KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

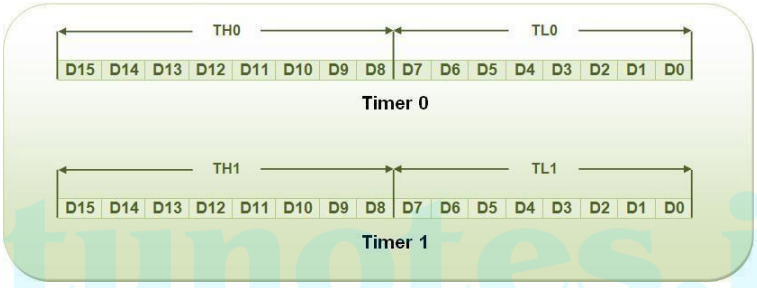
MODULE IV
ADVANCED CONCEPTS

TIMERS AND COUNTERS

Timers/Counters are used generally for

- Time reference
- Creating delay
- Wave form properties measurement
- Periodic interrupt generation
- Waveform generation

8051 has two timers, Timer 0 and Timer 1.



Timer in 8051 is used as timer, counter and baud rate generator. Timer always counts up irrespective of whether it is used as timer, counter, or baud rate generator: Timer is always incremented by the microcontroller. The time taken to count one digit up is based on master clock frequency.

If Master CLK=12 MHz,

Timer Clock frequency = Master CLK/12 = 1 MHz
Timer Clock Period = 1micro second

This indicates that one increment in count will take 1 micro second.

The two timers in 8051 share two SFRs (TMOD and TCON) which control the timers, and each timer also has two SFRs dedicated solely to itself (TH0/TL0 and TH1/TL1).

The following are timer related SFRs in 8051.

SFR Name	Description	SFR Address
TH0	*Timer 0 High Byte*	*8Ch*
TL0	*Timer 0 Low Byte*	*8Ah*
TH1	*Timer 1 High Byte*	*8Dh*
TL1	*Timer 1 Low Byte*	*8Bh*
TCON	*Timer Control*	*88h*
TMOD	*Timer Mode*	*89h*

TMOD Register

TMOD : Timer/Counter Mode Control Register (Not Bit Addressable)

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
TIMER 1				TIMER 0			

GATE	When TRx (in TCON) is set and GATE = 1, TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).
C/$\overline{T}$	Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from Tx input pin).
M1	Mode selector bit (NOTE 1).
M0	Mode selector bit (NOTE 1).

Note 1 :

M1	M0	OPERATING MODE	
0	0	0	13–bit Timer
0	1	1	16–bit Timer/Counter
1	0	2	8–bit Auto–Reload Timer/Counter
1	1	3	(Timer 0) TL0 is an 8–bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an 8–bit Timer and is controlled by Timer 1 control bits.
1	1	3	(Timer 1) Timer/Counter 1 stopped.

8051 timers have both software and hardware controls. The start and stop of a timer is controlled by software using the instruction **SETB TR1** and **CLR TR1** for timer 1, and **SETB TR0** and **CLR TR0** for timer 0.

The SETB instruction is used to start it and it is stopped by the CLR instruction. These instructions start and stop the timers as long as GATE = 0 in the TMOD register. Timers can be started and stopped by an external source by making GATE = 1 in the TMOD register.

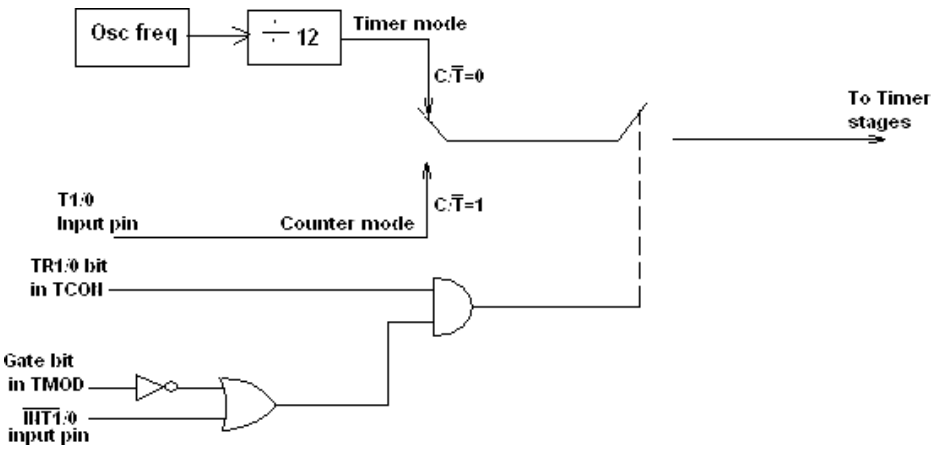
TCON Register

TCON : Timer/Counter Control Register (Bit Addressable)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when the Timer/Counter 1 overflows. Cleared by hardware as processor vectors to the interrupt service routine.
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn Timer/Counter ON/OFF.
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when the Timer/Counter 0 overflows. Cleared by hardware as processor vectors to the service routine.
TR0	TCON.4	Timer 0 run control bit. Set/cleared by software to turn Timer/Counter 0 ON/OFF.
IE1	TCON.3	External Interrupt 1 edge flag. Set by hardware when External interrupt edge is detected. Cleared by hardware when interrupt is processed.
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/flow level triggered External Interrupt.
IE0	TCON.1	External Interrupt 0 edge flag. Set by hardware when External Interrupt edge detected. Cleared by hardware when interrupt is processed.
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.

Timer/ Counter Control Logic.



TIMER MODES

Timers can operate in four different modes. They are as follows

Timer Mode-0: In this mode, the timer is used as a 13-bit UP counter as follows.

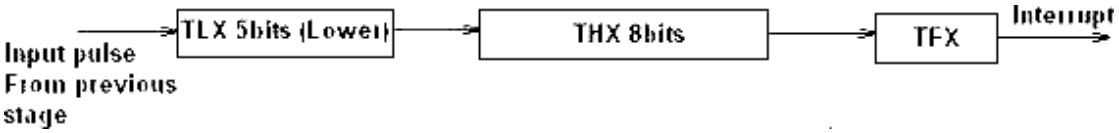


Fig. Operation of Timer on Mode-0

The lower 5 bits of TLX and 8 bits of THX are used for the 13 bit count.Upper 3 bits of TLX are ignored. When the counter rolls over from all 0's to all 1's, TFX flag is set and an interrupt is generated. The input pulse is obtained from the previous stage. If TR1/0 bit is 1 and Gate bit is 0,the counter continues counting up. If TR1/0 bit is 1 and Gate bit is 1, then the operation of the counter is controlled by input. This mode is useful to measure the width of a given pulse fed to input.

Timer Mode-1: This mode is similar to mode-0 except for the fact that the Timer operates in 16-bit mode.

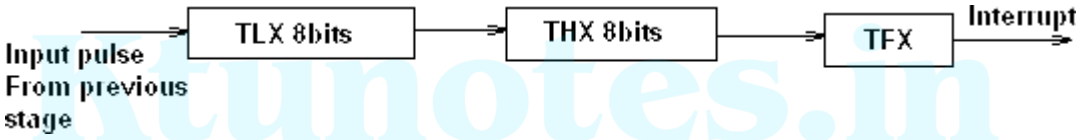


Fig: Operation of Timer in Mode 1

Timer Mode-2: (Auto-Reload Mode): This is a 8 bit counter/timer operation. Counting is performed in TLX while THX stores a constant value. In this mode when the timer overflows i.e. TLX becomes FFH, it is fed with the value stored in THX. For example if we load THX with 50H then the timer in mode 2 will count from 50H to FFH. After that 50H is again reloaded. This mode is useful inapplications like fixed time sampling.

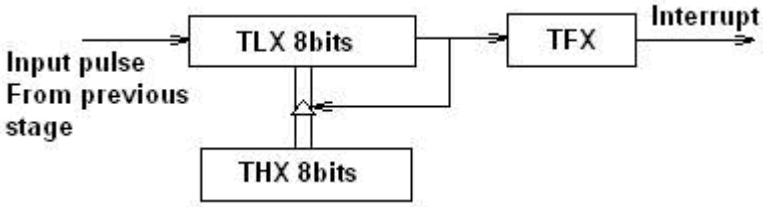


Fig: Operation of Timer in Mode 2

Timer Mode-3: Timer 1 in mode-3 simply holds its count. The effect is same as setting TR1=0.Timer0 in mode-3 establishes TL0 and TH0 as two separate counters.

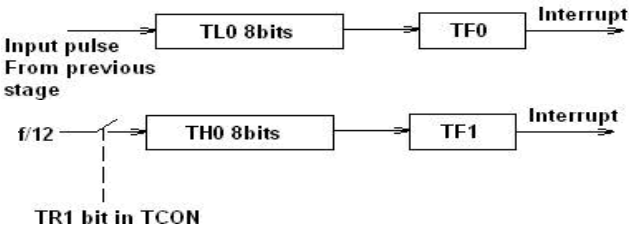


Fig: Operation of Timer in Mode 3

Control bits TR1 and TF1 are used by Timer-0 (higher 8 bits) (TH0) in Mode-3 while TR0 and TF0are available to Timer-0 lower 8 bits(TL0).

SERIAL COMMUNICATION.

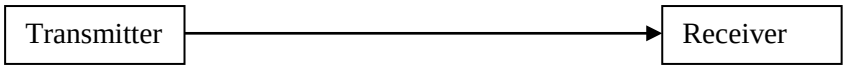
DATA COMMUNICATION

The 8051 microcontroller is parallel device that transfers eight bits of data simultaneously over eight data lines to parallel I/O devices. Parallel data transfer over a long is very expensive. Hence, a serial communication is widely used in long distance communication. In serial data communication, 8-bit data is converted to serial bits using a parallel in serial out shift register and then it is transmitted over a single data line. The data byte is always transmitted with least significant bit first.

BASICS OF SERIAL DATA COMMUNICATION,

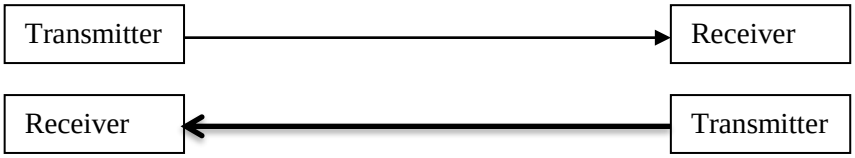
Communication Links

1. Simplex communication link: In simplex transmission, the line is dedicated for transmission.The transmitter sends and the receiver receives the data.



2. Half duplex communication link: In half duplex, the communication link can be used for eithertransmission or reception. Data is transmitted in only one direction at a time.

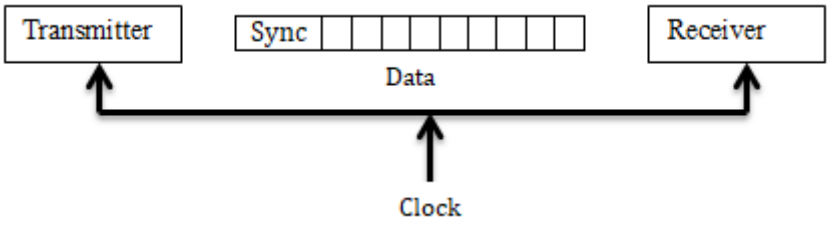
3. Full duplex communication link: If the data is transmitted in both ways at the same time, it is a full duplex i.e. transmission and reception can proceed simultaneously. This communication link requires two wires for data, one for transmission and one for reception.



Types of Serial communication:

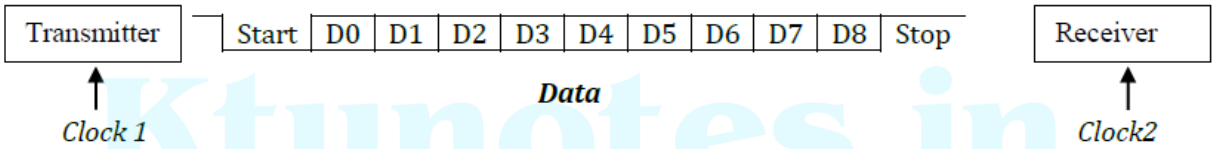
Serial data communication uses two types of communication.

1. Synchronous serial data communication: In this transmitter and receiver are synchronized. It uses a common clock to synchronize the receiver and the transmitter. First the synch character is sent and then the data is transmitted. This format is generally used for high speed transmission..



In Synchronous serial data communication a block of data is transmitted at a time

2. Asynchronous Serial data transmission: In this, different clock sources are used for transmitter and receiver. In this mode, data is transmitted with start and stop bits. A transmission begins with start bit, followed by data and then stop bit. For error checking purpose parity bit is included just prior to stop bit. In Asynchronous serial data communication a single byte is transmitted at a time.



Baud rate:

The rate at which the data is transmitted is called baud or transfer rate. The baud rate is the reciprocal of the time to send one bit. In asynchronous transmission, baud rate is not equal to number of bits per second. This is because; each byte is preceded by a start bit and followed by parity and stop bit. For example, in synchronous transmission, if data is transmitted with 9600 baud, it means that 9600 bits are transmitted in one second. For bit transmission time = 1 second/ 9600 = 0.104 ms.

6.1.1. 8051 SERIAL COMMUNICATION

The 8051 supports a full duplex serial port.

Three special function registers support serial communication.

1. **SBUF Register**: Serial Buffer (SBUF) register is an 8-bit register. It has separate SBUF registers for data transmission and for data reception. For a byte of data to be transferred via the TXD line, it must be placed in SBUF register. Similarly, SBUF holds the 8-bit data received by the RXD pin and read to accept the received data.
2. **SCON register**: The contents of the Serial Control (SCON) register are shown below. This register contains mode selection bits, serial port interrupt bit (TI and RI) and also the ninth data bit for transmission and reception (TB8 and RB8).

Serial Port Control (SCON) Register							
D7	D6	D5	D4	D3	D2	D1	D0
SM0	**SM1**	**SM2**	**REN**	**TB8**	**RB8**	**TI**	**RI**

- SM0 (SCON.7) : Serial communication mode selection bit
- SM1 (SCON.6) : Serial communication mode selection bit

SM0	SM1	Mode	Description	Baud rate
0	0	Mode 0	8-bit shift register mode	Fosc / 12
0	1	Mode 1	8-bit UART	Variable (set by timer 1)
1	0	Mode 2	9-bit UART	Fosc/ 32 or Fosc/64
1	1	Mode 3	9-bit UART	Variable (set by timer 1)

- SM2 (SCON.5) : Multiprocessor communication bit. In modes 2 and 3, if set this will enable multiprocessor communication.
- REN (SCON.4) : Enable serial reception
- TB8 (SCON.3) : This is 9th bit that is transmitted in mode 2 & 3.
- RB8 (SCON.2) : 9th data bit is received in modes 2 & 3.
- TI (SCON.1) : Transmit interrupt flag, set by hardware must be cleared by software.
- RI (SCON.0) : Receive interrupt flag, set by hardware must be cleared by software.

3. **PCON register**: The SMOD bit (bit 7) of PCON register controls the baud rate in asynchronous mode transmission.

Power mode Control (PCON) Register							
D7	D6	D5	D4	D3	D2	D1	D0
SMOD	**--**	**--**	**--**	**GF1**	**GF0**	**PD**	**IDL**

- SMD (PCON.7): Serial rate modify bit. Set to 1 by program to double baud rate using timer 1 for modes 1, 2, and 3. cleared by program to use timer 1 baud rate.
- GF1 (PCON.3) : General Purpose user flag bit.
- GF0 (PCON.2) : General Purpose user flag bit.
- PD (PCON.1) : Power down bit. Set to 1 by program to enter power down configuration for CHMOS processors.
- IDL (PCON.0) : Idle mode bit. Set to 1 by program to enter idle mode configuration for CHMOS processors.

SERIAL COMMUNICATION MODES

1. *Mode 0*

In this mode serial port runs in synchronous mode. The data is transmitted and received through RXD pin and TXD is used for clock output. In this mode the baud rate is 1/12 of clock frequency.

2. Mode 1

In this mode SBUF becomes a 10 bit full duplex transceiver. The ten bits are 1 start bit, 8 data bit and 1 stop bit. The interrupt flag TI/RI will be set once transmission or reception isover. In this mode the baud rate is variable and is determined by the timer 1 overflow rate.

Baud rate = $[2^{smod}/32]$ x Timer 1 overflow Rate

= $[2^{smod}/32]$ x [Oscillator Clock Frequency] / [12 x [256 – [TH1]]]

3. Mode 2

This is similar to mode 1 except 11 bits are transmitted or received. The 11 bits are, 1 startbit, 8 data bit, a programmable 9^{th} data bit, 1 stop bit.

Baud rate = $[2^{smod}/64]$ x Oscillator Clock Frequency

4. Mode 3

This is similar to mode 2 except baud rate is calculated as in mode 1

CONNECTIONS TO RS-232

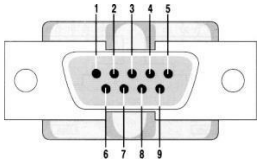
RS-232 standards:

To allow compatibility among data communication equipment made by various manufactures, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. Since the standard was set long before the advent of logic family, its input and output voltage levels are not TTL compatible.

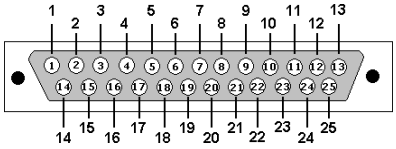
In RS232, a logic one (1) is represented by -3 to -25V and referred as MARK while logic zero

(0) is represented by +3 to +25V and referred as SPACE. For this reason to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic level to RS232 voltage levels and vice-versa. MAX232 IC chips are commonly referred as linedrivers.

In RS232 standard we use two types of connectors. DB9 connector or DB25 connector.



DB9 Male Connector



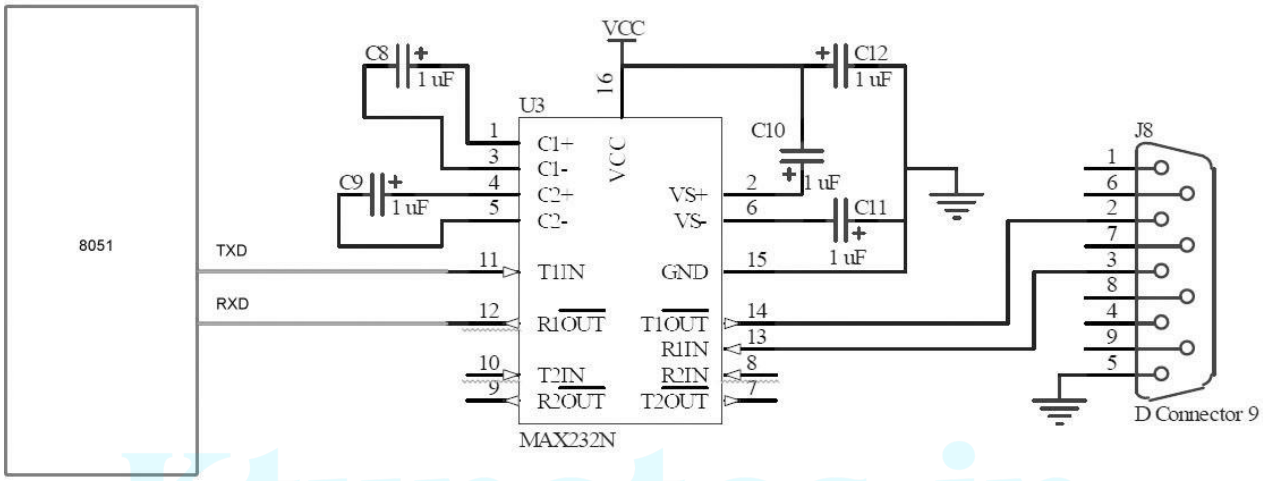
DB25 Male

ConnectorThe pin description of DB9 and DB25 Connectors are as follows

DB-25 Pin No.	DB-9 Pin No.	Abbreviation	Full Name
Pin 2	Pin 3	TD	Transmit Data
Pin 3	Pin 2	RD	Receive Data
Pin 4	Pin 7	RTS	Request To Send
Pin 5	Pin 8	CTS	Clear To Send
Pin 6	Pin 6	DSR	Data Set Ready
Pin 7	Pin 5	SG	Signal Ground
Pin 8	Pin 1	CD	Carrier Detect
Pin 20	Pin 4	DTR	Data Terminal Ready
Pin 22	Pin 9	RI	Ring Indicator

The 8051 connection to MAX232 is as follows.

The 8051 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TXD, RXD. Pin 11 of the 8051 (P3.1) assigned to TXD and pin 10 (P3.0) is designated as RXD. These pins TTL compatible; therefore they require line driver (MAX 232) to make them RS232 compatible. MAX 232 converts RS232 voltage levels to TTL voltage levels and vice versa. One advantage of the MAX232 is that it uses a +5V power source which is the same as the source voltage for the 8051. The typical connection diagram between MAX 232 and 8051 is shown below.



SERIAL COMMUNICATION PROGRAMMING IN ASSEMBLY AND C.

Steps to programming the 8051 to transfer data serially

1. The TMOD register is loaded with the value 20H, indicating the use of the Timer 1 inmode 2 (8-bit auto reload) to set the baud rate.
2. The TH1 is loaded with one of the values in table to set the baud rate for serialdata transfer.

Baud Rate	TH1 (Decimal)	TH1 (Hex)
9600	–3	FD
4800	–6	FA
2400	–12	F4
1200	–24	E8

Note: XTAL = 11.0592 MHz.

3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an8-bit data is framed with start and stop bits.
4. TR1 is set to 1 start timer 1.
5. TI is cleared by the "CLR TI" instruction.
6. The character byte to be transferred serially is written into the SBUF register.
7. The TI flag bit is monitored with the use of the instruction JNB TI, target to see if thecharacter has been transferred completely.
8. To transfer the next character, go to step 5.

Example 1. Write a program for the 8051 to transfer letter 'A' serially at 4800- baud rate, 8 bit data,1 stop bit continuously.

```
ORG 0000H
LJMP START
START: MOV TMOD, #20H    ; select timer 1 mode 2
       MOV TH1, #0FAH    ; load count to get baud rate of 4800
       MOV SCON, #50H    ; initialize UART in mode 2
                         ; 8 bit data and 1 stop bit
       SETB TR1          ; start timer
AGAIN: MOV SBUF, #'A'    ; load char 'A' in SBUF
BACK:  JNB TI, BACK      ; Check for transmit interrupt flag
       CLR TI            ; Clear transmit interrupt flag
       SJMP AGAIN
       END
```

Example 2. Write a program for the 8051 to transfer the message 'EARTH' serially at 9600 baud, 8 bit data, 1 stop bit continuously.

```
ORG 0000H
LJMP START
START: MOV TMOD, #20H          ; select timer 1 mode 2
MOV TH1, #0FDH                 ; load count to get reqd. baud rate of 9600
MOV SCON, #50H                 ; initialise uart in mode 2
                               ; 8 bit data and 1 stop bit
SETB TR1                       ; start timer
LOOP: MOV A, #'E'              ; load 1st letter 'E' in A
ACALL LOAD                     ; call load subroutine
MOV A, #'A'                    ; load 2nd letter 'A' in A
ACALL LOAD                     ; call load subroutine
MOV A, #'R'                    ; load 3rd letter 'R' in A
ACALL LOAD                     ; call load subroutine
MOV A, #'T'                    ; load 4th letter 'T' in A
ACALL LOAD                     ; call load subroutine
MOV A, #'H'                    ; load 4th letter 'H' in A
ACALL LOAD                     ; call load subroutine
SJMP LOOP                      ; repeat steps

LOAD:  MOV SBUF, A
HERE: JNB TI, HERE             ; Check for transmit interrupt flag
CLR TI                         ; Clear transmit interrupt flag
RET
END
```

Example 3. Write a program for the 8051 to transfer the message 'KTU' serially at 4800 baud, 8bit data, 1 stop bit continuously.

```
        ORG 0000H
        LJMP START
START:  MOV TMOD, #20H          ; select timer 1 mode 2
        MOV TH1, #0FAH          ; load count to get reqd. baud rate of 4800
        MOV SCON, #50H          ; initialise uart in mode 2
                                ; 8 bit data and 1 stop bit
        SETB TR1                ; start timer
LOOP:   MOV A, #'K'             ; load 1st letter 'K' in A
        ACALL LOAD              ; call load subroutine
        MOV A, #'T'             ; load 2nd letter 'T' in A
        ACALL LOAD              ; call load subroutine
        MOV A, #'U'             ; load 3rd letter 'U' in A
        ACALL LOAD              ; call load subroutine
        SJMP LOOP               ; repeat steps

LOAD:   MOV SBUF, A
HERE:   JNB TI, HERE            ; Check for transmit interrupt flag
        CLR TI                  ; Clear transmit interrupt flag
        RET
        END
```

Example 4. Write a program for the 8051 to transfer the message 'JCET' serially at 4800 baud, 8 bit data, 1 stop bit continuously.


```
        ORG 0000H
        LJMP START
START:  MOV TMOD, #20H          ; select timer 1 mode 2
        MOV TH1, #0FAH          ; load count to get reqd. baud rate of 4800
        MOV SCON, #50H          ; initialize UART in mode 2
                                ; 8 bit data and 1 stop bit
        SETB TR1                ; start timer
LOOP:   MOV A, #'J'             ; load 1st letter 'J' in A
        ACALL LOAD              ; call load subroutine
        MOV A, #'C'             ; load 2nd letter 'C' in A
        ACALL LOAD              ; call load subroutine
        MOV A, #'E'             ; load 3rd letter 'E' in A
        ACALL LOAD              ; call load subroutine
        MOV A, #'T'             ; load 4th letter 'T' in A
        SJMP LOOP               ; repeat steps

LOAD:   MOV SBUF, A
HERE:   JNB TI, HERE            ; Check for transmit interrupt flag
        CLR TI                  ; Clear transmit interrupt flag
        RET
        END
```

ARM 7 MICROCONTROLLER

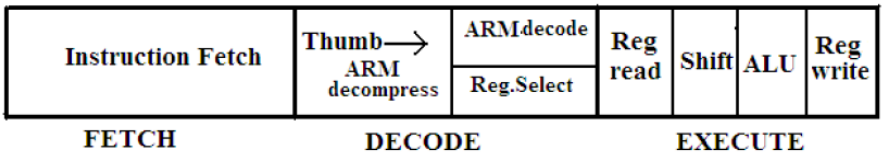
INTRODUCTION:

The ARM was originally developed at Acorn Computers Limited of Cambridge , England, between 1983 and 1985. It was the first RISC microprocessor developed for commercial use and has some significant differences from subsequent RISC architectures. In 1990 ARM Limited was established as a separate company specifically to widen the exploitation of ARM technology and it is established as a market-leader for low-power and cost-sensitive embedded applications. The ARM is supported by a toolkit which includes an instruction set emulator for hardware modelling and software testing and benchmarking, an assembler, C and C++ compilers, a linker and a symbolic debugger.

The 16-bit CISC microprocessors that were available in 1983 were slower than standard memory parts. They also had instructions that took many clock cycles to complete (in some cases, many hundreds of clock cycles), giving them very long interrupt latencies. As a result of these frustrations with the commercial microprocessor offerings, the design of a proprietary microprocessor was considered and ARM chip was designed.

ARM 7TDMI-S Processor :

The ARM7TDMI-S processor is a member of the ARM family of general-purpose 32-bit microprocessors. The ARM family offers high performance for very low-power consumption and gate count. The ARM7TDMI-S processor has a Von Neumann architecture, with a single 32-bit data bus carrying both instructions and data. Only load, store, and swap instructions can access data from memory. The ARM7TDMI-S processor uses a three stage pipeline to increase the speed of the flow of instructions to the processor. This enables several operations to take place simultaneously, and the processing, and memory systems to operate continuously. In the three-stage pipeline the instructions are executed in three stages



The three stage pipelined architecture of the ARM7 processor is shown in the above figure.

ARM7TDMIS stands for

T: THUMB ;

D for on-chip Debug support, enabling the processor to halt in response to a debug request,

M: enhanced Multiplier, yield a full 64-bit result, high performance

I: Embedded ICE hardware (In Circuit emulator)
S : Synthesizable

FEATURES OF ARM PROCESSORS

The ARM processors are based on RISC architectures and this architecture has provided small implementations, and very low power consumption. Implementation size, performance, and very low power consumption remain the key features in the development of the ARM devices.
The typical RISC architectural features of ARM are :

- A large uniform register file
- A load/store architecture, where data-processing operations only operate on register
- contents, not directly on memory contents
- Simple addressing modes, with all load/store addresses being determined from register
- contents and instruction fields only uniform and fixed-length instruction fields, to
- simplify instruction decode.
- Control over both the Arithmetic Logic Unit (ALU) and shifter in most data-processing
- instructions to maximize the use of an ALU and a shifter
- Auto-increment and auto-decrement addressing modes to optimize program loops
- Load and Store Multiple instructions to maximize data throughput
- Conditional execution of almost all instructions to maximize execution throughput.

ARM Processor Families

Architecture version	Processor Families	Processor	Features	Microcontroller
ARM v4T	ARM7TDMI (1995)	ARM720T ARM740T	Von Neumann, 3-stage pipeline	LPC2100 series
	ARM9TDMI	ARM920T ARM922T ARM942T	MMU, Harvard, 5-stage pipeline	SAM9G, LPC29xx, LPC3xxx, STR9
ARM v5TE, ARM v5TEJ	ARM9E (1997)	ARM926EJ-S,	MMU, DSP, Jazelle,	SAM9XE
		ARM946E-S,	MPU, DSP	
		ARM966HS	MPU (optional), DSP	
	ARM10E	ARM1020E	MMU, DSP	

	(1999)	ARM1026EJ-S	MMU/MPU, DSP, Jazelle	
ARM v6	ARM11 (2003)	ARM1136J(F)-S	MMU, TrustZone, DSP, Jazelle	MSM7000, i.MX3x
		ARM1156T2(F)-S	MPU, DSP	
		ARM1176JZ(F)-S,	MMU, TrustZone, DSP, Jazelle	BCM2835
		ARM11 MP core N	MMU, Multiprocessor cache support DSP, Jazelle	
ARM v4T	ARM7TDMI (1995)	ARM720T ARM740T	Von Neumann, 3-stage pipeline	LPC2100 series
	ARM9TDMI	ARM920T ARM922T ARM942T	MMU, Harvard, 5-stage pipeline	SAM9G, LPC29xx, LPC3xxx, STR9
ARM v5TE, ARM v5TEJ	ARM9E (1997)	ARM926EJ-S,	MMU, DSP, Jazelle,	SAM9XE
		ARM946E-S,	MPU, DSP	
		ARM966HS	MPU (optional), DSP	
	ARM10E (1999)	ARM1020E	MMU, DSP	
		ARM1026EJ-S	MMU/MPU, DSP, Jazelle	
ARM v6	ARM11 (2003)	ARM1136J(F)-S	MMU, TrustZone, DSP, Jazelle	MSM7000, i.MX3x
		ARM1156T2(F)-S	MPU, DSP	
		ARM1176JZ(F)-S,	MMU, TrustZone, DSP, Jazelle	BCM2835
		ARM11 MP core N.	MMU/MPU, DSP, Jazelle	
ARM v6-M	Cortex	Cortex-M0	NVIC	LPC1200, 1100 series STM32F0x0, x1, x2

		Cortex-M1	FPGA TCM Interface, NVIC	STM32F1, F2, L1, W
ARM v7-M	Cortex	Cortex-M3	MPU (optional), NVIC	ST32F512-M, LPC1300, 1700, 1800
ARM v7-R	Cortex	Cortex-R4	MPU, DSP	STA1095, SAM4L, SAM4N, SAM4S
		Cortex-R4F	MPU, DSP, Floating Point	SAM4C, SAM4E, LPC40xx, 43xx, STM32 F3, F4
ARM v7-A	Cortex	Cortex-A8	MMU, Trust Zone, DSP, Jazelle, Neon, Floating Point	Freescale i.MX5X
		Cortex-A9	**MMU, Trust Zone, Multiprocessor, DSP, Jazelle, Neon Floating Point**	Freescale i.MX6QP

There are three basic instruction sets for ARM.

- A 32- bit ARM instruction set
- A 16 –bit Thumb instruction set and
- The 8-bit Java Byte code used in Jazelle state

The Thumb instruction set is a subset of the most commonly used32-bit ARM instructions. Thumb instructions operate with the standard ARM register configurations ,enabling excellent interoperability between ARM and Thumb states. This Thumb state is nearly 65% of the ARM code and can provide 160%of the performance of ARM code when working ona 16-bit memory system. This Thumb mode is used in embedded systems where memory resources are limited. The Jazelle mode is used in ARM9 processor to work with 8-bit Java code.

ARCHITECTURE OF ARM PROCESSORS:

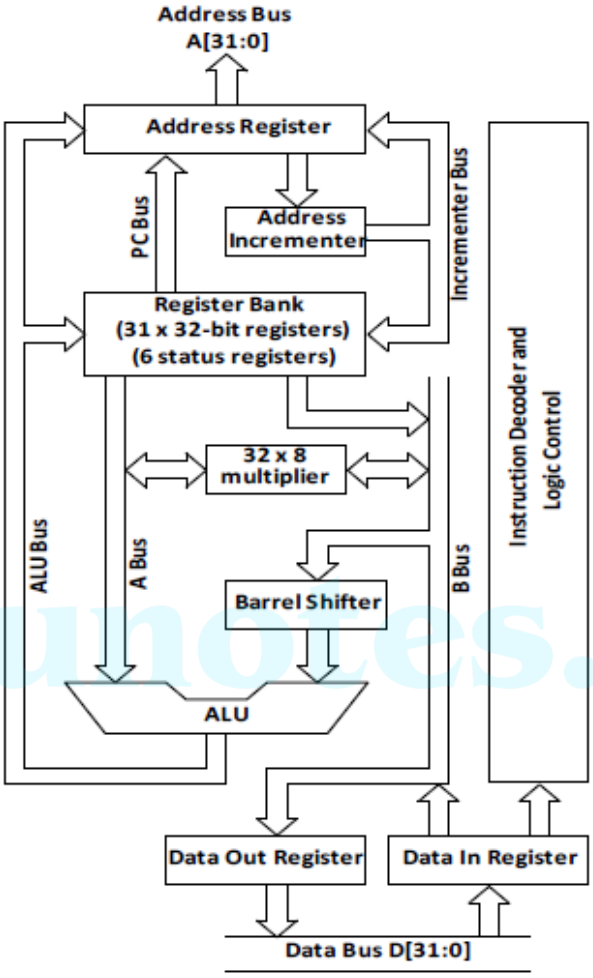
The ARM 7 processor is based on Von Neman model with a single bus for both data and instructions..(The ARM9 uses Harvard model).Though this will decrease the performance of ARM, it is overcome by the pipe line concept. ARM uses the Advanced Microcontroller Bus Architecture (AMBA) bus architecture. This AMBA include two system buses: the AMBA High-Speed Bus (AHB) or the Advanced System Bus (ASB), and the Advanced Peripheral Bus (APB).

The ARM processor consists of

- Arithmetic Logic Unit (32-bit)
- One Booth multiplier(32-bit)
- One Barrel shifter

- One Control unit
- Register file of 37 registers each of 32 bits.

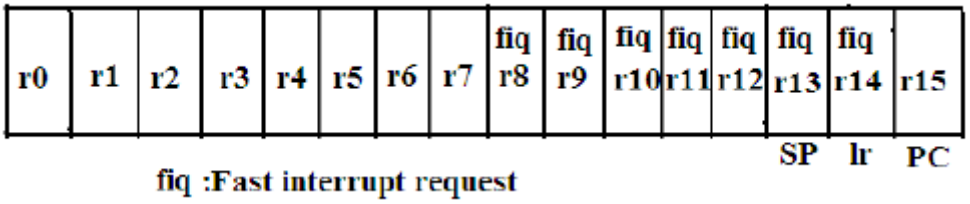
In addition to this the ARM also consists of a **Program status register** of 32 bits, Some special registers like the **instruction register**, memory data read and write register and memory address register ,one **Priority encoder** which is used in the multiple load and store instruction to indicate which register in the register file to be loaded or stored and Multiplexers etc.



ARM Registers :

ARM has a total of 37 registers .In which - 31 are general-purpose registers of 32-bits, and six status registers .But all these registers are not seen at once. The processor state and operating mode decide which registers are available to the programmer. At any time, among the 31 general purpose registers only 16 registers are available to the user. The remaining 15 registers are used to speed up exception processing. there are two program status registers: CPSR and SPSR (the current and saved program status registers, respectively .In ARM state the registers r0 to r13 are orthogonal—any instruction that you can apply to r0 you can equally well apply to any of the other registers.

The main bank of 16 registers is used by all unprivileged code. These are the User mode registers. User mode is different from all other modes as it is unprivileged. In addition to this register bank ,there is also one 32-bit Current Program status Register(CPSR)



fiq :Fast interrupt request

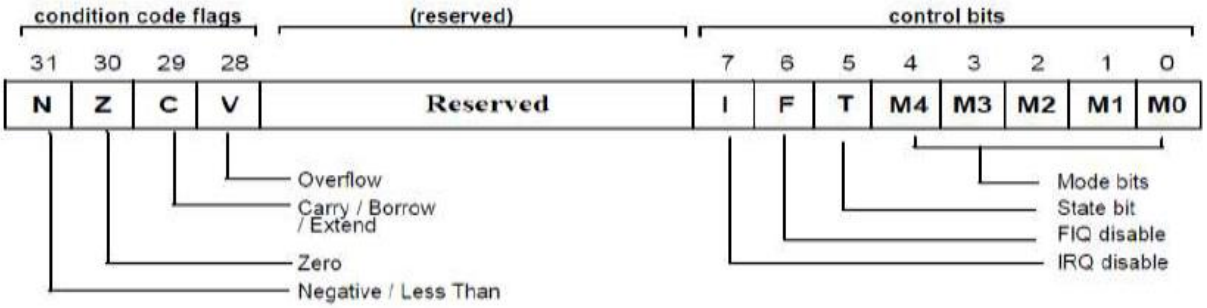
In the 15 registers ,the r13 acts as a stack pointer register and r14 acts as a link register and r15 acts as a program counter register. Register r13 is the sp register ,and it is used to store the address of the stack top. R13 is used by the PUSH and POP instructions in T variants, and by the SRS and RFE instructions from ARMv6.

Register 14 is the Link Register (LR). This register holds the address of the next instruction after a Branch and Link (BL or BLX) instruction, which is the instruction used to make a subroutine call. It is also used for return address information on entry to exception modes. At all other times, R14 can be used as a general-purpose register.

Register 15 is the Program Counter (PC). It can be used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed.

The remaining 13 registers have no special hardware purpose.

CPSR : The ARM core uses the CPSR register to monitor and control internal operations. The CPSR is a dedicated 32-bit register and resides in the register file. The CPSR is divided into four fields, each of 8 bits wide : flags, status, extension, and control. The extension and status fields are reserved for future use. The control field contains the processor mode, state, and interrupt mask bits. The flags field contains the condition flags. The 32-bit CPSR register is shown below.



Processor Modes: There are seven processor modes .Six privileged modes abort, fast interrupt request, interrupt request, supervisor, system, and undefined and one non-privileged mode called user mode.

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

Banked Registers : Out of the 32 registers , 20 registers are hidden from a program at different times. These registers are called banked registers and are identified by the shading in the diagram. They are available only when the processor is in a particular mode; for example, abort mode has banked registers r13_abt , r14_abt and spsr _abt. Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or _mode.

When the T bit is 1, then the processor is in Thumb state. To change states the core executes a specialized branch instruction and when T= 0 the processor is in ARM state and executes ARM instructions. There are two interrupt request levels available on the ARM processor core— interrupt request (IRQ) and fast interrupt request (FIQ).

V, C , Z , N are the Condition flags .

V (oVerflow) : Set if the result causes a signed overflow

C (Carry) : Is set when the result causes an unsigned carry

Z (Zero) : This bit is set when the result after an arithmetic operation is zero, frequently used to indicate equality

N (Negative) : This bit is set when the bit 31 of the result is a binary 1.

THE ARM PROGRAMMER'S MODEL

A processor's instruction set defines the operations that the programmer can use to change the state of the system incorporating the processor. This state usually comprises the values of the data items in the processor's visible registers and the system's memory. Each instruction can be viewed as performing a defined transformation from the state before the instruction is executed to the state after it has completed. Note that although a processor will typically have many invisible registers involved in executing an instruction, the values of these registers before and after the instruction is executed are not significant; only the values in the visible registers have any significance. The visible registers in an ARM processor are shown in Figure

When writing user-level programs, only the 15 general-purpose 32-bit registers (r0 to r14), the program counter (r15) and the current program status register (CPSR) need be considered. The remaining registers are used only for system-level programming and for handling exceptions (for example, interrupts).

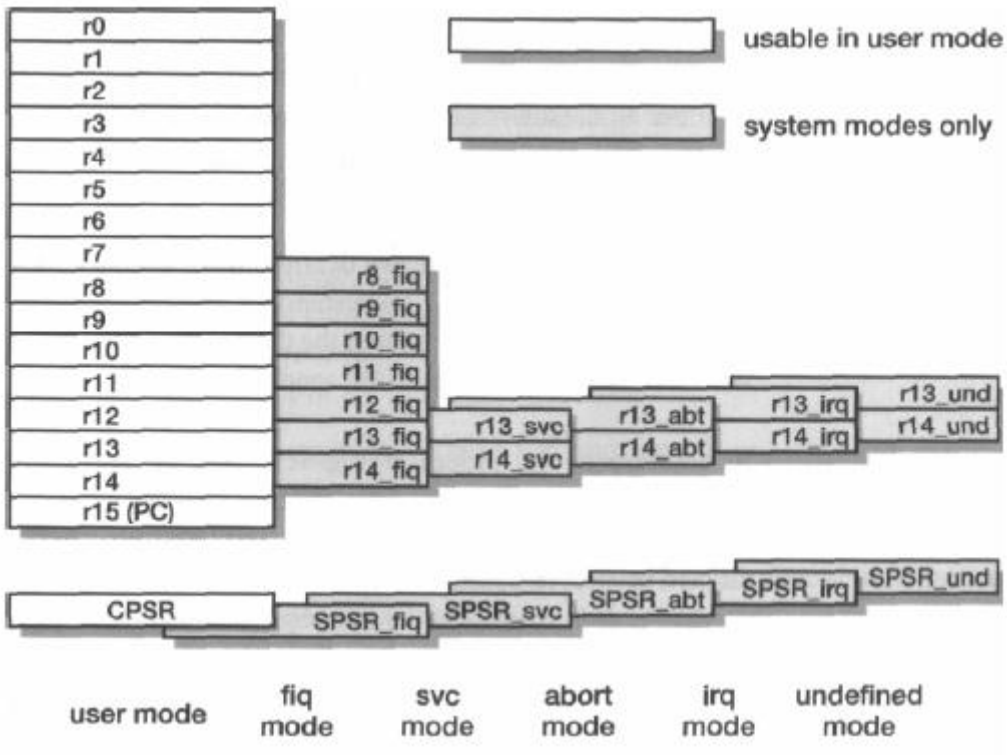


Figure : ARM's visible registers.

The Current Program Status Register (CPSR)

The CPSR is used in user-level programs to store the condition code bits. These bits are used, for example, to record the result of a comparison operation and to control whether or not a conditional branch is taken. The user-level programmer need not usually be concerned with how this register is configured, but for completeness the register is illustrated in Figure .

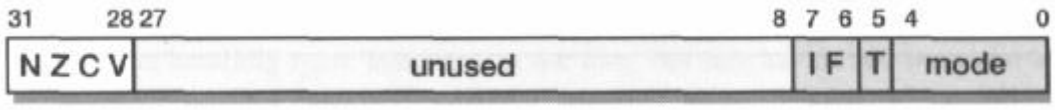


Figure: ARM CPSR format

The bits at the bottom of the register control the processor mode ,instruction set and interrupt enables ('I' and 'F') are protected from change by the user-level program. The condition code flags are in the top four bits of the register and have the following meanings:

- N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 32-bit result was a one).
- Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 32-bit result was zero).

- C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.
- V: oVerflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

The memory system

In addition to the processor register state, an ARM system has memory state. Memory may be viewed as a linear array of bytes numbered from zero up to 2^{32}-l. Data items may be 8-bit bytes, 16-bit half-words or 32-bit words. Words are always aligned on 4-byte boundaries (that is, the two least significant address bits are zero) and half-words are aligned on even byte boundaries.

The memory organization is illustrated in Figure . This shows a small area of memory where each byte location has a unique number. A byte may occupy any of these locations, and a few examples are shown in the figure. A word-sized data item must occupy a group of four byte locations starting at a byte address which is a multiple of four, and again the figure contains a couple of examples. Half-words occupy two byte locations starting at an even byte address.

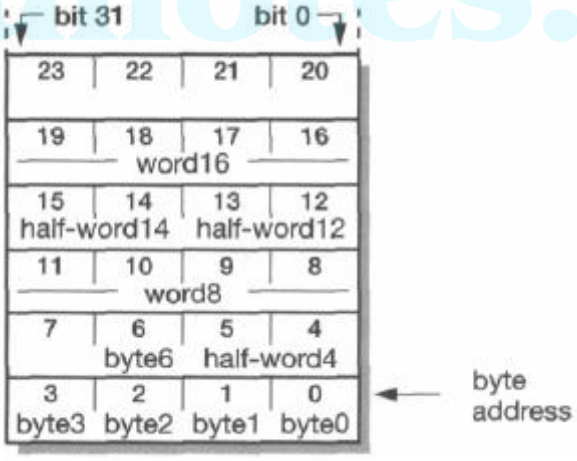


Figure: ARM memory organization

Load-store architecture

In common with most RISC processors, ARM employs a load-store architecture. This means that the instruction set will only process (add, subtract, and so on) values which are in registers (or specified directly within the instruction itself), and will always place the results of such processing into a register. The only operations which apply to memory state are

ones which copy memory values into registers(load instructions) or copy register values into memory (store instructions).

CISC processors typically allow a value from memory to be added to a value in a register, and sometimes allow a value in a register to be added to a value in memory. ARM does not support such 'memory-to-memory' operations. Therefore all ARM instructions fall into one of the following three categories:

1. Data processing instructions. These use and change only register values. For example, an instruction can add two registers and place the result in a register.
2. Data transfer instructions. These copy memory values into registers (load instructions) or copy register values into memory (store instructions). An addi tional form, useful only in systems code, exchanges a memory value with a register value.
3. Control flow instructions. Normal instruction execution uses instructions stored at consecutive memory addresses. Control flow instructions cause execution to switch to a different address, either permanently (branch instructions) or saving a return address to resume the original sequence (branch and link instructions) or trapping into system code (supervisor calls).

Supervisor mode

The ARM processor supports a protected supervisor mode. The protection mechanism ensures that user code cannot gain supervisor privileges without appropriate checks being carried out to ensure that the code is not attempting illegal operations.

The upshot of this for the user-level programmer is that system-level functions can only be accessed through specified supervisor calls. These functions generally include any accesses to hardware peripheral registers, and to widely used operations such as character input and output. User-level programmers are principally concerned with devising algorithms to operate on the data 'owned' by their programs, and rely on the operating system to handle all transactions with the world outside their programs.

The ARM instruction set

All ARM instructions are 32 bits wide (except the compressed 16-bit Thumb instructions) and are aligned on 4-byte boundaries in memory. The most notable features of the ARM instruction set are:

- The load-store architecture;
- 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified);
- Conditional execution of every instruction;
- The inclusion of very powerful load and store multiple register instructions;
- The ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;
- Open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model;
- A very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

The I/O system

The ARM handles I/O (input/output) peripherals (such as disk controllers, network interfaces, and so on) as memory-mapped devices with interrupt support. The internal registers in these devices appear as addressable locations within the ARM's memory map and may be read and written using the same (load-store) instructions as any other memory locations.

Peripherals may attract the processor's attention by making an interrupt request using either the normal interrupt *(IRQ)* or the fast interrupt *(FIQ)* input. Both interrupt inputs are level-sensitive and maskable. Normally most interrupt sources share the IRQ input, with just one or two time-critical sources connected to the higher-priority FIQ input. Some systems may include direct memory access (DMA) hardware external to the processor to handle high-bandwidth I/O traffic. Interrupts are a form *of exception* and are handled as outlined below.

ARM exceptions

The ARM architecture supports a range of interrupts, traps and supervisor calls, all grouped under the general heading of exceptions. The general way these are handled is the same in all cases:

- The current state is saved by copying the PC into *rl4_exc* and the CPSR into SPSR_exc (where *exc* stands for the exception type).
- The processor operating mode is changed to the appropriate exception mode.

- The PC is forced to a value between 0016 and 1C16, the particular value depending on the type of exception.

The instruction at the location the PC is forced to (the *vector address)* will usually contain a branch to the exception handler. The exception handler will use rl3_exc, which will normally have been initialized to point to a dedicated stack in memory, to save some user registers for use as work registers.

The return to the user program is achieved by restoring the user registers and then using an instruction to restore the PC and the CPSR atomically. This may involve some adjustment of the PC value saved in *rl4_exc* to compensate for the state of the pipeline when the exception arose.

SYSTEM SOFTWARE

❖ The ARM C compiler

The ARM C compiler is compliant with the ANSI (American National Standards Institute) standard for C and is supported by the appropriate library of standard functions. It uses the ARM Procedure Call Standard for all externally available functions. It can be told to produce assembly source output instead of ARM object format, so the code can be inspected, or even hand optimized, and then assembled subsequently. The compiler can also produce Thumb code.

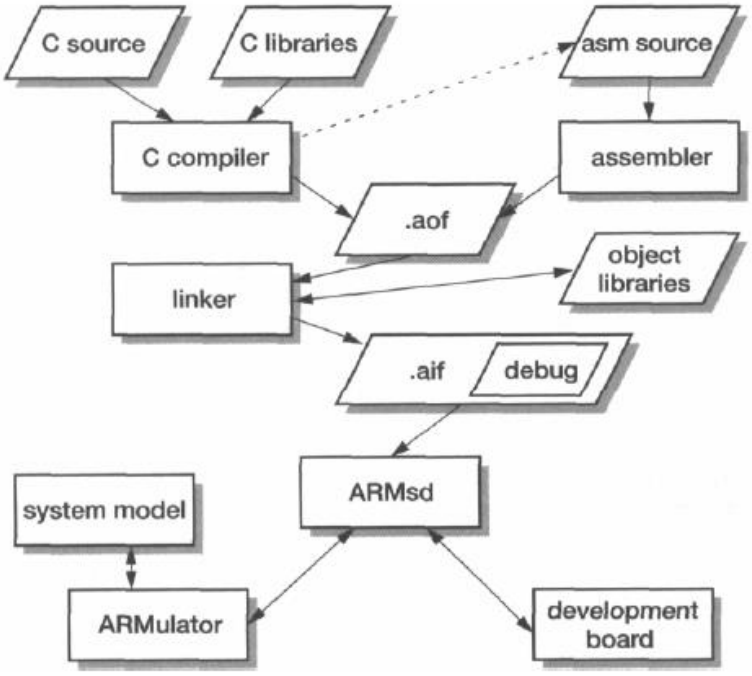


Figure : The structure of the ARM cross-development toolkit

❖ The ARM assembler

The ARM assembler is a full macro assembler which produces ARM object format output that can be linked with output from the C compiler. Assembly source language is near machine-level, with most assembly instructions translating into single ARM (or Thumb) instructions.

❖ The linker

The linker takes one or more object files and combines them into an executable program.

It resolves symbolic references between the object files and extracts object modules from libraries as needed by the program. It can assemble the various components of the program in a number of different ways, depending on whether the code is to run in RAM (Random Access Memory, which can be read and written) or ROM (Read Only Memory), whether overlays are required, and so on.

Normally the linker includes debug tables in the output file. If the object files were compiled with full debug information, this will include full symbolic debug tables (so the program can be debugged using the variable names in the source program). The linker can also produce object library modules that are not executable but are ready for efficient linking with object files in the future.

❖ ARMsd

The ARM symbolic debugger is a front-end interface to assist in debugging programs running either under emulation (on the ARMulator) or remotely on a target system such as the ARM development board. The remote system must support the appropriate remote debug protocols either via a serial line or through a JTAG test interface. Debugging a system where the processor core is embedded within an application-specific system chip is a complex issue.

At its most basic, ARMsd allows an executable program to be loaded into the ARMulator or a development board and run. It allows the setting of breakpoints, which are addresses in the code that, if executed, cause execution to halt so that the processor state can be examined. In the ARMulator, or when running on hardware with appropriate support, it also allows the setting of watchpoints. These are memory addresses that, if accessed as data addresses, cause execution to halt in a similar way.

At a more sophisticated level ARMsd supports full source level debugging, allowing the C programmer to debug a program using the source file to specify breakpoints and using variable names from the original program.

❖ ARMulator

The ARMulator *(ARM emulator)* is a suite of programs that models the behaviour of various ARM processor cores in software on a host system. It can operate at various levels of accuracy:

- *Instruction-accurate* modelling gives the exact behaviour of the system state without regard to the precise timing characteristics of the processor.
- *Cycle-accurate* modelling gives the exact behaviour of the processor on a cycleby-cycle basis, allowing the exact number of clock cycles that a program requires to be established.
- *Timing-accurate* modelling presents signals at the correct time within a cycle, allowing logic delays to be accounted for.

All these approaches run considerably slower than the real hardware, but the first incurs the smallest speed penalty and is best suited to software development.

At its simplest, the ARMulator allows an ARM program developed using the C compiler or assembler to be tested and debugged on a host machine with no ARM processor connected. It allows the number of clock cycles the program takes to execute to be measured exactly, so the performance of the target system can be evaluated. At its most complex, the ARMulator can be used as the centre of a complete, timing-accurate, C model of the target system, with full details of the cache and memory management functions added, running an operating system. In between these two extremes the ARMulator comes with a set of model prototyping modules including a rapid prototype memory model and coprocessor interfacing support. The ARMulator can also be used as the core of a timing-accurate ARM behavioural model in a hardware simulation environment based around a language such as VHDL. (VHDL is a standard, widely supported hardware description language.) A VHDL 'wrapper' must be generated to interface the ARMulator C code to the VHDL environment.