example : C Compiler	Example : PYTHON

Modes of python interpreter:

Python Interpreter is a program that reads and executes Python code. It uses 2 modes of Execution.

1. Interactive mode
2. Script mode

Interactive mode:

- ❖ Interactive Mode, as the name suggests, allows us to interact with OS.
- ❖ When we type Python statement, **interpreter displays the result(s) immediately.**

Advantages:

- ❖ Python, in interactive mode, is good enough to learn, experiment or explore.
- ❖ Working in interactive mode is convenient for beginners and for testing small pieces of code.

Drawback:

- ❖ We cannot save the statements and have to retype all the statements once again to re-run them.

In interactive mode, you type Python programs and the interpreter displays the result:

```
>>> 1 + 1
```

```
2
```

The chevron, >>>, is the prompt the interpreter uses to indicate that it is ready for you to enter code. If you type 1 + 1, the interpreter replies 2.

```
>>> print('Hello, World!')
```

```
Hello, World!
```

This is an example of a print statement. It displays a result on the screen. In this case, the result is the words.

```

Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2+5
7
>>> 2**6
64
>>> a=3
>>> b=6
>>> c=a-b
>>> print(c)
-3
>>> print("good morning")
good morning
>>> |

```

Script mode:

- ☐ In script mode, we type python program in a file and then use interpreter to execute the content of the file.
- ☐ Scripts can be saved to disk for future use. **Python scripts have the extension .py**, meaning that the filename ends with .py
- ☐ Save the code with **filename.py** and run the interpreter in script mode to execute the script.

Example:

```

print(1)
x = 2
print(x)

```

Output:

```

>>>1
2

```

Interactive mode	Script mode
A way of using the Python interpreter by typing commands and expressions at the prompt.	A way of using the Python interpreter to read and execute statements in a script.
Can't save and edit the code	Can save and edit the code
If we want to experiment with the code, we can use interactive mode.	If we are very clear about the code, we can use script mode.
we cannot save the statements for further use and we have to retype all the statements to re-run them.	we can save the statements for further use and we no need to retype all the statements to re-run them.
We can see the results immediately.	We can't see the code immediately.

Integrated Development Learning Environment(IDLE):

- ☐ Is a **graphical user interface** which is completely written in Python.
- ☐ It is bundled with the default implementation of the python language and also comes with optional part of the Python packaging.

Features of IDLE:

- ☐ Multi-window text editor with syntax highlighting.

- ❑ Auto completion with **smart indentation**.
- ❑ **Python shell** to display output with syntax highlighting.

2. VALUES AND DATATYPES

Value:

Value can be any letter, number or string.

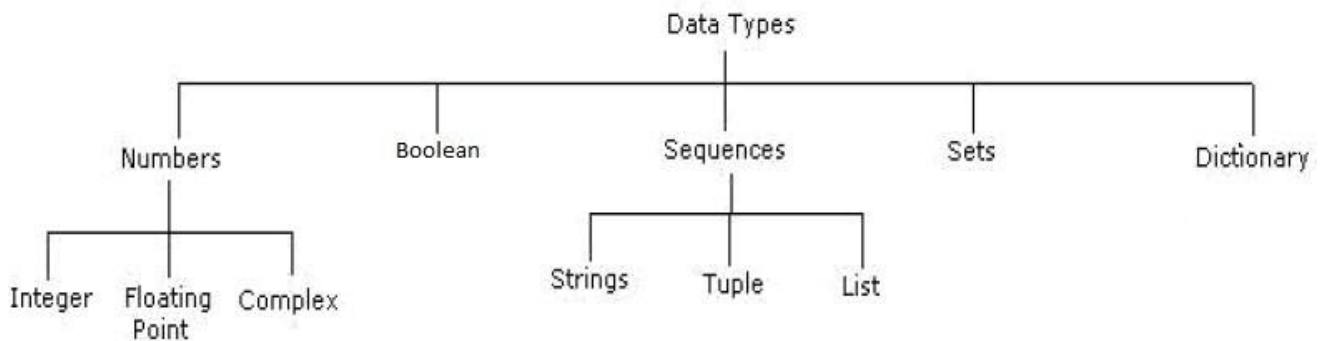
Eg, Values are 2, 42.0, and 'Hello, World!'. (These values belong to different datatypes.)

Data type:

Every value in Python has a data type.

It is a set of values, and the allowable operations on those values.

Python has four standard data types:



Numbers:

- ❖ Number data type stores **Numerical Values**.
- ❖ This data type is immutable [i.e. values/items cannot be changed].
- ❖ Python supports integers, floating point numbers and complex numbers. They are defined as,

Integers	Long	Float	Complex
<ul style="list-style-type: none"> - They are often called just integers or int. - They are positive or negative whole numbers with no decimal point. 	<ul style="list-style-type: none"> -They are long integers. -They can also be represented in <u>octal</u> and hexadecimal representation. 	<ul style="list-style-type: none"> -They are written with a decimal point dividing the integer and the fractional parts. 	<ul style="list-style-type: none"> -They are of the form $a + bj$, where a and b are floats and j represents the square root of -1 (which is an imaginary number). -The real part of the number is a, and the imaginary part is b.
Eg, 56	Eg, 5692431L	Eg, 56.778	Eg, square root of -1 is a complex number

Sequence:

- ❖ A sequence is an **ordered collection of items**, indexed by positive integers.
- ❖ It is a combination of **mutable** (value can be changed) and **immutable** (values cannot be changed) datatypes.

❖ There are three types of sequence data type available in Python, they are

1. **Strings**
2. **Lists**
3. **Tuples**

Strings:

- A String in Python consists of a series or sequence of characters - letters, numbers, and special characters.
- Strings are marked by quotes:
 - Single quotes(' ') E.g., 'This a string in single quotes'
 - double quotes(" ") E.g., "This a string in double quotes"
 - triple quotes(""" """)E.g., """This is a paragraph. It is made up of multiple lines and sentences."""
- Individual character in a string is accessed using a subscript(index).
- Characters can be accessed using indexing and slicing operations .Strings are Immutable i.e the contents of the string cannot be changed after it is created._

Indexing:

String A	H	E	L	L	O
Positive Index	0	1	2	3	4
Negative Index	-5	-4	-3	-2	-1

- Positive indexing helps in accessing the string from the beginning
- Negative subscript helps in accessing the string from the end.
- Subscript 0 or –ven(where n is length of the string) displays the first element.

Example: A[0] or A[-5] will display “H”

- Subscript 1 or –ve (n-1) displays the second element.

Example: A[1] or A[-4] will display “E”

Operations on string:

- i. Indexing
- ii. Slicing
- iii. Concatenation
- iv. Repetitions
- v. Membership

Creating a string	>>> s="good morning"	Creating the list with elements of different data types.
Indexing	>>>print(s[2]) o >>>print(s[6]) O	❖ Accessing the item in the position0 ❖ Accessing the item in the position2
Slicing(ending position -1)	>>>print(s[2:]) od morning	- Displaying items from 2 nd till last.

<u>Slice operator is used to extract part of a data type</u>	>>>print(s[:4]) Good	- Displaying items from 1 st position till 3 rd .
Concatenation	>>>print(s+"friends") good morning friends	-Adding and printing the characters of two strings.
Repetition	>>>print(s*2) good morning good morning	Creates new strings, concatenating multiple copies of the same string
in, not in (membership operator)	>>> s="good morning" >>>"m" in s True >>> "a" not in s True	Using membership operators to check a particular character is in string or not. Returns true if present.

Lists

- ❖ List is an ordered sequence of items. Values in the list are called elements /items.
- ❖ It can be written as a list of comma-separated items (values) between **square brackets[]**.
- ❖ Items in the lists can be of different datatypes.

Operations on list:

Indexing
Slicing
Concatenation
Repetitions
Updation, Insertion, Deletion

Creating a list	>>>list1=["python", 7.79, 101, "hello"] >>>list2=["god",6.78,9]	Creating the list with elements of different data types.
Indexing	>>>print(list1[0]) python >>>list1[2] 101	❖ Accessing the item in the position0 ❖ Accessing the item in the position2
Slicing(ending position -1) <u>Slice operator is used to extract part of a string, or some part of a list</u> <u>Python</u>	>>>print(list1[1:3]) [7.79, 101] >>>print(list1[1:]) [7.79, 101, 'hello']	- Displaying items from 1st till2nd. - Displaying items from 1 st position till last.
Concatenation	>>>print(list1+list2) ['python', 7.79, 101, 'hello', 'god',	-Adding and printing the items of two lists.

	6.78, 9]	
Repetition	<pre>>>>list2*3 ['god', 6.78, 9, 'god', 6.78, 9, 'god', 6.78, 9]</pre>	Creates new strings, concatenating multiple copies of the same string
Updating the list	<pre>>>>list1[2]=45 >>>print(list1) ['python', 7.79, 45, 'hello']</pre>	Updating the list using index value
Inserting an element	<pre>>>>list1.insert(2,"program") >>> print(list1) ['python', 7.79, 'program', 45, 'hello']</pre>	Inserting an element in 2 nd position
Removing an element	<pre>>>>list1.remove(45) >>> print(list1) ['python', 7.79, 'program', 'hello']</pre>	Removing an element by giving the element directly

Tuple:

- ❖ A tuple is same as list, except that the set of elements is **enclosed in parentheses** instead of square brackets.
- ❖ **A tuple is an immutable list.** i.e. once a tuple has been created, you can't add elements to a tuple or remove elements from the tuple.
- ❖ Benefit of Tuple:
- ❖ Tuples are faster than lists.
- ❖ If the user wants to protect the data from accidental changes, tuple can be used.
- ❖ Tuples can be used as keys in dictionaries, while lists can't.

Basic Operations:

Creating a tuple	<pre>>>>t=("python", 7.79, 101, "hello")</pre>	Creating the tuple with elements of different data types.
Indexing	<pre>>>>print(t[0]) python >>>t[2] 101</pre>	<ul style="list-style-type: none"> ❖ Accessing the item in the position0 ❖ Accessing the item in the position2
Slicing(ending position -1)	<pre>>>>print(t[1:3]) (7.79, 101)</pre>	❖ Displaying items from 1st till 2nd.
Concatenation	<pre>>>>t+("ram", 67) ('python', 7.79, 101, 'hello', 'ram', 67)</pre>	❖ Adding tuple elements at the end of another tuple elements
Repetition	<pre>>>>print(t*2) ('python', 7.79, 101, 'hello', 'python', 7.79, 101, 'hello')</pre>	❖ Creates new strings, concatenating multiple copies of the same string

Altering the tuple data type leads to error. Following error occurs when user tries to do.

```
>>>t[0]="a"
```

Trace back (most recent call last):

File "<stdin>", line 1, in <module>

Type Error: 'tuple' object does not support item assignment

Mapping

-This data type is unordered and mutable.

-Dictionaries fall under Mappings.

Dictionaries:

- ❖ Lists are ordered sets of objects, whereas **dictionaries are unordered sets**.
- ❖ Dictionary is created by using **curly brackets**. i.e.{ }
- ❖ Dictionaries **are accessed via keys** and not via their position.
- ❖ A dictionary is an associative array (also known as hashes). Any key of the dictionary is associated (or mapped) to a value.
- ❖ The values of a dictionary can be any Python data type. So dictionaries are **unordered key-value-pairs**(The association of a key and a value is called a key- value pair)

Dictionaries don't support the sequence operation of the sequence data types like strings, tuples and lists.

Creating a dictionary	<pre>>>> food = {"ham": "yes", "egg": "yes", "rate": 450 } >>> print(food) {'rate': 450, 'egg': 'yes', 'ham': 'yes'}</pre>	Creating the dictionary with elements of different data types.
Indexing	<pre>>>>> print(food["rate"]) 450</pre>	Accessing the item with keys.
Slicing(ending position -1)	<pre>>>> print(t[1:3]) (7.79, 101)</pre>	Displaying items from 1st till 2nd.

If you try to access a key which doesn't exist, you will get an error message:

```
>>> words = {"house": "Haus", "cat": "Katze" }
```

```
>>> words["car"]
```

Traceback (most recent call last): File

"<stdin>", line 1, in <module> KeyError: 'car'

Data type	Compile time	Run time
int	a=10	a=int(input("enter a"))
float	a=10.5	a=float(input("enter a"))
string	a="panimalar"	a=input("enter a string")
list	a=[20,30,40,50]	a=list(input("enter a list"))
tuple	a=(20,30,40,50)	a=tuple(input("enter a tuple"))

3.Variables,Keywords Expressions, Statements, Comments, Docstring ,Lines And Indentation, Quotation In Python, Tuple Assignment:

VARIABLES:

- ❖ A variable allows us to store a value by assigning it to a name, which can be used later.
- ❖ Named memory locations to store values.
- ❖ Programmers generally choose names for their variables that are meaningful.
- ❖ It can be of any length. No space is allowed.
- ❖ We don't need to declare a variable before using it. In Python, we simply assign a value to a variable and it will exist.

Assigning value to variable:

Value should be given on the right side of assignment operator(=) and variable on left side.

```
>>>counter =45  
print (counter)
```

Assigning a single value to several variables simultaneously:

```
>>> a=b=c=100
```

Assigning multiple values to multiple variables:

```
>>>a,b,c=2,4,"ram"
```

KEYWORDS:

- ❖ Keywords are the reserved words in Python.
- ❖ We cannot use a keyword as name, function name or any other identifier.
- ❖ They are used to define the syntax and structure of the Python language.
- ❖ Keywords are case sensitive.

<i>False</i>	<i>class</i>	<i>finally</i>	<i>is</i>	<i>return</i>
<i>None</i>	<i>continue</i>	<i>for</i>	<i>lambda</i>	<i>try</i>
<i>True</i>	<i>def</i>	<i>from</i>	<i>nonlocal</i>	<i>while</i>
<i>and</i>	<i>del</i>	<i>global</i>	<i>not</i>	<i>with</i>
<i>as</i>	<i>elif</i>	<i>if</i>	<i>or</i>	<i>yield</i>
<i>assert</i>	<i>else</i>	<i>import</i>	<i>pass</i>	
<i>break</i>	<i>except</i>	<i>in</i>	<i>raise</i>	

IDENTIFIERS:

Identifier is the name given to entities like class, functions, variables etc. in Python.

- ❖ Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).

- ❖ all are valid example.
- ❖ An identifier cannot start with a digit.
- ❖ Keywords cannot be used as identifiers.
- ❖ Cannot use special symbols like!, @, #, \$, % etc. in our identifier.
- ❖ Identifier can be of any length.

Example:

Names like myClass, var_1, and **this_is_a_long_variable**

Valid declarations	Invalid declarations
Num	Number 1
Num	num1
Num1	addition of program
_NUM	1Num
NUM_temp2	Num.no
IF	if
Else	else

STATEMENTS AND EXPRESSIONS:

Statements:

- Instructions that a Python interpreter can executes are called statements.
- A statement is a unit of code like creating a variable or displaying avalue.

```
>>> n = 17
>>> print (n)
```

Here, The first line is an assignment statement that gives a value to n. The second line is a print statement that displays the value of n.

Expressions:

- An expression is a combination of values, variables, and operators.**
- A value all by itself is considered an expression, and also a variable.
- So the following are all legal expressions:

```
>>> 42
42
>>> a=2
>>>a+3+2 7
>>> z=("hi"+"friend")
>>>print(z) hifriend
```

INPUT AND OUTPUT

INPUT: Input is data entered by user (end user) in the program. In python, **input()** function is available for input.

Syntax for input() is:
variable = input ("data")

Example:

```
>>> x=input("enter the name:")
```

```
enter the name: george
```

```
>>>y=int(input("enter the number"))
```

```
enter the number 3
```

#python accepts string as default data type. Conversion is required for type.

OUTPUT: Output can be displayed to the user using Print statement .

Syntax:

```
print (expression/constant/variable)
```

Example:

```
>>> print ("Hello")
```

```
Hello
```

COMMENTS:

- ☐ A **hash sign (#)** is the beginning of a comment.
- ☐ Anything written after # in a line is ignored by interpreter.
Eg: percentage = (minute * 100)/60 **# calculating percentage of an hour**
- ☐ Python **does not have multiple-line commenting feature.** You have to comment each line individually as follows:

Example:

```
# This is a comment.  
# This is a comment, too.  
# I said that already.
```

DOCSTRING:

- ☐ Docstring is short for documentation string.
- ☐ It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring.
- ☐ **Triple quotes** are used while writing docstrings.

Syntax:

functionname__doc__Example:

```
def double(num):  
    """Function to double thevalue"""  
    return 2*num  
>>>print (double.__doc_)  
Function to double the value
```

LINES AND INDENTATION:

- ☐ Most of the programming languages like C, C++, Java use braces { } to define a block of code. But, python uses indentation.
- ☐ Blocks of code are denoted by line indentation.
- ☐ It is a space given to the block of codes for class and function definitions or flow control.

Example:

```
a=3
b=1
if a>b:
    print("a is greater")
else:
    print("b is greater")
```

QUOTATION IN PYTHON:

Python accepts single ('), double (") and triple (" or """) quotes to denote string literals. Anything that is represented using quotations are considered as string.

- ☐ Single quotes(' ') Eg, 'This a string in single quotes'
- ☐ double quotes(" ") Eg, "This a string in double quotes"
- ☐ triple quotes(""" """) Eg, This is a paragraph. It is made up of multiple lines and sentences."""

TUPLE ASSIGNMENT

- ☐ An assignment to all of the elements in a tuple using a single assignment statement.
- ☐ Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.
- ☐ The left side is a tuple of variables; the right side is a tuple of values.
- ☐ Each value is assigned to its respective variable.
- ☐ All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.
- ☐ Naturally, the number of variables on the left and the number of values on the right have to be the same.

```
>>>(a, b, c, d) = (1, 2, 3)
ValueError: need more than 3 values to unpack
```

Example:

-It is useful to swap the values of two variables. With **conventional assignment statements**, we have to use a temporary variable. For example, to swap **a** and **b**:

Swap two numbers	Output:
a=2;b=3 print(a,b) temp = a a = b b = temp print(a,b)	(2, 3) (3, 2) >>>

-Tuple assignment solves this problem neatly:

```
(a, b) = (b, a)
```

-One way to think of tuple assignment is as **tuple packing/unpacking**.

In tuple packing, the values on the left are ‘packed’ together in a tuple:

```
>>>b = ("George",25,"20000")    # tuplepacking
```

-In tuple unpacking, **the values in a tuple on the right are ‘unpacked ‘into the variables/names on the right:**

```
>>>b = ("George", 25, "20000")    # tuple packing
>>>(name, age, salary)=b          # tupleunpacking
>>>name
'George'
>>>age
25
>>>salary
'20000'
```

-The right side can be any kind of sequence (string, list,tuple)

Example:

-To split an email address in to user name and a domain

```
>>>mailid='god@abc.org'
>>>name, domain=mailid.split('@')
>>>print name god
>>> print (domain) abc.org
```

4.OPERATORS:

- ☐ Operators are the constructs which can manipulate the value of operands.
- ☐ Consider the **expression** $4 + 5 = 9$. Here, **4 and 5 are called operands** and **+ is called operator**
- ☐ Types of Operators:

-Python language supports the following types of operators

- Arithmetic Operators
- Comparison (Relational)Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic operators:

They are used to perform **mathematical operations** like addition, subtraction, multiplication etc.

Assume, a=10 and b=5

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a ** b = 10$ to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed	$5 // 2 = 2$

Examples

```
a=10
b=5
print("a+b=",a+b)
print("a-b=",a-b)
print("a*b=",a*b)
print("a/b=",a/b)
print("a%b=",a%b)
print("a//b=",a//b)
print("a**b=",a**b)
```

Output:

```
a+b=15
a-b= 5
a*b= 50
a/b= 2.0
a%b=0
a//b=2
a**b= 100000
```

Comparison (Relational) Operators:

- Comparison operators are used to compare values.
- It either returns True or False according to the condition. Assume, a=10 and b=5

Operator	Description	Example
==	If the values of two operands are equal, then the condition	(a == b) is

	becomes true.	not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!=b) is true
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Example

```
a=10
b=5
print("a>b=>",a>b)
print("a>b=>",a<b)
print("a==b=>",a==b)
print("a!=b=>",a!=b)
print("a>=b=>",a<=b)
print("a>=b=>",a>=b)
```

Output: a>b=>

```
True a>b=>
False a==b=>
False a!=b=>
True a>=b=>
False a>=b=>
True
```

Assignment Operators:

-Assignment operators are used in Python to assign values to variables.

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to leftoperand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c -a

<code>*=</code> AND	Multiply	It multiplies right operand with the left operand and assign the result to left operand	<code>c *= a</code> is equivalent to <code>c = c *a</code>
<code>/=</code> AND	Divide	It divides left operand with the right operand and assign the result to left operand	<code>c /= a</code> is equivalent to <code>c = c /ac</code> <code>/= a</code> is equivalent to <code>c = c /a</code>
<code>%=</code> AND	Modulus	It takes modulus using two operands and assign the result to left operand	<code>c %= a</code> is equivalent to <code>c = c % a</code>
<code>**=</code> Exponent AND		Performs exponential (power) calculation on operators and assign value to the left operand	<code>c **= a</code> is equivalent to <code>c = c ** a</code>
<code>//=</code> Division	Floor	It performs floor division on operators and assign value to the left operand	<code>c //= a</code> is equivalent to <code>c = c // a</code>

Example

```

a =21
b =10
c = 0
c = a + b
print("Line 1 - Value of c is ",c)
c += a
print("Line 2 - Value of c is ", c)
c *= a
print("Line 3 - Value of c is ",c)
c /= a
print("Line 4 - Value of c is ", c)
c = 2
c %=a
print("Line 5 - Value of c is ",c)
c **= a
print("Line 6 - Value of c is ",c)
c //= a
print ("Line 7 - Value of c is ", c)

```

Output

```

Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52.0
Line 5 - Value of c is2
Line 6 - Value of c is 2097152
Line 7 - Value of c is99864

```

Logical Operators:

-Logical operators are the and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example

```
a = True
b = False
print('a and b is', a and b)
print('a or b is', a or b)
print('not a is', not a)
```

Output

```
x and y is False
x or y is True
not x is False
```

Bitwise Operators:

- A **bitwise operation** operates on one or more **bit** patterns at the level of individual bits

Example: Let x = 10 (0000 1010 in binary) and

y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
	Bitwise OR	x y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

Example

```
a = 60      # 60 = 0011 1100
b = 13      # 13 = 0000 1101
c = 0
c = a & b;   # 12 = 0000 1100
print "Line 1 - Value of c is ", c
c = a | b;   # 61 = 00111101
print "Line 2 - Value of c is ", c
c = a ^ b;   # 49 = 00110001
print "Line 3 - Value of c is ", c
c = ~a;      # -61 = 11000011
```

Output

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

```

print "Line 4 - Value of c is ", c
c = a<<2;      # 240 = 11110000
print "Line 5 - Value of c is ", c
c = a>>2;      # 15 = 00001111
print "Line 6 - Value of c is ", c

```

Membership Operators:

- ❖ Evaluates to find a value or a variable is in the specified sequence of string, list, tuple, dictionary or not.
- ❖ Let, **x=[5,3,6,4,1]**. To check particular item in list or not, **in** and **not in** operators are used.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example:

```

x=[5,3,6,4,1]
>>>5 in x
True
>>>5 not in x
False

```

Identity Operators:

- They are used to check if two values (or variables) are located on the same part of the memory.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example

```

x =5
y =5
x2 = 'Hello'
y2= 'Hello'
print(x1 is not y1)
print(x2 is y2)

```

Output
False
True

5. OPERATOR PRECEDENCE:

When an expression contains **more than one operator**, the **order of evaluation** depends on the order of operations.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>><<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR' and regular 'OR'
<= <> >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

-For mathematical operators, Python follows mathematical convention.

-The acronym **PEMDAS** (Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction) is a useful way to remember the rules:

- Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, $2 * (3-1)$ is 4, and $(1+1)**(5-2)$ is 8.
- You can also use parentheses to make an expression easier to read, as in $(minute * 100) / 60$, even if it doesn't change the result.
- Exponentiation has the next highest precedence, so $1 + 2**3$ is 9, not 27, and $2**3**2$ is 18, not 36.
- Multiplication and Division have higher precedence than Addition and Subtraction. So $2*3-1$ is 5, not 4, and $6+4/2$ is 8, not 5.
- Operators with the same precedence are evaluated from left to right (except exponentiation).

Examples:

a=9-12/3+3*2-1 a=? a=9-4+3*2-1 a=9-4+6-1 a=5+6-1 a=11-1 a=10	A=2*3+4%5-3/2+6 A=6+4%5-3/2+6 A=6+4-3/2+6 A=6+4-1+6 A=10-1+6 A=9+6 A=15	find m=? m=-43 8&&0 -2 m=-43 0 -2 m=1 -2 m=1
--	--	--

6.Functions, Function Definition And Use, Function call, Flow Of Execution, Function Prototypes, Parameters And Arguments, Return statement, Arguments types, Modules

FUNCTIONS:

- **Function is a sub program which consists of set of instructions used to perform a specific task. A large program is divided into basic building blocks called function.**

Need For Function:

- When the program is too complex and large they are divided into parts. Each part is separately coded and combined into single program. Each subprogram is called as function.
- Debugging, Testing and maintenance becomes easy when the program is divided into subprograms.
- Functions are used to avoid rewriting same code again and again in a program.
- Function provides code re-usability
- The length of the program is reduced.

Types of function:

Functions can be classified into two categories:

- i) user defined function
- ii) Built in function

i) Built in functions

- Built in functions are the functions that are already created and stored in python.
- These built in functions are always available for usage and accessed by a programmer. It cannot be modified.

Built in function	Description
>>>max(3,4) 4	# returns largest element
>>>min(3,4) 3	# returns smallest element
>>>len("hello") 5	#returns length of an object
>>>range(2,8,1) [2, 3, 4, 5, 6, 7]	#returns range of given values
>>>round(7.8) 8.0	#returns rounded integer of the given number
>>>chr(5) '\x05'	#returns a character (a string) from an integer
>>>float(5) 5.0	#returns float number from string or integer
>>>int(5.0) 5	# returns integer from string or float
>>>pow(3,5) 243	#returns power of given number
>>>type(5.6) <type 'float'>	#returns data type of object to which it belongs
>>>t=tuple([4,6.0,7]) (4, 6.0, 7)	# to create tuple of items from list
>>>print("good morning") Good morning	# displays the given object
>>>input("enter name:") enter name : George	# reads and returns the given string

ii) User Defined Functions:

- User defined functions are the functions that programmers create for their requirement and use.
- These functions can then be **combined to form module** which can be used in other programs by importing them.
- Advantages of user defined functions:
 - Programmers working on large project can divide the workload by making different functions.
 - If repeated code occurs in a program, function can be used to include those codes and execute when needed by calling that function.

Function definition: (Sub program)

- def keyword is used to define a function.
- Give the function name after def keyword followed by parentheses in which arguments are given.
- End with colon (:)
- Inside the function add the program statements to be executed
- End with or without return statement

Syntax:

```
def fun_name(Parameter1,Parameter2...Parameter n): statement1
    statement2...
    statement n return[expression]
```

Example:

```
def my_add(a,b):
    c=a+b
    return c
```

Function Calling: (Main Function)

- Once we have defined a function, we can call it from another function, program or even the Pythonprompt.
- To call a function we **simply type the function name with appropriate arguments.**

Example:

```
x=5
y=4
my_add(x,y)
```

Flow of Execution:

- The order in which statements are executed is called the **flow of execution**
- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the **def** statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called.

Function Prototypes:

- i. Function without arguments and without return type
- ii. Function with arguments and without return type
- iii. Function without arguments and with return type
- iv. Function with arguments and with return type

i) Function without arguments and without return type

- In this type no argument is passed through the function call and no output is return to main function
- The sub function will read the input values perform the operation and print the result in the same block

ii) Function with arguments and without return type

- Arguments are passed through the function call but output is not return to the main function

iii) Function without arguments and with return type

- In this type no argument is passed through the function call but output is return to the main function.

iv) Function with arguments and with return type

- In this type arguments are passed through the function call and output is return to the main function

Without Return Type	
Without argument	With argument
<pre>def add(): a=int(input("enter a")) b=int(input("enter b")) c=a+b print(c) add()</pre>	<pre>def add(a,b): c=a+b print(c) a=int(input("enter a")) b=int(input("enter b")) add(a,b)</pre>
OUTPUT: enter a5 enter b 10 15	OUTPUT: enter a5 enter b 10 15

With return type	
Without argument	With argument
<pre>def add(): a=int(input("enter a")) b=int(input("enterb")) c=a+b return c c=add() print(c)</pre>	<pre>def add(a,b): c=a+b return c a=int(input("enter a")) b=int(input("enter b")) c=add(a,b) print(c)</pre>
OUTPUT: enter a5 enter b 10 15	OUTPUT: enter a5 enter b 10 15

Parameters And Arguments:

Parameters:

- Parameters are the value(s) provided in the parenthesis when we write function header.
- These are the values required by function to work.
- If there is more than one value required, all of them will be listed in parameter list separated by **comma**.
- Example: defmy_add(a,b):

Arguments :

- Arguments are the value(s) provided in function call/invoke statement.
- List of arguments should be supplied in same way as parameters are listed.
- Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.
- Example:my_add(x,y)

RETURN STATEMENT:

- The **return statement is used to exit a function** and go back to the place from where it was called.
- If the return statement has no arguments, then it will not return any values. But exits from function.

Syntax:

return[expression]

Example:

```
def my_add(a,b):  
    c=a+b  
    return c  
x=5  
y=4  
print(my_add(x,y))
```

Output:

9

ARGUMENT TYPES:

1. Required Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable length Arguments

Required Arguments : The number of arguments in the function call should match exactly with the function definition.

```
defmy_details( name, age ):  
    print("Name: ", name)  
    print("Age ", age)  
    return  
my_details("george",56)
```

Output:

```
Name:
georgeAge56
```

Keyword Arguments:

Python interpreter is able to use the keywords provided to match the values with parameters even though if they are arranged in out of order.

```
def my_details( name, age ):
    print("Name: ", name)
    print("Age ", age)
    return
my_details(age=56,name="george")
```

Output:

```
Name:
georgeAge56
```

Default Arguments:

Assumes a default value if a value is not provided in the function call for that argument.

```
def my_details( name, age=40 ):
    print("Name: ", name)
    print("Age ", age)
    return
my_details(name="george")
```

Output:

```
Name:
georgeAge40
```

Variable length Arguments

If we want to specify more arguments than specified while defining the function, variable length arguments are used. It is denoted by * symbol before parameter.

```
def my_details(*name ):
    print(*name)
my_details("rajan","rahul","micheal", "ärjun")
```

Output:

```
rajanrahulmichealärjun
```

7. MODULES:

- A module is a file containing Python definitions ,functions, statements and instructions.
- Standard library of Python is extended as modules.
- To use these modules in a program, programmer needs to import the module.

- Once we import a module, we can reference or use to any of its functions or variables in our code.
 - There is large number of standard modules also available in python.
 - Standard modules can be imported the same way as we import our user- defined modules.
 - Every module contains many functions.
 - To access one of the function , you have to specify the name of the module and the name of the function separated by dot .This format is called dot notation.

Syntax:

import module_name.module_name.function_name(variable)	
Importing Builtin Module:	Importing User Defined Module:
import math x=math.sqrt(25) print(x)	import calx=cal.add(5,4) print(x)

There are **four ways to import a module** in our program, they are










<u>Import:</u> It is simplest and most common way to use modules in our code. Example: <pre>import math x=math.pi print("The value of pi is", x)</pre> Output: The value of pi is 3.141592653589793	<u>from import :</u> It is used to get a specific function in the code instead of complete file. Example: <pre>from math import pi x=pi print("The value of pi is", x)</pre> Output: The value of pi is 3.141592653589793
<u>import with renaming:</u> We can import a module by renaming the module as our wish. Example: <pre>import math as m x=m.pi print("The value of pi is", x)</pre> Output: The value of pi is 3.141592653589793	<u>import all:</u> We can import all names(definitions) form a module using * Example: <pre>from math import * x=pi print("The value of pi is", x)</pre> Output: The value of pi is 3.141592653589793

Built-in python modules are,

1.math– mathematical functions:

some of the functions in math module is,

- 🚦 math.ceil(x) - Return the ceiling of x, the smallest integer greater

- than or equal to x
-  `math.floor(x)` - Return the floor of x , the largest integer less than or equal to x .
-  `math.factorial(x)`-Return x factorial.
-  `math.gcd(x,y)`-Return the greatest common divisor of the integers a and b
-  `math.sqrt(x)`- Return the square root of x
-  `math.pi` - The mathematical constant $\pi = 3.141592$
-  `math.e` – returns The mathematical constant $e = 2.718281$
-  `random`-Generate pseudo-random numbers
-  `random.randrange(stop)` `random.randrange(start, stop[, step])` `random.uniform(a, b)`
-  -Return a random floating point number

8. ILLUSTRATIVE PROGRAMS

<u>Program for SWAPPING(Exchanging)of values</u>	<u>Output</u>
<pre>a = int(input("Enter a value ")) b = int(input("Enter b value")) c = a a = b b =c print("a=",a,"b=",b,)</pre>	<pre>Enter a value 5 Enter b value 8 a=8 b=5</pre>
<u>Program to find distance between twopoints</u>	<u>Output</u>
<pre>import math x1=int(input("enter x1")) y1=int(input("enter y1")) x2=int(input("enter x2")) y2=int(input("enter y2")) distance =math.sqrt((x2-x1)**2)+((y2- y1)**2) print(distance)</pre>	<pre>enter x17 enter y16 enter x25 enter y27 2.5</pre>
<u>Program to circulate n numbers</u>	<u>Output:</u>
<pre>a=list(input("enter the list"))</pre>	<pre>enter the list '1234'</pre>
<pre>print(a) for i in range(1,len(a),1): print(a[i:]+a[:i])</pre>	<pre>['1', '2', '3', '4'] ['2', '3', '4', '1'] ['3', '4', '1', '2'] ['4', '1', '2', '3']</pre>

What is Raspberry Pi | IoT Raspberry Pi Tutorial for Beginners

In IoT Tutorials, we saw different types of applications like Health, Education, Government etc. But today, we will talk about a new device called Raspberry Pi that can be incorporated into IoT systems to make work easy.

So, in this Raspberry Pi tutorial, we are going to learn about IoT Raspberry Pi introduction with its innovation. Moreover, we will discuss the difference between Raspberry Pi models in IoT. At last, we will see how to buy IoT Raspberry Pi.

So, let's start with Introduction to IoT Raspberry Pi. What is a Raspberry Pi?

The Raspberry Pi is a very small computer that is almost the size of your credit card. It costs between Rs 750 and Rs 4000. It can function as a proper desktop computer or use to build smart devices and is available anywhere in the world.

The Pi changed into what initially was meant to be a microcomputer to teach kids coding. Its scope can expand after hobbyists and engineers noticed its capacity, and it's far now one of the most famous objects inside the international era.



Who Invented IoT Raspberry Pi?

The Raspberry Pi Foundation was formed in the year 2008 after a group of technicians and academics who were concerned about

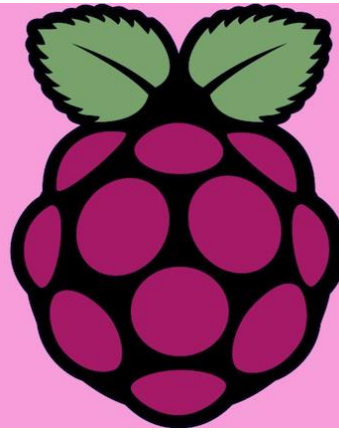
students' interest gradually drifting and declining in computer sciences. So they came up with a low-cost computer as a solution to inspire children and make it more accessible.

The basic motive was that these tiny computer systems might allow for very basic and simple programming. Its low electricity utilization and value expect to make Pis more easily to be used in school rooms.

The invention of Raspberry Pi

Why is it called Raspberry Pi?

The “Raspberry” name is a homage to computer companies in early times that were being named after a fruit, like Apple, Apricot Computers, Tangerine Computer Systems. The idea to make a small computer to run only the Python programming language is where the “Pi” derives from.



When was IoT Raspberry Pi launched?

The first Raspberry Pi unit which was available commercially was launched on February 19, 2012. This version featured 256MB of RAM, could run on Linux-based desktop operating systems, had one USB port, and no Ethernet port. This was named the Model A.

What's the Difference Between Raspberry Pi Models?

IoT Raspberry Pi models can be confusing because there are so many of them and there are two levels to the naming system.

The “generation” of the model, represent by Pi 1, Pi 2, and Pi 3 where Pi 1 is for models between 2012-14, Pi 2 is 2015 models, and Pi 3 is 2016 models. So 3 is the most recent which is better than 2, which is better than 1.

The power and features indicate by model A, A+, B, and B+. It’s not like grades though, A is lower than B.

Where is IoT Raspberry Pi’s used?

IoT Raspberry Pi can be used in a wide variety of tasks. It’s ideal and best suitable for projects where there is a computer requirement but you don’t require much processing power, you want to keep the costs low and want to save on space. Here’s a brief list of some ideal uses of the Pi.

- Teach kids (or yourself) the way to code.



How IoT Raspberry Pi Used

- Use it as a desktop pc.
- Construct a movement seize safety digital camera or a DIY pan and tilt digital camera with Raspberry Pi.



How IoT Raspberry Pi Uses

- Make your very own retro gaming console.



How IoT Raspberry Pi Uses

- You can make an FM radio or a global clock.
- Prepare time-lapse pictures digital camera with the digitalcamera module.

How to Build a Raspberry Pi Temperature Monitor

Temperature and humidity are vital data points in today's industrial world. Monitoring environmental data for server rooms, commercial freezers, and production lines is necessary to keep things running smoothly. There are lots of solutions out there ranging from basic to complex and it can seem overwhelming on what your business needs and where to start.

We'll walk through how to build and use a Raspberry Pi temperature sensor with different temperature sensors. This is a good place to start since these solutions are inexpensive, easy to do, and gives you a foundation to build off of for other environmental monitoring.

Raspberry Pi

A Raspberry Pi is an inexpensive single board computer that will allow you to connect to a temperature sensor and stream the data to a data visualization software. Raspberry Pi's started out as a learning tool and have evolved to an industrial workplace tool. The ease of use and ability to code with Python, the fastest growing programming language, has made them a go to solution.

You'll want a Raspberry Pi that has WiFi built in, which are any model 3, 4, and zero W/WH. Between those you can choose based on pricing and features. The Zero W/WH is the cheapest but if you need more functionality you can choose between the 3 and 4. You can only buy one Zero W/WH at a time due to limitations by the Raspberry Pi Foundation. Whatever Pi you choose, make sure to purchase a charger since that is how you'll power the Pi and an SD card with Raspbian to make installation of the operating system as easy as possible.

There are other single board computer that can work as well, but that's for another time and another article.

Sensors

There are four sensors we recommend using because they are inexpensive, easy to connect, and give accurate readings; DSB18B20, DHT22, BME280, and Raspberry PiSense HAT.

DHT22 — This temperature and humidity sensor has temperature accuracy of +/- 0.5 C and a humidity range from 0 to 100 percent. It is simple to wire up to the Raspberry Pi and doesn't require any pull up resistors.

DSB18B20 — This temperature sensor has a digital output, which works well with the Raspberry Pi. It has three wires and requires a breadboard and resistor for the connection.

BME280 — This sensor measures temperature, humidity, and barometric pressure. It can be used in both SPI and I2C.

Sense HAT — This is an add on board for Raspberry Pi that has LEDs, sensors, and a tiny joystick. It connects

directly on to the GPIO on the Raspberry Pi but using a ribbon cable

gives you more accurate temperature readings.

Raspberry Pi Setup

If this is the first time setting up your Raspberry Pi you'll need to install the Raspbian Operating System and connect your Pi to WiFi. This will require a monitor and keyboard to connect to the Pi. Once you have it up and running and connected to the WiFi, your Pi is ready to go.

Initial State Account

You'll need somewhere to send your data to keep a historical log and view the real-time data stream so we will use Initial State. Go to <https://iot.app.initialstate.com> and create a new account or log into your existing account.

Next, we need to install the Initial State Python module onto your Pi. At a command prompt (don't forget to SSH into your Pi first), run the following command:

```
$ cd /home/pi/  
$ \curl -sSL https://get.initialstate.com/python -o - | sudo bash
```

After you enter the curl command in the command prompt you will see something similar to the following output to the screen:

```
pi@raspberrypi ~ $ \curl -sSL https://get.initialstate.com/python -o  
- | sudo bash  
Password:  
Beginning ISStreamer Python Easy Installation!  
This may take a couple minutes to install, grab some coffee :) But don't forget to come  
back, I'll have questions later!
```

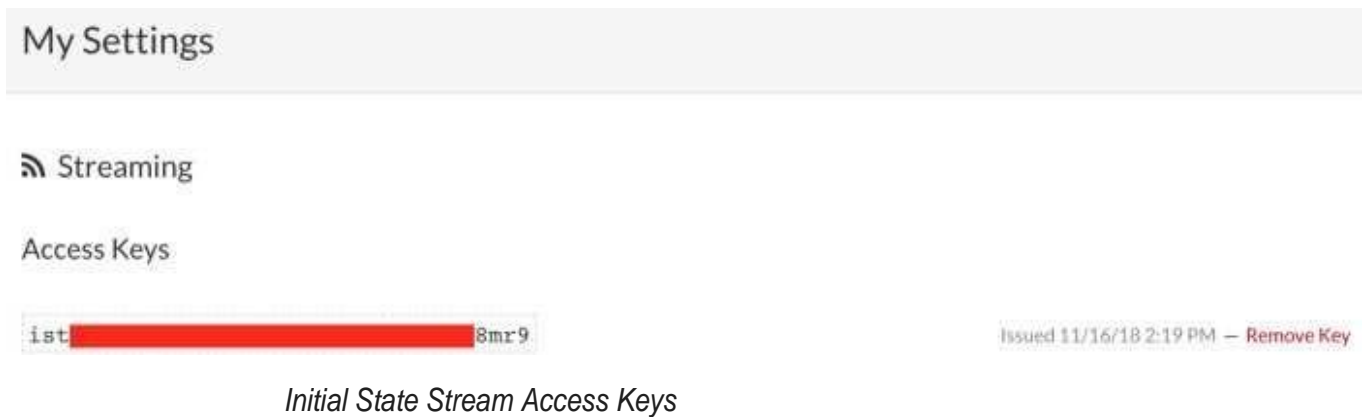
```
Found easy_install: setuptools 1.1.6  
Found pip: pip 1.5.6 from /Library/Python/2.7/site-packages/pip-1.5.6-py2.7.egg  
(python 2.7)  
pip major version: 1 pip  
minor version: 5  
ISStreamer found, updating...  
Requirement already up-to-date: ISStreamer in  
/Library/Python/2.7/site-packages  
Cleaning up...  
Do you want to automatically get an example script? [y/N]  
Where do you want to save the example? [default: ./is_example.py]  
Please select which Initial State app you're using:  
1. app.initialstate.com  
2. [NEW!] iot.app.initialstate.com Enter  
choice 1 or 2:  
Enter iot.app.initialstate.com user name:  
Enter iot.app.initialstate.com password:
```

When prompted to automatically get an example script, type y. This will create a test script that we can run to ensure that we can stream data to Initial State. The next prompt will ask where you want to save the example file. You can either type a custom local path or hit enter to accept the default location. Finally, you'll be asked which Initial State app you are using. If you've recently created an account, select option 2, enter your user name and password. After that the installation will be complete.

Let's take a look at the example script that was created.

```
$ nano is_example.py
```

On line 15, you will see a line that starts with `streamer = Streamer(bucket_.....`. This line creates a new data bucket named "Python Stream Example" and is associated with your account. This association happens because of the `access_key="..."` parameter on that same line. That long series of letters and numbers is your Initial State account access key. If you go to your Initial State account in your web browser, click on your username in the top right, then go to "my settings", you will find that same access key here under "Streaming Access Keys".



Every time you create a data stream, that access key will direct that data stream to your account (so don't share your key with anyone).

Run the test script to make sure we can create a data stream to your Initial State account. Run the following:

```
$ python is_example.py
```

Go back to your Initial State account in your web browser. A new data bucket called "Python Stream

Example” should have shown up on the left in your log shelf (you may have to refresh the page). Click on this bucket and then click on the Waves icon to view the test data.

Initial State Python Stream Example dashboard

If you are using Python 3 you can install the Initial State Streamer Module you can install using the following command:

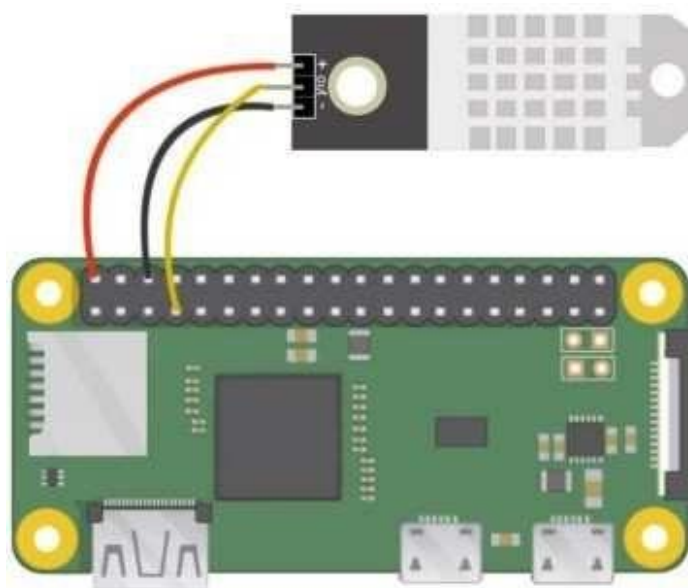
```
pip3 install ISStreamer
```

Now we are ready to setup the temperature sensor with the Pi to stream temperature to a dashboard.

DHT22 Solution

You'll need the following items to build this solution:

-DHT22 Temperature and Humidity Sensor



The DHT22 will have three pins — 5V, Gnd, and data. There should be a pin label for power on the DHT22 (e.g. '+' or '5V'). Connect this to pin 2 (the top right pin, 5V) of the Pi. The Gnd pin will be labeled '-' or 'Gnd' or something equivalent. Connect this to pin 6 Gnd (two pins below the 5V pin) on the Pi. The remaining pin on the DHT22 is the data pin and will be labeled 'out' or 's' or 'data'. Connect this to one of the GPIO pins on the Pi such as GPIO4 (pin 7). Once this is wired, power on your Pi.

For this solution we will need to use Python 3 and the CircuitPython library as Adafruit has deprecated the DHT Python library.

Install the CircuitPython-DHT Python module at a command prompt to make reading DHT22 sensor

data super easy:

```
$ pip3 install adafruit-circuitpython-dht
$ sudo apt-get install libgpiod2
```

With our operating system installed along with our two Python modules for reading sensor data and sending data to Initial State, we are ready to write our Python script.

The following script will create/append to an Initial State data bucket, read the DHT22 sensor data, and send that data to a real-time dashboard. All you need to do is modify lines 6-11.

```
1  import adafruit_dht
2  from ISStreamer.Streamer import Streamer
3  import time
4  import
board5
6  # ----- User Settings -----
7  SENSOR_LOCATION_NAME = "Office"
8  BUCKET_NAME = ":partly_sunny: Room Temperatures"
9  BUCKET_KEY = "dht22sensor"
10 ACCESS_KEY = "ENTER ACCESS KEY HERE"
11 MINUTES_BETWEEN_READS = 10
12 METRIC_UNITS = False
13 # .....
14
15 dhtSensor = adafruit_dht.DHT22(board.D4)
16 streamer = Streamer(bucket_name=BUCKET_NAME, bucket_key=BUCKET_KEY,
access_key=ACCESS_KEY)17
18 while True:
19     try:
20         humidity = dhtSensor.humidity
21         temp_c = dhtSensor.temperature
22     except RuntimeError:
23         print("RuntimeError, trying again. ")
24         continue
25
26     if METRIC_UNITS:
```

```
27         streamer.log(SENSOR_LOCATION_NAME + " Temperature(C)", temp_c)
28     else:
29         temp_f = format(temp_c * 9.0 / 5.0 + 32.0, ".2f")
30         streamer.log(SENSOR_LOCATION_NAME + " Temperature(F)", temp_f)
31     humidity = format(humidity, ".2f")
32     streamer.log(SENSOR_LOCATION_NAME + " Humidity(%)", humidity)
33     streamer.flush()
34     time.sleep(60*MINUTES_BETWEEN_READS)
```

tempsensor.py hosted with ❤ by GitHub

[view raw](#)

<https://gist.github.com/25be959d124f4b4c86f7160cf916f4d4.git>

- ◆ Line 7— This value should be unique for each node/temperature sensor. This could be your sensor node's room name, physical location, unique identifier, or whatever. Just make sure it is unique for each node to ensure that the data from this node goes to its own data stream in your dashboard.
- ◆ Line 8— This is the name of the data bucket. This can be changed at any time in the Initial State UI.
- ◆ Line 9— This is your bucket key. It needs to be the same bucket key for every node you want displayed in the same dashboard.
- ◆ Line 10— This is your Initial State account access key. Copy and paste this key from your Initial State account.
- ◆ Line 11— This is the time between sensor reads. Change accordingly.
- ◆ Line 12 — You can specify metric or imperial units on line 11.

After you have set lines 7–12 in your Python script on your Pi, save and exit the text editor. Run the script with the following command:

```
$python3 tempsensor.py
```


Introduction:

What is ThingSpeak?

ThingSpeak is an open source platform that provides various services exclusively targeted for development of IoT (Internet of Things) applications. It enables various services like real time data collection, analysis and visualisation of collected data via charts. It enables the creation of various plugins, apps collaborating with various web services, social networking and other APIs.

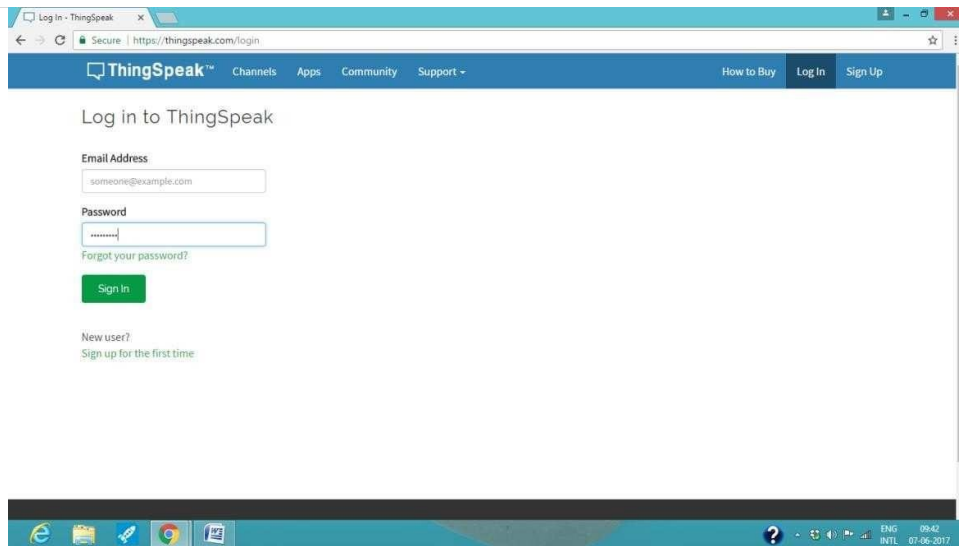
Thingspeak Channel:

'Thingspeak channel' is the core element of the thingspeak platform. This channel is used to store the real-time data or the data transferred through various sensors and embedded systems. Data stored at the channel is further used for analysis and visualisation.

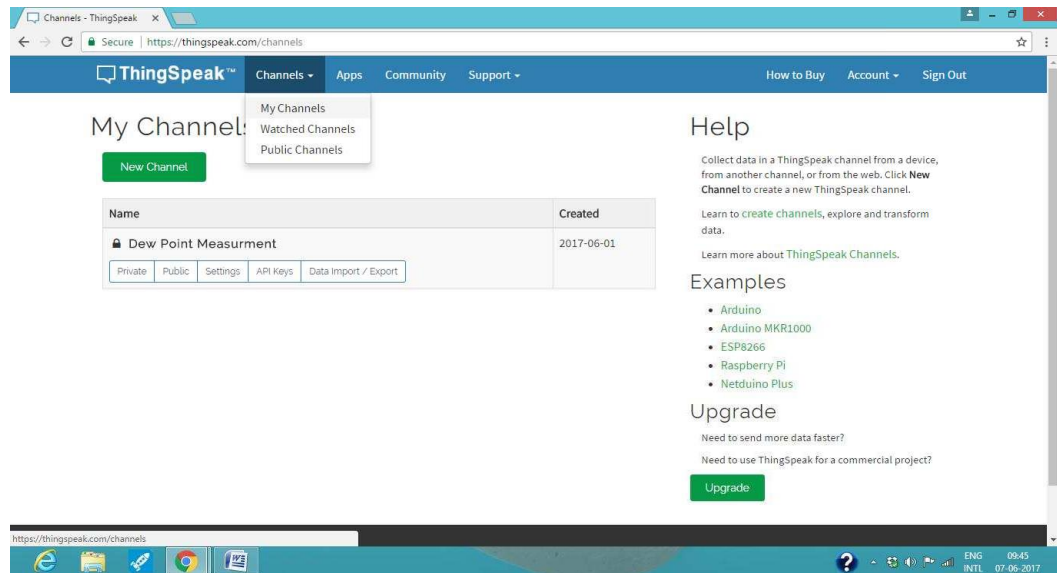
1. Creating a channel :

Before creating a channel you need to sign in to things speak. You can easily **sign in** either using your either **thingspeak account** or **mathswork account**, or **create a new mathswork account** via following link:

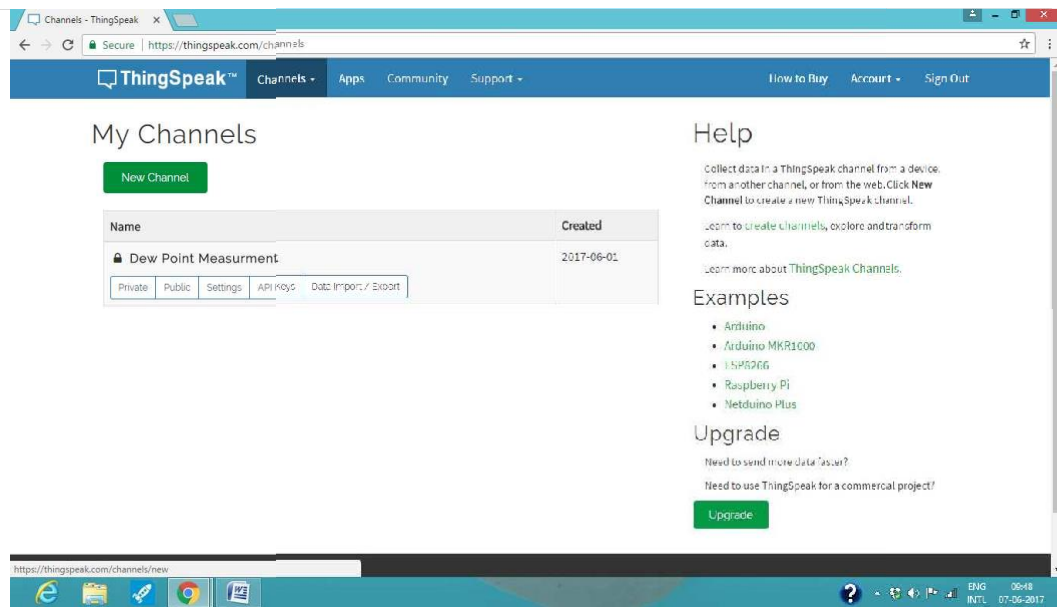
https://thingspeak.com/users/sign_up



1. Click on the menu bar **Channels > My Channels.**



2. Now on the channels page click on the button **'New Channel'**.



3. New channel page have various text box fields showing the settings of the channel.
 - a. **Name** – provide a unique name to your channel.
 - b. **Fields** – Click the check boxes next to the field and then enter the field name.

New Channel

Name:

Description:

Field 1: ☒

Field 2: ☒

Field 3: ☒

Field 4: ☐

Field 5: ☐

Field 6: ☐

Field 7: ☐

Field 8: ☐

Metadata:

Help

Channels store all the data that a ThingSpeak application collects. Each channel includes eight fields that can hold any type of data, plus three fields for location data and one for status data. Once you collect data in a channel, you can use ThingSpeak apps to analyze and visualize it.

Channel Settings

- Channel Name:** Enter a unique name for the ThingSpeak channel.
- Description:** Enter a description of the ThingSpeak channel.
- Fields:** Check the box to enable the field, and enter a field name. Each ThingSpeak channel can have up to 8 fields.
- Metadata:** Enter information about channel data, including JSON, XML, or CSV data.
- Tags:** Enter keywords that identify the channel. Separate tags with commas.
- Latitude:** Specify the position of the sensor or thing that collects data in decimal degrees. For example, the latitude of the city of London is 51.5072.
- Longitude:** Specify the position of the sensor or thing that collects data in decimal degrees. For example, the longitude of the city of London is -0.1275.
- Elevation:** Specify the position of the sensor or thing that collects data in meters. For example, the elevation of the city of London is 35.052.
- Make Public:** If you want to make the channel publicly available, check this box.
- URL:** If you have a website that contains information about your ThingSpeak channel, specify the URL.

- To make your channel public check the '**Make Public**' check box.
- Similarly, you can also add the location to your channel by clicking the '**Show Location**' check box.
- Check the '**Show video**' check box to make the video visible uploaded by you.
- Now click the '**Save channel**' button to save your channel.

Channel Settings

Make Public: ☐

URL:

Elevation:

Show Location: ☒

Latitude:

Longitude:

Show Video: ☒

* YouTube ☐ Vimeo ☐

Video ID:

Show Status: ☐

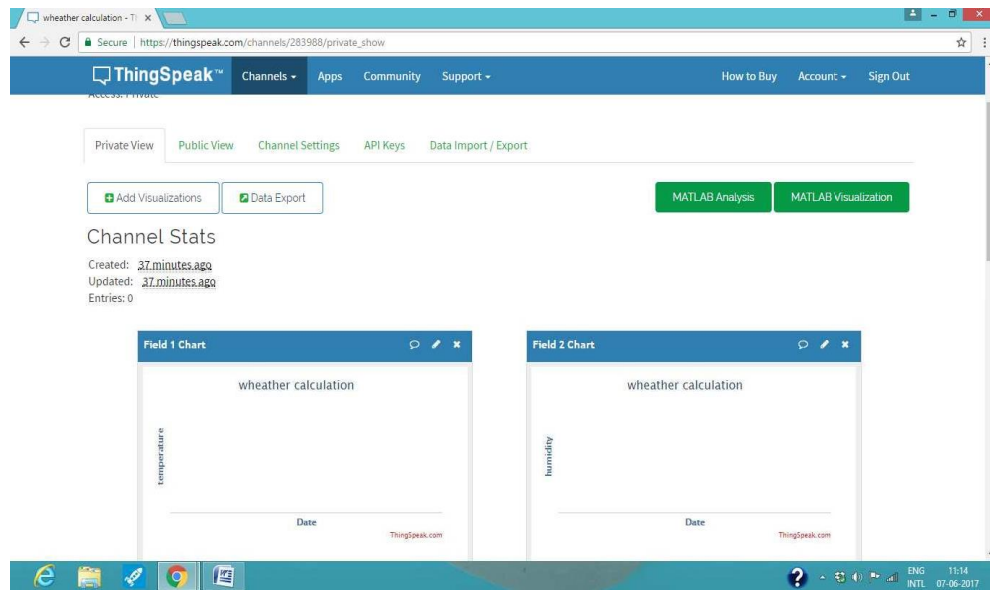
Save Channel

Can create visual data and traffic on ThingSpeak Apps.

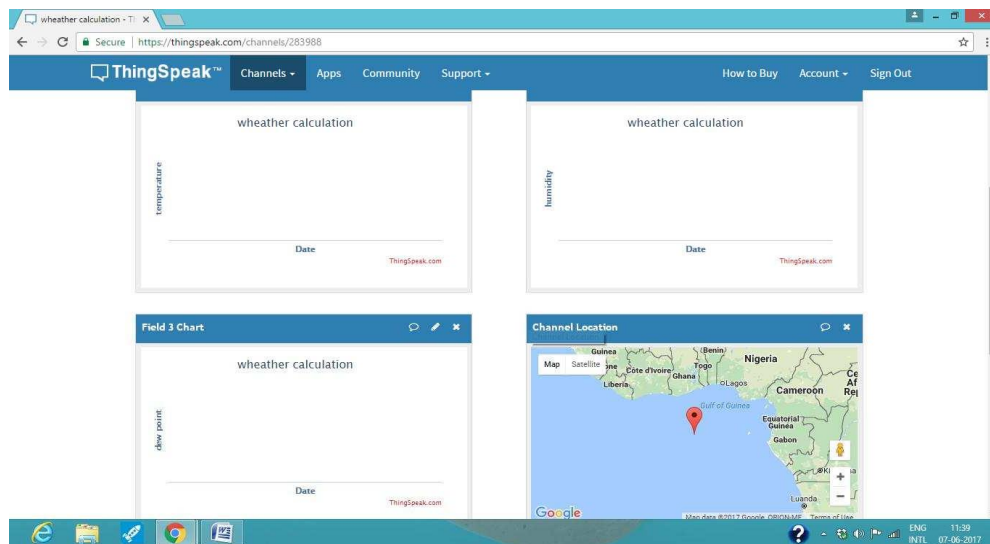
See Tutorial: [ThingSpeak and MATLAB](#) for an example of measuring dew point from a weather station that acquires data from an Arduino® device.

[Learn More](#)

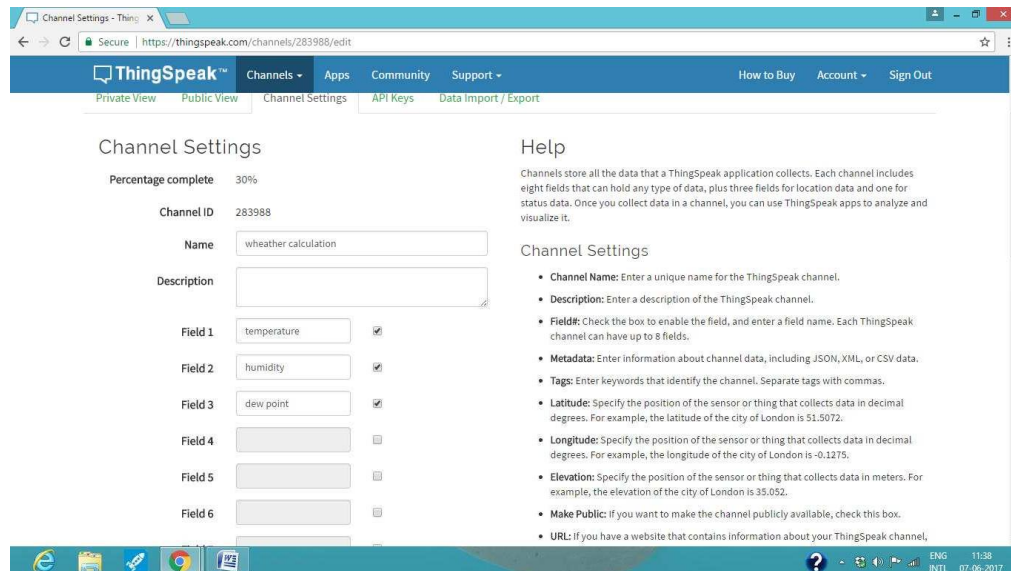
- Now , the channel page opens with the following tabs:
 - Private View**- It displays the information about your channel that is only visible to you.



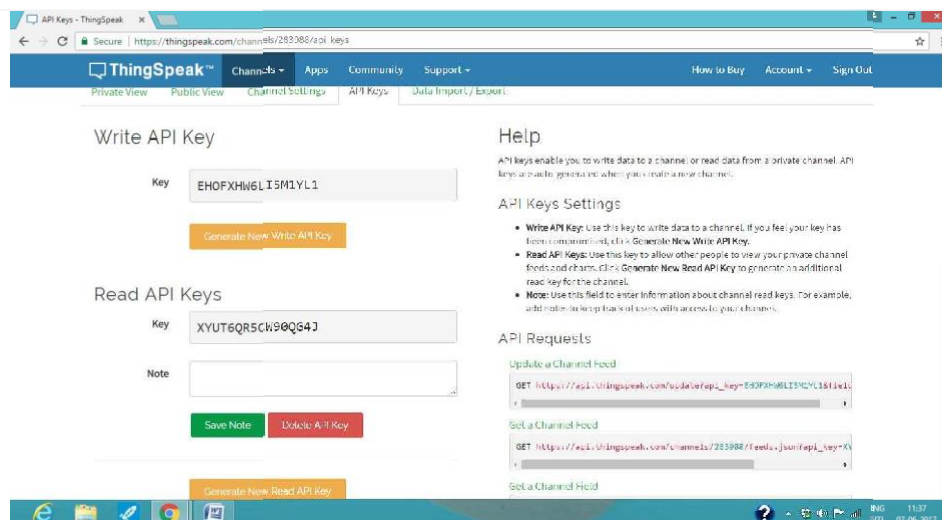
- b. **Public View-** If you have chosen to make your channel publicly visible then it will display the selected fields and information.



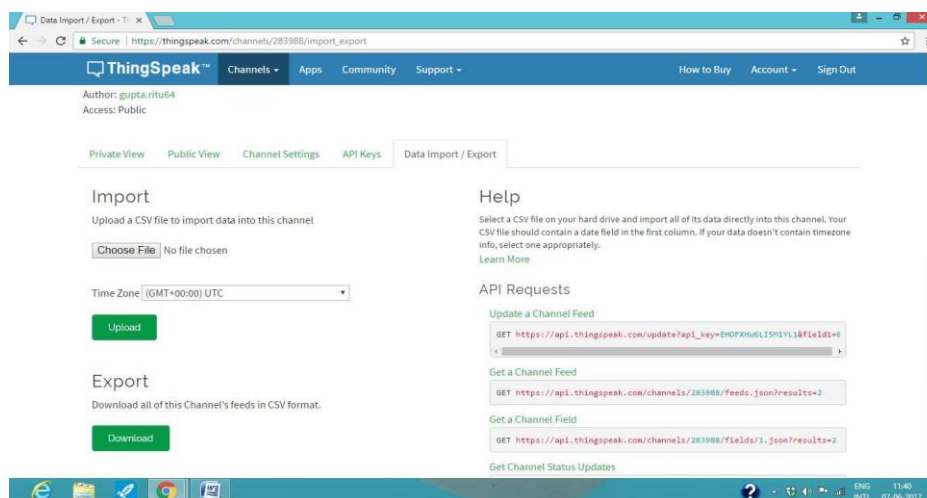
- c. **Channel Settings-** It will show all the options that are available during the channel creation.



- d. **API Keys-** In this tab you will have two API Keys – Read API key (to read from your channel), write API Key (to write to your channel).



- e. **Data Import/export-** It enables you to import and export the channel data.



5. In future your channel will be available to you just by clicking '**Channels >My Channels**'.

Implementing IoT Concepts with Python

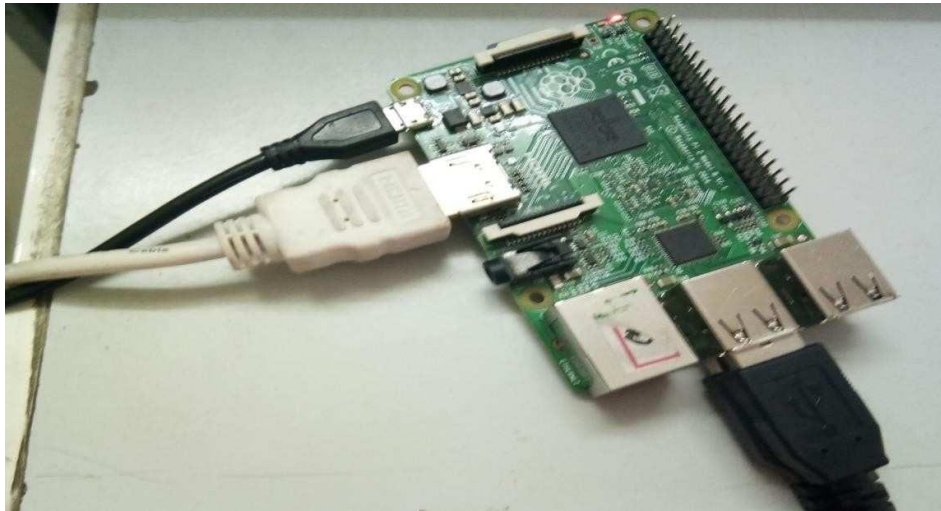
Objective: To upload DHT11 sensor data to ThingSpeak channel through Raspberry pi2.

1. Creating a Channel:

- a. Create a Channel over ThingSpeak.
- b. Edit the channel settings, add two fields first one for temperature and second for humidity .
- c. Save the channel.

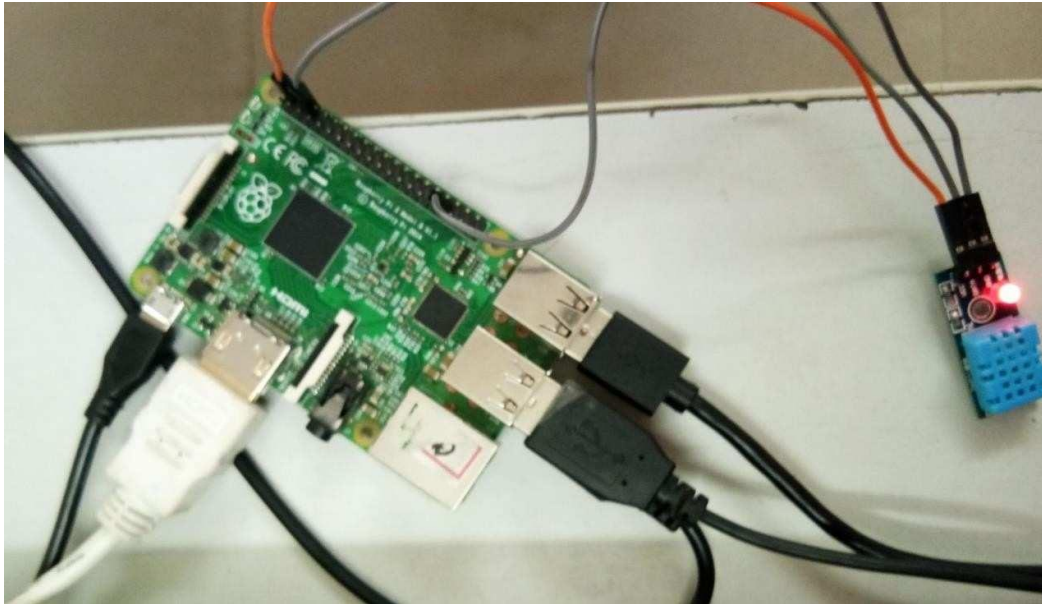
2. Establishing hardware infrastructure:

- a. Take a Raspberry Pi 2 board, connect it with the power of 12V approx..
- b. Make the connections with monitor, keyboard and mouse.



3. Connecting sensor with Raspberry Pi 2:

- a. The DHT11 sensor consists of three connecting pins :
 1. VCC: This pin used to connect with the voltage. Connect this pin with the pin number 2 at raspberry pi board via connecting wire.
 2. GND: This is called as 'ground'. Connect it at Pin number 6 of pi board.
 3. DATA: It is used as an output data port connect this at pin named GPIO 23 at pi board.
- b. The connections of sensor with the raspberry Pi is shown the following diagram:



4. Coding for DHT11 Sensor:

- a. The python code for reading the sensor data is as follows:

```
import sys
import RPi.GPIO as GPIO
from time import sleep
import Adafruit_DHT
import urllib2

def getSensorData():
    RH, T = Adafruit_DHT.read_retry(Adafruit_DHT.DHT11, 23)
    # return dict
    return (str(RH), str(T))

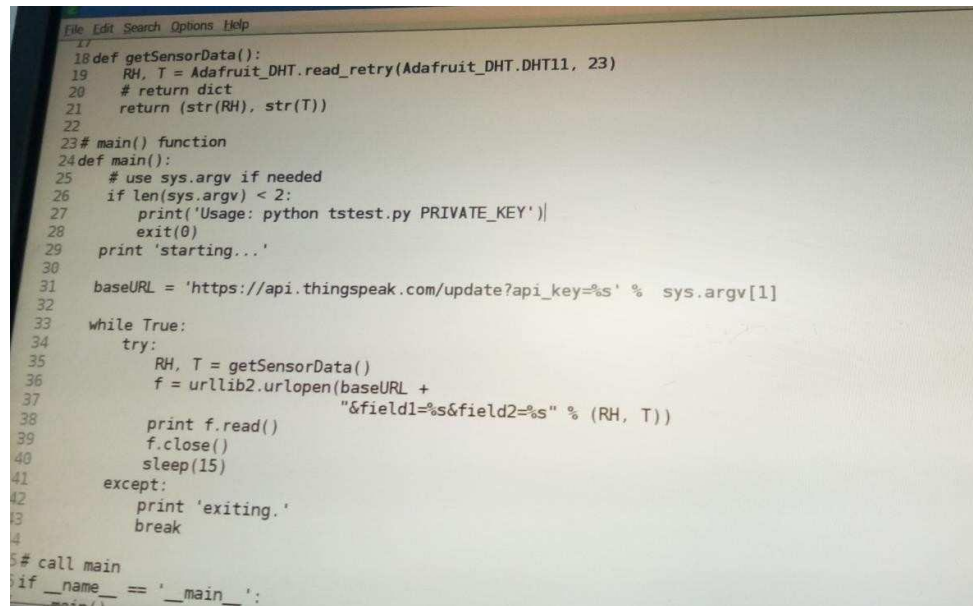
# main() function
def main():
    # use sys.argv if needed
    if len(sys.argv) < 2:
        print('Usage: python tstest.py PRIVATE_KEY')
        exit(0)
    print('starting...')

    baseURL = 'https://api.thingspeak.com/update?api_key=%s'%sys.argv[1]

    while True:
        try:
            RH, T = getSensorData()
            f = urllib2.urlopen(baseURL +
                                "&field1=%s&field2=%s"%(RH, T))
            printf.read()
            f.close()
            sleep(15)
        except:
            print('exiting.')
            break

# call main
```

```
if __name__ == '__main__':  
    main()
```



```
17  
18 def getSensorData():  
19     RH, T = Adafruit_DHT.read_retry(Adafruit_DHT.DHT11, 23)  
20     # return dict  
21     return (str(RH), str(T))  
22  
23 # main() function  
24 def main():  
25     # use sys.argv if needed  
26     if len(sys.argv) < 2:  
27         print('Usage: python tstest.py PRIVATE_KEY')  
28         exit(0)  
29     print 'starting...'  
30  
31     baseUrl = 'https://api.thingspeak.com/update?api_key=%s' % sys.argv[1]  
32  
33     while True:  
34         try:  
35             RH, T = getSensorData()  
36             f = urllib2.urlopen(baseUrl +  
37                               "&field1=%s&field2=%s" % (RH, T))  
38             print f.read()  
39             f.close()  
40             sleep(15)  
41         except:  
42             print 'exiting.'  
43             break  
44  
45 # call main  
46 if __name__ == '__main__':  
47     main()
```

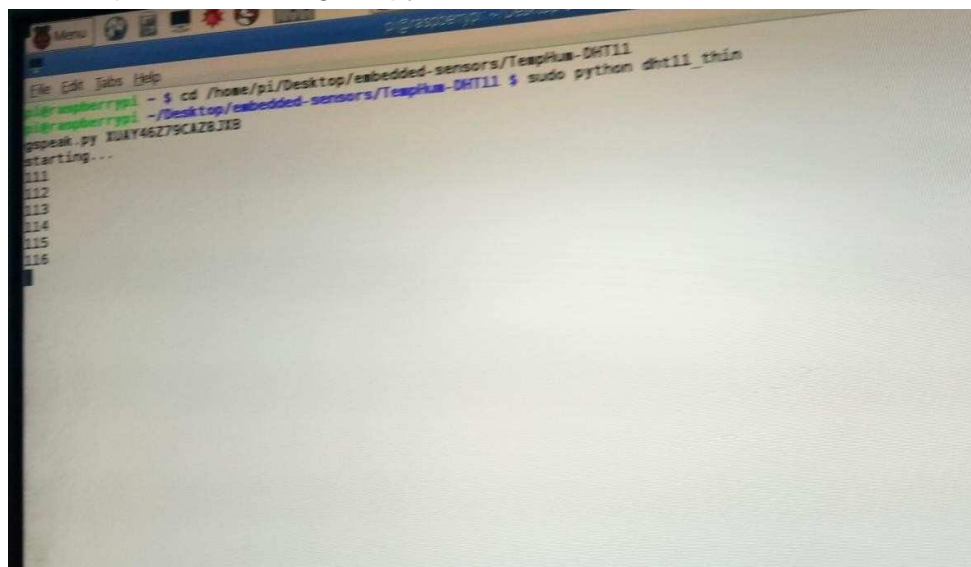
[snapshot of the coding](#)

5. Save the code as 'dht_thingspeak.py'.
6. Use Adafruit DHT Library to make this code run.
7. Now run the following command at the terminal to execute the code:

```
sudopython dht_thingspeak.py 'writeAPI Key'.
```

Here, mention the write API key of your thingspeak channel at the place 'writeAPIKey'.

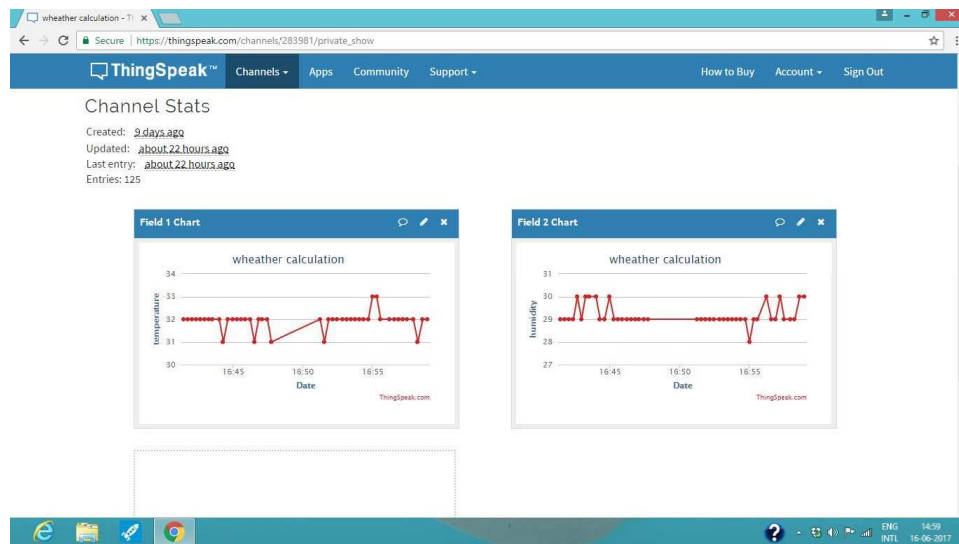
8. The snapshot of running the python code is as follows:



```
pi@raspberrypi ~ $ cd /home/pi/Desktop/embedded-sensors/Tespikun-DHT11  
pi@raspberrypi ~/Desktop/embedded-sensors/Tespikun-DHT11 $ sudo python dht11_thin  
thingspeak.py IUAY46Z79CAZ8JTB  
starting...  
111  
112  
113  
114  
115  
116
```


Acquiring Data in Think speak_:

The data at your Think Speak channel will be visible as follows:





SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

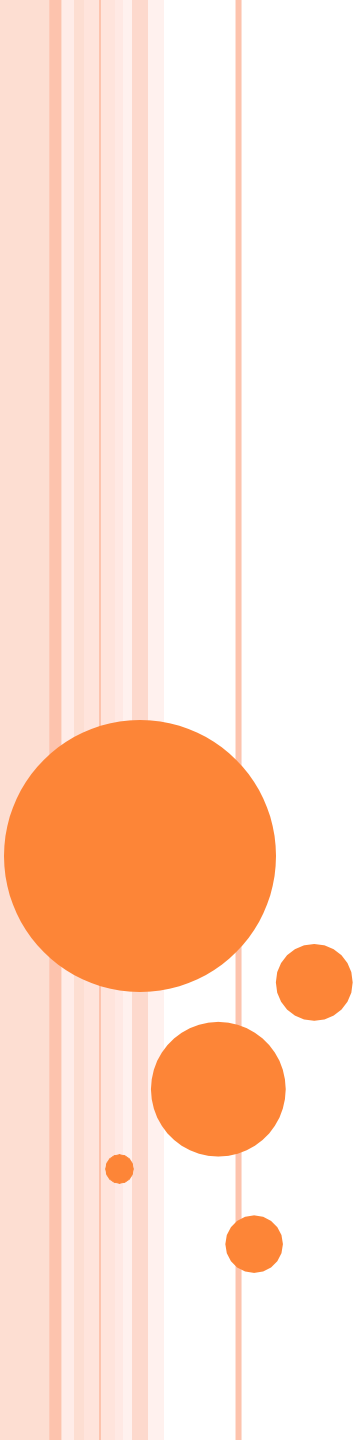
SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT V – EMBEDDED SYSTEM DESIGN AND DEVELOPMENT
SPHA5204

UNIT V

IDE, or Integrated Development Environment, is a software application that combines all of the features and tools needed by a software developer. Examples of IDEs include NetBeans, Eclipse, IntelliJ, and Visual Studio.



CASE STUDY OF AN EMBEDDED SYSTEM FOR ASMART CARD

SMART CARD

- Smart card is one of the most used embedded system today. It is used for credit, debit bank card, e-wallet card, identification card, medical card (for history and diagnosis details) and card for a number of new innovative application.



EMBEDDED HARDWARE COMPONENTS

- Microcontroller or ASIP
- RAM for temporary variables and Stack
- OTP ROM for application codes and RTOS codes for scheduling the tasks
- Flash for storing user data, user address, user identification codes, card number and expiry date
- Timer and interrupt controller
- A carrier frequency generating circuit and ASK modulator
- Interfacing circuit for the IOs.
- Charge pumps for delivering power to the antenna for transmission and for the system circuits.



EMBEDDED SOFTWARE COMPONENTS

- Boot-up, initialization and OS program
- Smart card secure file system
- Connection establishment and termination
- Communication with the host
- Cryptography algorithm
- Host authentication
- Card authentication
- Saving addition parameters or recent new data sent by the host(ex- balance receipt)



12.4 CASE STUDY OF AN EMBEDDED SYSTEM FOR A SMART CARD

Section 1.10.3 introduced the smart card system hardware and software. Section 12.4.1 gives the requirements and functioning of smart card communication system. Section 12.4.2 gives the class diagram. Figure 1.13 showed smart card-system hardware components for a contact-less smart. Sections 12.4.3 and 12.4.4 give the hardware and software architecture and synchronization model. Section 12.4.5 gives the exemplary codes.

12.4.1 Requirements

Assume a contact-less smart card for bank transactions. Let it not be magnetic. [The earlier card used a magnetic strip to hold the nonvolatile memory. Nowadays, it is EEPROM or flash that is used to hold nonvolatile application data.] Requirements of smart card communication system with a host can be understood through a requirement-table given in Table 12.4.

12.4.2 Class Diagram

Table 12.4 listed the functions and the different tasks. An abstract class is Task_CardCommunication. Figure 12.17 shows the class diagram of Task_CardCommunication. A cycle of actions and card-host synchronization in the card leads us to the model Task_CardCommunication for system tasks. Card system communicates to host for identifying host and authenticating itself to the host. ISR1_Port_IO, ISR2_Port_IO and ISR3_Port_IO are interfaces to the tasks. [A class gives the implementation methods of the interfacing routines.] The task_Appl, task_PW, task_ReadPort, and resetTask are the objects of Task_Appl, Task_PW, Task_ReadPort and Task_Reset, respectively. These classes are extended classes of abstract class Task_CardCommunication.

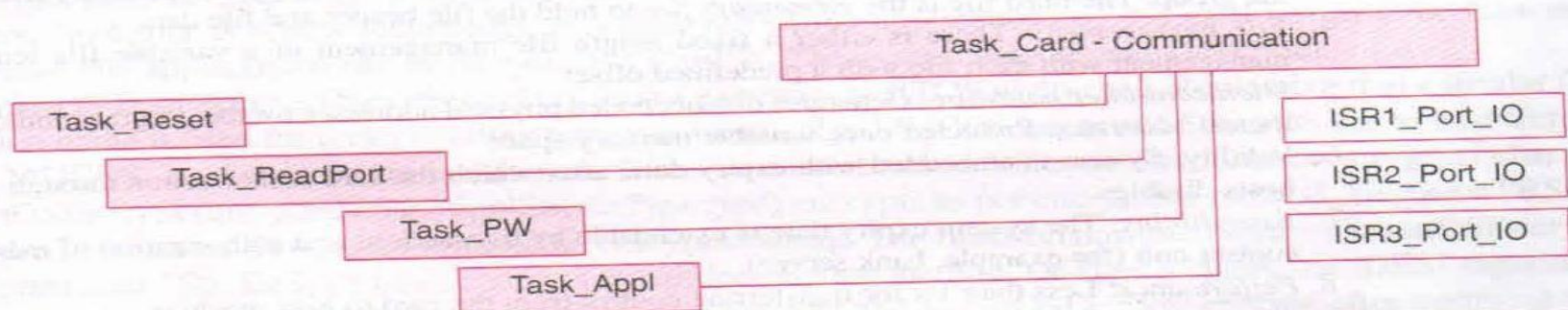


Fig. 12.17 Class diagrams of Task_CardCommunication

Table 12.4 Requirements of smart card communication system with a host

<i>Requirement</i>	<i>Description</i>
Purpose	1. Enabling authentication and verification of card and card holder by a host and enabling GUI at host machine to interact with the card holder/user for the required transactions; for example, financial transactions with a bank or credit card transactions.
System Functioning	<ol style="list-style-type: none"> 1. The card inserts at host machine. The radiations from the host activate a charge pump at card. The charge pump powers the SoC circuit, which consists of card processor, memory, timer, interrupt handler and Port_IO. 2. On power up, system-reset signals resetTask to start. The resetTask sends the messages—<i>requestHeader</i> and <i>requestStart</i> for waiting task task_ReadPort. 3. task_ReadPort sends requests for host identification and reads through the Port_IO the host-identification message and request from host for card identification. 4. The task_PW sends through Port_IO the requested card identification after system receives the host identity through Port_IO. 5. The task_Appl then runs required API. The <i>requestApplClose</i> message closes the application. 6. The card can now be withdrawn and all transactions between card-holder/user now takes place through GUIs using at the host control panel (screen or touch screen or LCD display panel).
Inputs	1. Received header and messages at IO port <i>Port_IO</i> from host through the antenna.
Signals, Events and Notifications	<ol style="list-style-type: none"> 1. On power up by radiation-powered charge-pump supply of the card, a <i>signal</i> to start the system boot program at resetTask. 2. Card start <i>requestHeader</i> message to task_ReadPort from resetTask. 3. Host authentication request <i>requestStart</i> message to task_ReadPort from resetTask to enable requests for Port_IO. 4. <i>UserPW</i> verification message (notification) through Port_IO from host. 5. Card application close request <i>requestApplClose</i> message to Port_IO.
Outputs	Transmitted headers and messages at Port_IO through antenna.
Control Panel	No control panel is at the card. The control panel and GUIs activate at the host machine (for example, at ATM or credit card reader).

Design metrics

1. *Power Source and Dissipation*: Radiation powered contact-less operation.
2. *Code size*: Code-size generated should be optimum. The card system memory needs should not exceed 64 kB memory. Limited use of data types; multidimensional arrays, long 64-bit integer and floating points and very limited use of the error handlers, exceptions, signals, serialization, debugging and profiling.
3. *File system(s)*: Three-layered file system for the data. One file for the *master file* to store all file headers. A header has strings for file status, access conditions and file-lock. The second file is a *dedicated file* to hold a file grouping and headers of the immediate successor elementary files of the group. The third file is the *elementary file* to hold the file header and file data.
4. *File management*: There is either a fixed length file management or a variable file length management with each file with a predefined offset.
5. *Microcontroller hardware*: Generates distinct coded physical addresses for the program and data logical addresses. Protected once writable memory space.
6. *Validity*: System is embedded with expiry date, after which the card authorization through the hosts disables.
7. *Extendibility*: The system expiry date is extendable by transactions and authorization of master control unit (for example, bank server).
8. *Performance*: Less than 1 s for transferring control from the card to host machine.

Test and validation conditions

9. *Process Deadlines*: None.
10. *User Interfaces*: At host machine, graphic at LCD or touchscreen display on LCD and commands for card holder (card user) transactions.
11. *Engineering Cost*: US\$ 50000 (assumed).
12. *Manufacturing Cost*: US\$ 1 (assumed).
1. Tested on different host machine versions for fail proof card-host communication.

SMART CARD HARDWARE COMPONENTS

2.4.3 Hardware and Software Architecture

Smart card hardware was introduced in Section 1.10.3. Figure 12.18 shows hardware inside the card.

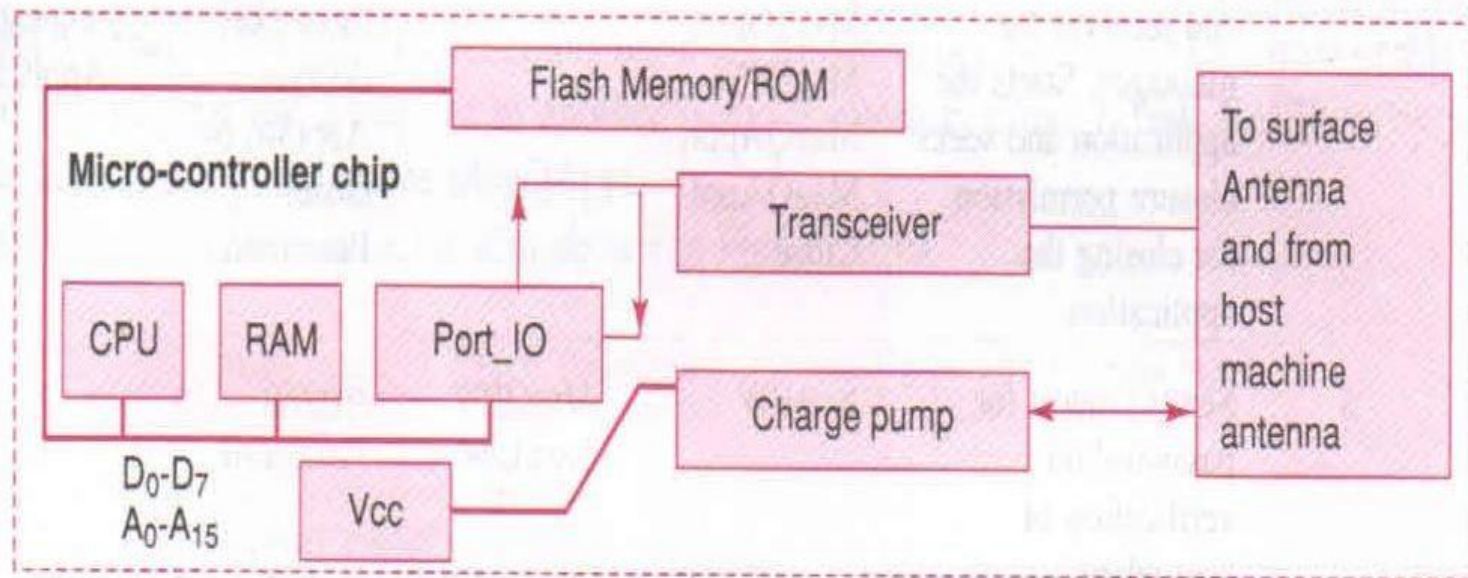


Fig. 12.18 Smart card hardware

LIST OF TASKS, FUNCTIONS AND IPC

Table 12.5 List of tasks, Functions and IPCs

<i>Task Function</i>	<i>Priority</i>	<i>Action</i>	<i>IPCs pending</i>	<i>IPCs posted</i>	<i>String or System or Host input</i>	<i>String or System or Host Output</i>
resetTask	1	Initiates system timer ticks, creates tasks, sends initial messages and suspends itself.	None	SigReset, MsgQStart	SmartOS call to the main	<i>request-Header, requestStart</i>
task_Read Port	2	Wait for resetTask Suspension, sends the queue messages and receives the messages. Starts the application and seeks closure permission for closing the application.	SigReset, Messages from MsgQStart, MsgQPW, MsgQAppl, MsgQAppl-Close	SePW	Functions Smart OS-Encrypt, SmartOS-decrypt, ApplStr, Str, close-Permitted	<i>request-password, request-Appl, request-ApplClose</i>
task_PW	3	Sends request for password on verification of host when SemPW = 1.	SemPW	MsgQPW, SemAppl	<i>request-Password</i>	—
task_Appl	8	when SemPW = 1. runs the application program.	SemAppl	MsgQAppl.	—	—

TASKS AND THE SYNCHRONIZATION MODEL

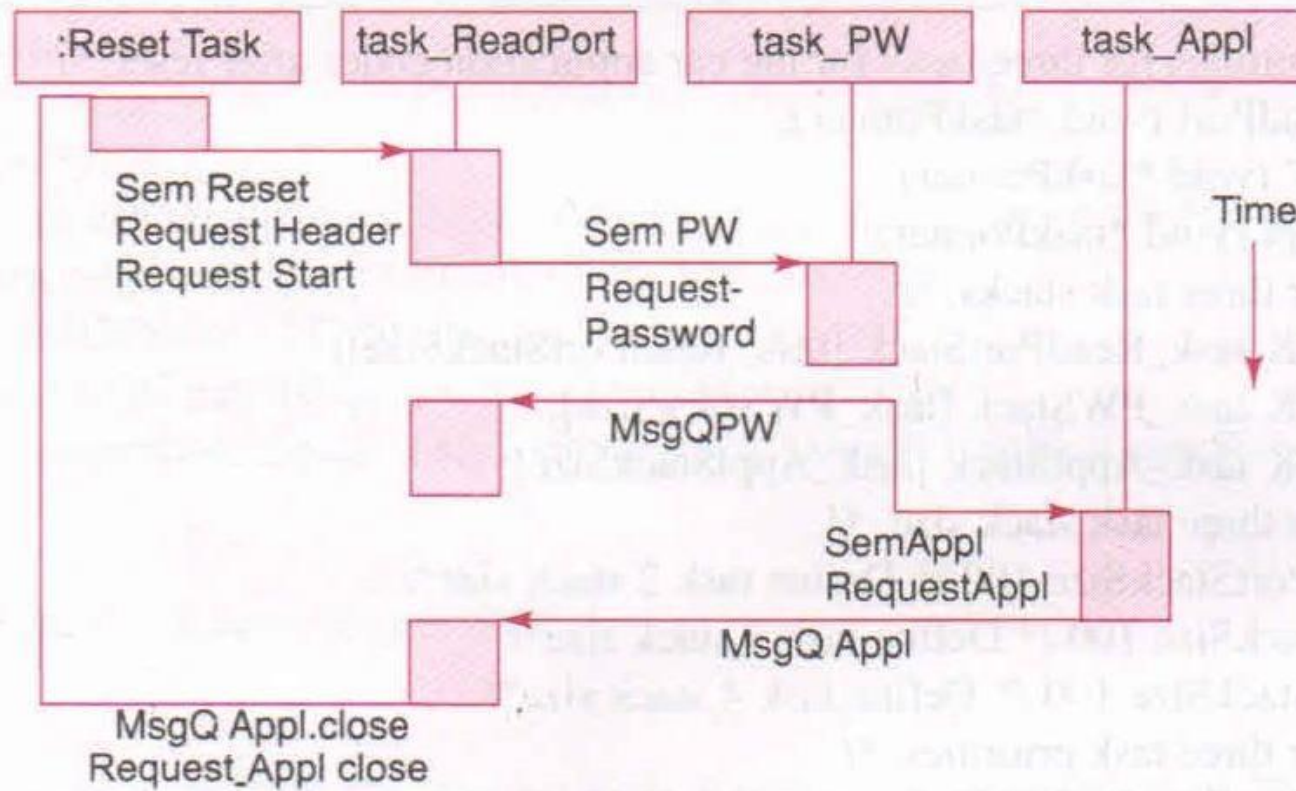


Fig. 12.19 Tasks and the synchronization model



Case Study: Cruise Control

Murray Cole

Cruise Control

Cruise Control

Basic idea

allow driver to set a speed to be maintained without his/her intervention (*e.g.* 70mph down a long straight motorway)

no need to keep accelerator pressed (less driver fatigue)

Specification

Pin down some requirements

Driver can request the system to maintain the current speed
Driver can always turn it off

System should not operate after braking

System should allow the driver to travel faster than the set speed

Design an FSM

We need to

specify the inputs specify

the outputs

decide on the required states (and a start state) specify the transitions

Driver Inputs

on: on/off button

set: set the cruise speed to the current speed

brake: the brake has been pressed accP: the accelerator has been

pressed accR: the accelerator has been released

resume: resume travelling at the set speed

Sensor Inputs

`correct`: indicates the car is travelling at the correct speed.
`slow`: indicates the car is going slower than the set speed

`fast`: indicates the car is going faster than the set speed

Control Outputs

store: store the current speed as the cruise speed

inc: increase the throttle

dec: decrease the throttle

States

Off: System is not operational.

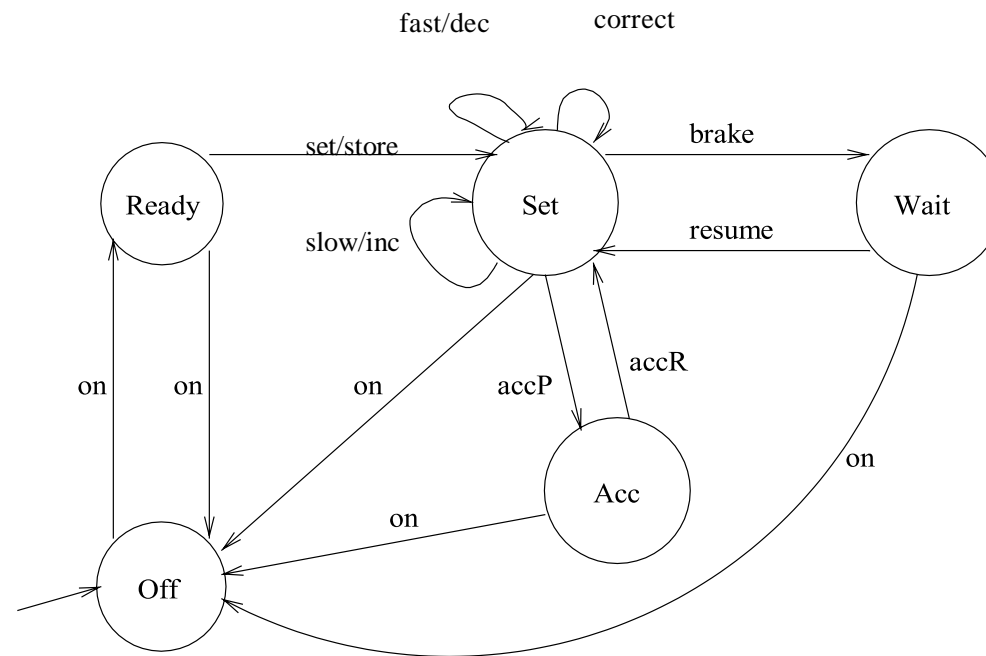
Ready: Switched on but no cruise speed set.

Set: Actively controlling speed.

Wait: Speed set but subsequently overridden by brake. Wait to be told to resume control.

Acc: Accelerator is currently pressed down (so override)

The Controller



All other inputs are loops with no output (ie ignored). Omitted herefor clarity!

Java Implementation

Model (real system would interact with car hardware) inputs with strings from the keyboard

outputs with strings to the monitor

state by a variable holding an integer

transitions by code which changes state and makes outputs in response to inputs

Overview

```
// Initialise Values

// Repeat Foreverwhile(true) {
    // Display Current State
    // Read input from keyboard
    // Make a transition
    // Display any output
}
```



```
public class CruiseControl {  
    // Inputs  
    public static final int on = 1; public static final  
    int set = 2; public static final int brake = 3; public  
    static final int accP = 4;  
  
    .....  
    // Outputs  
    public static final int store = 10; public static  
    final int inc = 11; public static final int dec = 12;  
  
    // States  
    public static final int OFF = 13; public static final  
    int READY = 14;  
  
    .....
```

```
int input    = 0; int output = 0; int state    = OFF;

while(true) {
    String in = keyboard.readLine();if (in.equals("on"))
    input = on;
    else if (in.equals("set")) input = set;
    .....
    else input = 0;

    // Make the appropriate transition (NEXT OVERHEAD)switch(output) { // Display any

    output
        case store: System.out.println(" store "); break;
        .....
    }
}
```

```
switch(state) {  
    .....  
    case SET:  
        switch(input)  
        {  
            case on:  
                state = OFF; break; case brake:  
                state = WAIT; break; case accP:  
                state = ACC; break; case fast:  
                output = dec; break; case slow:  
                output = inc; break; case correct:  
                break;  
  
            default: break;  
        }  
        break;  
        .....  
}
```

Maintenance

The incident occurred when the driver was on a highway on a rainy night. The traffic was slow, travelling at about 40 mph. The driver engaged cruise control and set it to 40 mph. Later the rain cleared and the traffic got faster so the driver used the accelerator to increase the speed to 60 mph and travelled in this mode for some miles (the controller still in set mode but overridden by the accelerator).

Coming to the exit ramp the driver turned off and released the accelerator to coast up the ramp. At that point the cruise control aimed to stabilise the speed at the set level (40 mph). The driver was taken by surprise and lost control of the car which travelled through a stop sign without braking. Fortunately no accident occurred.

PART A

1. Discuss about Real time Operating system.

2. Analyze the Embedded System in real time
3. Explain the Software Development Environment
4. Grade the Hardware debugging techniques
5. Justify the RTOS
6. Compare the assembler, compiler and linker
7. Justify the applications of Embedded System.
8. Explain the working of Robotics
9. Assess the Home Automation in Real Time Application
10. Describe the Industrial Automation in Real Time Application.

PART B

1. Evaluate the role of Embedded system in Robotics.
2. Create the design diagram for the Adaptive cruise control in car.
3. Construct the diagram for smart card using Embedded System.
4. Explain the working of the Home Automation with suitable Example.
5. Measure the parameters used for Industrial automation.

TEXT / REFERENCE BOOKS

1. A.K Ray and K M Bhurchandi, Advanced Microprocessors and Peripherals,3RD edition, TMH,2017.
2. Joseph Yiu, The Definitive Guide to the ARM Cortex-M3,2nd Edition, Newnes,2015.
3. Dr.MarkFisher, ARM Cortex M4 Cookbook, Packt, 2016.
4. David Hanes, “IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things”, Cisco press, 2017.
5. Olivier Hersent , David Boswarthick , Omar Elloumi, “The Internet of Things: Key Applications and Protocols”, 2nd Edition, Wiley, 2012.
6. Rajkamal, “Embedded system-Architecture, Programming, Design”, TMH, 2011.
7. Jonathan W.Valvano, “Embedded Microcomputer Systems,Real Time Interfacing”,Cengage Learning,3rd Edition, 2012.