Embedded Systems

By
Chandra Sekhar G.
Assistant Professor
Electronics and Communication Engineering Dept.
Bharati Vidyapeeth's College of Engineering
New Delhi

UNIT - II

- ARM Processors:
  - Comparison of ARM architecture with PIC micro controller.
  - ARM 7 Data Path, Registers, Memory Organization, Instruction set.
  - Programming, Exception programming, Interrupt Handling, Thumb mode Architecture.
- Bus structure:
  - Time multiplexing, serial, parallel communication bus structure.
  - Bus arbitration, DMA, PCI, AMBA, I2C and SPI Buses.

Note: Prerequisites: Clear Conceptual knowledge in Microprocessors, Microcontrollers and their Functional Units.

This Notes is prepared mostly using NPTEL Videos of Embedded Systems, IIT Delhi. Some concepts are taken from Internet.

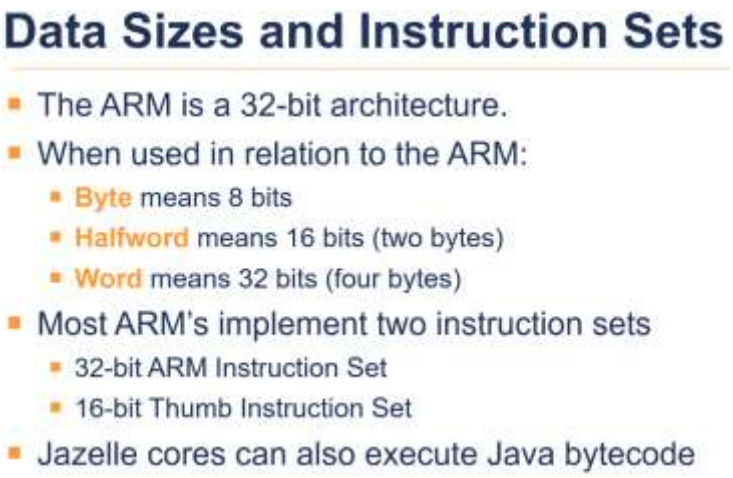
This notes is prepared to complete the syllabus in the given number of lectures and in the exam point of view. For more and detailed theory go through Mazidi (For overview of Embedded Systems) and Peatman (For PIC).

ARM Processor

In the first unit we studied PIC processors which were targeted primarily for low end applications. Now, we study ARM processor which are basically 32 bit processors and are meant for particularly high end applications, in fact, applications which involve more complex computations.

History of ARM processors: It was first developed at Acorn computer limited of Cambridge England between 1983 and 85. It is just after 1980 when the concept of RISC was introduced at Stanford and Berkley. Subsequently, ARM limited was formed in 1990 and what ARM popularized is a concept of ARM core, the processor core which they have licensed to a number of other manufactures to make variety of chips around that same processor core. So, what we shall be starting is it is just not a family of processors but conceptually a CPU architecture which may figure in a number of different chips intended for embedded applications.

*****Applications:** ARM processors are specifically used in High end portable devices like digital cameras, mobile phones, home networking modules and wireless communication technologies and other embedded systems projects due to the benefits, such as high code density, low power consumption, smaller silicon area, reasonable performance MIPS (several million instructions per second) etc.



***ARM Architecture:

The ARM architecture is based upon RISC architecture but it is not a purely RISC architecture because it has been enhanced to meet requirements of embedded applications. The requirements emerged for embedded applications because we need to have **high code density, low power consumption as well as low and smaller silicon footprint**. Architecturally we will find that it satisfies various conditions and properties of RISC processors as well. **It has got a large uniform register file and it basically is a load-store architecture where data processing operations are only with registers and does not really involve memory locations.** It is **basically 32 bit processor**, but we will find the variants of that as well. In fact we have got ARM processors which can support both 32 bit as well as 16 bit operations. So, **there is a 16 bit variant embedded into a 32 bit processor**. Why? We shall come to that point when we discuss **thumb extensions of ARM architecture**. It has got good speed versus power consumption ratio and high code density as needed for embedded systems. But where are its real exceptions or departure from classical RISC architecture. In this case, we have got variety of interesting add-on features in ARM. ARM Processors have got a barrel shifter in the data path which can maximize the usage of hardware available on chip. Then there is an auto increment and auto decrement addressing modes to optimize program loops. This is not very common with RISC processors. Also it supports load and store of multiple data elements through a single instruction and subsequently we will find that there is

an interesting set of branch instructions. In these branch instructions, they are not just individual branch instructions but this branching can be used in conjunction with other operations. So, we have got a large variety of branching possibilities and these has also been used to maximize the execution throughput.

In ARM we shall find various enhancements which have been made into it to make it more suited for embedded applications. And these actually distinguish ARM from other typical RISC processors. There are various variations of this architecture, in fact the whole idea is that this architecture has undergone a number of versions and these versions have been implemented into number of processors having distinct identities and numbers. So, here we have listed four such versions, version 1, version 2, version 3, version 4 we will find that initially the processor was a 26 bit addressing processor, it had no multiply or provision for core processors. The version 2 really made a 32 bit addressing mode. Version 3 really made it into a 32 bit addressing register capable and version 2 included multiply as an option. Now, this is a definite departure from PIC.

PIC, like other simple micro-controllers simply support add and subtract operations. Here, in this instructions set itself I have got multiply. In fact we shall see also variations of multiply in the instruction set of all. In fact version 4 was characterized by enhanced set of instruction and in version 4 itself they introduced what is called the thump mode and this version is called version 4T which included a 16 bit mode. Now, this 16 bit mode obviously implies that given the cell memory if I can use 16 bit instructions I can pack additional instructions. So, I can use this effectively same memory for more instructions and thereby I can increase code density. And this is particularly useful when we really do not require 32 bit operations. And this embedding of a 16 bit variant inside the 32 bit architecture came in version 4T is for thumb. Then we have got version 5T which is superset of 4T adding new instructions and the version 5TE add its signal processing instructions. In fact the instruction which are targeted for signal processing applications into the basic set. In fact this is again are departure from classical RISC concept.

Now, these architectures have been implemented into a variety of chips. So, there are some examples, ARM6 actually implements version 3. ARM7 which is a very popular ARM version implements version 3 but another variant of this ARM7 which is ARM7TDMI, this actually implements version 4T, that means it has got a thump embedded into the basic processor. The StrongARM is version 4 implementation, but this implementation is from Intel and not from ARM implemented. In fact it has been the point I was making that ARM core that has been licensed to a number of manufactures and they have taken the basic core, done the appropriate modifications and designed processors for various targeted applications. The example is StrongARM here, which Intel head made around the ARM core targeted for again embedded applications. ARM 9E-S this example is an implementation of the version 5TE that is extension architecture which implements digital signal processing instructions set and it is thought primarily targeted for embedded system which need to process lots of digital signals like speech, video etc.

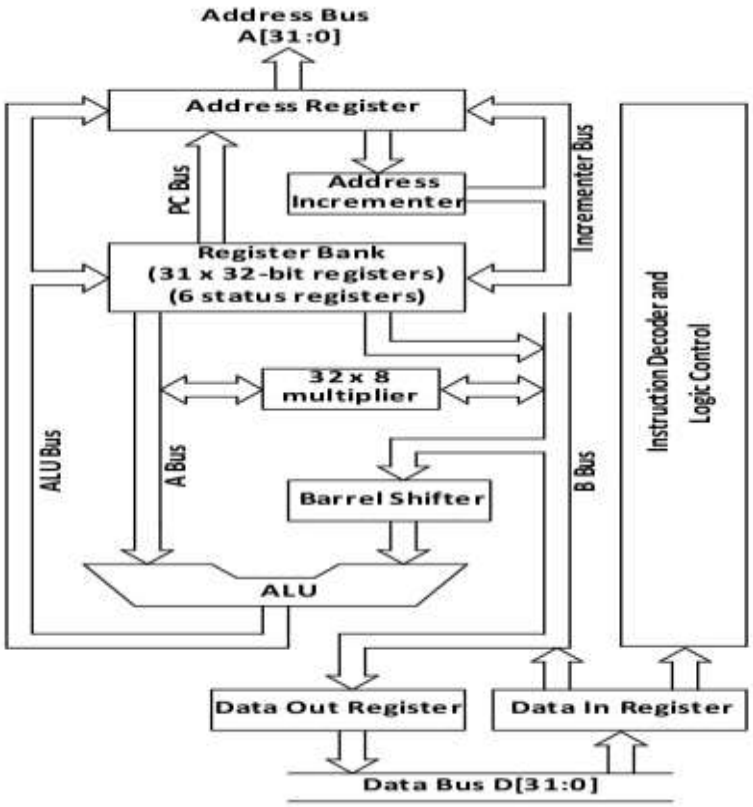
Now we shall look into the basic architecture, in fact we have seen that once we are talking about so many variations of architecture, we will find that the ARM architecture is not synonymous with the single organization. But there are certain commonality across the different variants. We shall look at this common feature before we go in to the variants in detail. First thing is any architecture is characterized by its data path as well as by the control path. Our current focus will be on data path and we shall try to understand the instruction set of ARM with reference to this core data path. This data path is organized in such a way that the operands are not directly fetched from memory because that is the basic feature of a RISC. So the operands are typically stored in the register file and instruction typically use 2 source registers and single result or destination registers wherever required in the operations involving 3 registers. If the operation involves just 2 registers then obviously the third register will not be there. But the more interesting thing is last two points, there is a barrel shifter on the data path which can preprocess data before it enters ALU.

***What is the Barrel shifter? Barrel shifter is basically a combinational circuit which can shift the data bits to the left or to the right by arbitrary number of positions in the same cycle itself. In a classical shift registers involving flip flops the number of shifts require the equivalent number of clock cycles because a shifting takes place on the basis of clock. But here it is combinational circuits so that the shift takes place at one go itself. In fact the shift takes place in the same instruction cycle. This is very basic enhancement in the ARM data path.

The other interesting feature is the increment and decrement logic which can operate on the registers independent of the ALU. So, we have got a facility for implementation of what we call auto increment and auto decrement addressing modes.

In fact this is not uncommon with CISC processors but this has been brought in ARM to facilitate certain operations that is particularly block move kind of operations and that way it is an enhancement on the key RISC model.

***ARM Core Datapath: So let us look at the basic ARM organization. Now what is interesting here is that there is a register bank which is connected to the ALU where two data parts one is A bus, another data part is B bus and this B bus goes via the Barrel shifter. So, this Barrel shifter can actually preprocess the data which can come from one of this source registers; and the Barrel shifter can shift to the left, shift to the right or even rotate the data before it is fed to the ALU. Basically, ALU & Barrel shifter are combinational circuits. So, all the operations that is operation that ALU carries out as well as operation that Barrel shifter carries out can take place in one cycle itself and that actually splits up the operation execution speed.



We can use the register bank for generation of the address as well. In fact the PC also is part of the register bank and that can generate the address. As well as the other register banks can be made useful for generation & for manipulation of address. Because registers are in a way symmetric they can have both address as well as the data. The PC generates the address for the instruction. Now here, we have not indicated whether it would be a Harvard or VanNeuman architecture typically. So, what we have concentrating primarily here, the data path. So, what we will find in this case, the incremental block, this incremental block enables us to decrement or increment the register values independent of the ALU. In fact the PC value can be incremented and put back to the registers so on and so forth. Other operations can also be done with the registers using the incremental block. There is an instruction decode & control to decodes the instruction and generate the control signals for the operation. There is an address bus is 0 to 31 that means it is a 32 bit. Data buses are also 32 bits, so it is basically a 32 bit processor. It can operate on 32 bit operands and the addresses that it generates are also 32 bit. In fact

the one of the very interesting feature that we should note, not only an interesting, a very basic concept here is, since the registers can handle data and address in a symmetric fashion it is very easy to handle same number of bits for address and data and use the similar kind of operations for manipulating addresses and data in the registers. Now, let us look at the register bank because we are found that in the data path this register bank has a very prominent role.

All registers have to be 32 bits because my data bus is 32 bit; I am operating at 32 bit operands as well as my addresses are also 32 bits. **In user mode there are 16 data registers and 2 status registers which are visible**. What it implies, that there could be some invisible registers as well and we shall look at this mystery slightly later on.

User mode is a common operating mode that means when we will be running your program on ARM; typically we will be operating in user mode. Data registers are typically r0 to r15 and in fact in ARM, all registers are refer to by r followed by a number. So, here we are talking about **data registers r0 to r15 which are visible in the user mode**. Out of these registers, **3 registers perform special function they are r13, r14 and r15. R13 is a stack pointer**, so this stack pointer refers to the entry point on the stack and this is critical for implementation of a stack in the memory. **R14 is a link register**, this is also found in variety of other processor as well. This link register is a register where return address is put whenever a subroutine is called. So, whenever a subroutine is called the return address typically we will expect the model wise, we have talked about earlier in the contest of PIC; the return address goes into stack. In PIC, it was a hardware stack different from the program or data area. Here, we have got a single link register and in the link register the return address is put in. Then **r15 is the program counter** and obviously the current instruction what is being executed will be pointed to by the content of r15.

Program Counter (r15)

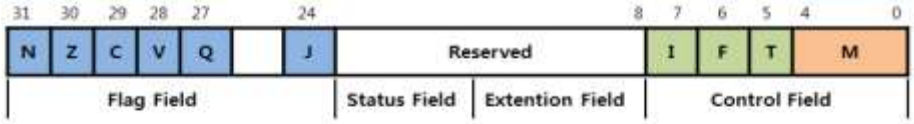
- **When the processor is executing in ARM state:**
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the pc value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned)
- **When the processor is executing in Thumb state:**
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the pc value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned)
- **When the processor is executing in Jazelle state:**
  - All instructions are 8 bits wide
  - Processor performs a word access to read 4 instructions at once

Now, depending on the context registers r13 and r14 can also be used as general purpose registers, although this is not a very common usage because we understood that r13 and r14 has got a special role to play. Any instruction which uses r0 can also use to any other GPR and in addition there are 2 status registers. **CPSR, current program status register** and **SPSR what is called saved program status register**. These are basically the status registers which are not data registers. So, here in this registers effectively the status of the current execution is being captured. In fact the status can include status of the program as well as that of the processor. The r15, the register is basically the PC of 32 bit wide that will have the address of the instructions.

And when ARM is operating in your 32 bit mode, all instructions will be 32 bit and the addresses are also 32 bit and what we assume that all instructions are word aligned. That means all 32 bit instructions start at 32 bit boundary; this is very important. And that implies that PC value is stored effectively in bits from 2 to 31, with bits 1, 0 effectively undefined or not really useful for referring to an instruction. Now, obviously this discussion refers to one fact that my 32 bit address in ARM refers to byte locations. Each byte which associated with a unique address so, if I am talking about 32 bit boundaries that means effectively I am talking about blocks of 4

bytes. So, if I have one instruction starting at location 0 then that instruction will occupy address location 0, 1, 2 as well as 3. The next instruction would be located at 4 so, therefore the 2 bits [1:0], the least significant bits of PC that is r15 do not care for operations. So, that is why we say that PC value is effectively stored in bits from 2 to 31.

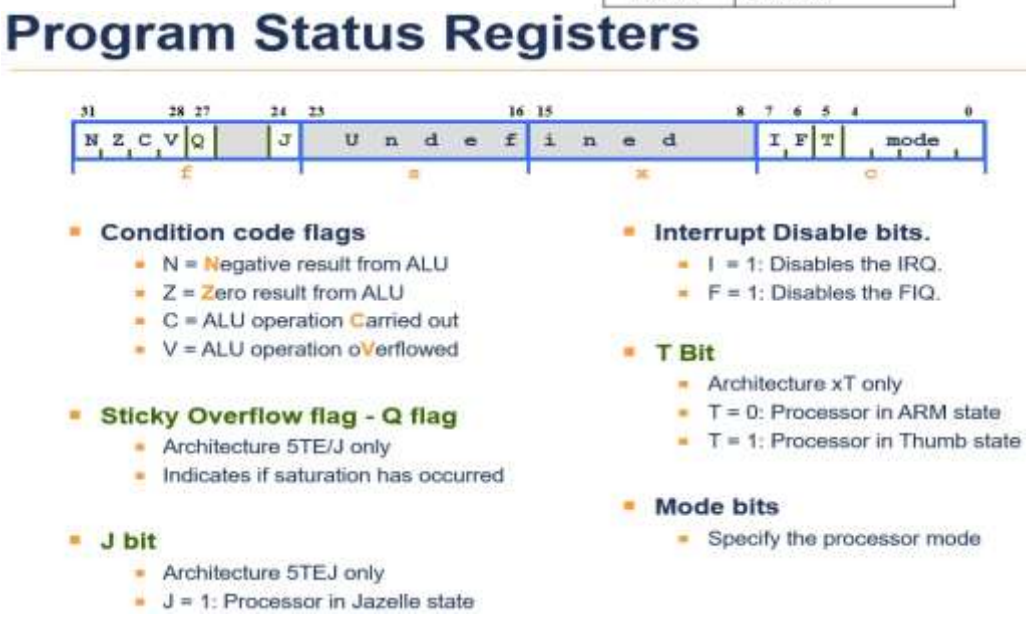
*****CPSR - Current Program Status Register:** It has got a number of bits. Again it will be a 32 bit register; it is not that all bits are used at the same time. **The condition code flags which occupy the higher MSBs** that is most significant bits in the status register; they are standard flags which reflect various arithmetic conditions. I have got negative flag if there is a negative result from ALU which is typically the most significant bit. If it is one then it can be interpreted as a negative result when we are doing signed arithmetic set, Z indicates zero, C is the carry and V is overflow. There is this sticky overflow flag, this is with reference to saturation arithmetic and there is interrupt disable bits, there are two levels of interrupts. We shall come to that point later on. So, we can enable or disable these two levels of interrupts by using the 2 bits. The T bit indicates whether we are in thump mode or not thumb mode because when we actually have an embedded 16 bit processor into the 32 bit architecture, we shall be making use of this T bit to know whether I am operating in the thumb mode or ordinary 32 bit mode. And rest are mode bits and these mode bits to define what is called the mode of processors operation. We can use 16 data registers in your program in normal operation and that is user mode, that mode specified in these bits. Now, before going into these processor modes in detail, let us briefly look at this sticky overflow flag. What is saturation? Saturation means when we reach the maximum value or the minimum value because of an arithmetic operation which may have overflow or underflow. For example, I have got the maximum value, it is all ones and if I add 1 to it then actually 32 bit content would become 0.



Flag Field	
N	Negative result from ALU
Z	Zero result from ALU
C	ALU operation caused Carry
V	ALU operation oVerflowed
Q	ALU operation saturated
J	Java Byte Code Execution

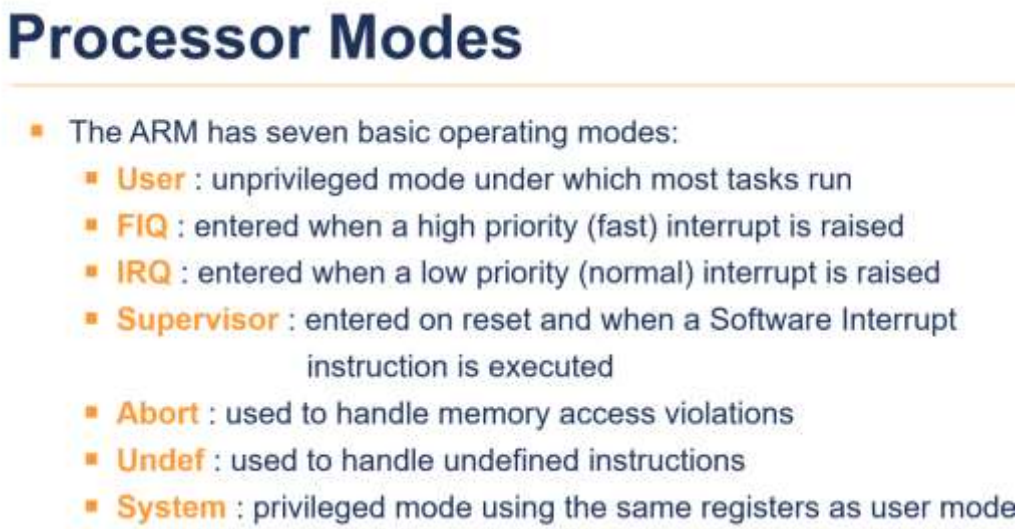
Control bits	
I	1: disables IRQ
F	1: disables FIQ
T	1: Thumb, 0: ARM

Mode bits M[4:0]	
0b10000	User
0b11111	System
0b10001	FIQ
0b10010	IRQ
0b10011	SVC(Supervisor)
0b10111	Abort
0b11011	Undefined



Now, I may not like the 32 bit, 32 bit value to becomes 0,I may like it to stick to all one's. That is a maximum value be retained. Now, that is enabled in saturation arithmetic and that is required for a number of signal processing tasks. In fact the architecture 5TE has got signal processing extension and that is why this Q flag, the saturation arithmetic flag becomes significant or important or relevant with respective to that architecture.

*****Processor Modes:** The processors modes are there to define various kinds of registers; their visibility as well as rights to modify CPSR register. So, we call the processor modes either privileged or non-privileged mode. In a privileged mode we expect to have full read-write access to the CPSR. In a non-privileged mode only read access to the control field of CPSR but read-write access to the condition flags. Now, try to understand what is the implication of these privileged and non-privileged modes? In a privileged mode as we can change the CPPSR control bits, which means we can have a full read as well as write access of the control bits. We can actually change the processor mode, we can enable, disable the interrupts. So, this is a privileged operation. In a non- privileged mode, CPSR control fields can be simply read but cannot be changed, but the condition flags can be changed because the condition flags would normally reflect the status of the arithmetic operation and that should remain write enable even in non-privileged modes. So, typically we will find that when we talk about these kind of operations, a typically user program is expected to run in a non- privileged mode because user program is normally not expected to change the control bits.



And in a privileged mode typically we will expect the OS or the supervisory cell to run. Since we are targeting for ARM for more sophisticated applications, typically there would be an OS running in an ARM based system under which user programs are expected to execute. The OS is typically expected to be running in privileged mode and user applications running in non- privileged mode. In fact ARM has got 7 modes and these 7 modes can be now classified as privileged and non- privileged. In fact the privileged modes are abort, fast interrupt request, normal interrupt request, supervisor, system and undefined. Non- privileged mode is user mode and it is used for programs and applications.

Now, privileged modes represent different scenarios. **Abort** is a mode when there is a failed attempt to access memory. This can happen for variety of reasons but the reasons we shall look at when we consider the memory architecture subsequently. But this is a particular mode in which the processor goes in, when it detects that there is a failure to access the memory location. The **fast interrupt request** and **interrupt request** correspond to interrupt levels available on ARM. So, when a particular kind of interrupt occurs ARM processor goes into either fast interrupt mode or interrupt request mode. **Supervisor mode** is a state in which **processor** goes in after **reset** and generally it is a mode in which the OS kernel is supposed to operate because obviously when the processor is reset, the first thing that it is excepted to execute is an operating system code and not user application of program. So, this is a supervisor mode in which the processor goes in when the reset happens. The other two privileged modes are system mode and another mode is called undefined. In a **system mode**, is a **special version**

of user mode that allows full read-write access of CPSR. And it is also targeted for supervisory applications; many of the OS routines can be configured to run in the system mode. The **undefined mode**, **processer enters this undefined mode when it encounters an undefined instruction** that means when we are trying to use an illegal op-code for undefined instruction, the instruction undefined for particular processor, then it goes into an undefined.

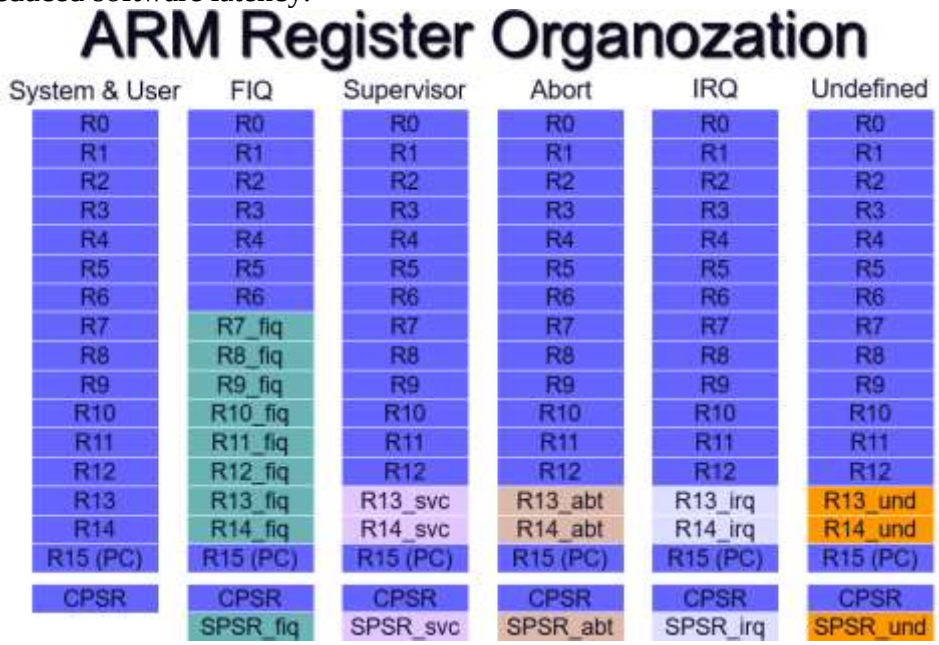
So, what we have found is that these privileged modes are primarily targeted for OS handling of special error conditions as well as that of interrupts and user mode is a mode intended for running user applications. Now, these modes have got associated with them a very interesting capability to manage the registers.
Now, we go into the concept of what we call banked register in ARM architecture. ARM has got 37 registers in all and typically 20 registers are hidden from program at different times.

So, they are not visible registers and they are actually called banked registers and this banked registers becomes available only when processors is in a particular mode. In fact processors modes other than system mode have a set of associated banked registers that are subset of the 16 registers in the user mode. And the banked registers have one-to-one mapping with the user mode registers. So, what happens, let us look at this, I am operating, let us say in a user mode, in the user mode there are the 16 data registers which are available and the current program which is getting executed, that status would get reflected in the CPSR register. Now, if the processor goes into some other mode, let us say that FIQ mode. FIQ is what- fast interrupt; IRQ is interrupt request model. Now, in an FIQ mode, what we will find that I have got banked register r8, r9, r10, r11, r12 becoming available as well as r13 and r14. These registers got a one-to-one correspondence with the registers in the user mode that means effectively I am getting a fresh copy of r8 to r14 in a fast interrupt mode.

It implies that if I am having an interrupt service routine which is operating in FIQ, it can use r8, r9 to r14 without bothering about what happens to the original content of these registers. When we go to the interrupt service routine and if I want to come back to the original program from there and to come back to the correct state of computation, I need to store the registers in the stack. Now, storing the registers in the stack is consisting of push operation that will have the overhead time and that is actually the software latency for interrupt processing. In the previous topics, we have looked at hardware latency, this becomes the software latency. The moment I have got a fresh copy of the registers; if my ISR do not use any other register, I really do not need to push these registers onto stack. So, I can minimize my software latency and in fact that is a reason why this mode is called fast interrupt mode. In other one, the interrupt request mode what we have is simply r13 and r14 fresh copy is generated. So, other registers the copy is not generated. So, other registers, we need to save if we are using them; so obviously it is not as fast as that of a fast interrupt mode because here there will be some amount of software latency which will be involved. The similar thing is true for the other modes- the supervisory mode, undefined, abort. All these modes have got a copy of r13, r14 but other register the fresh copy is not

generated. So, if these original user mode registers have to be used; they are to be retained, they have to be safe in memory. Corresponding to CPSR, in privileged modes we have what are called SPSR. So, what is the **SPSR-Saved Program Status Register**. So, CPSR is copied into SPSR which becomes available in FIQ mode. So, when I return from this mode to say user mode, I have to take this content of SPSR back to CPSR because CPSR would be storing the current status; so when I am going back I should have the current status of the computation back with me. So, pictorially what we say here about SPSR is each privileged mode except system mode has associated with it a save program status register or SPSR. This SPSR is used to save the status of CPSR when the privileged mode is entered in order that the user state can be fully restored when the user process is resumed; this is particularly the job of SPSR. So, now if we summarize, in the user mode I have got 16 registers as well as CPSR. In FIQ mode what I have got, I have got same r0 to r7 registers as that of user mode but a fresh copy of these registers r8 – r14 and SPSR. Similar thing in IRQ; IRQ have got I can use the registers same as that of r0 to r12, fresh copy of r13 and r14 and CPSR is copied into SPSR.

But, obviously in FIQ mode I got to have a CPSR which will reflect the state of the processors in that mode during execution of that program. So, effectively what we are telling is that in these modes if we are writing onto these registers I shall overwrite the original data. If I have to get this data back, I need to save it in the stack. But in case of FIQ, I get a fresh copy of registers; so I need not save this register on to the stack and thereby we have reduced software latency.



The mode changes can be made by directly writing onto CPSR or by hardware when the processor responds to exception or interrupt. And to return to user mode, a special return instruction is used that instructs the core to restore the original CPSR and banked registers.

***Exception Programming:

Now, we shall look at other features of the ARM architecture. In particular exception processing and the other components how they go into the core and also, the internal organizational details of the processor core.

So, what I am showing here is the exceptions and the modes, in which the ARM processor is expected to be in when such an exception occur. The interesting feature that corresponding to exception there is a mode switch. That the mode in which processor works changes, when an exception occurs. So, along with mode change the core saves CPSR to the SPSR of the exception mode. There is save PC to the link register of the exception mode. Sets the CPSR of the exception mode and then, set PC to the address of the exception handler. In fact, all of this exceptions are associate with the vector. That is a memory address at which the exception handler is expected to be located. But, it is strictly not always that way. Because, the vector table which is the table of addresses, where ARM code branches when exception occurs, may not have space and really it does not have space to put in the complete service routine in that table itself. So, in the table we will actually have a branch instruction. It can be explicitly of branch instruction or it can be any other instruction which modifies PC.

Exception Entry

- Force PC to relevant vector address

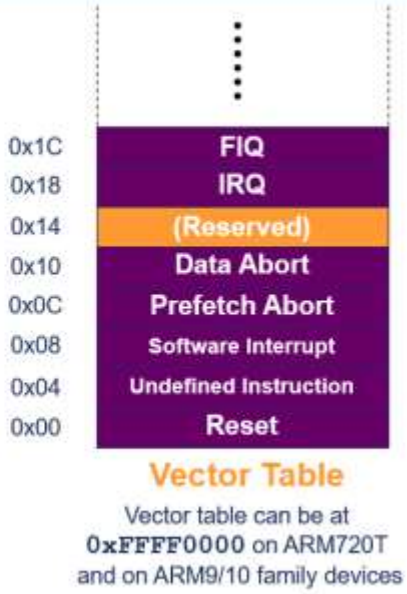
Priority	Exception	Mode	vector address
1	Reset	SVC	0x00000000
2	Data abort (data access memory fault)	Abort	0x00000010
3	FIQ (fast interrupt)	FIQ	0x0000001C
4	IRQ (normal interrupt)	IRQ	0x00000018
5	Prefetch abort (instruction fetch memory fault)	Abort	0c0000000C
6	Undefined instruction	UND	0x00000004
	Software interrupt (SWI)	SVC	0x00000008

- ❑ Normally the vector address contains a branch to the relevant routine
- ❑ Exception handler use `r13_<mode>` and `r14_<mode>` to hold the stack point and return address

Exception Handling

- When an exception occurs, the ARM:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

This can only be done in ARM state.



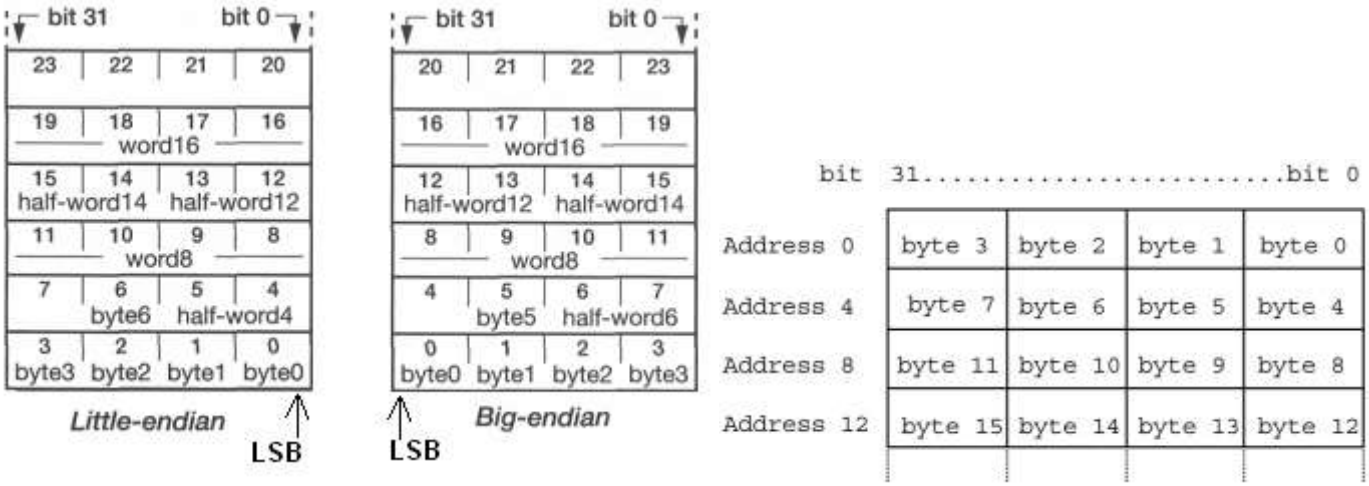
*****ARM memory organization**: ARM can be configured either in little endian form or in big endian form. PC is incremented by 4 to fetch the new instruction. There are two ways to store words(32bit) in a byte(8bit)-addressed memory: ***Little-endian*** and ***Big-endian***.

• Most ARM chips can be configured to work with either memory arrangement, though they default to little-endian. ARM is capable to store words in two ways depending on where the significant byte is stored.

• If word MSB is stored at highest byte then operation is called "little-endian"

• If MSB is stored at lowest position, then it is called " bigendian".

• Usually it is easier to work with little-endian for people as they expect to be LSB at lowest position and MSB at highest position. Endian is selected in compiler settings



ARM instruction set: We know now the data path of ARM; so we shall see through instructions how we can use the data path.

The instructions process data held in registers and access memory with load and store instructions. And that is typical of any RISC architecture and the classes of instructions are data processing, branch, load-store, software interrupt, program status register instructions because these are the different roles.

Instruction set Features:

•All instructions are 32bits long
• Most instructions execute in a single cycle.
• Every instruction can be conditionally executed.
• A load/store architecture
• Three operand format
• Combined ALU and shifter for high speed bit manipulation
• Instruction set extension via coprocessors

Data Processing Instructions

- Consist of :
  - Arithmetic: `ADD` `ADC` `SUB` `SBC` `RSB` `RSC`
  - Bit wise Logical: `AND` `ORR` `EOR` `BIC`
  - Comparisons: `CMP` `CMN` `TST` `TEQ`
  - Data movement: `MOV` `MVN`
- These instructions only work on registers, NOT memory.
  Syntax:
  `<Operation>{<cond>}{S} Rd, Rn, Operand2`
  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn
- Second operand is sent to the ALU via barrel shifter.

Now, we shall simply look at data processing instructions. Typically the ARM instruction set has got 3 address data processing instructions - the two operands and one for destination. The other interesting feature of the instruction set is that we can do a conditional execution of each instruction. That means depending on certain condition, we can decide whether to execute an instruction or not. In fact this is a very special kind of branch instructions and I have told we, this is an enhancement in the RISC architecture to increase the computation throughput. The Barrel shifter enables shift and ALU together; so we shall have that enhancement of instruction set to do these operations and the other interesting thing we shall look at subsequently that we can actually

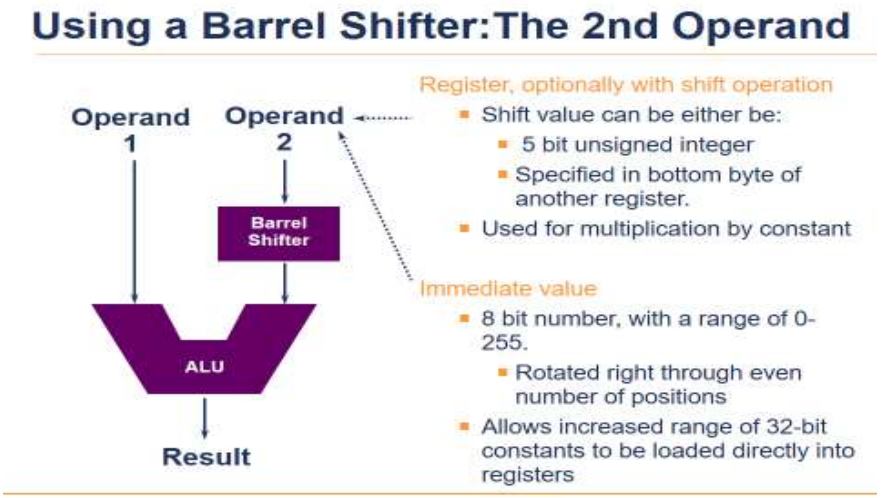
increase the instruction set of the processor by adding on co-processors. And that will be a very interesting feature for ARM architecture. Now, before we look into data processing instructions, we need to know what kind of data types are supported in ARM. Obviously ARM is the 32 bit processor.

ARM data types

- Word can be divided into four bytes (each 8 bit)
- Word is 32 bit long In ARM
- ARM processors support six data types:
  - 8-bit signed and unsigned bytes.
  - 16-bit signed and unsigned half-words; these are aligned on 2-byte boundaries.
  - 32-bit signed and unsigned half-words; these are aligned on 4-byte boundaries.

So, we will be having 32 bit word as a basic data type. But we can also look at each word as 8 bit, four 8 bit bytes and there can be operations on this bytes as well as 16 bit components as well. And we can configure the processor at power-up as little-endian or big-endian because that would define the definition of a value of a particular data whether it is being considered as a big-endian or small-endian. Now, data processing instructions obviously are concerned with manipulation of data within registers. We have MOVE instructions, arithmetic instructions; I have indicated multiply instructions because it implements multiplication and some variants of multiplication as well, then logical instructions, comparison instructions, this is what is standard in any instruction set of processors. And all these instructions can have a variant with a suffix S or not. When we add a suffix S to an instruction, it means that the condition flags will be set appropriately depending on the condition emerging out of the computation. If it is not, if we are not using a suffix, these condition flags will not be updated. These operands are typically 32 bit wide, can come from registers or specified as literal in the instruction itself.

That means I can have registers as well as immediate operands. The second operand can be sent via barrel shifter and 32 bit result is again placed in register only. In other cases that when we really do a 32 bit multiplication we generate 64 bit result which can go into multiple registers. So, **MOV Rd, N** is an example of a move instruction. So move will obviously involve two operands and **Rd** destination register and **N** can be immediate value or source register. A simple example is **mov r7, r5**; it means that I am moving content of r5 to r7. There is an interesting variant of move which is **MVN Rd, N;** the move negative. So, we move into Rd not of the 32bit value from source. So, this N can be a register, can also be an immediate value; so what we move in is not of that value into the destination. The MVN instruction takes the value of Operand2, performs a bitwise logical NOT operation on the value, and places the result into Rd.

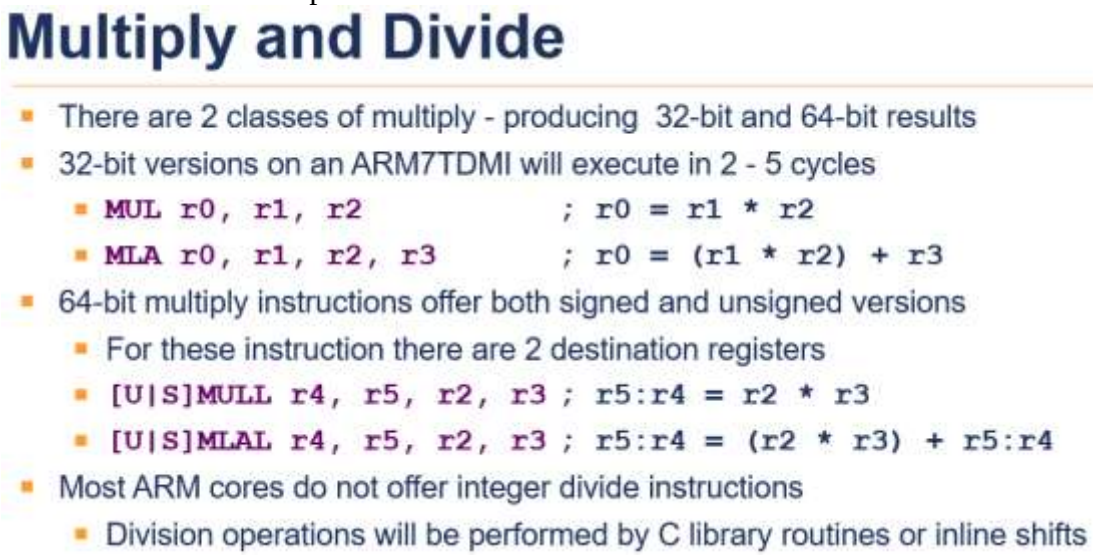


Now, we can use barrel shifter with the move instruction and this barrel shifter can shift left, shift right and rotate right by fixed number of bits. And what does it therefore help in, we can do a fast multiply and division and let us look at a simple example: **mov r7, r5, LSL #2**. Here, what I have done? I have specified that move

content of r5 to r7 but before that I am doing a logical shift left by 2 positions;i.e. shifting the 2nd operand, #2 indicates that immediate value 2. So, it multiplies the content of r5 by 4 and puts the result in r7. That means effectively what I have done, combined two operations into a single instruction and that is the basic role that barrel shifter plays. So, effectively this is the whole picture; again I am repeating, we have already seen that on the data path. So, I have got a barrel shifter and this operand 2 comes to barrel shifter, it can be an immediate value of the operand or a value specified in a register. And I would specify the number of positions that I need to shift as part of the instruction itself.

It can be an immediate value or it can be specified in a register as well. Similarly, we have arithmetic instructions, simple arithmetic instructions implement 32 bit addition and subtractions. Obviously, in this case I shall have 3 operands, 2 source set and destination.

Here, I have given some example of subtraction instructions, **SUB r0, r1, r2**. Here the value stored in r2 is subtracted from that of r1 and the result is stored in r0. In **SUBS r0, r1, #1**, this is an example of subtraction using an immediate operand and here I have used suffix S that means the result, that is the effect emerging out of this computation will be reflected in the bits of the CPSR. But that is not true essentially in this case of **SUB r0, r1, r2**, in this instruction. Now, we can use these operations with Barrel shifter; the moment we use them with Barrel shifter, then the possibilities also increases. So, look at this **ADD r0, r1, r1 LSL#1**- this is an example of an ADD instruction and I have added content of r1 and what I have written here I want to again do a shift, shift by one position. So if I shift by one position, it is effectively means what, it is multiplied by 2. And then I am adding to r1, adding to itself and putting the result back to r0, so effectively I am multiplying the content of r1 by 3 and putting the result in r0. So, these are the various possibilities which can be there. So, we can use the Barrel shifter and these operations with all of these arithmetic instructions.



Next, we have multiply instruction, in fact multiply as a block is implemented in ARM. And this multiply can be looked at in two forms- one is called long multiply; in case of long multiply we are expected to generate 64 bits results. Otherwise, the result is a 32 bit result. So, here we have given an example **MUL r0, r1, r2** where we are telling that we multiply r1 and r2 and put the result back to r0. U**MUL r0, r1, r2, r3 : t**his is a case of a 32 bit multiplication that is the result expected is 32 bit, so that it can be accommodated in r0. This is the case of a long multiplication and case of a long unsigned multiplication. So we multiply the results r2 and r3 and the result is stored in r0 and r1. The number of cycles taken for execution of multiply instruction depends upon processor implementation. So, that will be architecture dependent and we also have other than unsigned, we have got signed multiplication where the sign bit is taken care of while doing the multiplication. So, with sign, we get sign long multiplication; but along with multiplication there is another very interesting instruction which is multiply and accumulate- what is called MLA.

MLA Rd, Rm, Rs, Rn :if we look at here we actually multiply Rm into Rs, this is added with Rn, that is another register content and the register result is stored in Rd, the destination register **(Rd = (Rm*Rs) + Rn)**. And where

do we expect MLA operation, a very simple operation when we do convolution, we actually multiply and add and if I have to implement convolution which is very basic operation for a large number of signal processing task then MLA becomes a very useful instruction. And that is basically the motivation for implementing MLA as an instruction in the ARM processor itself.

The other variant of MLA is unsigned long MLA; **MLA Rdlo, Rdhi, Rm, Rs:** so here **Rdlo, Rdhi** 2 registers will contain the final result and in this case content of these registers is added with the product of **(Rm*Rs)**. And this product is expected to be a 64 bit product and so this 64 bit product is added with this 64 bit content of the registers and that result is stored **Rdlo, Rdhi**.

There are bit wise logical instructions AND, OR, EX-OR, bit clear. In fact bit clear is a very interesting instruction; I have given an example here BIC r0, r1, r2: in this case, r2 contains a binary pattern, for every binary one in r2 clears a corresponding bit location in register r1. So, that means I can specify a bit mask, I can specify a bit mask in r2, depending on that bit mask the bits of r1 will be cleared and the result will be stored in r0. So, this instruction is obviously useful in manipulating status flags as well as interrupt masks. And it is something like that in bit manipulation facility. Compare instructions are another set of instruction. So, I have got simple compare, TEQ for equality and TST

So, we will find that depending on the operation, I have given an example CMP r0, r9 which involving registers r0 and r9. So, depending on the instruction involved, in this case of a compare, a subtraction is done and depending on the result of the subtraction the flags are affected but the content of registers are not affected.
In case of TEQ, there is an ex-or operation, bit-wise ex-or operation between content of r0 and r9 and flags are accordingly changed as it gets reflected on the basis of the result, but the content of r0 and r9 is not changed. This is a similar thing and in this case, we use AND and not ex-or.

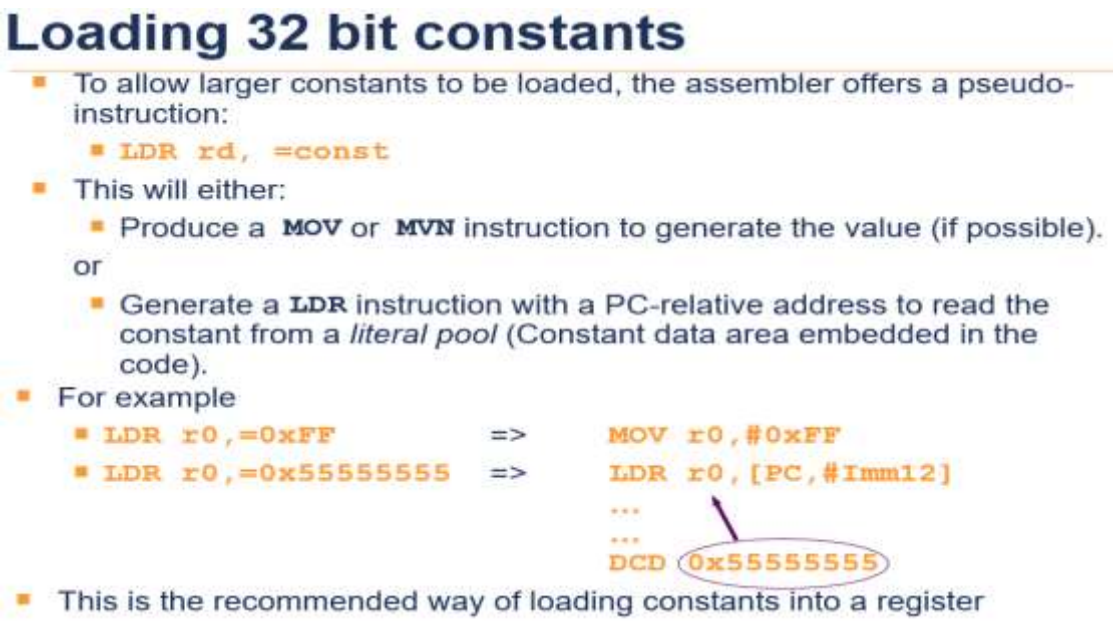
So, these are the typical comparisons instructions available in R. So, what we have seen today, we have studied very basics of ARM architecture; ARM architecture is much more complex then what we have looked at today. We have understood the different modes of operation of the processor. We have examined the data path in some detail and discussed the basic data processing instructions. We shall look at other instructions- branching instructions, then software interrupt instruction, the processors status register manipulation instruction and the other aspects of this architecture in subsequent lectures.

More ARM Instructions

Now, ARM is an example of RISC architecture. So, basically memory access is through load and store. And this load-store instructions are used for data transfer between memory and processor registers. **There are 3 basic types of load store instructions - single register transfer, multiple register transfer and swap**. In fact multiple register transfer in case of ARM is a significant departure from classical RISC model or RISC instructions sets. The single register transfer supports signed and unsigned 32 bit transfer, half - words (16bit) transfer as well as byte transfer.

Single transfer instructions: These load and stored instructions are particularly for transferring data at a boundary alignment. What do we mean by boundary alignment? It means that the data is expected to be aligned at the correct memory address. So, when we load or store a word, the address should be at the 32 bit boundary; when we load or store a half-word it should be at the 16 bit boundary. In fact these load store instructions support a variety of addressing modes. The simplest addressing mode is register indirect. Ex: **LDR r0, [r1]**: Here the memory address is specified in a register **r1**. Variation of this will be found in **LDR r0, [r1, #4]**: in this case where I have specified an immediate mode offset. This offset is added to the base register **r1** to get the memory address.

We can even have register operation specified, **LDR r0, [r1, -r2]** in this case I have specified a second register with a minus sign. It means that there would be an arithmetic operation performed between the content of r1 and r2 for obtaining the memory address. This is true for store instructions as well.
There are more addressing modes, one of them is scaled addressing mode. Now, I hope we remember that there is a barrel shifter in the data path of ARM. In **scaled addressing mode,** the barrel shifter is used for calculation of address. So, address is calculated using the base address register and a barrel shift operation Ex: **LDR r0, [r1, r2, LSL#2]**. The same set of operations that we had discussed in the context of data processing instructions are applicable here also.



Then we have pre and post indexing this is also a very interesting scheme of addressing memory locations.
The two distinct modes are pre indexed with write back and post indexed. The normal mode that we have seen so far is typical pre index addressing mode. When we have pre index with write back the interesting thing is that when we calculate the address of the memory location, we have added the immediate mode offset with the content of r1 and then we update the address of the base register **r1** with the new address Ex: **LDR r0, [r1, #4]!**. Now, why it is a pre index because when we fetch the data we use the address of r1 plus 4, in this case four is an immediate value and then the content at the address of r1 plus 4 value is loaded on to r0 and r1. In contrast to this we can look at post index. In the post index we will find that syntax of the instruction is slightly different.

LDR r0, [r1], #4 Here what happens is that we update the address register after address it used. What does that mean that means we use r1 for referring to the memory i.e. first load the contents at r1 into r0 and after we have accessed the memory we modify r1 and load the modified value of r1 back to r1. So, this is post indexing. So, the pre index means that the offset is added to the base register for accessing memory. In post index the base

value is used for accessing memory and then base value is added with the offset and this new value is stored back to r1. See, if I again use this instruction in a loop then in that case the modified r1 value would be used for accessing next memory location. Let us take an example to understand it better. Here I am illustrating a case of pre indexing with write back. So, we have got r0 as the target register and if we look here what we are telling is that we have got the memory location 9000. Let us say this memory location has got some value and then the memory location 9004 has got some other value. Now this register r1 which is my base is initially loaded with 9000. Now, after I execute the instruction what happens? My r0 is loaded with content of the memory location 9004 because I have added 4 to r1 to access the memory location. So, my r0 is now loaded with contents of the memory location 9004 and then r1 content is changed to 9004. This is the basic operation that can find out in case of pre indexing. Since I am using a load instruction, contents of the base register as well as that of the target register are modified.

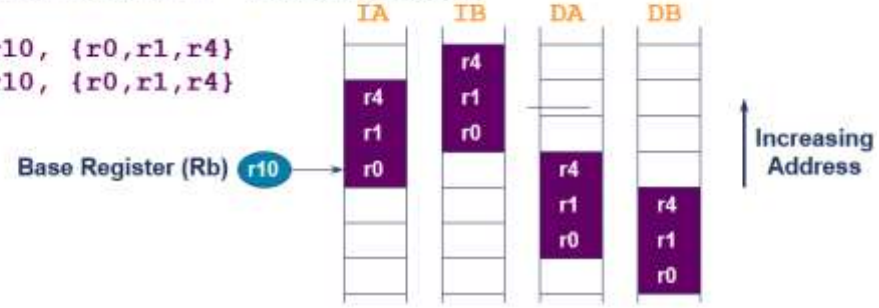
Address accessed

- Address accessed by LDR/STR is specified by a base register with an offset
- For word and unsigned byte accesses, offset can be:
  - An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes)
    ```
    LDR r0, [r1, #8]
    ```
  - A register, optionally shifted by an immediate value
    ```
    LDR r0, [r1, r2]
    LDR r0, [r1, r2, LSL#2]
    ```
- This can be either added or subtracted from the base register:
    ```
    LDR r0, [r1, #-8]
    LDR r0, [r1, -r2, LSL#2]
    ```
- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (i.e. 0 - 255 bytes)
  - A register (unshifted)
- Choice of *pre-indexed* or *post-indexed* addressing
- Choice of whether to update the base pointer (pre-indexed only)
    ```
    LDR r0, [r1, #-8]!
    ```

Load and Store Multiples

- Syntax:
  - **<LDM|STM>**{<cond>}<addressing_mode> Rb{!}, <register list>
- 4 addressing modes: xx = IA/IB/DA/DB
  - **LDMIA / STMIA** increment after
  - **LDMIB / STMIB** increment before
  - **LDMDA / STMDA** decrement after
  - **LDMDB / STMDB** decrement before

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```



Next, we have got **multiple register transfers**. In this case we transfer contents of multiple registers to the memory or content of multiple memory locations to multiple registers using a single instruction. Obviously this provides an efficient way of moving blocks of data between memory and the set of registers. And since I am using this multiple byte transfer or word transfer in a single instruction, what happens is that this instruction cannot be interrupted under normal circumstances. So, this instruction may increase interrupt latency. That means if there is an interrupt pending, that interrupt can only be serviced if and only if I have completed transfer of all memory locations specified in the instruction to memory or register bank. So, what is the mnemonic used for this multiple byte load store. In this case I use LDM or SDM. Ex: **LDM Rn, reglist**
And the base register Rn determines the source or destination address depending on the instruction whether it is a load instruction or whether it is a store instruction.
Now, there are also number of addressing modes which are supported here. So, I have listed these addressing modes. We can have increment after, we can have increment before, we can have decrement after, we can have decrement before.

Multiple Register Load / Store Instructions (2)

LDM	Load multiple registers
STM	Store multiple registers

Addressing mode	Description	Starting address	End address	Rn!
IA	Increment After	Rn	Rn+4*N-4	Rn+4*N
IB	Increment Before	Rn+4	Rn+4*N	Rn+4*N
DA	Decrement After	Rn-4*Rn+4	Rn	Rn-4*N
DB	Decrement Before	Rn-4*N	Rn-4	Rn-4*N

Addressing mode for multiple register load and store instructions

The start address when I am using addressing mode IA; I shall have the one instruction LDMIA Rn and I have got the set of registers. This Rn has the base address and I am trying to transfer the content memory locations starting from the base address to registers r1 r2 r3. So, the start address is Rn, the end address is Rn+4*N-4. Why I am having this value because I am transferring the content of the memory to registers. Since I am transferring content of memory to registers, this address has to be incremented and they have to be incremented by 4. If there are N registers to be loaded then it has to be multiplied by 4 because ARM has got byte addressing because each byte has got a unique address and I am subtracting minus 4 because it is a increment after.
So, final value of Rn is Rn+4*N.
Similarly when I have got an increment before, the start address is Rn+4 and the last address is Rn+4*N. So, base address is added with the offset 4 and content is stored in the first register value. Here the end address would become Rn plus 4 into N and here also the final value of Rn will be Rn plus 4 into N. Similarly, I can have decrement after and the decrement before modes. In this case instead of increment I have got decrement as the basic operation. So, what we have therefore seen is that using a single instruction by specifying the base register and by specifying the set of registers here I can load; the memory location pointed to by the base will be loaded on to the registers. If instead of that if I got store I shall be storing the content of the registers on to the corresponding memory locations. And the addresses of the memory locations will be given by the content of Rn.

STACK Processing:****

Now, this multiple register instructions in various ways facilitate stack processing.

- Stack is implemented as a linear data structure which grows up (Ascending)or down (descending) stack.
- stack pointer register hold the address of the current top of the stack.

Modes of Stack Operation

- ARM multiple register transfer instructions support
- Full ascending: grows up, SP points to the highest address containing a valid item
- Empty ascending: grows up, SP points to the first empty location above stack
- Full descending: grows down, SP points to the lowest address containing a valid data
- Empty descending: grows down, SP points to the first location below the stack

Now, what is the stack? Stack is implemented as a linear data structure which grows up or down. So, I can have growing up stack or I can have a growing down or a descending stack and stack pointer register hold the address of the current top of the stack. Now, how shall I use these different addressing modes that I have discussed in the contest of stack? Do we really need special push and pop instructions? Strictly speaking I do not need, why? Because, I have got a symmetric organization with any of the register as space registers I can use the addressing modes. If I am using the addressing modes then using my stack pointer register I can implement either ascending or descending stack.

Hence we shall have in ARM all this possible modes of stack operations. We have full ascending, empty ascending, full descending, empty descending. In case of full descending, the stack grows down but the stack pointer points to the highest address containing a valid item.

In case of empty ascending the stack grows up but SP points to the first empty location above stack. So, that is why it is empty ascending. Similarly I can have a full descending whether the stack grows down and SP points to the lowest address containing a valid data. In case of empty descending the stack grows down and SP points to the first location below the stack. Now, we can realize that if I used stack pointer as my base registers with addressing modes for multiple byte transfer that I have discussed i.e. increment after, increment before, decrement after and decrement before. If I use these addressing modes, I can implement stack in any of these modes that I have listed. And this is a flexibility that this ARM architecture provides.

If I compare this with a typical processor like your 8085 or 8086 depending on the operations defined for push or pop instructions, the nature or mode of the stack gets defined. And that is defined by the architecture itself, but here the mode of the stack operation is programmer defined. So, we can now talk about stack instructions which can implement full ascending or empty descending. Full Ascending: LDMFA, STMFA, SP points to last item in stack. Empty Descending: LDMED & STMED, SP points to first unused location.

Similarly I can have the other counter parts as well. So, we talk about instructions LDMFA. LDMFA actually translates to this LDMDA (POP) and STMFA translates to STMIB (PUSH) and SP points to the last item in the stack. When I am using empty descending this translates to IB that is increment before and STMIA is increment after and SP used to the first unused location in the stack. Now, in this case the interesting feature is that these are not real instructions. In many case the ARM assembler provides these instructions and these instructions translates to one of the instruction modes IA, DA, IB & DB.

Next, we have got swap instruction. In case of swap instruction what happens- a word is swapped between memory and register. Special case of load store instruction. **SWP :** swap a word between memory and register. In case of SWPB, we swap a byte between memory and register and this is useful for implementing synchronization primitives like semaphore. If we have already done a course on OS, we know what a semaphore is and we have also discussed in the context of PIC that is if we want to prevent access to a common memory location by concurrent threads, that we want to restrict the access to a single thread when a memory location is shared between two concurrent threads, we would like to use semaphores. To implement such operations swap is a hardware supports.

Control flow instructions: *****

We have got branch instructions, conditional branches, conditional executions, in fact this is the very interesting feature of ARM. ARM enables execution of each instruction conditionally. Then we have got branch and link instructions as well as subroutine return instructions. And these instructions are all used for controlling your program flow. Typically branch instruction has got 2 variance; one is branch which is actually an unconditional branch or jump.

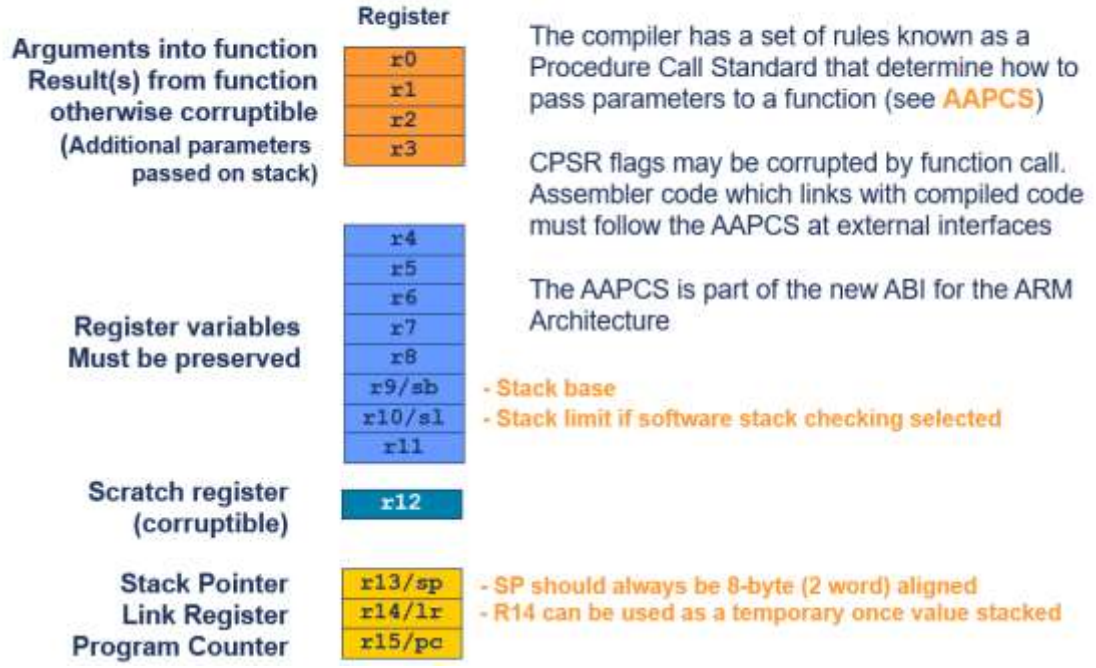
Branch instructions

- Branch : B{<cond>} label
- Branch with Link : BL{<cond>} subroutine_label

The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC

- ± 32 Mbyte range
- How to perform longer branches?

Register Usage



Example: Block memory copy: r9 points to source of data, r10 points to start of destination data, r11 points to end of the source

```
Loop    LDMIA r9!,{r0-r7}
        STMIA r10!,{r0-r7}
        CMP r9,r11
        BNE Loop
```

The other one is conditional branch that is we branch on certain condition. In both the cases we have address label which is part of the instruction and is a signed pc-relative offset. So, we jump to the location whose address

is calculated with reference to the current value of PC. Now, we can look at an example how to use this conditional jump instruction. In this case, we use multiple transfers, multiple byte transfer instructions. This is load and this is store and then we have compared and then we have jumped. So, I can have r9 pointing to source of data and r10 can point to the start of destination data; r0 and r11 points to end of the source. So, what we are doing here? I am loading this r9 onto this registers and then I am storing them back. So, effectively these two instructions are doing block memory copy. If we look into it I am copying what- a set of memory locations from one base address, starting from one base address to another base address because I have got the source in r9 and destination base in r10. So, these are two distinct values so I can actually do a block memory copy using just 2 instructions. In fact here I am checking whether my r9 that is it is end of the source because whether I have actually reached the end of the source and if it is not again I am going back and I am actually doing the loop. So, this is the simple code snippet, of ARM instruction by which we have shown how this load store compare as well as conditional branch can be used.

This loop is typically a label which will be obviously used by that assembler to calculate the address. This address will be replacing these loop a symbolic reference and it will be calculated with respect to the current value of PC. The assembler will calculate the current value of the PC and with respect to the PC, the offset will be provided here for doing the actual jump. An unusual feature of ARM instruction set is conditional execution of each and every instruction. We have already shown that I can have branch instruction with condition code but it is not that we can use condition code only with branch instruction. We can use condition code with other instructions as well. Here is an example: **ADDEQ r0, r1, r2;** where we have shown that this addition; so this addition instruction is associated with the condition code. Now, what does that mean? That is addition instruction will only be executed when the zero flag is set to 1. That is exactly the condition code that I am referring to here. Now, if it is not so what will happen- this addition instruction will be skipped and next instruction will be executed. So, effectively this ADDEQ will be converted to a no operation instruction. So, what are the advantages? Why is that, that in ARM instruction set these kind of instructions have been provided? Obviously it reduces the number of possible branches. As I reduce branching if I am implementing a pipelined architecture, then the number of pipelining flushes reduces because if I have a pipelined architecture then what happens; I need to pre-fetch the instructions and when there is a branch the pre-fetch instructions have to be removed from the pipeline.

Condition Codes

- The possible condition codes are listed below
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Now, when I have a conditional instruction and I am not using branching then the conditional instruction remains in pipeline only that this instruction is not really executed and I replace that by nop, no operation. As a result, since I am reducing the number of pipeline flushes I have got an improvement in the performance. Also it increases code density (ability of an instruction to perform more operations). Why? Because a branch could essentially mean that I have to actually use a branch instruction if I think in terms of the previous example, I

have to use a branch on the condition code, branch on equality and then I have to use the add instruction. So, the instruction count becomes 2. If I use a conditional instruction in this case instruction count is 1 and obviously my code density increases. So, a thumb rule says that whenever the conditional sequence is 3 instructions or feyour it is useful to exploit conditional execution than to use a branch. But if it is really a number of instructions that is to be executed on a particular condition is bigger than this, what will happen. If we use conditional instructions, your pipeline will only have effectively nops. So, when the condition is not getting satisfied, the pipeline will effectively execute nop. So, your CPU becomes underutilized. So, when the branching set of instruction is small enough, we should use conditional instructions rather than branch. That would increase code density and at the same time increase efficiency of your code.

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
    CMP   r3,#0
    BEQ   skip
    ADD   r0,r1,r2
skip
```

```
    CMP   r3,#0
    ADDNE r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

```
loop
  ...
  SUBS r1,r1,#1   <- decrement r1 and set flags
  BNE loop        <- if Z flag clear then branch
```

Conditional execution examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

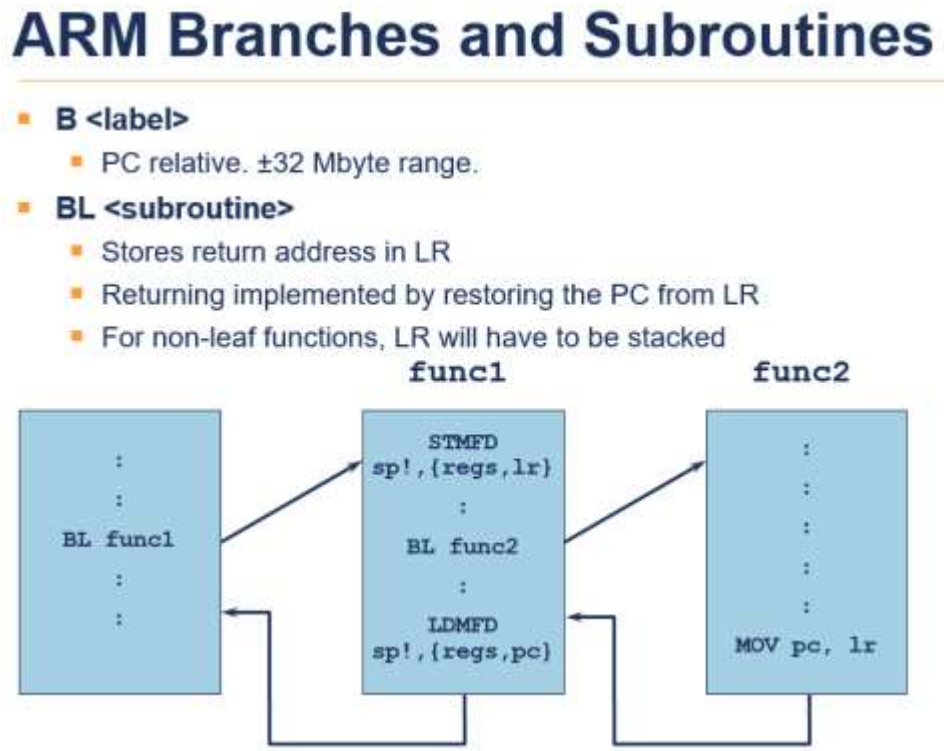
conditional

```
CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1
...
```

- 3 instructions
- 3 words
- 3 cycles

Next, is branch and link instruction is like CALL. This branch and link instruction is primarily used for subroutine call. So, it performs the branch; using this instruction we can perform a branch. But along with branching, the address following the branch is saved in the link register (R14), i.e. the next value of PC that is the return address is saved it in the link register. So, the basic different between ordinary branch and branch and link is the use of the link register. In case of a branch the next value of PC is not saved in the link register. In this case the next value is saved in the link register. So, here we are showing an example **BL sub1:** that is when I do a subroutine call, I use branch and link subroutine. So, here I am branching to the beginning of the subroutine and the return address which could be the instruction following this branch will be stored in the link register. Now, I have got only 1 link register and there maybe nested subroutine calls. What is to be done under that condition? For nested subroutine, we will be pushing r14 that is the link register and some work registers

in the stack and stack will be set up in the memory. Say we are now inside the first subroutine using BL sub1. So, the return address is stored in a link register. So, from inside subroutine one we would like to call another subroutine.



So, I need to save the link register as well as the current working registers. So, I use multiple byte store instruction. So, where I am storing, I am storing to the location pointed to by r13 which is my stack pointer. What I am storing? I am storing the work register as well as the link. So, the link register is link of the previous subroutine call. Now, when I execute this BL sub2, the return address from this will be stored in the r14, the current value of r14 and the previous r14 is now saved in the stack. So, this is how the nested subroutine call is to be managed in ARM processors. Then how do we return from subroutine. Now, there are no specific instructions like return because the moment I can load my PC with the value of the link register I will return to the main flow from where the subroutine was called. So, the simplest thing would be to move r14 to the PC which is the r15 that is the register which is your program counter.

But when the return address has been pushed into the stack then we can use a load instruction which uses the stack pointer register r13 and we load the value onto the set of target registers. Now, what is interesting in both these cases we will find that when I am really returning from the subroutine, if I am using the multiple word or multiple byte transfer instructions, it ensures that the registers are always correctly loaded because this register transfer cannot be interrupted. If I am using, say for example, we do not have this multiple byte or word transfer instructions or we are not using this instructions, we are using single register transfer instruction for loading the parameter pack onto the registers while returning from the subroutine, what can happen? If an interrupt occurs in between we will jump possibly to an interrupt service routine.
And the register will be lost and the state of the computation will not be correctly restored when we come back and in fact typically when I need to return from the kind of subroutines I would like to disable interrupt. So, that the status of the registers are correctly saved. If I am using this multiple data transfer instructions, I make sure that the state of the computation cannot be corrupted by an interrupt. But what is the consequence, I already told we that interrupt latency increases. So, when we are writing a software these has to be kept in mind and your timing calculations have to be appropriately done.

Software interrupt instruction (SWI): Instruction: **SWI {<Cond>} SWI_number**
A software interrupt instructions causes what we call a software interrupt exception and these provides a mechanism for applications to call OS routines. Now, typically if we have this instruction that SWI; now just

like any other ARM instruction I can have associated with it a condition code and I have a software interrupt number associated with it. In fact in a way we can realize that software interrupts are actually calling a routine. That means we are calling a routing which is part of the operating system and not part of your program that is user code. We are calling a routine which is part of your operating system and not part of your own set of code. Now, what is the difference between a software interrupt and subroutine call; this is the basic difference that in case of a subroutine we actually call a subroutine which may be part of your code and the subroutine can be located anywhere in the memory.

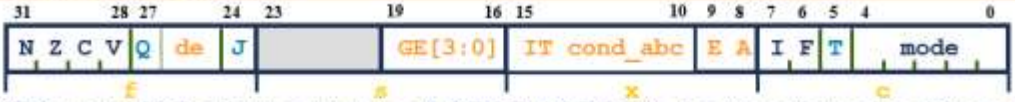
But when we are actually using a software interrupt, the software interrupt servicing has to start from fixed locations, fixed vector locations and that is why we can established a kind of a universal protocol for accessing OS utilities from the applications of the users because if the OS locates the utilities at different memory locations and if we have to use a subroutine call to do that, then it becomes an unmanageable situation; we have to remember and we have to be notified and told about the location of all these OS routines. So, that we can use them through a subroutine call when I am using a software interrupt, the protocol gets fixed, we exactly know where the interrupt handler is located. And there is another real advantage of using software interrupts in case of ARM there is a mode switch because I have already told we that your **application program will run in user mode, but OS routines will run in supervisor mode**. So, when we have to actually call the OS routines, we have to switch mode from user to supervisor mode and supervisor mode is a privileged mode. So, software interrupt enables this switching of mode as well. In this case, in case of a ARM it sets the program counter PC to the offset 08 in the vector table. In fact I am not going into the details; this can be a different address as well. So as I have already told we, these software interrupt instruction is typically part of user program. So, it is executed in the user mode and instruction forces the processor mode to become supervisor and this allows the OS routine to be executed in privileged mode. Each software interrupt instruction has an associated number which is used to represent the particular function caller feature. But this number is not directly used by this instruction; In fact what happens is, the software interrupt handler routine or the exception handler can use this number for identifying the service to be provided. We need to pass parameters, so use typical registers for passing the parameters. In fact return value is also passed using registers. So, let us take an example; this is an example of a software interrupt instruction and in this case we have got CPSR which is the program status register I am showing we these are the flags- condition flags. These are your interrupt enable disable flags, this is the thumb mode flag and this is the mode bit which is now user mode.
This is currently of this value. So, currently this is the instruction which is to be executed and this is the software interrupt instruction. This lr is the link register value, some value okay which is not really consign right now in this context of discussion and this lr is what the r14 value.
Example:

PRE: cpsr = nzcvqift_USER	POST: cpsr = nzcvqift_SVC
PC = 0x00008000	PC = 0x00008004
lr = 0x003fffff (lr = r14)	lr = 0x003fffff (lr = r14_SCV)
r0 = 0x12	r0 = 0x12
0x00008000 SWI 0x123456	

And we can have this register r0, okay. We may use r0 for passing parameters; so I am just showing some value 12. So, what happens when the software interrupt instruction is executed? Obviously the mode now switches to the system. So, it is SVC and this is what, this is saved program status register. I hope we remember this register that I had talked about in the last class; this is the saved program status register. This register will have the previous value, the previous value the mode was user, so that is saved, but in this case we will find that these bits remain unchanged. The other bits remain unchanged. Now, the PC value has been changed to the desired location and now this lr is what, this lr value if we see, lr value is will be this location and what is the interesting feature. The interesting feature here is that your lr is what; now linked register is r14 is SVC. r14 SVC is what the copy of r14 which becomes available in SVC.

Now, we shall look at some of the program status register instructions and there are typically two instructions to control PSR directly. One is MRS another is MSR.

PSR access



- MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register or take an immediate value
  - MSR allows the whole status register, or just parts of it to be updated
- Interrupts can be enable/disabled and modes changed, by writing to the CPSR
  - Typically a read/modify/write strategy should be used:

```
MRS r0,CPSR        ; read CPSR into r0
BIC r0,r0,#0x80    ; clear bit 7 to enable IRQ
MSR CPSR_c,r0      ; write modified value to 'c' byte only
```

- In User Mode, all bits can be read but only the condition flags (_f) can be modified

MRS transfers contents of either cpsr or spsr into a register and MSR transfers contents of register to cpsr or spsr. Now, there are this example, example is that of enabling IRQ interrupt. So, how will we do it? So, the code uses these instructions; this is for accessing cpsr, so I get the cpsr. Then I use the bit manipulation, the PIC instruction and then I load it back to cpsr. So, the PRE in this case I was not set and in this case I was set I am showing in a small the change.

Example: Enabling IRQ interrupt

PRE: cpsr = nzcvqIFt_SVC

```
MRS r1,CPSR
BIC r1,r1,#0x80
MSR cpsr,r1
```

POST cpsr = nzcvqiFt_SVC

So, now this is the modified status of the IRQ flag that is this is the masking bit. So, what I have, these instructions are typically executed in SVC mode. So, I told we that SVC is the privileged mode, so in that case we can actually modify this, these bits. So, that is why we have got these instructions which are available in your privileged mode. In your user mode we can only change the flag bits and not the status and mode bits.

Co-processor Instructions:

- Used to extend the instruction set
- Used by cores with a co-processor
- Co-processor specific operations Ex: Memory management unit
- Syntax: Co-processor data processing

  CDP {<Cond>} Cp, Opcode1, Cd, Cn, Cm,{Opcode2}

  Cp -Co-processor number between p0top15

  Opcode field describes co-processor operation

  Cd, Cn, Cm- Co-processor registers
- Also Co-processor register transfer and memory transfer instructions

****Thumb instruction set:

- Compressed form of ARM
  - Instructions stored as 16-bit,
  - Decompressed into ARM instructions and
  - Executed
- Thumb encodes a subset of the 32-bit instruction set into a 16-bit subspace
- Thumb has higher performance than ARM on a processor with a 16-bit databus
- Thumb has higher code density (THUMB saves 30% space)
- For memory constrained embedded system
- Only low registers r0 to r7 is fully accessible

- Higher registers are accessible with MOV, ADD,CMP instructions
- Barrel shift operations are separate instructions
- PUSH/POP for stack manipulation
  - Descending stack (SP hardwired to R13)
- No *MSR* and *MRS*, must change to ARM to modify CPSR (change using *BX* or *BLX*)
- ARM entered automatically after RESET or entering exception mode
- Maximum 255 SWI calls

*****Thumb Architecture:** The ARM family grew to include the ARM7 which extended the architecture to full 32-bit addressing (early ARMs were limited to 26 bit addressing), increased maximum clock rate to 40 MHz, and expanded the cache to 8 KB. An optional feature in the ARM7 architecture was 'Thumb', a combination of a new instruction set with a 16 bit long instruction format, and a hardware logic unit that sat in the instruction fetch path and translated Thumb instructions to regular, full 32-bit length ARM instructions. The Thumb hardware scheme is shown in Figure. Thumb is a rather clever feature that adds very little complexity to an ARM7 MPU, only about 3000 transistors. Yet Thumb improves ARM instruction density (already quite good, roughly comparable to 32-bit x86 code) by about 25 to 35%. Not only that, but in systems with 16-bit wide memory (not uncommon in cost sensitive embedded applications) an ARM7 would run an application compiled into Thumb faster than if it had been compiled for 32-bit ARM code.

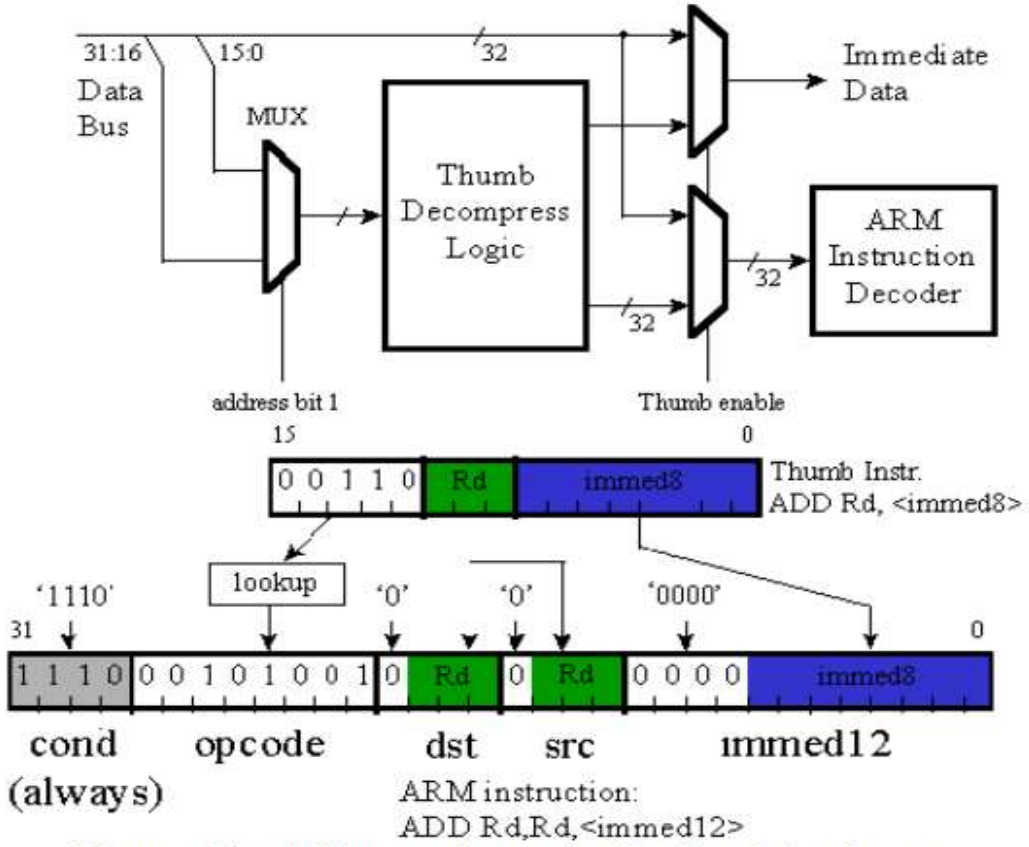


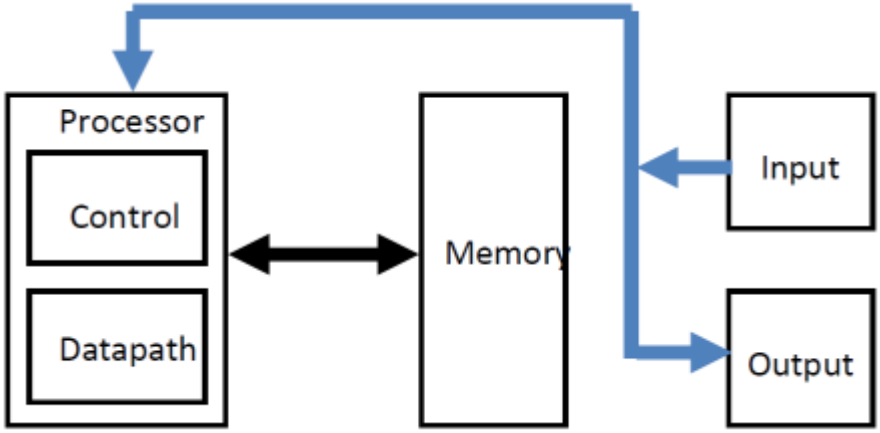
Figure . Thumb Compact Instruction Set Translation System

BUS STRUCTURE

Topics Covered: Time multiplexing, serial, parallel communication bus structure. Bus arbitration, DMA, PCI, AMBA, I2C and SPI Buses.

Bus Structure:**** Bus Structure is how a CPU communicates with memory and devices.
So, for this communication CPU vs bus; bus is the mechanism by which CPU communicates with memory and IO devices. And it is not just a collection of wires. In fact, bus defines the protocol for communication. Bus is a transport device then the bus in the context of Embedded System or computers is also defining the corresponding transport mechanism between the CPU and memory and IO devices.

So, typically bus is a shared communication link, because there are a mini devices which can actually sit on the bus. A single set of wires used to connect multiple subsystems. Bus is also fundamental tool for composing large and complex systems because a bus defines the mechanism for communication and once I have to put in a complex sub system along with CPU that subsystem should have the ability to communicate with the communication scheme specified by the bus. So, a bus enables us to build a system on the basis of individual components which are compatible with the bus. Here, we have shown an example and in this example the interesting thing to notice that I have shown not 1 bus, but 2 buses. One bus connecting the memory and the processor another bus input at output devices to the processor we may have this kind of two distinct buses or else we can have a single bus and both memory and input at output devices sitting on the same bus itself.



A Generic bus structure will have address, data and control. So, I am showing the number of lines I can have for address, for data, for control m n and c respectively. And accordingly I shall be setting up the connection or the communication link.

Control lines actually implements the transaction protocol. The signals which flow along the control lines are really instrumental in implementing the transport protocol. The signals request an acknowledgement. So, these are request and acknowledgement signals are the 2 basic and generic types of signals we will find on the bus.
Data lines carry information between the source and the destination. In fact, we have clubbed data and address together, because address is nothing, but a special form of data. Also there may be complex commands which may be given to devices through data lines through the bus.

A bus is a common pathway to connect various subsystems in a computer system. A bus consists of the connection media like wires and connectors, and a bus protocol. Buses can be serial or parallel, synchronous or asynchronous. Depending on these and other features, several bus architectures have been devised in the past.

The Universal Serial Bus (USB) and IEEE 1394 are examples of serial buses while the ISA and PCI buses are examples of popular parallel buses. This article first describes fundamental information on bus architectures and bus protocols, and then provides specific information on various industry standard bus architectures from the past and the present, and their advantages and disadvantages. It also describes how different types of bus architectures are used simultaneously in different parts of a modern personal computer.

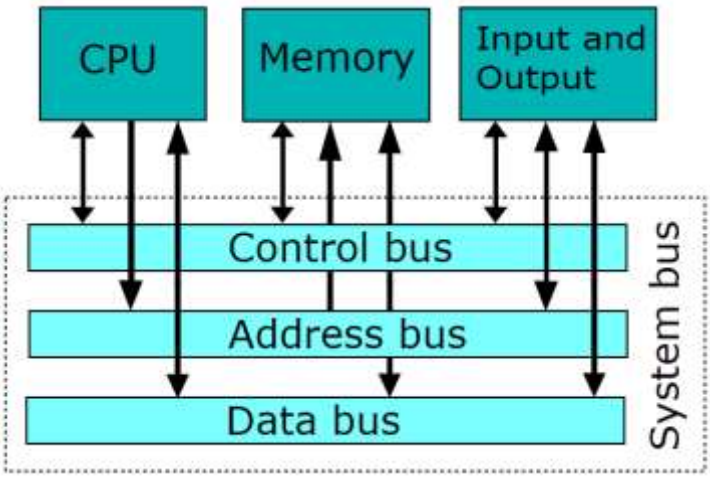
Introduction A typical embedded system is composed of several components such as the Central Processing Unit (CPU), memory chips, and Input / Output (I/O) devices. A bus is a common pathway or a set of wires that interconnect these various subsystems. The bus thus allows the different components to communicate with each other.

A bus can also be defined as a channel over which information flows between units or devices. It typically has access points, or places into which a device can tap to become part of the channel. Most buses are bidirectional and devices can send or receive information. A bus is a shared communication link between the different devices. It allows to add new devices easily and facilitates portability of peripheral devices between different systems. However, if too many devices are connected to the same bus, the bandwidth of the bus can become a bottleneck. Typically more than two devices or subsystems are involved in a bus, and channels connecting only two components are sometimes referred to as ports instead of buses.

Buses often include wires to carry signals for addresses, data, control, status, clock, power and ground. The address lines indicate the source or destination of the data on the data lines. Control lines are used to implement the bus protocol. Often there are lines to request bus control, to handle interrupts, etc. Status lines indicate the progress of the current transaction. Clock signals are used in synchronous bus systems to synchronize bus operation.

The communications needs of the different devices in a computer system vary. For instance, fast high bandwidth communication is needed between processors and memory whereas bandwidth requirements are not so high on buses to I/O devices. This has led to the creation of different kinds of buses differing in their width, latency and bandwidth capabilities.

There are normally three types of bus in any processor system:****



Address bus: This determines the location in memory that the processor will read data from or write data to. The physical location of the data in memory is carried by the address bus. An internal hardware component, having received the address from the address bus and about to receive the data, enables a buffer to allow the flow of signals to or from the location that was designated by the address bus. The address bus carries only the information regarding the address, and is synchronized with the data bus to accomplish read/write tasks from the processor. The address bus is only as wide as is necessary to address all memory in the system.

Data bus: This contains the contents that have been read from the memory location or are to be written into the memory location. The data bus "width" of an MCU is typically 8-, 16-, 32- or 64-bits, although MCUs of just a 4-bit data bus or greater than 64-bit width are possible. The width of the data bus reflects the maximum amount of data that can be processed and delivered at one time. A 64-bit processor has a 64-bit data bus and can communicate 64-bits of data at a time, and whether the data is read or written is determined by the control bus.

Control bus: this manages the information flow between components indicating whether the operation is a read or a write and ensuring that the operation happens at the right time.

Bus Characteristics:**** Bus signals are usually tri stated. Let us consider the interface of the device to the bus. So, if a device would like to disconnect itself from the bus, it would drive its interface lines to tri state high impedance state. So, effectively it gets disconnected from the bus. So, all these devices are really sharing the common set of signal lines. So, typically bus signals are tri stated. In many cases address and data lines may be multiplexed. This would save on the number of actual connections or wires. The other interesting thing to be noted, that every device on the bus must be able to drive the maximum bus load. Because maximum bus load would determine what are the maximum number of device that we can actually put on the bus. So, it has to drive the bus where is another bus devices to deliver the signal so, that is the compatibility with regard to the drive. Bus may include a clock signal. It is not necessary that all the buses are clocked, but some of the buses do include clock signal. And all the timing, particularly for bus signals which implement the complete communication protocol is defined relative to the bus clock. Now, the bus clock may not be same as that of the system clock of the processor.

In the next page, we are showing the time multiplexing. So, when there is a single signal line which is used for carrying two kinds of signals we do have time multiplexing. I am showing here a master another is a servant device. Now, master is requesting the servant to receive data or to get data from and the data is flowing here. Now, the interestingly there is no address pass that is where will these data be stored by the servant is not being specified by a separate address lines, what is happening here is the same data line which is of 8 bit width is being used for these purpose. So, see how it has been done in this case. So, we have the address as well as the data. So, when we first send the request we have the address which is being sent next time with the next request we are sending the data. So, the address and data is getting synchronized with respect to the request signal. In these cases what is happening is the master is likely to transfer 16 bit data, but you have got 8 data lines available. So, you have to do multiplexing. So, if in the first request, you are sending the more significant byte and then you are sending the less significant byte. It can be other way also. So, what you can see is here the 16 bit transfer I would have required 16 lines instead of that I am doing it using 8 lines. But obviously, I am saving wires at the expense of time, because I would do require 2 clock periods for transferring 16 bit data.

Now, obviously, I would like to increase the bus bandwidth the way I have shown if I am using time multiplexed mechanism for transferring data or address and data. The basic limitation that comes is with regard to the bus bandwidth. So, obviously, if I do a non-multiplexed address and data lines I shall have increased bus bandwidth. So, you have got more bus lines and increase complexity in terms of the physical layout as well as the space requirement. The other option is increasing data bus width. In fact, this is done in many of the main frame systems. That means, you increase the width of data bus and transfers multiple words therefore, will require few bus cycles. So, consider as system is having its basic word length to be 16 bits that is sitting on the bus of say 64 bits. A bus which can support 64 bit data lines and if you really doing a sequential access to the memory you can transfer 4 words in 1 bus cycle. So, effectively increases the bandwidth of the bus, but obviously, here the drawback is you are using additional wire. The other option is block transfers. Allow the bus to transfer multiple words in back to back bus cycles. This is also using the same concept of using sequential address. That is I shall just provide a single address and then I shall do a number of this data transfer cycles. So, only a 1 address needs to be send at the beginning and the bus is not released until the last word is transferred. Logically this has an increased complexity and decreased response time for request from the other devices which are willing to use the bus, but for a transfer you can have a large volume of data transfer and a fast data transfer. In fact, in

DRAMS, use of sequential signal from ARM for accessing the DRAM, you do a row select and then transfer the data in the row buffer for all column sequentially, that can be very easily done through a block transfer mechanism.

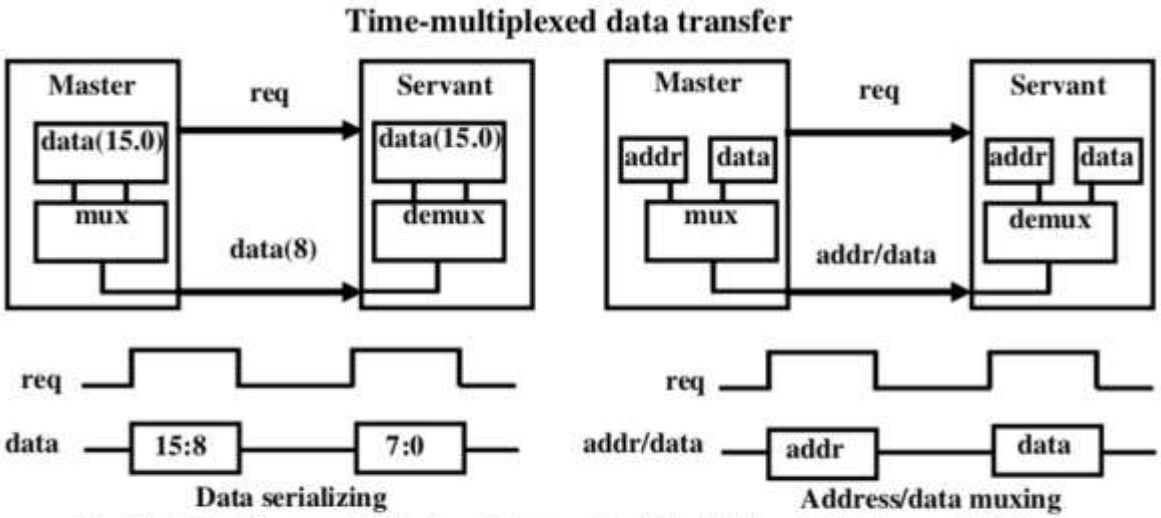


Fig. 13.4 The Time multiplexing data transfer. The left hand side transmits 16-bits of data in an 8-bit line MSB after the LSB. The transfer is synchronized with the *req* signal. In the example shown on the right hand side the same set of wires carry address followed by data in synchronism with the *req* signal. mux: stands for multiplexer

TIME MULTIPLEXING:**** Bus lines can be separated into two generic types: dedicated and multiplexed.

- **A dedicated bus line** is permanently assigned either to one function or to a physical subset of computer components. —An example of functional dedication is the use of separate dedicated address and data lines, which is common on many buses.

- **In a multiplexed bus**, address and data information may be transmitted over the same set of lines using an Address Valid control line. At the beginning of a data transfer, the address is placed on the bus and the Address Valid line is activated. At this point, each module has a specified period of time to copy the address and determine if it is the addressed module. The address is then removed from the bus, and the same bus connections are used for the subsequent read or write data transfer. This method of using the same lines for multiple purposes is known as time multiplexing.

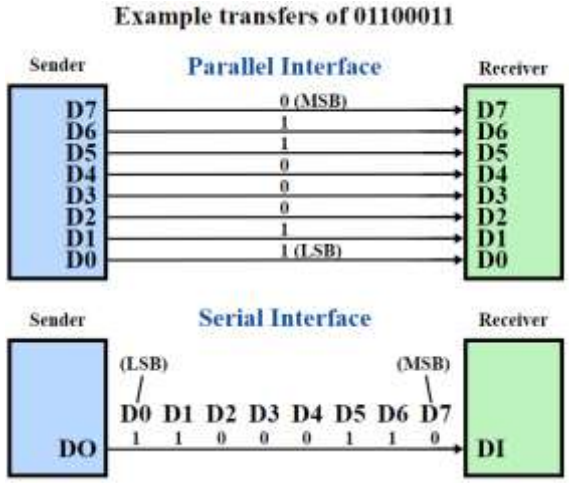
The advantage of time multiplexing is the use of fewer lines, which saves space and, usually, cost.
The disadvantage of time multiplexing is that more complex circuitry is needed within each module. Also, there is a potential reduction in performance because certain events that share the same lines cannot take place in parallel.

So, what are the advantages of bus? Since we are using a bus we have an agreement of the communication protocol. So, **Versatility**, **new devices can be added easily**. It is not that each device having its peculiar communication protocol. If you have a 1 to 1 connection then each device can have its own communication protocol and then its implementation in terms of connecting into the processor will become more complex. It is a **low cost**, because a single set of wires is shared in multiple ways depending on the requirement of the device.

What are the disadvantages? It obviously, creates the communication battled although we had started from the time multiplexed organization and talked about ways and means of increasing bus bandwidth. But the bandwidth of the bus can actually limit the maximum IO throughput. And that can actually be the limitation on the

processing capability not really processing capability, but output of the processor. So, if you may have a processor working at 2.4 Giga Hertz, but if it is not getting data at a rate more than 133 Mega Hertz then obviously, you will not be using the processor to full protection. So, bus can face that kind of a bottle neck. The maximum bus speed is largely limited by length of the bus and the number of devices on the bus. And there is a need to support a range of devices with widely varying latencies and widely varying data transfer rates. Because these are the inherent limitations of the devices, because I am looking at a bus if we look at the simplest picture of the model of the bus, where we are having a single bus and the single bus I have memory and other devices hanging from it. So, in that case the latencies are different, because the devices would take different times to become ready to put the data on the bus and there would be widely varying data transfer rates as well. So, we shall see there are various ways by which you can try to minimize this kind of disadvantages.

**********Below Figure is most important for serial and parallel communicaiton**********



Parallel Communication:****

So far the key model or the basic model by which you have looked at is mode of a parallel communication. That is multiple data control and possibly power wires, an effective data transfer unit is 1 bit per wire. So, this is useful for high data throughput with short distances and typically used when connecting devices on same IC or same circuit board. The motivation is, this passes must be kept short, because long parallel wires will result in high capacitance values which requires more time to charge and discharge. That will limit the data transfer rate and data misalignment between wires, increases as a length increases, because there is a capacity cost associated with each wire. So, higher cost means bulky. So, this wired is a classical model where we have parallel lines defining the bus and you will find that particularly when we are talking about system or a general purpose computer system.

The bus on your mother board is typically the parallel bus, because these buses are exceptive to run for shorter length. But you will find that Embedded Systems is not always the devices are connected by this kind of parallel buses, because you can realize that with these parallel buses and so many signal lines, there are lots of problems which we need to face. A problem would be that of the space. Just consider that PIC as a microcontroller which has got inbuilt peripherals and it has to supports some external peripheral as well. If it supports a full-fledged parallel bus by which external peripherals can be connected then it has to provide a large number of external beams. So, that is an area requirement even for the device itself putting so many connection lines can actually seek additional physical area.

Serial Communication:*****

So, the other option what we have is serial communication. Serial protocols and a very few lines for doing the actual communication you may have a single data wire possibly also a single control and power wires. Words are transmitted 1 bit at a time. So, effectively you have a higher data throughput with longer distances, because your capacity for load does not build up. Because when you have multiple parallel lines long parallel lines there

will be mutual capacitance which actually reduces or puts constraint in the transfer rate. So, my bus is cheaper it is less bulky. You have got more complex interfacing logic and communication protocol. Sender needs to decompose word into bits for serial communication. Receiver needs to recompose bits into word. And control signals are often sent on the same wire as data. So, that increases protocol complexity, because you do not have separate data or control lines. So, you might like to use a same line to send the control information followed by data. And since sender needs to decompose word into bits, because it has to send the words in a bit by bit fashion.

Synchronous Bus:

The other way of looking at bus is whether a bus is a synchronous bus or an asynchronous bus. A synchronous bus includes a clock in control lines. So, you have a fixed protocol for communication that is relative to the clock that means, timing for your signals, control signals are defined always relative to that of the clock. Since it is defined with respect to the clock you can understand that your logic can be simple and it can run very fast. A disadvantage; every device on the bus must run at the same clock rate means it should have the ability to deliver data the same clock rate. And to avoid clocks skew they cannot be long if they are fast, because a skew will buildup, if the skew buildup then there would be timing errors for data transfer. So, you will find the most processor buses which are expected to be fast buses, in such cases you have synchronous bus. But otherwise when you are really using peripherals the buses are not strictly synchronous bus.

Asynchronous Bus:

So, you got to have what you call asynchronous bus? Asynchronous bus is not strictly clocked. Since it is not clocked it can accommodate wide range of devices they can work with different latencies. They need not respond with respect to the same clock. The bus can be lengthened that without worrying about clock skew, but it definitely requires what we call a handshaking protocol. So, that these devices can talked to the processor in a proper fashion

Basic Protocol Concepts:

So, this takes us to the concept of protocols on the bus. So, what we say a protocol? at the basic element of a protocol is bus transaction. A bus transaction includes two parts - request and action. So, typically a request consists of a command and an address an action is basically transferring of data. Master in fact, we have already seen a master and a slave in a previous diagram in a time multiplexed bus organization. Master is a one who starts the bus transaction by issuing the command and address. Slave is the one responds to the address by sending data to the master if the master asks for data or receives data if the master wants to send data. But the device which is on the bus, may not be assigned permanently to this master and slave role. A slave in one transaction can become master in some other transaction.

Next question is that of Bus Arbitration. How the bus is reserved by a device that wishes to use it? If there is a single master there is a simplest system, when we say processor is the only bus master then really arbitration problem is not there. But if the slave on the master can have interchange role then we really need an arbitration skin, but let us start looking at master slave arrangement. So, in case of a master slave arrangement only the bus master can control access to the bus. It initiates and controls all bus requirements. As slave responds to read and write request. And in the simplest system of this master slave arrangement, the processor based system, but here the major drawback is - the processor is involved in every transaction. You might not like to have processor involved in all that transactions.

Bus Protocols: A bus is a communication channel shared by many devices and hence rules need to be established in order for the communication to happen correctly. These rules are called bus protocols. Design of a bus architecture involves several tradeoffs related to the width of the data bus, data transfer size, bus protocols, clocking, etc. Depending on whether the bus transactions are controlled by a clock or not, buses are classified into synchronous and asynchronous buses. Depending on whether the data bits are sent on parallel wires or

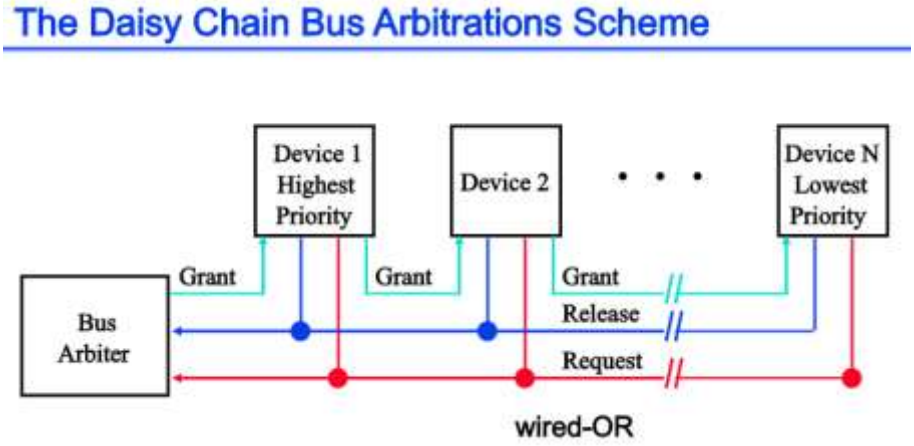
multiplexed onto one single wire, there are parallel and serial buses. Control of the bus communication in the presence of multiple devices necessitates defined procedures called arbitration schemes. In this section, different kinds of buses and arbitration schemes are described.

- The device that is allowed to initiate data transfers on the bus at any given time is called the bus master. In a computer system there may be more than one bus master such as processor, DMA controller etc.
- They share the system bus. When current master relinquishes control of the bus, another bus master can acquire the control of the bus.
- **Bus arbitration****** is the process by which the next device to become the bus master is selected and bus mastership is transferred to it. The selection of bus master is usually done on the priority basis.
- There are two approaches to bus arbitration: Centralized and distributed.

1. **Centralized Arbitration**
   In centralized bus arbitration, a single bus arbiter performs the required arbitration. The bus arbiter may be the processor or a separate controller connected to the bus.
   There are three different arbitration schemes that use the centralized bus arbitration approach. There schemes are:
   a) Daisy chaining
   b) Polling method
   c) Independent request

a) Daisy chaining

- The system connections for Daisy chaining method are shown in fig below.

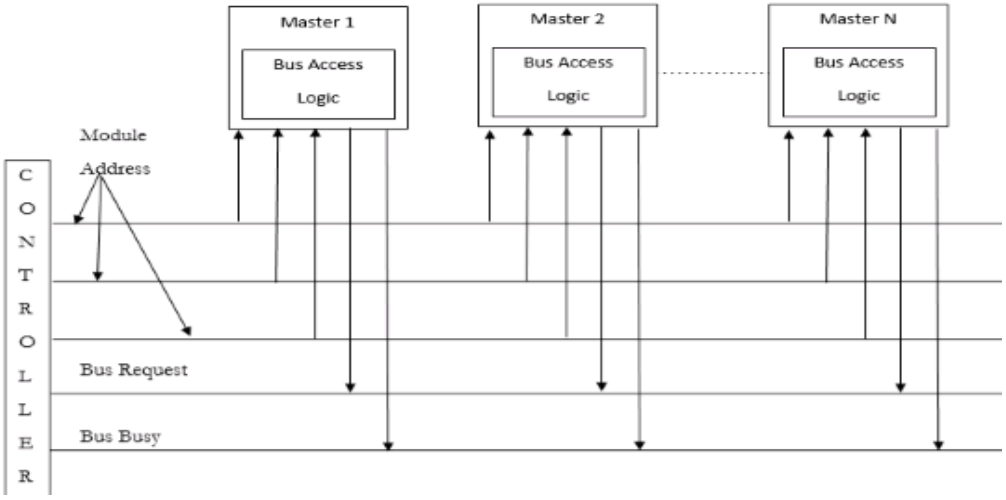


- It is simple and cheaper method. All masters make use of the same line for bus request.
- In response to the bus request the controller sends a bus grant if the bus is free.
- The bus grant signal serially propagates through each master until it encounters the first one that is requesting access to the bus. This master blocks the propagation of the bus grant signal, activities the busy line and gains control of the bus.
- Therefore any other requesting module will not receive the grant signal and hence cannot get the bus access.

b) Polling method

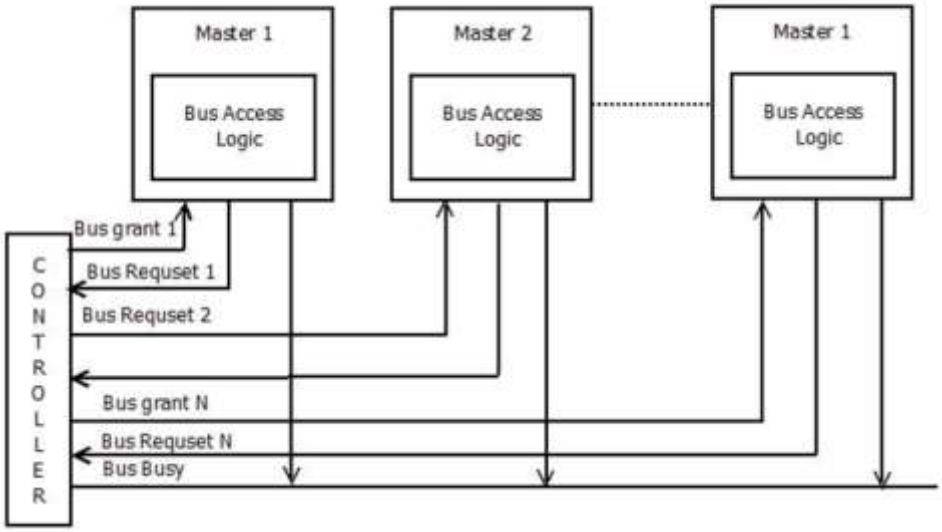
- The system connections for polling method are shown in figure above.
- In this the controller is used to generate the addresses for the master. Number of address line required depends on the number of master connected in the system.
- For example, if there are 8 masters connected in the system, at least three address lines are required.
- In response to the bus request controller generates a sequence of master address. When the requesting master recognizes its address, it activated the busy line ad begins to use the bus.
- Each device compares the code formed on the arbitration line to its own ID, starting from the most significant bit. If it finds the difference at any bit position, it disables its drives at that bit position and for all lower-order bits.

- It does so by placing a 0 at the input of their drive. In our example, device detects a different on line ARB2 and hence it disables its drives on line ARB2, ARB1 and ARB0. This causes the code on the arbitration lines to change to 0110. This means that device B has won the race.
- The decentralized arbitration offers high reliability because operation of the bus is not dependent on any single device.



c) Independent request

- The figure below shows the system connections for the independent request scheme.

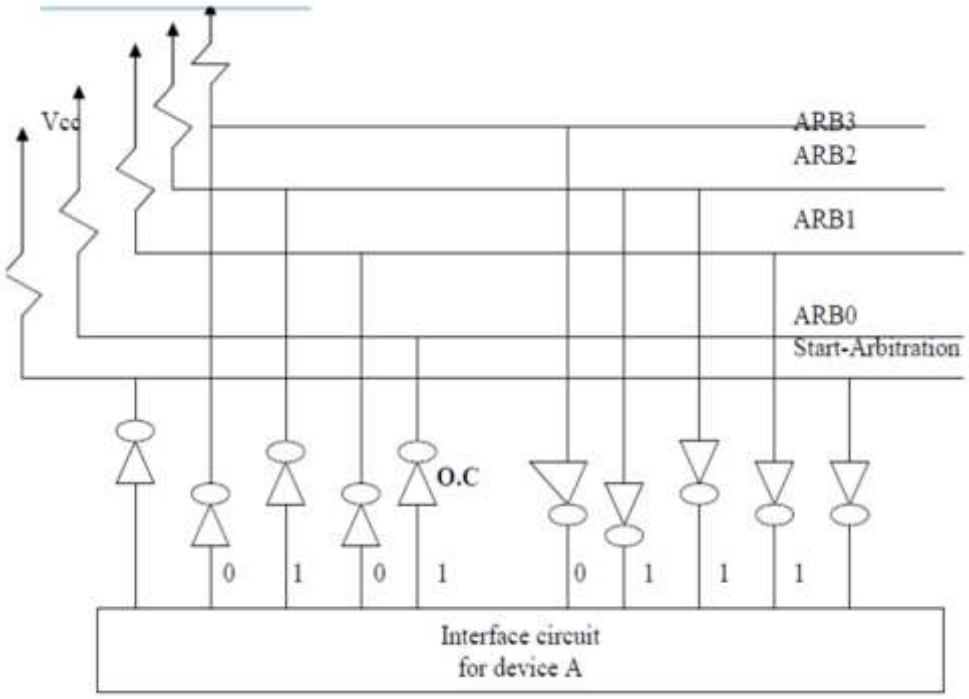


- In this scheme each master has a separate pair of bus request and bus grant lines and each pair has a priority assigned to it.
- The built in priority decoder within the controller selects the highest priority request and asserts the corresponding bus grant signal.

2. Distributed Arbitration

- In distributed arbitration, all devices participate in the selection of the next bus master.
- In this scheme each device on the bus is assigned a4-bit identification number.
- The number of devices connected on the bus when one or more devices request for the control of bus, they assert the start-arbitration signal and place their 4-bit ID numbers on arbitration lines, ARB0 through ARB3.
- These four arbitration lines are all open-collector. Therefore, more than one device can place their 4-bit ID number to indicate that they need to control of bus. If one device puts 1 on the bus line and another device puts 0 on the same bus line, the bus line status will be 0. Device reads the status of all lines through inverters buffers so device reads bus status 0as logic 1. Scheme the device having highest ID number has highest priority.

- When two or more devices place their ID number on bus lines then it is necessary to identify the highest ID number on bus lines then it is necessary to identify the highest ID number from the status of bus line. Consider that two devices A and B, having ID number 1 and 6, respectively are requesting the use of the bus.
- Device A puts the bit pattern 0001, and device B puts the bit pattern 0110. With this combination the status of bus-line will be 1000; however because of inverter buffers code seen by both devices is 0111.
- Each device compares the code formed on the arbitration line to its own ID, starting from the most significant bit. If it finds the difference at any bit position, it disables its drives at that bit position and for all lower-order bits.
- It does so by placing a 0 at the input of their drive. In our example, device detects a different on line ARB2 and hence it disables its drives on line ARB2, ARB1 and ARB0. This causes the code on the arbitration lines to change to 0110. This means that device B has won the race.
- The decentralized arbitration offers high reliability because operation of the bus is not dependent on any single device.



Direct Memory Access (DMA)****

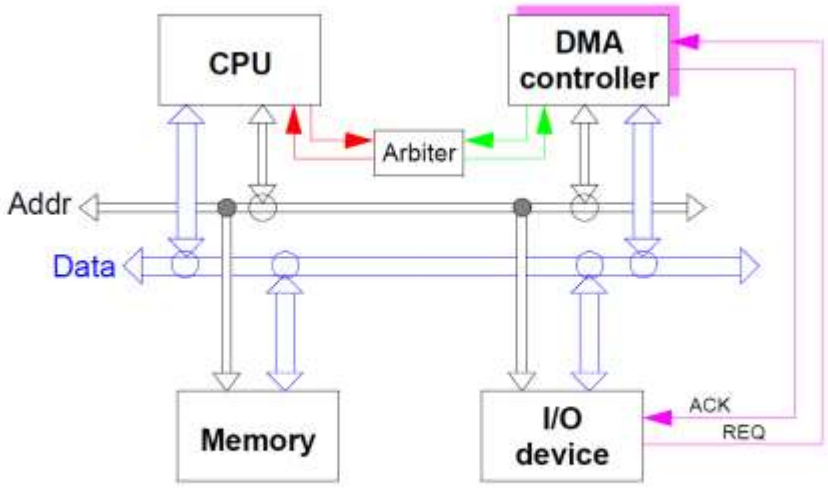
Let us look at direct memory transfer, because this is the most common form of having multiple bus master. In DMA is what we are bypassing the CPU to transfer the data from peripheral to the memory. And you are using a DMA controller and DMA controller is strictly speaking as single purpose processor which can take over the control of the bus. So, in the very common, DMA controller would be the most important other device to become bus master. So, microprocessor relinquishes control of the system bus to DMA controller and while DMA controller is carrying out the data transfer microprocessor can execute its regular program. So, what we have is no inefficient storing and restoring of state due to ISR call.

And time which goes in for this kind of state storage and retrieval, the interrupt latency overhead that gets completely eliminated if I use DMA for data transfer. Further, regular program need not wait unless it requires a system bus, because if it is a pipeline processing, if it already fetches the instructions. Then it can continue execution of its instructions while there is a data transfer on the external bus between memory and peripheral. And this is so, it is very interesting in the context of a Harvard architecture, because processor can fetch and execute instructions. Because its instruction memory is different from data memory and when you are transferring a data from peripheral you will be transferring the data to data memory and not to the instruction memory. In fact, this is another reason why we have found that to those microcontrollers which are targeted for embedded systems pickup Harvard architecture. Because you can have parallel data transfer from slow peripherals while execution of the code can really continue by fetching instruction from instruction memory.

DMA Basics: ***Definition: A direct memory access (DMA) is an operation in which data is copied (transported) from one resource to another resource in a computer system without the involvement of the CPU.***
The task of a DMA-controller (DMAC) is to execute the copy operation of data from one resource location to another. The copy of data can be performed from:

- I/O-device to memory
- memory to I/O-device
- memory to memory
- I/O-device to I/O-device

**********Below Figure is most important for DMA**********



simplified logical structure of a system with DMA

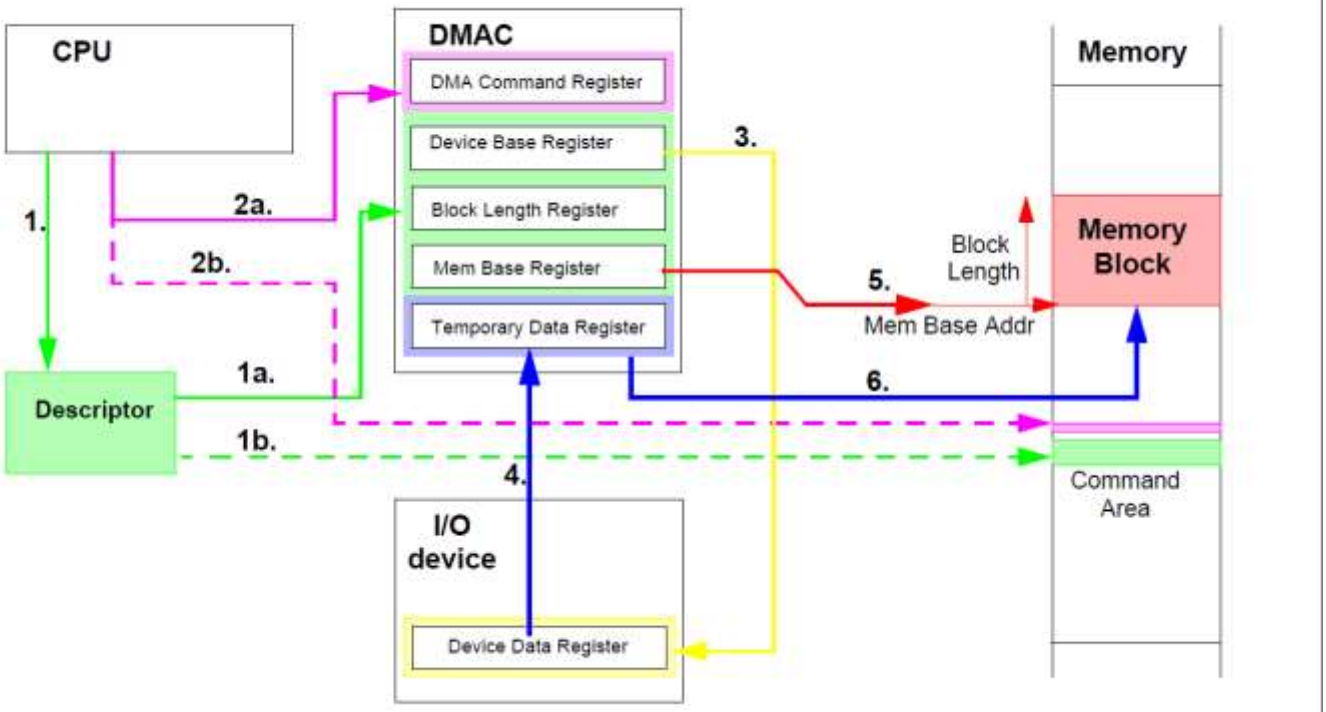
A DMAC is an independent (from CPU) resource of a computer system added for the concurrent xecution of DMA-operations. The first two operation modes are 'read from' and 'write to' transfers of an I/O-device to the main memory, which are the common operation of a DMA-controller. The other two operations are slightly more difficult to implement and most DMA-controllers do not implement device to device transfers.

The DMAC replaces the CPU for the transfer task of data from the I/O-device to the main memory (or vice versa) which otherwise would have been executed by the CPU using the programmed input output (PIO) mode. PIO is realized by a small instruction sequence executed by the processor to copy data. The 'memcpy' function supplied by the system is such a PIO operation. The DMAC is a master/slave resource on the system bus, because it must supply the addresses for the resources being involved in a DMA transfer. It requests the bus whenever a data value is available for transport, which is signaled from the device by the REQ signal. The functional unit DMAC may be integrated into other functional units in a computer system,
e.g. the memory controller, the south bridge, or directly into an I/O-device.

DMA Operations:
A lot of different operating modes exist for DMACs. The simplest one ist the single block transfer copying a block of data from a device to memory. For the more complex operations please refer to the literature [Mot81]. Here, only a short list of operating modes is given:

- single block transfer
- chained block transfers
- linked block transfers
- fly-by transfers

All these operations normally access the block of data in a linear sequence. Nevertheless, there are more usefull access functions possible, as there are: constant stride, constant stride with offset, incremental stride, ...



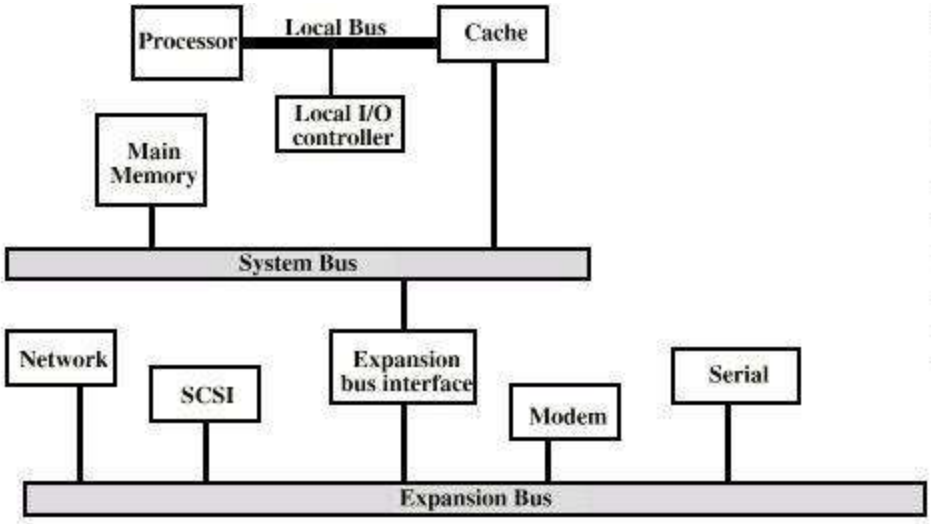
Execution of a DMA-operation (single block transfer)
The CPU prepares the DMA-operation by the construction of a descriptor (1), containing all necessary information for the DMAC to independently perform the DMA-operation (offload engine for data transfer). It initializes the operation by writing a command to a register in the DMAC (2a) or to a special assigned memory area (command area), where the DMAC can poll for the command and/or the descriptor (2b). Then the DMAC addresses the device data register (3) and read the data into a temporary data register (4). In another bus transfer cycle, it addresses the memory block (5) and writes the data from the temporary data register to the memory block (6). The DMAC increments the memory block address and continue with this loop until the block length is reached. The completion of the DMA operation is signaled to the processor by sending an IRQ signal or by setting a memory semaphore variable, which can be tested by the CPU. DMA control signals (REQ, ACK) are

used to signal the availability of values in the I/O device for transportation. DMAC is using bus bandwidth which may slow down processor execution by bus conflicts.

Multi-level Buses

Now, I had already talked about the disadvantages of the bus if there are various devices with various kind of latencies. So, one solution to this problem is called multilevel bus architectures. Because if you have one bus for all communication, peripherals would need high speed processors, specific bus interface which is not possible always to provide. Also the cost of the peripheral devices would increase, that means, it would lead to access power consumption and cost unless portable.

So, these you would not like to have. In fact, this is more true in the context of Embedded Systems. And also many peripherals putting on the same bus slows down the bus, so system. One solution is to have hierarchal bus structure what we call a processor local bus and a peripheral bus? So, processor local bus is where your microprocessor that is CPU, cache memory, memory controller, DMA controller all these things really set. And on the peripheral bus you have the peripherals. And the 2 buses communicate via bridge using expansion bus interface.



Processor local bus: High speed, wide bus and most frequent communication happens on this bus which connects microprocessor, cache and memory.

Peripheral bus: lower speed, narrow bus and less frequent communication happens on this bus which connects microprocessor, peripherals. Typically we follow some industry standards bus (ISA (Industry Standard Architecture, PCI) for the purpose of portability.

Bridge which is the single purpose processor which conveys communication between the buses. Like, in many a bus standards, it will actually accumulate number of transactions from the slower bus and then post it to the high speed bus.

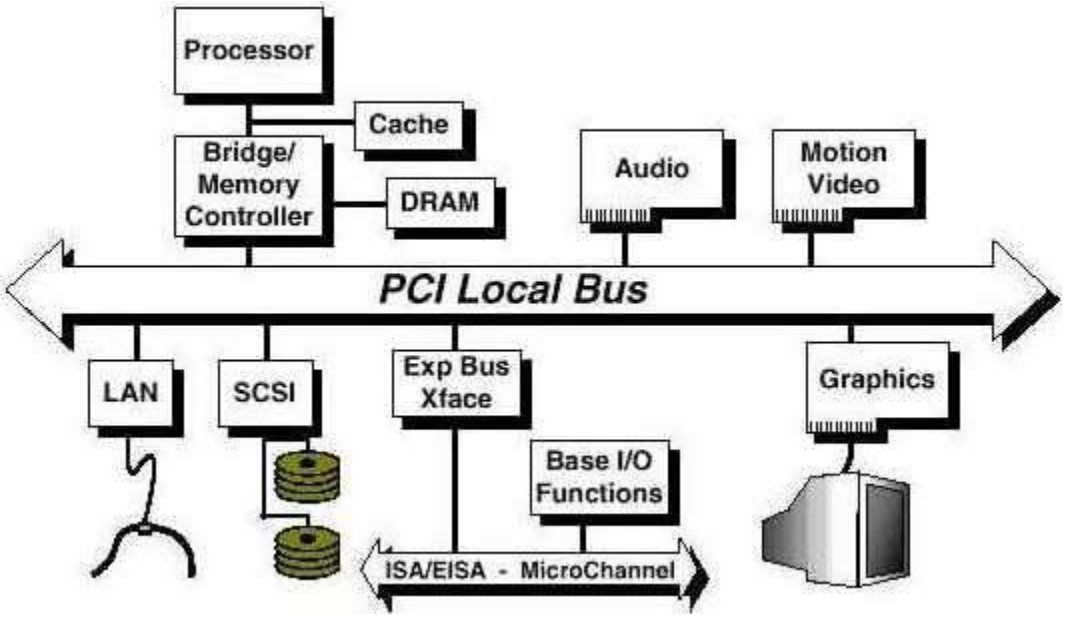
PCI - Peripheral Component Interconnect****

The most well-known and the common hierarchal bus structure is PCI - Peripheral Component Interconnect bus because it can have a data transfer rates of 127.2 to 508.6Mbits/s and 32 bit addressing. High performance bus originated at Intel in the early 1990's. Standard adopted by industry and administered by PCISIG (PCI Special Interest Group. Interconnects chips, expansion boards, processor memory subsystems. It uses synchronous bus architecture, but still it has got multiplexed data or address lines.

Basic protocol is based on the architecture. In the architecture we have got the basic CPU or processor bus which is high speed bus then we have got PCI bus as well as we have got ISA bus which we are talking so far for connecting the low speed device. And between each one of them you have got the corresponding bridge. There is the host bridge between CPU bus which is the high speed and that of the PCI bus the basic bus and that of the ISA Bridge for bridging between PCI and ISA bus. Obviously, L2 cache is sit on the CPU bus, but for

trying to write onto the DRAM it will go via the host bridge because DRAM is a slower device. So, L2 and L1 cache are on the bus, CPU bus, but ROM & DRAM are connected via PCI host bridge, because they are slower device.

**********Below Figure is most important for PCI**********



PCI Read/Write Transactions:
FRAME # de-asserted when master intends to complete only one more data transfer

So, what we say the whole signals in the PCI bus sampled on rising edge and follows a centralized parallel arbitration. And all transfers nothing but bursts and address phase starts by asserting FRAME. Frame is a basic signal and next cycle initiates asserts common address. That means, it is something like a complete cycle transfer. Data transfers are done with regard to 2 signals IRDY and TRDY. When IRDY is asserted then the master is ready to transfer. TRDY is asserted when the target is ready to transfer and transfer when both asserted on rising edge. I hope you can understand that the master is ready to transfer and the device has to agree then only you can have the transfer. The frame is de-asserted when master intends to complete only one more data transfer.

What is really a bridge? A bridge is a slave on the fast bus and master of the slow bus, because bridge is actually sitting on both the buses. And if we look at a typical master slave configuration then the CPU will be the master for the faster bus and bridge will be slave over there. So, the bridge will take commands from the fast bus which is our CPU and it will become master in the slower bus for carrying out the transactions read and write transactions with the slower devices. So, it will issue command to the slower bus, effectively it can become a master on the slow bus. It also functions as what is call a protocol translator because I can have the 2 buses working with different transaction protocols. So, request from one bus has to be translated into a request compatible with the protocol of the slower bus. So, the translation is also carried out by the bridge. In fact, bridge actually an additional hardware. In very simple case, a bridge would be a simply implementing a protocol machine that is a finite machine, but in more complex cases like PCI bridge, it will as good as that of a controller with it's own registers and control logic.

Introduction to the AMBA Buses***

1.1 Overview of the AMBA specification**

The *Advanced Microcontroller Bus Architecture* (AMBA) specification defines an on chip communications standard for designing high-performance embedded microcontrollers.

Three distinct buses are defined within the AMBA specification:****

- the *Advanced High-performance Bus* (AHB)
- the *Advanced Peripheral Bus* (APB).
- the *Advanced System Bus* (ASB)

A test methodology is included with the AMBA specification which provides an infrastructure for modular macro-cell test and diagnostic access.

1.1.1 Advanced High-performance Bus (AHB)**

The AMBA AHB is for high-performance, high clock frequency system modules. The AHB acts as the high-performance system *backbone* bus. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macro-cell functions. AHB is also specified to ensure ease of use in an efficient design flow using synthesis and automated test techniques.

1.1.2 Advanced System Bus (ASB)**

The AMBA ASB is for high-performance system modules. AMBA ASB is an alternative system bus suitable for use where the high-performance features of AHB are not required. ASB also supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macro-cell functions.

1.1.3 Advanced Peripheral Bus (APB)****

The AMBA APB is for low-power peripherals. AMBA APB is optimized for minimal power consumption and reduced interface complexity to support peripheral functions. APB can be used in conjunction with either version of the system bus.

**********The Below Figure is most Important for AMBA**********

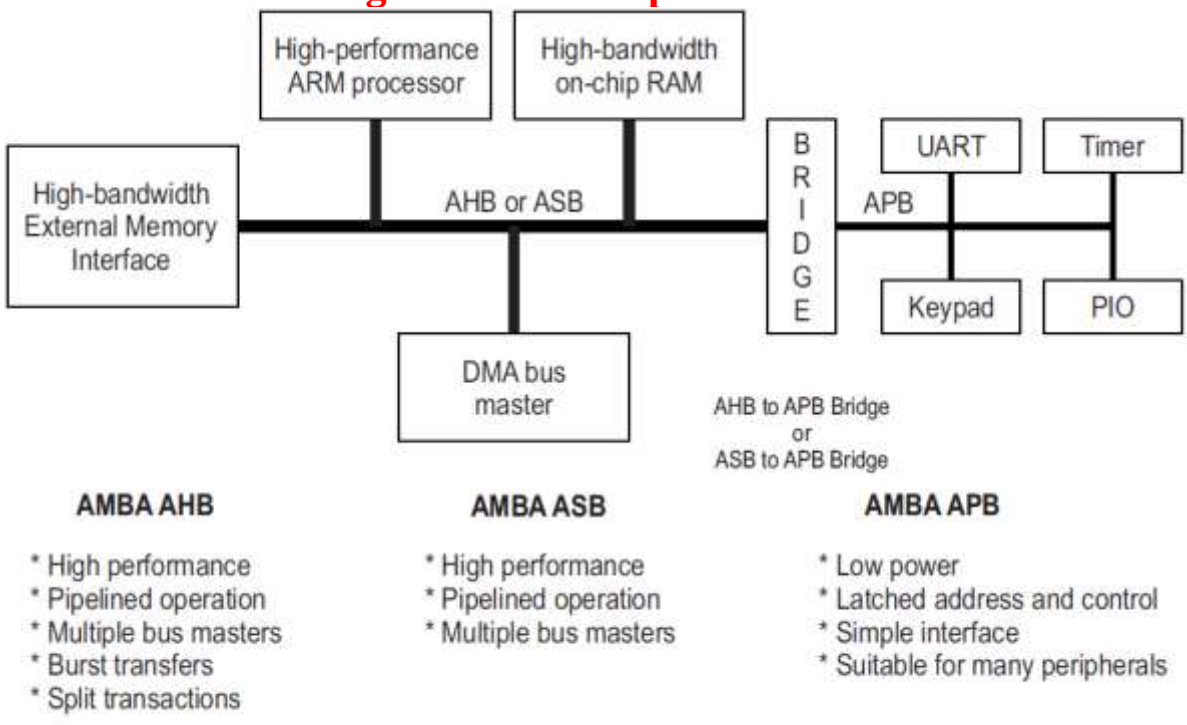


Figure 1-1 A typical AMBA system

1.2 Objectives of the AMBA specification

The AMBA specification has been derived to satisfy four key requirements:

- to facilitate the *right-first-time* development of embedded microcontroller products with one or more CPUs or signal processors
- to be *technology-independent* and ensure that highly reusable peripheral and system macro-cells can be migrated across a diverse range of IC processes and be appropriate for full-custom, standard cell and gate array technologies
- to encourage *modular system design* to improve processor independence, providing a development road-map for advanced cached CPU cores and the development of peripheral libraries
- to minimize the silicon infrastructure required to support efficient on-chip and off-chip communication for both operation and manufacturing test.

1.3 A typical AMBA-based microcontroller

An AMBA-based microcontroller typically consists of a high-performance system *backbone* bus (AMBA AHB or AMBA ASB), able to sustain the external memory bandwidth, on which the CPU, on-chip memory and other *Direct Memory Access* (DMA) devices reside. This bus provides a high-bandwidth interface between the elements that are involved in the majority of transfers. Also located on the high performance bus is a bridge to the lower bandwidth APB, where most of the peripheral devices in the system are located (see Figure 1-1).

AMBA APB provides the basic peripheral macrocell communications infrastructure as a secondary bus from the higher bandwidth pipelined main system bus. Such peripherals typically:

- have interfaces which are memory-mapped registers
- are accessed under programmed control.
- have no high-bandwidth interfaces

The external memory interface is application-specific and may only have a narrow data path, but may also support a test access mode which allows the internal AMBA AHB, ASB and APB modules to be tested in isolation with system-independent test sets.

1.4 Terminology

The following terms are used throughout this specification.

- **Bus cycle** A bus cycle is a basic unit of one bus clock period and for the purpose of AMBA AHB or APB protocol descriptions is defined from rising-edge to rising-edge transitions. An ASB bus cycle is defined from falling-edge to falling-edge transitions. Bus signal timing is referenced to the bus cycle clock.
- **Bus transfer** An AMBA ASB or AHB bus transfer is a read or write operation of a data object, which may take one or more bus cycles. The bus transfer is terminated by a *completion* response from the addressed slave.The transfer sizes supported by AMBA ASB include byte (8-bit), halfword (16-bit) and word (32-bit). AMBA AHB additionally supports wider data transfers, including 64-bit and 128-bit transfers. An AMBA APB bus transfer is a read or write operation of a data object, which always requires two bus cycles.
- **Burst operation** A burst operation is defined as one or more data transactions, initiated by a bus master, which have a consistent width of transaction to an incremental region of address space. The increment step per transaction is determined by the width of transfer (byte, halfword, word). No burst operation is supported on the APB.

1.5 Introducing the AMBA AHB

AHB is a new generation of AMBA bus which is intended to address the requirements of high-performance synthesizable designs. It is a high-performance system bus that supports multiple bus masters and provides high-bandwidth operation. AMBA AHB implements the features required for high-performance, high clock frequency systems including:

- burst transfers
- split transactions
- single-cycle bus master handover
- single-clock edge operation
- non-tristate implementation
- wider data bus configurations (64/128 bits).

Bridging between this higher level of bus and the current ASB/APB can be done efficiently to ensure that any existing designs can be easily integrated. An AMBA AHB design may contain one or more bus masters, typically a system would contain at least the processor and test interface. However, it would also be common for a *Direct Memory Access* (DMA) or *Digital Signal Processor* (DSP) to be included as bus masters. The external memory interface, APB bridge and any internal memory are the most common AHB slaves. Any other peripheral in the system could also be included as an AHB slave. However, low-bandwidth peripherals typically reside on the APB.

A typical AMBA AHB system design contains the following components:

AHB master A bus master is able to initiate read and write operations by providing an address and control information. Only one bus master is allowed to actively use the bus at any one time.

AHB slave A bus slave responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.

AHB arbiter The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as *highest priority* or *fair* access can be implemented depending on the application requirements. An AHB would include only one arbiter, although this would be trivial in single bus master systems.

AHB decoder The AHB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. A single centralized decoder is required in all AHB implementations.

Bus interconnection:

The AMBA AHB bus protocol is designed to be used with a central multiplexor interconnection scheme. Using this scheme all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexor, which selects the appropriate signals from the slave that is involved in the transfer.
Figure 3-2 illustrates the structure required to implement an AMBA AHB design with three masters and four slaves.

3.3 Overview of AMBA AHB operation

Before an AMBA AHB transfer can commence the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus.

A granted bus master starts an AMBA AHB transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as an indication if the transfer forms part of a burst.

Two different forms of burst transfers are allowed:

- incrementing bursts, which do not wrap at address boundaries
- wrapping bursts, which wrap at particular address boundaries.

A write data bus is used to move data from the master to a slave, while a read data bus is used to move data from a slave to the master.

Every transfer consists of:

- an address and control cycle
- one or more cycles for the data.

**********Below Figure is most important for AHB Operation**********

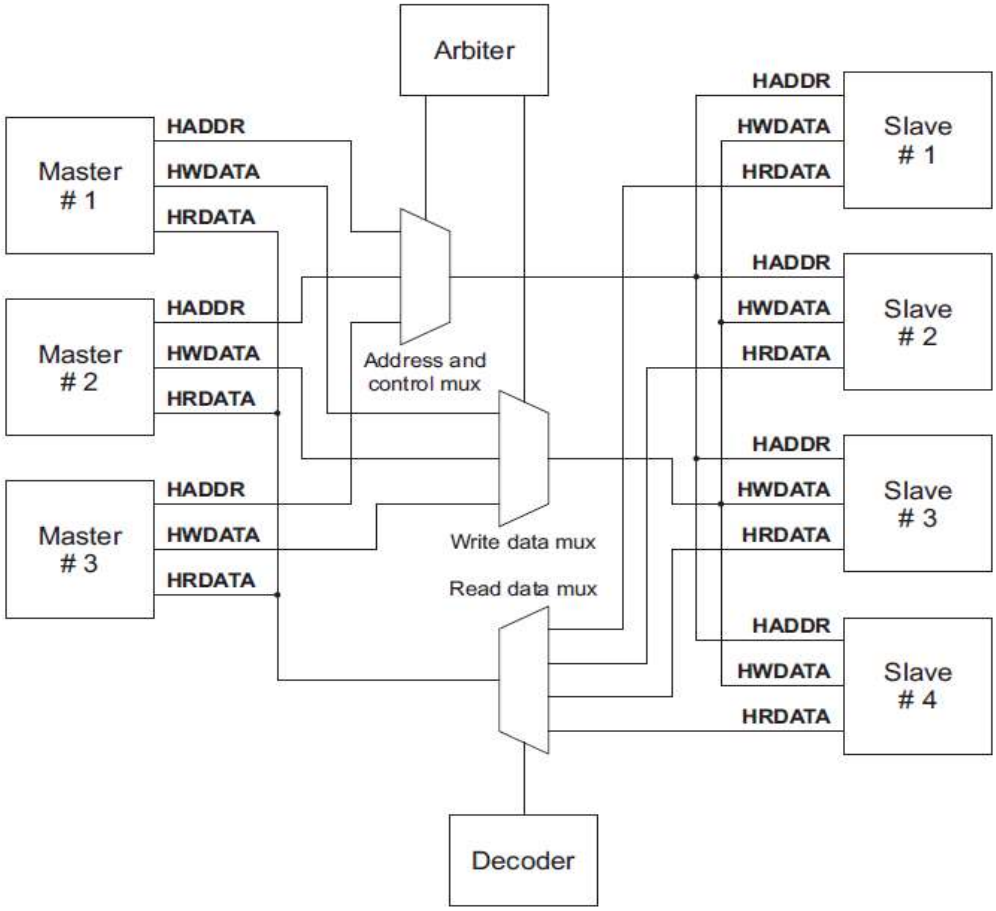


Figure 3-2 Multiplexor interconnection

The address cannot be extended and therefore all slaves must sample the address during this time. The data, however, can be extended using the **HREADY** signal. When LOW this signal causes wait states to be inserted into the transfer and allows extra time for the slave to provide or sample data.
During a transfer the slave shows the status using the response signals, **HRESP[1:0]**:

OKAY	The OKAY response is used to indicate that the transfer is progressing normally and when **HREADY** goes HIGH this shows the transfer has completed successfully.
ERROR	The ERROR response indicates that a transfer error has occurred and the transfer has been unsuccessful.
RETRY and SPLIT	Both the RETRY and SPLIT transfer responses indicate that the transfer cannot complete immediately, but the bus master should continue to attempt the transfer.

In normal operation a master is allowed to complete all the transfers in a particular burst before the arbiter grants another master access to the bus. However, in order to avoid excessive arbitration latencies it is possible for the

arbiter to break up a burst and in such cases the master must re-arbitrate for the bus in order to complete the remaining transfers in the burst.

1.6 Introducing the AMBA ASB

ASB is the first generation of AMBA system bus. ASB sits above the current APB and implements the features required for high-performance systems including:

• burst transfers • pipelined transfer operation • multiple bus master.

A typical AMBA ASB system may contain one or more bus masters. For example, at least the processor and test interface. However, it would also be common for a *Direct Memory Access* (DMA) or *Digital Signal Processor* (DSP) to be included as bus masters.

The external memory interface, APB bridge and any internal memory are the most common ASB slaves. Any other peripheral in the system could also be included as an ASB slave. However, low-bandwidth peripherals typically reside on the APB.

An AMBA ASB system design typically contains the following components:

ASB master A bus master is able to initiate read and write operations by providing an address and control information. Only one bus master is allowed to actively use the bus at any one time.

ASB slave A bus slave responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.

ASB decoder The bus decoder performs the decoding of the transfer addresses and selects slaves appropriately. The bus decoder also ensures that the bus remains operational when no bus transfers are required. A single centralized decoder is required in all ASB implementations.

ASB arbiter The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as *highest priority* or *fair* access can be implemented depending on the application requirements.An ASB would include only one arbiter, although this would be trivial in single bus master systems.

1.7 Introducing the AMBA APB

The APB is part of the AMBA hierarchy of buses and is optimized for minimal power consumption and reduced interface complexity. The AMBA APB appears as a local secondary bus that is encapsulated as a single AHB or ASB slave device. APB provides a low-power extension to the system bus which builds on AHB or ASB signals directly. The APB Bridge appears as a slave module which handles the bus handshake and control signal retiming on behalf of the local peripheral bus. By defining the APB interface from the starting point of the system bus, the benefits of the system diagnostics and test methodology can be exploited. The AMBA APB should be used to interface to any peripherals which are low bandwidth and do not require the high performance of a pipelined bus interface. The latest revision of the APB is specified so that all signal transitions are only related to the rising edge of the clock.

This improvement ensures the APB peripherals can be integrated easily into any design flow, with the following advantages:

- high-frequency operation easier to achieve
- performance is independent of the mark-space ratio of the clock
- static timing analysis is simplified by the use of a single clock edge
- no special considerations are required for automatic test insertion
- many *Application Specific Integrated Circuit* (ASIC) libraries have a better selection of rising edge registers
- easy integration with cycle-based simulators.

These changes to the APB also make it simpler to interface it to the new AHB. An AMBA APB implementation typically contains a single APB bridge which is required to convert AHB or ASB transfers into a suitable format for the slave devices on the APB. The bridge provides latching of all address, data and control signals, as well as providing a second level of decoding to generate slave select signals for the APB peripherals. All other modules on the APB are APB slaves.

The APB slaves have the following interface specification:

- address and control valid throughout the access (unpipelined)
- zero-power interface during non-peripheral bus activity (peripheral bus is static when not in use)
- timing can be provided by decode with strobe timing (unclocked interface)
- write data valid for the whole access (allowing glitch-free transparent latch implementations).

1.8 Choosing the right bus for your system

Before deciding on which bus or buses you should use in your system, you should consider the following:

- *Choice of system bus*
- *System bus and peripheral bus*
- *When to use AMBA AHB/ASB or APB* on page 1-13

1.8.1 Choice of system bus

Both AMBA AHB and ASB are available for use as the main system bus. Typically the choice of system bus will depend on the interface provided by the system modules required.

The AHB is recommended for all new designs, not only because it provides a higherbandwidth solution, but also because the single-clock-edge protocol results in a smoother integration with design automation tools used during a typical ASIC development.

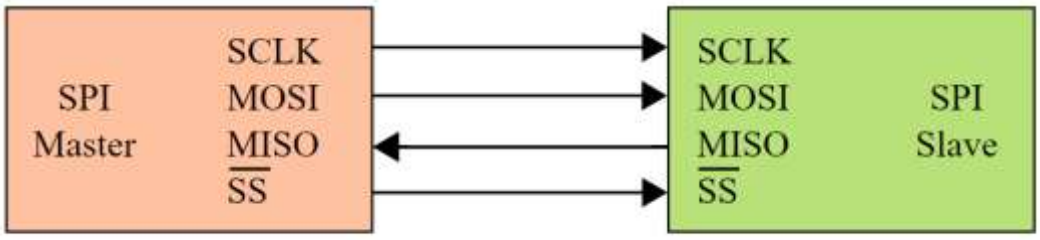
Serial Protocols: SPI and I2C

SPI:******

The **Serial Peripheral Interface bus** (SPI) is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. The interface was developed by Motorola in the late 1980s and has become a *de facto* standard. Typical applications include Secure Digital cards and liquid crystal displays.

SPI devices communicate in full duplex mode using a master-slave architecture with a single master. The master device originates the frame for reading and writing. Multiple slave devices are supported through selection with individual slave select (SS) lines.

**********Below Figure is most important for SPI**********



Single Master to Single Slave: basic SPI bus example

Sometimes SPI is called a *four-wire* serial bus, contrasting with three-, two-, and one-wire serial buses. The SPI may be accurately described as a synchronous serial interface, but it is different from the Synchronous Serial Interface (SSI) protocol, which is also a four-wire synchronous serial communication protocol. SSI Protocol employs differential signaling and provides only a single simplex communication channel.

The SPI bus specifies five logic signals:

- SCLK: Serial Clock (output from master).
- MOSI: Master Output Slave Input, or Master Out Slave In (data output from master).
- MISO: Master Input Slave Output, or Master In Slave Out (data output from slave).
- SDIO: Serial Data I/O (bidirectional I/O)
- SS: Slave Select (often active low, output from master).

While the above pin names are the most popular, in the past alternative pin naming conventions were sometimes used, and so SPI port pin names for older IC products may differ from those depicted in these illustrations:

Serial Clock:

- SCLK: SCK.

Master Output → Slave Input:

- MOSI: SIMO, SDO, DI, DIN, SI, MTSR.

Master Input ← Slave Output:

- MISO: SOMI, SDI, DO, DOUT, SO, MRST.

Serial Data I/O (bidirectional):

- SDIO: SIO

Slave Select:

- SS: $\overline{SS}$, SSEL, CS, $\overline{CS}$, CE, nSS, /SS, SS#.

The MOSI/MISO convention requires that, on devices using the alternate names, SDI on the master be connected to SDO on the slave, and vice versa. Slave Select is the same functionality as chip select and is used instead of an addressing concept. Pin names are always capitalized as in Slave Select, Serial Clock, and Master Output Slave Input.

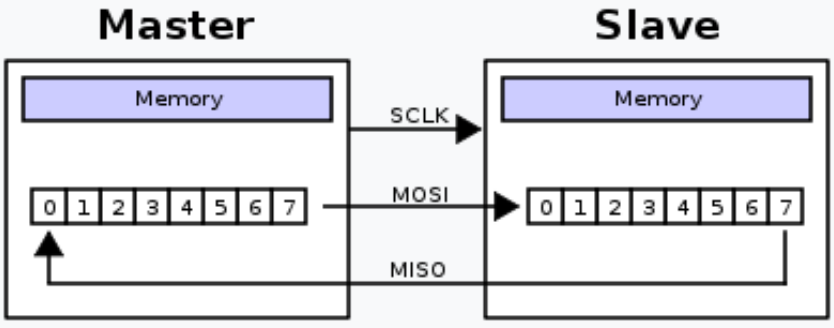
Operation

The SPI bus can operate with a single master device and with one or more slave devices.

If a single slave device is used, the SS pin *may* be fixed to logic low if the slave permits it. Some slaves require a falling edge of the chip select signal to initiate an action. An example is the Maxim MAX1242 ADC, which starts conversion on a high→low transition. With multiple slave devices, an independent SS signal is required from the master for each slave device.

Most slave devices have tri-state outputs so their MISO signal becomes high impedance (*logically disconnected*) when the device is not selected. Devices without tri-state outputs cannot share SPI bus segments with other devices; only one such slave could talk to the master.

Data transmission:



A typical hardware setup using two shift registers to form an inter-chip circular buffer

To begin communication, the bus master configures the clock, using a frequency supported by the slave device, typically up to a few MHz. The master then selects the slave device with a logic level 0 on the select line. If a waiting period is required, such as for an analog-to-digital conversion, the master must wait for at least that period of time before issuing clock cycles.

During each SPI clock cycle, a full duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it. This sequence is maintained even when only one-directional data transfer is intended.

Transmissions normally involve two shift registers of some given word size, such as eight bits, one in the master and one in the slave; they are connected in a virtual ring topology. Data is usually shifted out with the most-significant bit first, while shifting a new least-significant bit into the same register. At the same time, Data from the counterpart is shifted into the least-significant bit register. After the register bits have been shifted out and in, the master and slave have exchanged register values. If more data needs to be exchanged, the shift registers are reloaded and the process repeats. Transmission may continue for any number of clock cycles. When complete, the master stops toggling the clock signal, and typically deselects the slave.

Transmissions often consist of 8-bit words. However, other word sizes are also common, for example, 16-bit words for touch screen controllers or audio codecs, such as the TSC2101 by Texas Instruments, or 12-bit words for many digital-to-analog or analog-to-digital converters.

Every slave on the bus that has not been activated using its chip select line must disregard the input clock and MOSI signals, and must not drive MISO.

Mode numbers:

The combinations of polarity and phases are often referred to as modes which are commonly numbered according to the following convention, with CPOL as the high order bit and CPHA as the low order bit:

For "Microchip PIC" / "ARM-based" microcontrollers (note that NCPHA is the inversion of CPHA):

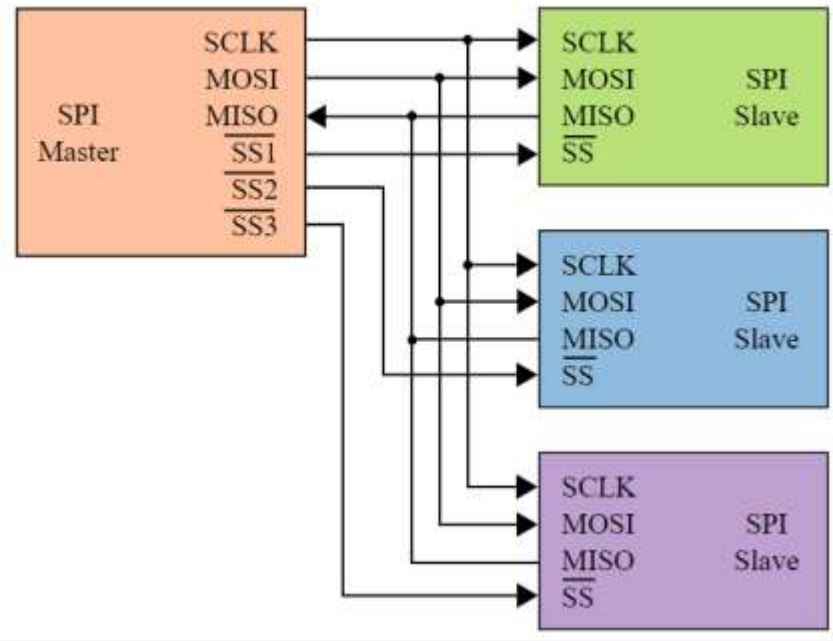
SPI Mode	Clock Polarity (CPOL/CKP)	Clock Phase (CPHA)	Clock Edge (CKE/NCPHA)
0	0	0	1
1	0	1	0
2	1	0	1
3	1	1	0

For PIC32MX : SPI mode configure CKP,CKE and SMP bits.Set SMP bit,and CKP,CKE two bits configured as above table.

Independent slave configuration:

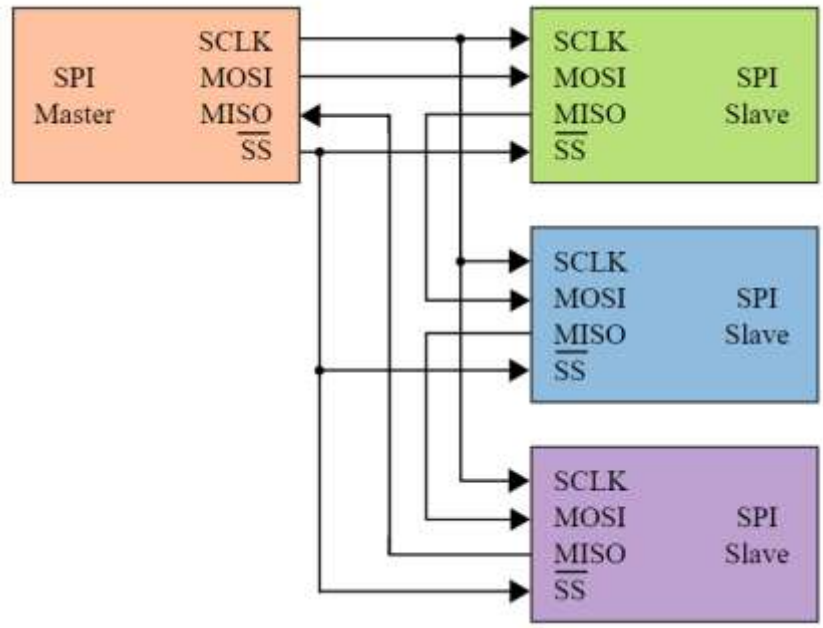
In the independent slave configuration, there is an independent chip select line for each slave. A pull-up resistor between power source and chip select line is highly recommended for each independent device to reduce cross-talk between devices. This is the way SPI is normally used. Since the MISO pins of the slaves are connected together, they are required to be tri-state pins (high, low or high-impedance).

**********Below Figure is most important for SPI**********



Typical SPI bus: master and three independent slaves

Daisy chain configuration:



Daisy-chained SPI bus: master and cooperative slaves

Some products that implement SPI may be connected in a daisy chain configuration, the first slave output being connected to the second slave input, etc. The SPI port of each slave is designed to send out during the second group of clock pulses an exact copy of the data it received during the first group of clock pulses. The whole chain acts as a communication shift register; daisy chaining is often done with shift registers to provide a bank of inputs or outputs through SPI. Such a feature only requires a single SS line from the master, rather than a separate SS line for each slave. Applications that require a daisy chain configuration include SGPIO and JTAG.

Valid communications:

Some slave devices are designed to ignore any SPI communications in which the number of clock pulses is greater than specified. Others do not care, ignoring extra inputs and continuing to shift the same output bit. It is common for different devices to use SPI communications with different lengths, as, for example, when SPI is used to access the scan chain of a digital IC by issuing a command word of one size (perhaps 32 bits) and then getting a response of a different size (perhaps 153 bits, one for each pin in that scan chain).

Interrupts:

SPI devices sometimes use another signal line to send an interrupt signal to a host CPU. Examples include pen-down interrupts from touchscreen sensors, thermal limit alerts from temperature sensors, alarms issued by real time clock chips, SDIO,[5] and headset jack insertions from the sound codec in a cell phone. Interrupts are not covered by the SPI standard; their usage is neither forbidden nor specified by the standard.

Example of bit-banging the master protocol:

Below is an example of bit-banging the SPI protocol as an SPI master with CPOL=0, CPHA=0, and eight bits per transfer. The example is written in the C programming language. Because this is CPOL=0 the clock must be pulled low before the chip select is activated. The chip select line must be activated, which normally means being toggled low, for the peripheral before the start of the transfer, and then deactivated afterward. Most peripherals allow or require several transfers while the select line is low; this routine might be called several times before deselecting the chip.

Pros and cons

Advantages:

- Full duplex communication in the default version of this protocol.
- Push-pull drivers (as opposed to open drain) provide good signal integrity and high speed
- Higher throughput than I²C or SMBus
- Complete protocol flexibility for the bits transferred
  - Not limited to 8-bit words
  - Arbitrary choice of message size, content, and purpose
- Extremely simple hardware interfacing
  - Typically lower power requirements than I²C or SMBus due to less circuitry (including pull up resistors)
  - No arbitration or associated failure modes
  - Slaves use the master's clock and do not need precision oscillators
  - Slaves do not need a unique address — unlike I²C or GPIB or SCSI
  - Transceivers are not needed
- Uses only four pins on IC packages, and wires in board layouts or connectors, much fewer than parallel interfaces
- At most one unique bus signal per device (chip select); all others are shared
- Signals are unidirectional allowing for easy Galvanic isolation
- Not limited to any maximum clock speed, enabling potentially high speed
- Simple software implementation

Disadvantages:

- Requires more pins on IC packages than I²C, even in the *three-wire* variant
- No in-band addressing; out-of-band chip select signals are required on shared buses
- No hardware flow control by the slave (but the master can delay the next clock edge to slow the transfer rate)
- No hardware slave acknowledgment (the master could be transmitting to nowhere and not know it)
- Typically supports only one master device (depends on device's hardware implementation)
- No error-checking protocol is defined

- Without a formal standard, validating conformance is not possible
- Only handles short distances compared to RS-232, RS-485, or CAN-bus. (its distance can be extended with use of transceivers like RS-422)
- Many existing variations, making it difficult to find development tools like host adapters that support those variations
- SPI does not support hot swapping (dynamically adding nodes).
- Interrupts must either be implemented with out-of-band signals or be faked by using periodic polling similarly to USB 1.1 and 2.0
- Some variants like Multi I/O SPI and three-wire serial buses defined below are half-duplex.

Applications:

The board real estate savings compared to a parallel I/O bus are significant, and have earned SPI a solid role in embedded systems. That is true for most system-on-a-chip processors, both with higher end 32-bit processors such as those using ARM, MIPS, or PowerPC and with other microcontrollers such as the AVR, PIC, and MSP430. These chips usually include SPI controllers capable of running in either master or slave mode. In-system programmable AVR controllers (including blank ones) can be programmed using an SPI interface.[6]

Chip or FPGA based designs sometimes use SPI to communicate between internal components; on-chip real estate can be as costly as its on-board cousin.

The full-duplex capability makes SPI very simple and efficient for single master/single slave applications. Some devices use the full-duplex mode to implement an efficient, swift data stream for applications such as digital audio, digital signal processing, or telecommunications channels, but most off-the-shelf chips stick to half-duplex request/response protocols.

SPI is used to talk to a variety of peripherals, such as

- Sensors: temperature, pressure, ADC, touchscreens, video game controllers
- Control devices: audio codecs, digital potentiometers, DAC
- Camera lenses: Canon EF lens mount
- Communications: Ethernet, USB, USART, CAN, IEEE 802.15.4, IEEE 802.11, handheld video games
- Memory: flash and EEPROM
- Real-time clocks
- LCD, sometimes even for managing image data
- Any MMC or SD card (including SDIO variant)

For high-performance systems, FPGAs sometimes use SPI to interface as a slave to a host, as a master to sensors, or for flash memory used to bootstrap if they are SRAM-based.

Although there are some similarities between the SPI bus and the JTAG (IEEE 1149.1-2013) protocol, they are not interchangeable. The SPI bus is intended for high speed, on board initialization of device peripherals, while the JTAG protocol is intended to provide reliable test access to the I/O pins from an off board controller with less precise signal delay and skew parameters. While not strictly a level sensitive interface, the JTAG protocol supports the recovery of both setup and hold violations between JTAG devices by reducing the clock rate or changing the clock's duty cycles. Consequently, the JTAG interface is not intended to support extremely high data rates. SGPIO is essentially another (incompatible) application stack for SPI designed for particular backplane management activities. SGPIO uses 3-bit messages.

I2C:******

I²C (**Inter-Integrated Circuit**), pronounced *I-squared-C* or *I-two-C*, is a multi-master, multi-slave, packet switched, single-ended, serial computer bus invented by Philips Semiconductor (now NXP Semiconductors). It is typically used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication. Alternatively I²C is spelled **I2C** (pronounced I-two-C) or **IIC** (pronounced I-I-C).

Since October 10, 2006, no licensing fees are required to implement the I²C protocol. However, fees are required to obtain I²C slave addresses allocated by NXP.

Several competitors, such as Siemens AG (later Infineon Technologies AG, now Intel mobile communications), NEC, Texas Instruments, STMicroelectronics (formerly SGS-Thomson), Motorola (later Freescale, now merged with NXP), Nordic Semiconductor and Intersil, have introduced compatible I²C products to the market since the mid-1990s.

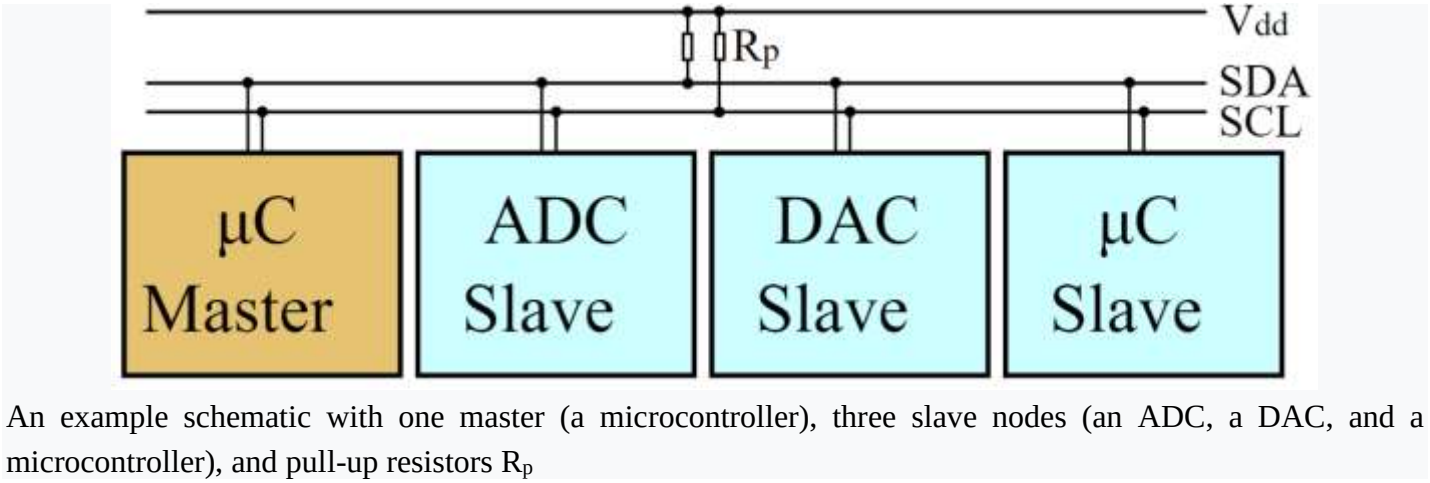
SMBus, defined by Intel in 1995, is a subset of I²C, defining a stricter usage. One purpose of SMBus is to promote robustness and interoperability. Accordingly, modern I²C systems incorporate some policies and rules from SMBus, sometimes supporting both I²C and SMBus, requiring only minimal reconfiguration either by commanding or output pin use.

Design

I²C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors. Typical voltages used are +5 V or +3.3 V, although systems with other voltages are permitted.

The I²C reference design has a 7-bit or a 10-bit (depending on the device used) address space. Common I²C bus speeds are the 100 kbit/s *standard mode* and the 10 kbit/s *low-speed mode*, but arbitrarily low clock frequencies are also allowed. Recent revisions of I²C can host more nodes and run at faster speeds (400 kbit/s *Fast mode*, 1 Mbit/s *Fast mode plus* or Fm+, and 3.4 Mbit/s *High Speed mode*). These speeds are more widely used on embedded systems than on PCs. There are also other features, such as 16-bit addressing.

**********Below Figure is most important for I2C**********



An example schematic with one master (a microcontroller), three slave nodes (an ADC, a DAC, and a microcontroller), and pull-up resistors R_p

Note the bit rates are quoted for the transactions between master and slave without clock stretching or other hardware overhead. Protocol overheads include a slave address and perhaps a register address within the slave device, as well as per-byte ACK/NACK bits. Thus the actual transfer rate of user data is lower than those peak bit rates alone would imply. For example, if each interaction with a slave inefficiently allows only 1 byte of data to be transferred, the data rate will be less than half the peak bit rate.

The maximal number of nodes is limited by the address space and also by the total bus capacitance of 400 pF, which restricts practical communication distances to a few meters. The relatively high impedance and low noise immunity requires a common ground potential, which again restricts practical use to communication within the same PC board or small system of boards.

Reference design:

The aforementioned reference design is a bus with a clock (SCL) and data (SDA) lines with 7-bit addressing. The bus has two roles for nodes: master and slave:

- Master node – node that generates the clock and initiates communication with slaves.
- Slave node – node that receives the clock and responds when addressed by the master.

The bus is a multi-master bus, which means that any number of master nodes can be present. Additionally, master and slave roles may be changed between messages (after a STOP is sent).

There may be four potential modes of operation for a given bus device, although most devices only use a single role and its two modes:

- master transmit – master node is sending data to a slave,
- master receive – master node is receiving data from a slave,
- slave transmit – slave node is sending data to the master,
- slave receive – slave node is receiving data from the master.

The master is initially in master transmit mode by sending a start bit followed by the 7-bit address of the slave it wishes to communicate with, which is finally followed by a single bit representing whether it wishes to write (0) to or read (1) from the slave.

If the slave exists on the bus then it will respond with an ACK bit (active low for acknowledged) for that address. The master then continues in either transmit or receive mode (according to the read/write bit it sent), and the slave continues in its complementary mode (receive or transmit, respectively).

The address and the data bytes are sent most significant bit first. The start bit is indicated by a high-to-low transition of SDA with SCL high; the stop bit is indicated by a low-to-high transition of SDA with SCL high. All other transitions of SDA take place with SCL low.

If the master wishes to write to the slave, then it repeatedly sends a byte with the slave sending an ACK bit. (In this situation, the master is in master transmit mode, and the slave is in slave receive mode.)

If the master wishes to read from the slave, then it repeatedly receives a byte from the slave, the master sending an ACK bit after every byte except the last one. (In this situation, the master is in master receive mode, and the slave is in slave transmit mode.)

The master then either ends transmission with a stop bit, or it may send another START bit if it wishes to retain control of the bus for another transfer (a "combined message").

Message protocols:

I²C defines basic types of messages, each of which begins with a START and ends with a STOP:

- Single message where a master writes data to a slave.
- Single message where a master reads data from a slave.
- Combined messages, where a master issues at least two reads or writes to one or more slaves.

In a combined message, each read or write begins with a START and the slave address. After the first START in a combined message these are also called *repeated START* bits. Repeated START bits are not preceded by STOP bits, which is how slaves know that the next transfer is part of the same message.

Any given slave will only respond to certain messages, as specified in its product documentation.

Pure I²C systems support arbitrary message structures. SMBus is restricted to nine of those structures, such as *read word N* and *write word N*, involving a single slave. PMBusextends SMBus with a *Group* protocol, allowing multiple such SMBus transactions to be sent in one combined message. The terminating STOP indicates when those grouped actions should take effect. For example, one PMBus operation might reconfigure

three power supplies (using three different I²C slave addresses), and their new configurations would take effect at the same time: when they receive that STOP.

With only a few exceptions, neither I²C nor SMBus define message semantics, such as the meaning of data bytes in messages. Message semantics are otherwise product-specific. Those exceptions include messages addressed to the I²C *general call* address (0x00) or to the SMBus *Alert Response Address*; and messages involved in the SMBus *Address Resolution Protocol* (ARP) for dynamic address allocation and management.

In practice, most slaves adopt request-response control models, where one or more bytes following a write command are treated as a command or address. Those bytes determine how subsequent written bytes are treated or how the slave responds on subsequent reads. Most SMBus operations involve single-byte commands.

Messaging example: 24c32 EEPROM:

One specific example is the 24c32 type EEPROM, which uses two request bytes that are called Address High and Address Low. (Accordingly, these EEPROMs are not usable by pure SMBus hosts, which only support single-byte commands or addresses.) These bytes are used to address bytes within the 32 kbit (4 kB) supported by that EEPROM; the same two-byte addressing is also used by larger EEPROMs, such as 24c512 ones storing 512 kbits (64 kB). Writing and reading data to these EEPROMs uses a simple protocol: the address is written, and then data is transferred until the end of the message. (That data transfer part of the protocol also makes trouble for SMBus, since the data bytes are not preceded by a count, and more than 32 bytes can be transferred at once. I²C EEPROMs smaller than 32 kbit, such as 2 kbit 24c02 ones, are often used on SMBus with inefficient single-byte data transfers.)

A single message writes to the EEPROM. After the START, the master sends the chip's bus address with the direction bit clear (*write*), then sends the two-byte address of data within the EEPROM and then sends data bytes to be written starting at that address, followed by a STOP. When writing multiple bytes, all the bytes must be in the same 32-byte page. While it is busy saving those bytes to memory, the EEPROM will not respond to further I²C requests. (That is another incompatibility with SMBus: SMBus devices must always respond to their bus addresses.)

To read starting at a particular address in the EEPROM, a combined message is used. After a START, the master first writes that chip's bus address with the direction bit clear (*write*) and then the two bytes of EEPROM data address. It then sends a (repeated) START and the EEPROM's bus address with the direction bit set (*read*). The EEPROM will then respond with the data bytes beginning at the specified EEPROM data address — a combined message: first a write, then a read. The master issues an ACK after each read byte except the last byte, and then issues a STOP. The EEPROM increments the address after each data byte transferred; multi-byte reads can retrieve the entire contents of the EEPROM using one combined message.

Physical layer:

At the physical layer, both SCL and SDA lines are of open-drain design, thus pull-up resistors are needed. A logic "0" is output by pulling the line to ground, and a logic "1" is output by letting the line float (output high impedance) so that the pull-up resistor pulls it high. A line is never actively driven high. This wire-ANDing allows multiple nodes to connect to the bus without short circuits from signal contention. High-speed systems (and some others) may use a current source instead of a resistor to pull-up on SCL or both SCL and SDA, to accommodate higher bus capacitance and enable faster rise times.

An important consequence of this is that multiple nodes may be driving the lines simultaneously. If *any* node is driving the line low, it will be low. Nodes that are trying to transmit a logical one (i.e. letting the line float high) can detect this and conclude that another node is active at the same time.

When used on SCL, this is called *clock stretching* and used as a flow-control mechanism for slaves. When used on SDA, this is called arbitration and ensures that there is only one transmitter at a time.

When idle, both lines are high. To start a transaction, SDA is pulled low while SCL remains high. Releasing SDA to float high again would be a stop marker, signaling the end of a bus transaction. Although legal, this is typically pointless immediately after a start, so the next step is to pull SCL low.

Except for the start and stop signals, the SDA line only changes while the clock is low; transmitting a data bit consists of pulsing the clock line high while holding the data line steady at the desired level.

While SCL is low, the transmitter (initially the master) sets SDA to the desired value and (after a small delay to let the value propagate) lets SCL float high. The master then waits for SCL to actually go high; this will be delayed by the finite rise time of the SCL signal (the RC time constant of the pull-up resistor and the parasitic capacitance of the bus) and may be additionally delayed by a slave's clock stretching.

Once SCL is high, the master waits a minimum time (4 μs for standard-speed I²C) to ensure that the receiver has seen the bit, then pulls it low again. This completes transmission of one bit.

After every 8 data bits in one direction, an "acknowledge" bit is transmitted in the other direction. The transmitter and receiver switch roles for one bit, and the original receiver transmits a single "0" bit (ACK) back. If the transmitter sees a "1" bit (NACK) instead, it learns that:

- (If master transmitting to slave) The slave is unable to accept the data. No such slave, command not understood, or unable to accept any more data.
- (If slave transmitting to master) The master wishes the transfer to stop after this data byte.

During the acknowledgment, SCL is always controlled by the master.

After the acknowledge bit, the master may do one of three things:

- Prepare to transfer another byte of data: the transmitter set SDA, and the master pulses SCL high.
- Send a "Stop": Set SDA low, let SCL go high, then let SDA go high. This releases the I²C bus.
- Send a "Repeated start": Set SDA high, let SCL go high, and pull SDA low again. This starts a new I²C bus transaction without releasing the bus.

Clock stretching using SCL:

One of the more significant features of the I²C protocol is clock stretching. An addressed slave device may hold the clock line (SCL) low after receiving (or sending) a byte, indicating that it is not yet ready to process more data. The master that is communicating with the slave may not finish the transmission of the current bit, but must wait until the clock line actually goes high. If the slave is clock-stretching, the clock line will still be low (because the connections are open-drain). The same is true if a second, slower, master tries to drive the clock at the same time. (If there is more than one master, all but one of them will normally lose arbitration.)

The master must wait until it observes the clock line going high, and an additional minimal time (4 μs for standard 100 kbit/s I²C) before pulling the clock low again.

Although the master may also hold the SCL line low for as long as it desires (this is not allowed in newest Rev. 6 of the protocol – subsection 3.1,1), the term "clock stretching" is normally used only when slaves do it. Although in theory any clock pulse may be stretched, generally it is the intervals before or after the acknowledgment bit which are used. For example, if the slave is a microcontroller, its I²C interface could stretch the clock after each byte, until the software decides whether to send a positive acknowledgment or a NACK.

Clock stretching is the only time in I²C where the slave drives SCL. Many slaves do not need to clock stretch and thus treat SCL as strictly an input with no circuitry to drive it. Some masters, such as those found inside custom ASICs may not support clock stretching; often these devices will be labeled as a "two-wire interface" and not I²C.

To ensure a minimal bus throughput, SMBus places limits on how far clocks may be stretched. Hosts and slaves adhering to those limits cannot block access to the bus for more than a short time, which is not a guarantee made by pure I²C systems.

Differences between modes:

There are several possible operating modes for I²C communication. All are compatible in that the 100 kbit/s standard mode may always be used, but combining devices of different capabilities on the same bus can cause issues, as follows:

- Fast mode is highly compatible and simply tightens several of the timing parameters to achieve 400 kbit/s speed. Fast mode is widely supported by I²C slave devices, so a master may use it as long as it knows that the bus capacitance and pull-up strength allow it.
- Fast mode plus achieves up to 1 Mbit/s using more powerful (20 mA) drivers and pull-ups to achieve faster rise and fall times. Compatibility with standard and fast mode devices (with 3 mA pull-down capability) can be achieved if there is some way to reduce the strength of the pull-ups when talking to them.
- High speed mode (3.4 Mbit/s) is compatible with normal I²C devices on the same bus, but requires the master have an active pull-up on the clock line which is enabled during high speed transfers. The first data bit is transferred with a normal open-drain rising clock edge, which may be stretched. For the remaining seven data bits, and the ack, the master drives the clock high at the appropriate time and the slave may not stretch it. All high-speed transfers are preceded by a single-byte "master code" at fast/standard speed. This code serves three purposes:
    1. it tells high-speed slave devices to change to high-speed timing rules,
    2. it ensures that fast/normal speed devices will not try to participate in the transfer (because it does not match their address), and
    3. because it identifies the master (there are eight master codes, and each master must use a different one), it ensures that arbitration is complete before the high-speed portion of the transfer, and so the high-speed portion need not make allowances for that ability.
- Ultra-Fast mode is essentially a write-only I²C subset, which is incompatible with other modes except in that it is easy to add support for it to an existing I²C interface hardware design. Only one master is permitted, and it actively drives both clock and data lines at all times to achieve a 5 Mbit/s transfer rate. Clock stretching, arbitration, read transfers, and acknowledgements are all omitted. It is mainly intended for animated LED displays where a transmission error would only cause an inconsequential brief visual glitch. The resemblance to other I²C bus modes is limited to:
    - the start and stop conditions are used to delimit transfers,
    - I²C addressing allows multiple slave devices to share the bus without SPI bus style slave select signals, and
    - a ninth clock pulse is sent per byte transmitted marking the position of the unused acknowledgement bits.

In all modes, the clock frequency is controlled by the master(s), and a longer-than-normal bus may be operated at a slower-than-nominal speed by under clocking.

Circuit interconnections:

I²C is popular for interfacing peripheral circuits to prototyping systems, such as the Arduino and Raspberry Pi. I²C does not employ a standardized connector, however, and board designers have created various wiring schemes for I²C interconnections. To minimize the possible damage due to plugging 0.1-inch headers in backwards, some developers have suggested using alternating signal and power connections of the following wiring schemes: (GND, SCL, VCC, SDA) or (VCC, SDA, GND, SCL).

The vast majority of applications use I²C in the way it was originally designed -- peripheral ICs directly wired to a processor on the same printed circuit board, and therefore over relatively short distances of less than a foot, without a connector. However a few applications use I²C to communicate between 2 boards, in some cases over a dozen meters apart, using pairs of I²C bus buffers to boost the signal or re-encode it as a differential signal traveling over the CAT5 or other cable.

Several standard connectors carry I²C signals. For example, the UEXT connector carries I²C; the 10-pin iPack connector carries I²C; the 6P6C Lego Mindstorms NXT connector carries I²C; a few people use the 8p8c connectors and CAT5 cable normally used for Ethernet physical layer to instead carry differential-encoded I²C signals or boosted single-ended I²C signals; and every HDMI and most DVI and VGA connectors carry DDC2 data over I²C.