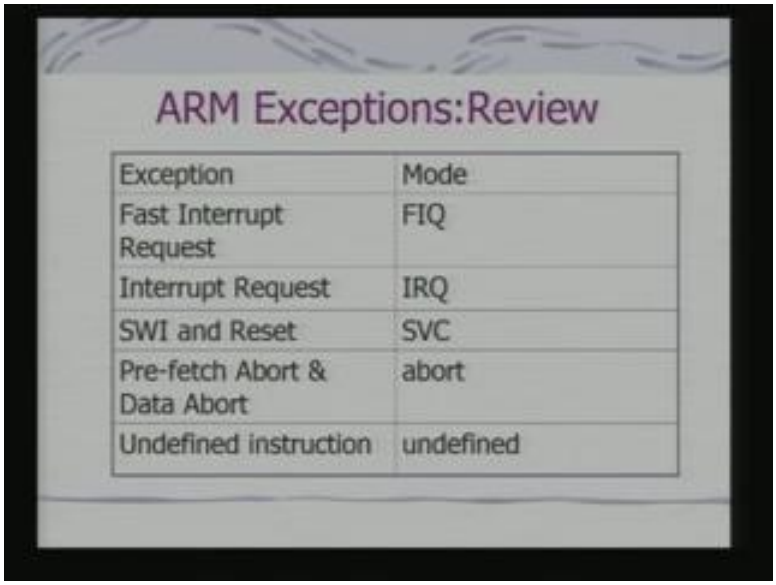
Embedded Systems
Dr. Santanu Chaudhury
Department of Electrical Engineering
Indian Institute of Technology, Delhi

Lecture - 07
ARM: Interrupt Processing

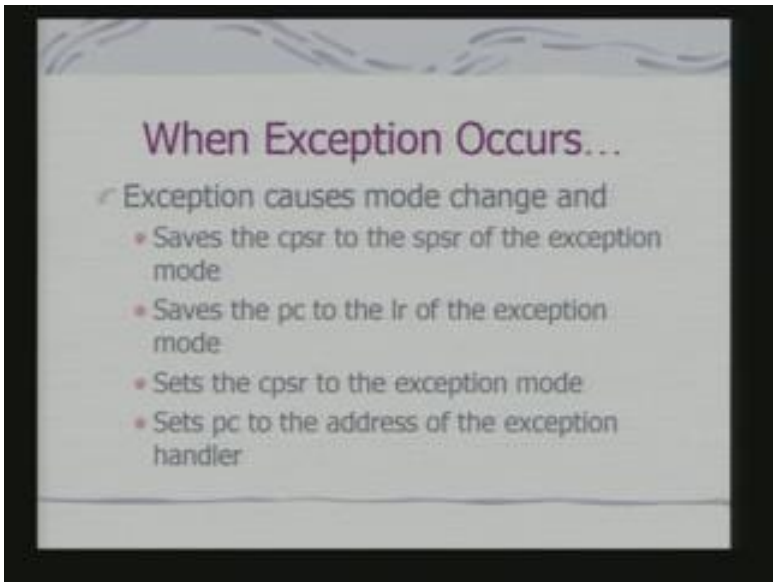
In the last class, we had discussed ARM instruction set. In a way we have developed and understanding of ARM instruction set architecture. Today, we shall look at other features of the ARM architecture. In particular exception processing and the other components how they go into the core and also, the internal organizational details of the processor core.

(Refer Slide Time: 01:51)



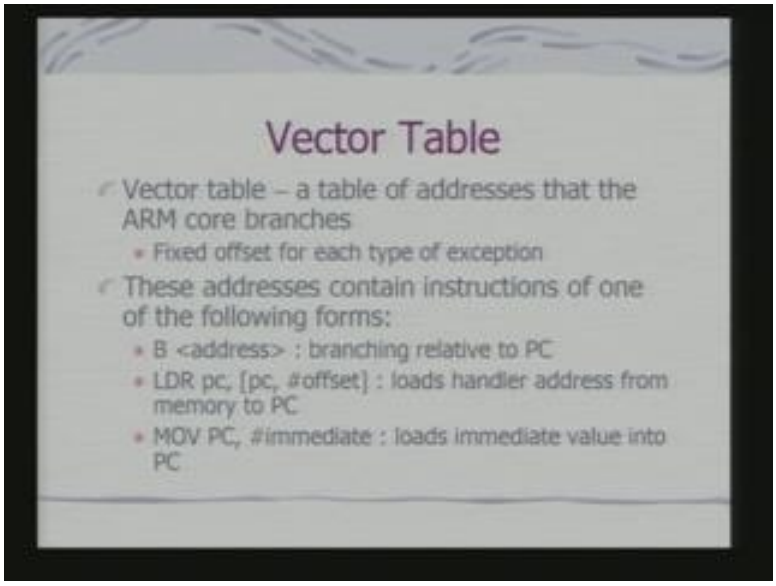
So, before going into the exception processing let us of a quick review. Actually we have discussed this modes earlier. So, what I am showing here is the exceptions and the modes, in which the ARM processor is expected to be in when such an exception occur. The interesting feature is what you find and we have discuss with earlier as well, that corresponding to exception there is a mode switch. That the mode in which processor works changes, when an exception occurs.

(Refer Slide Time: 02:31)



So, along with mode change what will happens? The core saves CPSR, that is Current Status Register to the SPSR of the exception mode that is saved a registers. 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.

(Refer Slide Time: 03:18)



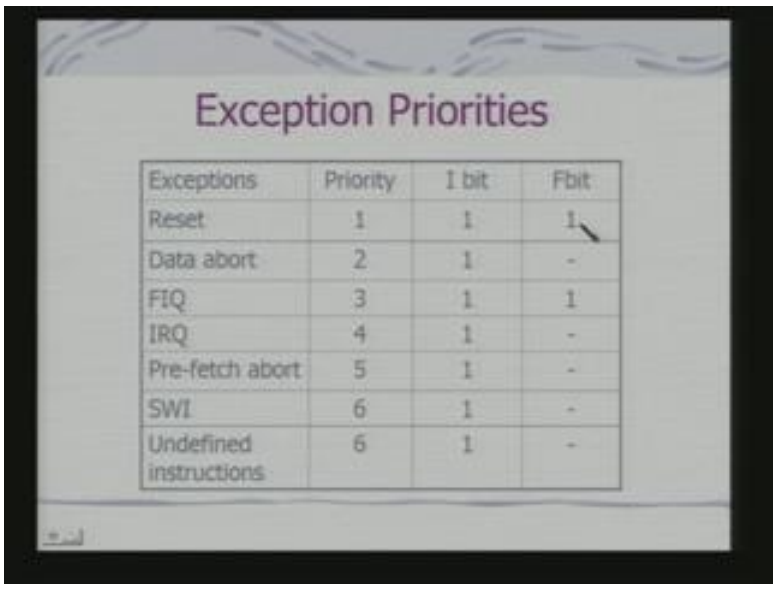
But, it is strictly not always that way. Because, the vector table which is the table of addresses, where ARM code branches when exception occurs. May not have space and really it does not have space to put in the complete service routine in that table itself. So, in the table you will actually have a branch instruction. It can be explicitly of branch instruction or it can be any other instruction which modifies PC.

So, on execution of this instruction actually the control gets switch to the interrupt handler. So, what we have shown here is example of a branch address, another is LDR which is loading the PC. Now, in this case and offset is add it to the PC. So, it can be signed offset as well add it to the PC. And that value is loaded on to the PC.

What does that mean? It means the instruction located at that address will be executed as part of the interrupt handler. Obviously, this would involve a memory access to get the address. And this will therefore, increase the interrupt latency. Now, in this case MOV PC immediate, this is also an immediate address which can be loaded onto the PC. This gives you an ability to get or locate the interrupt handler anywhere in the memory.

Because, the immediate operand can be appropriately changed and loaded on to the PC to provide the address. In this case it is always a relative address.

(Refer Slide Time: 05:11)



Now, this exception have associated with them priorities. So, I have listed here the exceptions and their priorities. And will find the reset exception has got the maximum

priority. And what does these two columns indicate, the indicate the value of I bit and that of F bit, which are bits in the status register, when such an exception occurs.

What is the meaning of this? Say, in reset exception when it occurs, this I bit is set F bit is set, which in effect means that first interrupt request as well as interrupt request both are getting disabled. Similarly, when we see the data about interrupt occurs. In fact, in this case the I bit is set to 1, but F bit is not set. So, interrupt that is normal interrupt request that is disabled. But, first interrupt request is not disabled.

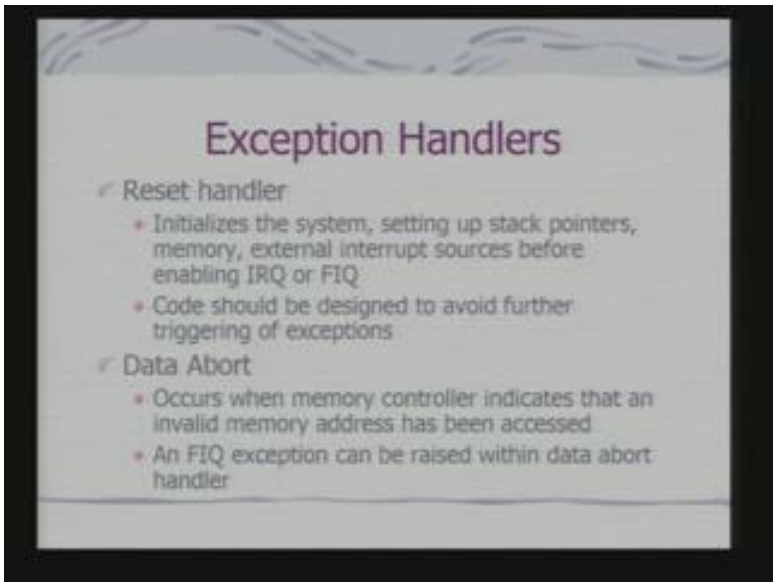
When, FIQ occurs you will find that your both I and F bit is disabled. That means, if I am servicing FIQ and if I do not explicitly enable F bit or I bit, another FIQ cannot be serviced. Now, we will find for all this interrupts, you will find that I bit is getting disabled. So, normal interrupts are getting disabled.

So, inside this service routine of this interrupts, if we are not explicitly enabling this interrupt. That is, generally interrupt cannot be serviced. But, FIQ can be serviced, so; obviously, FIQ can realize has got much greater weight age associated. Now, here also another interesting thing you should note, the basic difference between exceptions and interrupt with reference to the architecture.

Now, if you look at FIQ, IRQ this two interrupts typically occur because of external devices. External devices means, devices outside the CPU core. But, maybe residence on the same silicon area itself. Software interrupt instruction is an explicit instruction as part of your code. But, if you look at others say data abort pre-fetch about or undefined instructions, they can happen because of a some kind of an exceptional condition during execution of your code.

So, therefore, not really interrupts. Interrupts being explicitly generate to interrupt your current flow of execution. So, when you refer to interrupt explicitly, we shall be referring to FIQ, IRQ and software interrupts, others are exceptions. And everything put together is actually exceptions interrupt are also exceptions.

(Refer Slide Time: 08:38)



So, how do you designed exception handlers. Typically reset handler initialize the system, it is sets up stack pointers, memory, external interrupts sources, if you needs to initialize the peripherals before it enables IRQ or FIQ. Because, you would not like to have ((Refer Time: 09:05)) interrupts before the external devices have been initialized.

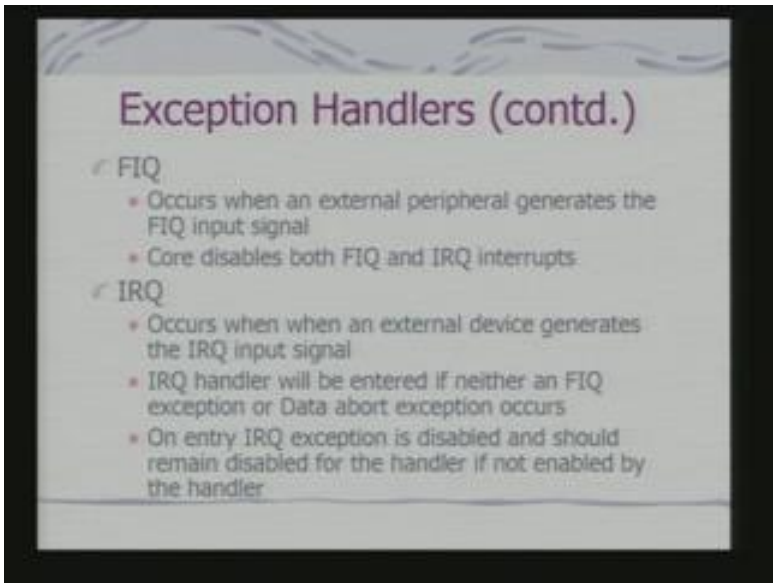
And the other interesting thing is that, you have to initialize stack pointers. Because, until and analyze you set up the stack, you cannot really do processing of the interrupts. Because, you might need to save the registers into the stack. And the reset handler code should be designed in such a way, that no other exceptions are really occur in or gets triggered while execution of this code.

Because, if that occurs that may not be correctly service. Because, you have not set up the vector tables, you have not set up the stack pointers correctly. So, all these things are to be done by the reset handler. The data abort interrupt occurs when memory controller indicates that an invalid memory address has been accessed.

That means, may not be and physical memory located at that address. And so when such a location is being accessed, there has to be an exception. Because, that an error condition and exception has to be handled differently. So, this data abort interrupt I should not say that an interrupt at data about exception is an error condition. And so I need to have an exception handler as part of my operating system to take care of that.

But, during this period itself and FIQ exception can occur. Because, it is not really disabling FIQ. Because, FIQ is an external device. So, it might require an immediate FIQ maybe generated from an external device and it might require an immediate service.

(Refer Slide Time: 10:59)



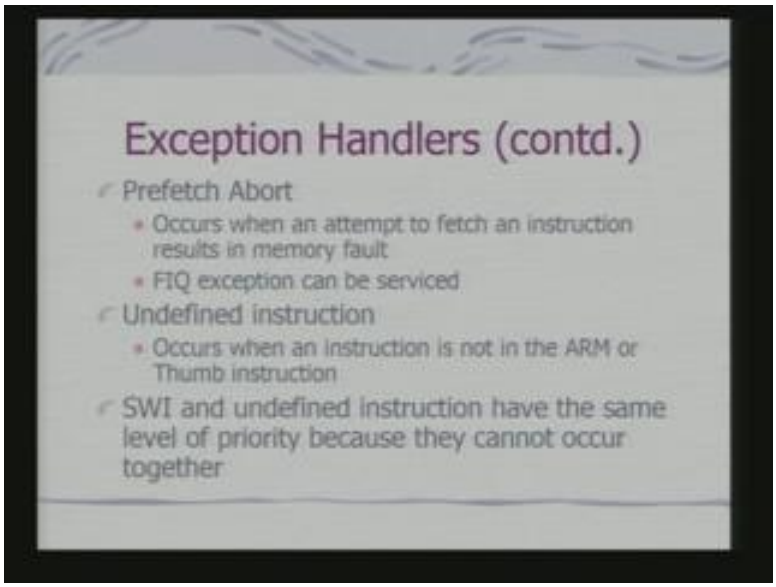
So, FIQ occurs when an external peripheral generates what I call FIQ input signal. And now, code disables both FIQ and IRQ interrupts. That is, when FIQ is being serviced I cannot have another FIQ coming or another IRQ which is of a lower priority occur in an interrupting my service routine. IRQ occurs when as external device generate a IRQ input signal.

So, in that sense the modality wise FIQ and IRQ are identical. But, FIQ has got a higher priority and more weight age. And also you will you I hope you remember, that there are a large number of register copies, which become available in FIQ mode. So, the interrupt latency the software latency would be much less compare to that of IRQ.

So, IRQ handler would be entered if neither an FIQ exception or data abort exception occurs. Because, the FIQ and data abort are of higher priority. And reset is not normally expected to occur, other than when the system is booting up that is the initial conditions. On entry IRQ exception is disabled and should remain disabled for the handler, if not enabled by the handler.

That means, if I am not entertaining nested interrupt processing I shall not do what enable IRQ, inside IRQ handler. If I am not enabling nested interrupt handling there may be increased interrupt latency.

(Refer Slide Time: 12:47)

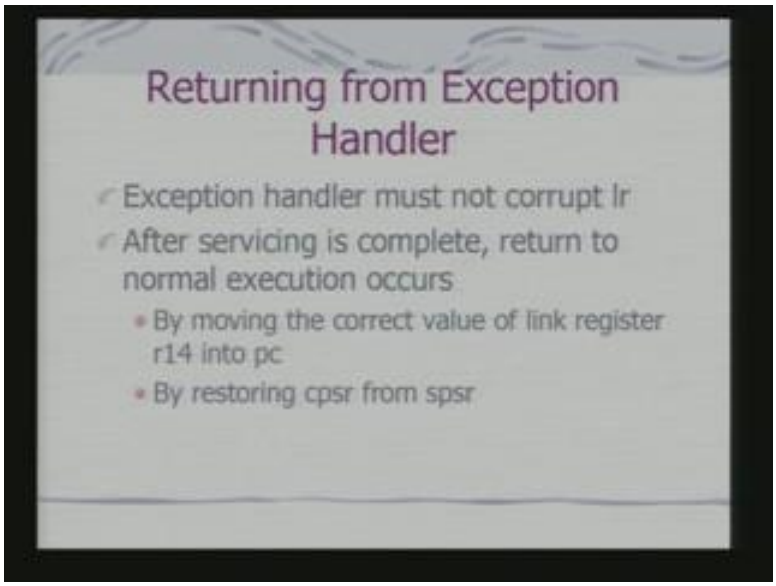


The pre-fetch abort occurs when an attempt to fetch an instruction results in the memory fault. In fact, it is very similar to data abort, data abort occurs when your executing an instruction trying to get the data or write the data pre-fetch about occurs, when you are trying to get the instruction the op-code.

And in the same case FIQ exception can be service, even when the pre-fetch about occurs. Undefined instructions occurs when an instruction, that is being that is the op-code which has been fetched and it is been attempted at decoding. And it is found, that is not in the ARM or thumb instructions.

In fact, when it is not even instruction of any of the coprocessor, which may exist on the core. And the interesting feature is the software interrupt, as well as undefined instruction they have the same priority level attach to them. In this obvious, because both this exceptions cannot occur simultaneously. And what happens and how we process software interrupt instruction, we have discussed in the last class itself.

(Refer Slide Time: 14:08)



So, how do you return from exception handler, it can be using any instruction for that matter, which will restore your PC to the corrected value of the link register. And what is important is that, the exception handler must prevent corruption of the link register value, which gets loaded when more switch takes place. This is very, very important.

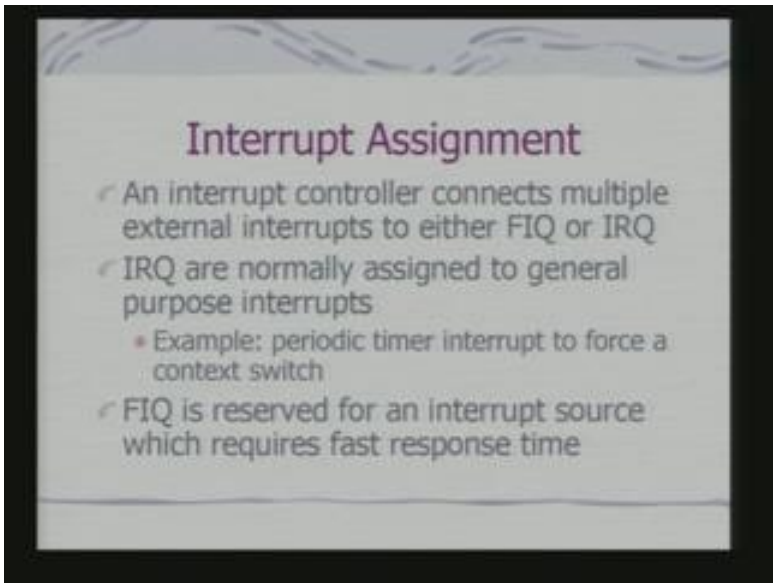
And the other important thing is that, the CPSR the current CPSR will be restored from saved program status register. So, that rest CPSR gets restore from the saved program status register. And what there is a certain interesting point here. What I have written is? That I am moving the correct value of link register r14 into PC.

It is not that I am loading exactly what is loaded in l r into PC. Why it is required we shall understand, when we study the pipeline architecture. It implies that before I actually return from the exception handler I need to correct the value stored in l r. So, that I come back to the correct instruction, which is to be executed.

And I have told you, that all this exceptions have associated with them a vector, that is a memory location. That location is accessed forgetting the instruction for the handler. We go into the handler and when I have completed servicing of the interrupt I come back from the handler. Before coming back from the handler I should adjust the l r value.

It may also so happen that I might like to store the l r value as well as other registers into the stack. Because, I would like to retain the values, then I need to do what, before I come back from the handler, restore the register values and then return from the handler.

(Refer Slide Time: 16:42)



Next question is interrupt assignment. And this interrupt assignment this is particularly important, when you are looking at the hardware devices. And we are assigning interrupts to the hardware devices. In many times why should I say many times in almost in all universal implementations, you have the interrupt controller with the core CPU.

The interrupt controller connects multiple external interrupts to either FIQ or IRQ. In fact, IRQ are normally assigned to general purpose interrupts. In fact, periodic timer interrupt to force the context switch, when you have a multiprocessing. When, we have concurrent process running and the CPU has to switch from one process to another process. That switch takes place, because of an interrupt from the timer.

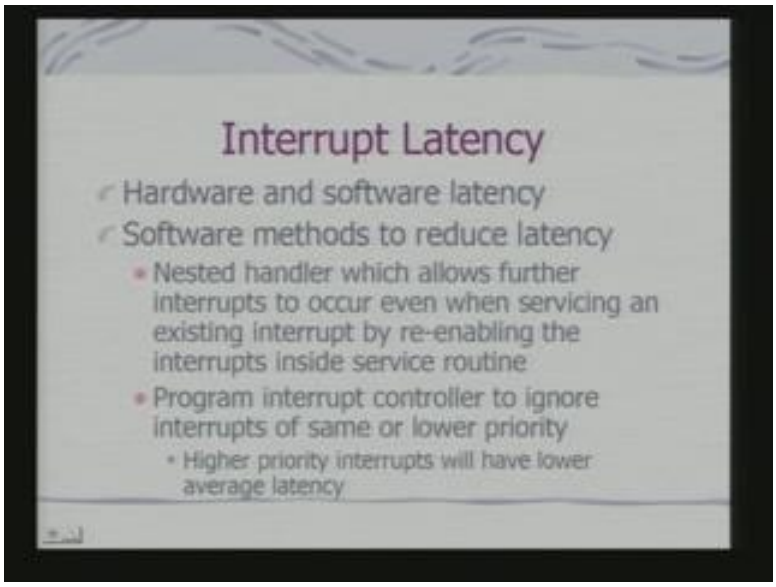
Because, timer is programmed to allocate a fixed time slot to the process. Now, these kind of general purpose interrupts, which is for the purpose of maintaining OS are associated with IRQ. The critical devices which are to be service with very less latency, they are associated with FIQ.

So, what we say that FIQ is always reserved for an interrupt source, which requires fast response time. Because, it is having minimum interrupt latency. And in fact, the issue is

that for an application in an embedded system. The environment, the external signal property decides, how much latency you can actually allow for.

In case of this periodic time and this latency is the more of an OS consideration. And not of the property of the external signal. But, when we have to handle a very first external signal with less interrupt latency, then becomes a signal coming from some kind of a dedicated peripherals or a dedicated application. So, that interrupt is associated with FIQ.

(Refer Slide Time: 19:00)



So obviously, now the question is interrupt latency. So, I had already discussed this point earlier and I am again coming back to this point. Because, interrupt latency is the very key issue in designing and embedded system. It has got both components, hardware as well as software components. Now, there are software methods to reduce latency.

Hardware is as you have been provided by the architecture. And you have found that FIQ, the example of ARM that provision of a copy registers becoming available is a hardware solution for reducing interrupt latency. Even in the software we can take certain steps, one of the step is what is called nested interrupts.

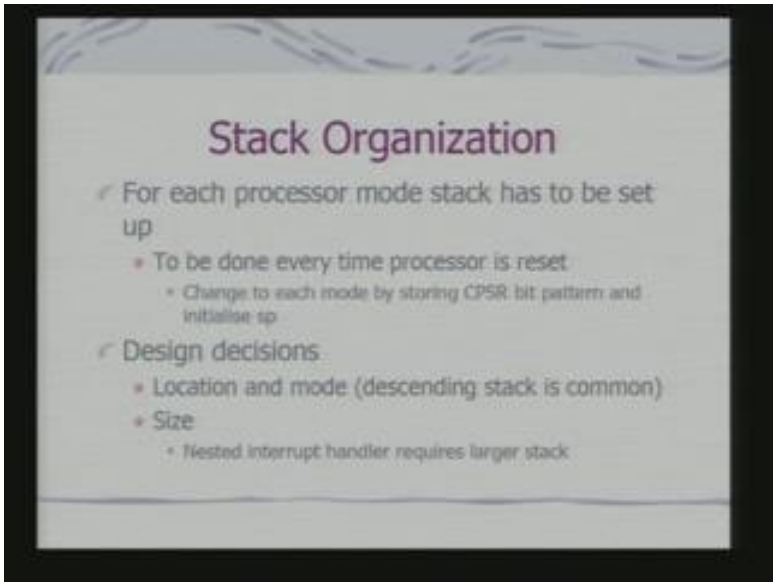
So, nested interrupt handler allows further interrupts to occur, even when servicing an existing interrupt by re-enabling the interrupts inside the service routine. And obviously, in this case what will happen? The interrupt which is now raising or request did not way till servicing of the previous interrupt is completed.

And when we are using an interrupt controller, we can associate different priorities with the devices. So, what can happen is that higher priority interrupt can actually interrupt or service provider of a lower priority interrupt. Although the, so what happens, if I have an IRQ handler, inside IRQ handler I enable IRQ bit. And interrupt controller is associate with priorities of the external devices.

Now, if an external device raises and interrupt. And these device has got a higher priority, than of the device whose interrupt is being serviced my interrupt controller will do what? We generate another IRQ interrupt request. Since, I am using a nested handler IRQ bit is now clear. So, that interrupt will occur and I shall now service interrupt of higher priority device.

What is the effect? The average latency of a higher priority device is less compare to that of a low priority device.

(Refer Slide Time: 22:39)



Now obviously, for doing all these exception handling, you have understood always the stack organization becomes critical. Because, for each processor mode I have got a stack pointer and each processor mode has got a different stack therefore. So, I need to set up this stack in the beginning itself. So, every time this stack this is not a single stack. So, what I shall have? I shall have a stack in the memory.

So, stack is what a memory area. And that memory area will be different, for different modes. The stack that I am using in user mode is definitely not same as that of the stack area, which I might be using in FIQ mode. So, I have to appropriately said the stack pointer values. And that is typically done during the reset mode.

So, change to each mode. So, during the reset mode, you change to each mode by adjusting the CPSR bit and adjust then writing the correct value on to SP. The other decision is and this is the design decision of the software developer. That, where in memory the stack to be located and what is the mode to be adopted. Because, I can have all possible modes of stack.

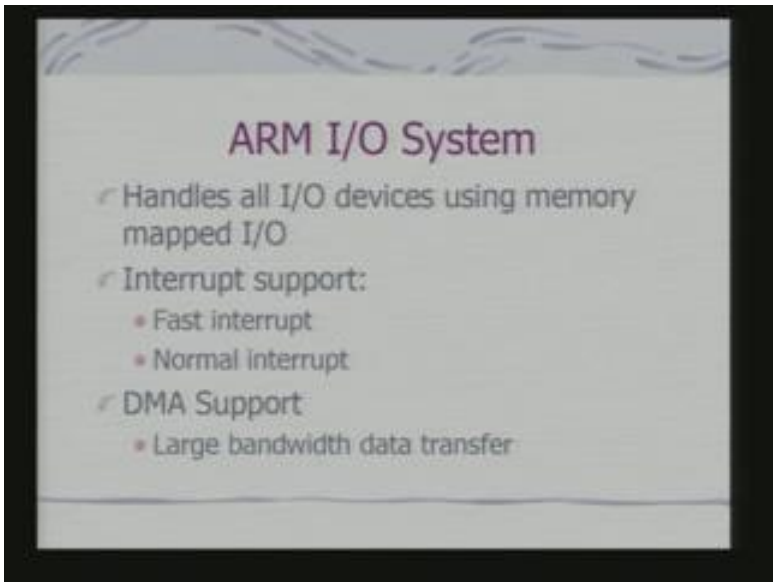
Typically, descending stack is the most common mode and the location is define in such a way, that you really do not have a chance to overrun the vector table. And that is very, very important. If you overrun your vector table by stack overrun and the error condition. So, you will have corrupt in the system as a whole.

So, you should associate a location with the stack. So, that you minimize your chance of overrunning the vector table. And in many cases, you can actually use a check to find out whether your overrunning the stack. And the size varies from implementation to implementation application to application, depending on how your designing your software.

Because, if you are really supporting nested interrupts, you will require a larger stack. Say for example, IRQ incase of nested interrupts I need to save the registers. So, that I can come back to the correct point of execution, where of servicing the lower priority interrupt. So, I shall need the space to save all this registers.

So, the stack size requirement for nested interrupt handling is always more, then when we are not having nested interrupt handlers. Now, you can realize since it is an embedded system is not a general purpose. I can have an assessment of the kind of external devices and the signals, that we generate the interrupts. And what are their latency constraints. And accordingly we can design the interrupt handling scheme and decide on the size of the stack.

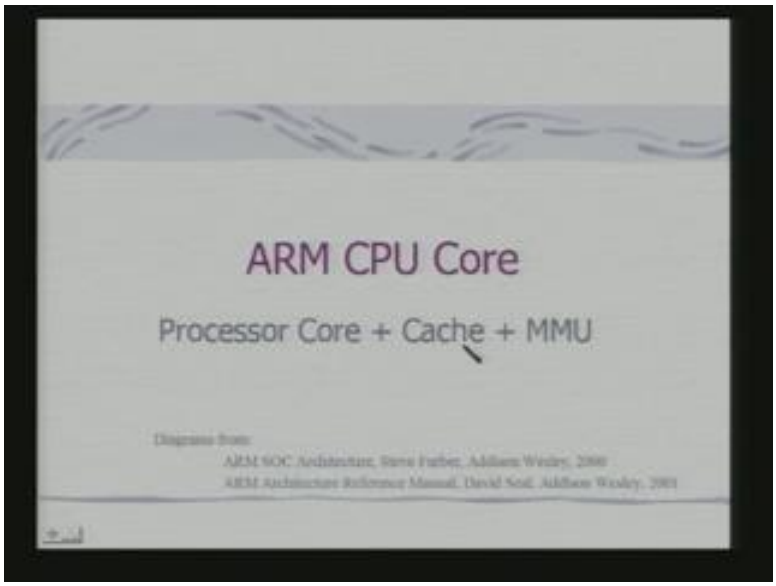
(Refer Slide Time: 25:00)



In fact, I/O system, that I/O devices they are primarily interface through this kind of interrupts. So, I have first interrupt in normal interrupt that we are already discussed. And there used for interrupt driven I/O processing and all I/O devices at typically located in the memory map of ARM.

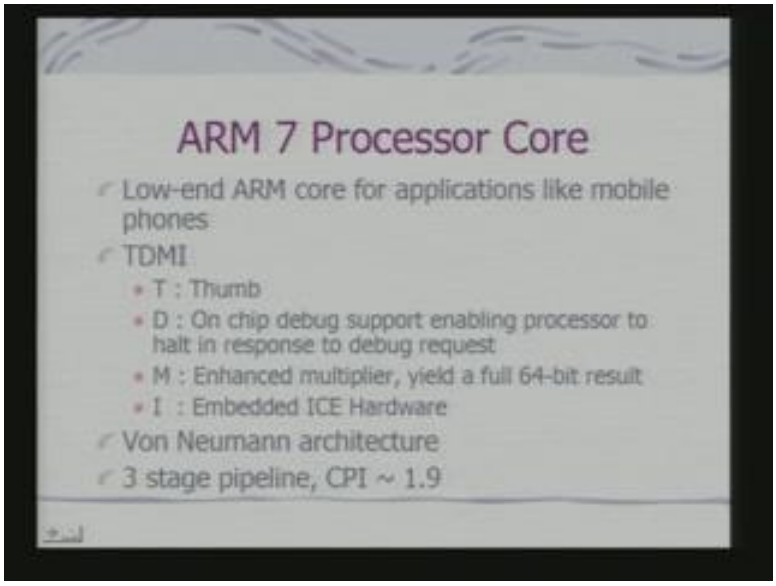
So, you have got memory mapped I/O, there is no separate I/O address bits. There is also a support for DMA. So in fact, this interrupts that we had actually discussed you can understand and you can realize. The basic job of this interrupts to handle this kind of various I/O which are external environment dependant. Next, we see shell look at these processors and processor codes in more detail.

(Refer Slide Time: 25:59)



In fact, what we say that ARM CPU core, typically consist of the processor core, cache memory and memory management unit. Now, this memory management unit and cache, we shall discuss separately. When, we discuss the memory organization in an embedded system. Today our concentration could be on the processor core.

(Refer Slide Time: 26:24)



We have look at the data path of ARM 7. But, not only the data path there are other; obviously, the hardware components, inside the ARM 7 processor. In fact, ARM 7 is the

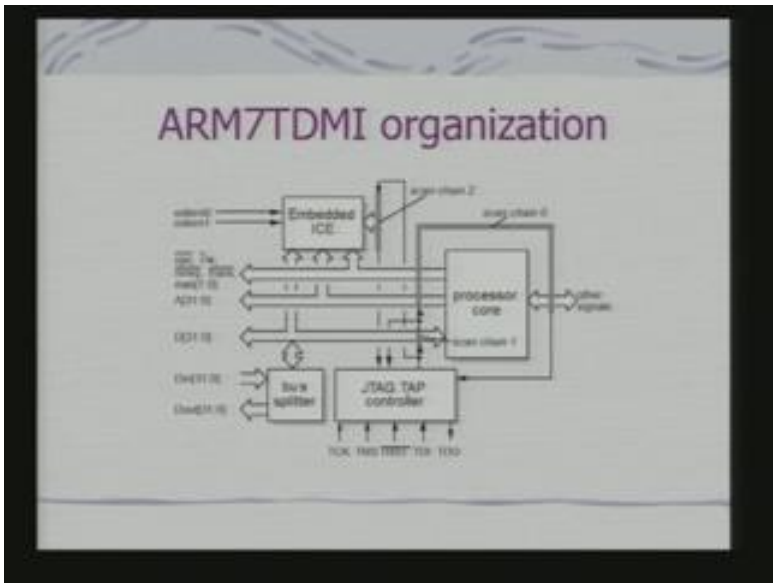
low-end ARM core, which are targeted for applications like mobile phones. Although today things of changed, you get ARM 9 and more sophisticated processor being used.

But, when ARM 7 came in one of the target applications are digital mobile phone. You have got these variance of the ARM core and which are indicated by this let us TDMI. T is basically thumb mode. That means, you have got an embedded 16 bit processor inside 32 bit. So, you can switch from ARM to thumb and back.

D means, you have got a one ship debug support, enabling the processor to halt in response to a debug request. M is an enhanced multiplier, which yields full 64 bit result you may not require it. So, if you do not require it. So, you do not have M, that is you have TDI and not TDMI, I is embedded ICE hardware.

The architecturally ARM 7 is the Von Neumann architecture has got an Von Neumann architecture with 3 stage pipelines. And the CPI the cycle instruction typically about 1.9.

(Refer Slide Time: 28:12)



So, let us look at this organization. So, this is a more abstract level or cross level organization. But, we are looked at earlier, we had looked at the data path which is sitting inside here. So, what we are looking at other than the processor core, there are other components, which are there in the CPU core.

The fundamentally these are the two blocks, the JTAG TAP controller and embedded ICE. In fact, this JTAG TAP controller is actually a kind of a port, which enables a direct

communication with the processor core, for the purpose of debugging and running the software. Consider a very simplified situation, you are trying to execute and instruction on this processor core.

Now, you know instruction set. So, you will be using an assembler or software development environment. The software development environment will typically run on a PC. And form that PC, I have generated the code and I would like to run the code on your ARM. So, one option is you load it on to a memory and connect that memory area to the processor core.

So, when it boots up that memory area, the initial memory area will contain the instruction and it will start execution from that. The other option is a using this JTAG port using a JTAG connector. So, if you using a JTAG connector, what happens is by using the JTAG connector, you can actually inject the instruction on to the processor core.

So, this can change this is the set of registers and you can actually inject your instructions into the processor core. And you are and what your embedded ICE, this is in-circuit emulator what it this hardware block provides for, the support to examine the processor state, while this instruction execution takes place.

So, what I am trying to say is that, these two enables say just consider this that some of this JTAG TAP controller enables what? Enables injection of instruction, instruction is what nothing but a binary bit pattern. Injection of that instruction on to the processor core and the processor core executes that instruction.

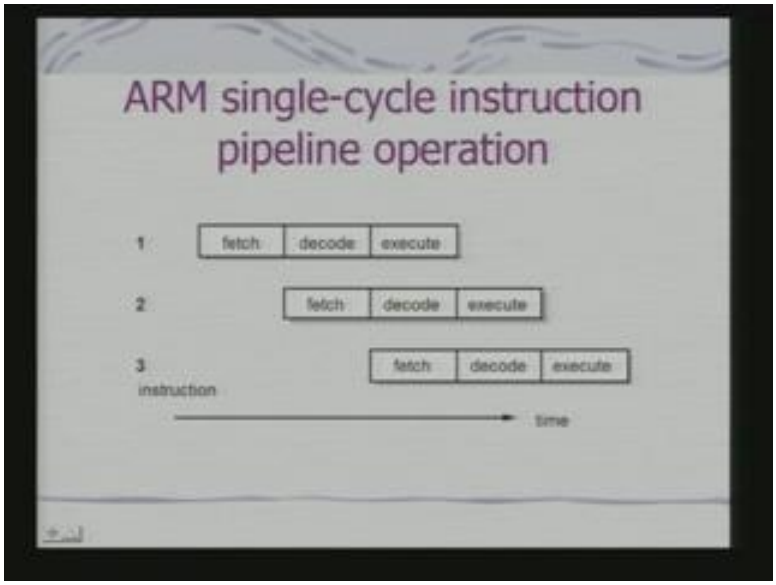
And this embedded ICE gives your provision for checking the processor state, as instruction execution takes place. Checking the processor state can mean what? Maybe checking the value of the registers, it also enables you to introduce break points. So, I have enable therefore, by adding to this two blocks in the basic core communication facility. And the ability to see and observed, what happens inside when instruction execution takes place.

This is the basic bus structure. So, what you have got here? You have got bidirectional data bus, here this is the bidirectional data bus and this is your input data bus. This is

your output data bus, these are the two both the options are available. This is actually your external address bus, this is essentially the control bus.

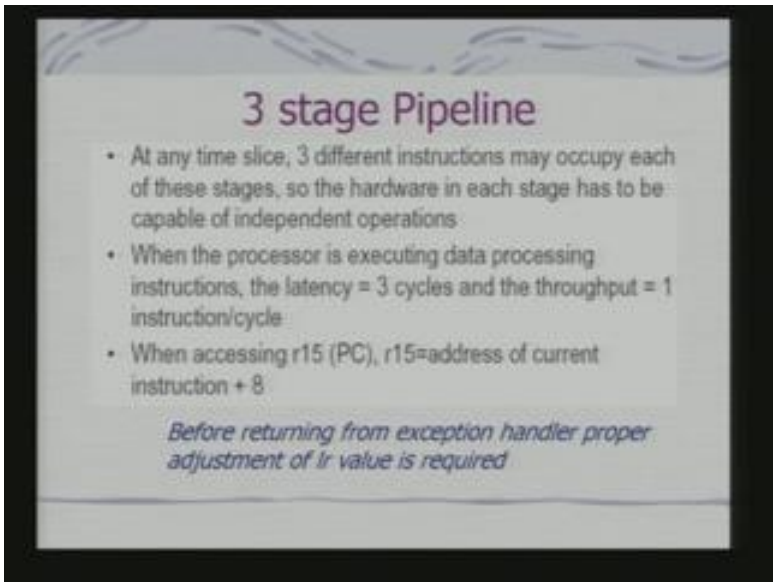
So, this is a very simplified picture of ARM 7 TDMI organization. In fact, the D and I actually indicated of the fact that I have got this JTAG, as well its embedded ICE sitting in the hardware.

(Refer Slide Time: 32:32)



It is supports some 7 supports 3 stage pipeline. So, effectively what we are seeing is that an instruction execution takes place 3 cycles. But, since I have a pipeline effectively what happens, once the pipeline is loaded for cycle, I can have one instruction executed. These the simplex three stage fetch decode execute.

(Refer Slide Time: 33:03)



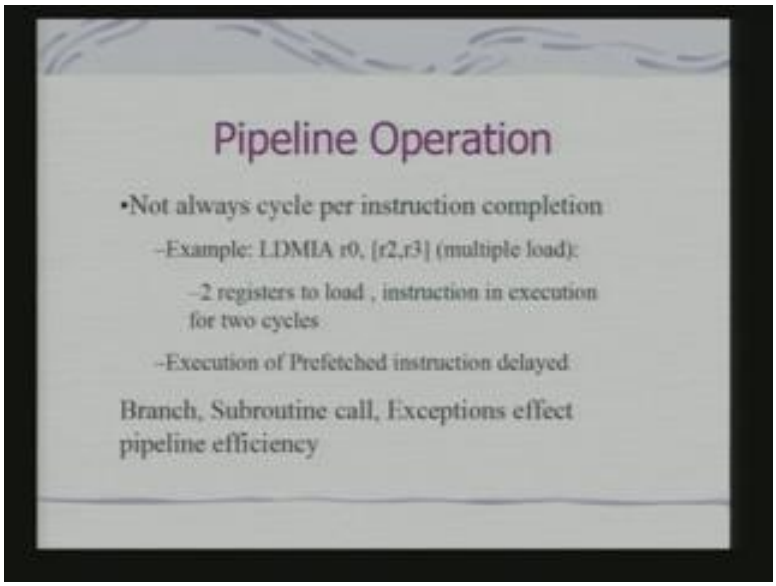
At any time slice therefore, 3 different instruction may occupy each of this stages. And when the processor is executing data processing instruction, the latency is 3 cycles and throughput is one instruction per cycle. I hope you understood why latency is 3. And when accessing r15. R15 is address of current instruction plus 8, this is the key point.

Why? Because, since you are doing pre-fetching, your PC when you are currently executing an instruction. Your PC gets pointed to the instruction, which is currently to be fetched. So, it is PC plus 8 and when interrupt occurs what happens? The current execution is completed and the interrupt is serviced.

If not is case of an exceptions, like data abort or pre-fetch abort, where actually the exception occurs, while execution of this instruction. But, in case of FIQ or IRQ the interrupt service only on completion of the current instruction. And in this case PC points to what? PC plus 8 and these value will get loaded onto l r.

But, when you coming back from the service routine. If you start execution from PC plus 8 you are not doing the right thing. You are skipping and instruction. So, l r value has to be correctly adjusted, inside the interrupts service handling routine. So, that you come back to the correct point for execution.

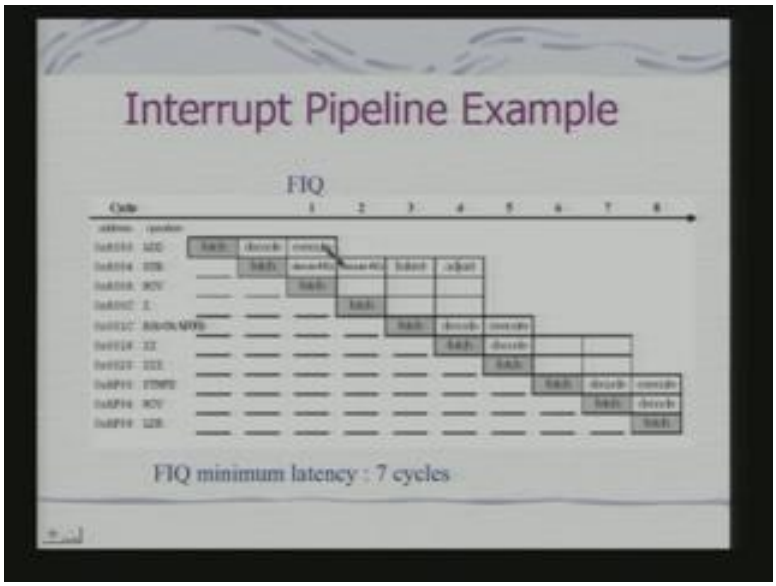
(Refer Slide Time: 35:07)



So, there are, but one thing with ARM instruction set is that, all instructions are not necessarily single cycle instructions. The typical example is load multiple byte or load multiple or stored multiple instructions. So, these the multiple load. So, it has got 2 registers to load and so the instruction has to be in execution for two cycles.

And hence, execution of pre-fetch instruction will be now delete. So, other instruction like branch, subroutine call, exceptions, effect pipeline efficiency. And you have already seen, that we would like to use conditional instructions rather than branch. When, you have small set of instruction to execute to have the pipeline working in an efficient fashion.

(Refer Slide Time: 36:04)



Let us see what happens when an interrupt occurs, in the pipelining case. So, I am executing an instructions. So, I have shown here at this point that FIQ occurs. So, if FIQ occurs at this point, then this instruction will be executed. But, next instruction will not be executed, this is already been fetched. And these are the different processing states, which would takes place.

So, now let us say from here where shall I go, if you look in to it, this move instruction I am just giving a typical examples. So, this move instruction, this is still this is what has been fetched. Because, it is a 3 stage pipeline and the next is X, because there nothing occurs. Because, now you are actually decoding IRQ, that is when IRQ or FIQ occurs.

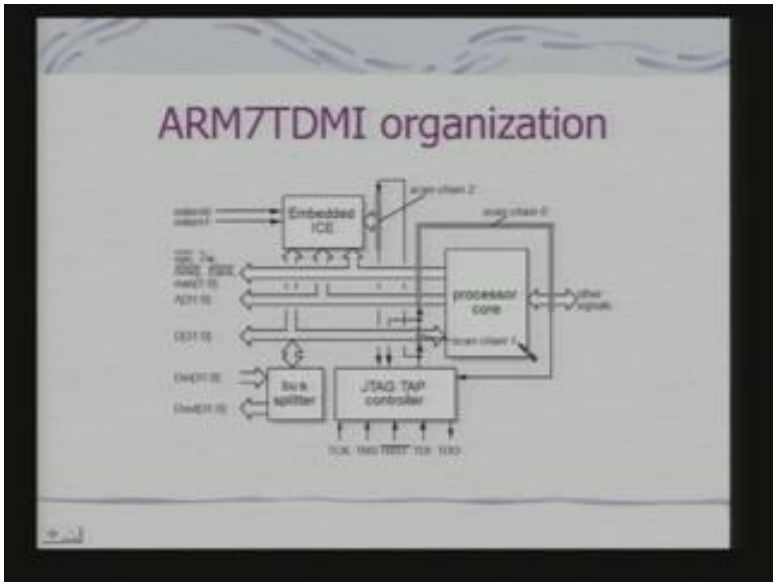
And then, you are trying to see what is to be done? See, FIQ occurs you actually go to 0 0 1 C which is actually the vector. Now, at this vector you have got a branch instruction, you are branching to AF00. So, when you are branching to AF00, then this pipeline is again has to be flashed. So, I am you are going to this location. And at this location you are starting execution of the first instruction of the interrupt handler.

So, effectively what we have seen here is the, minimum FIQ latency has to be here 7 cycles. These are typical situation I may do something else as well. But, here at this vector, what I have got at this vector I have got a branch instruction. So, these branch has to be executed to actually move into the interrupt handler. And so there is a delay of about 7 cycles.

And this is how the pipeline would be ((Refer Time: 38:13)). So, once at this point the interrupt occurs, I have to do this is taking place for the adjustments, corresponding to the interrupt. So, this has been fetched. So, I go to at this point, when I go to this point I am actually executing here, this is the branch instruction which gets executed.

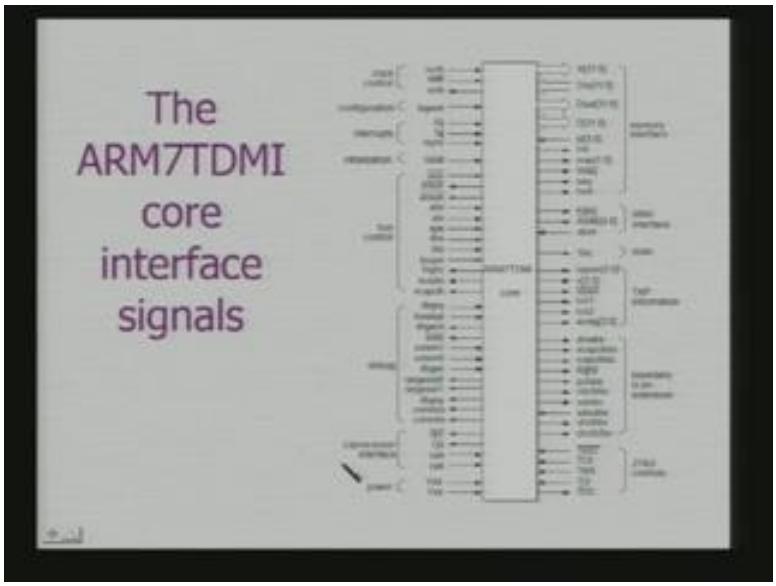
So, this fetch has already done and this has to be fleshed. And then, real execution starts from this points onwards. So, this is actually the first instruction of the interrupt handler. So, this is the scenario where you have got a 7 cycle in FIQ latency.

(Refer Slide Time: 39:00)



So, this we had already discussed now we are looking at this core. Because, next what we shall look at? We shall look at the signals, which are coming out and going in corresponding to this organization.

(Refer Slide Time: 39:13)

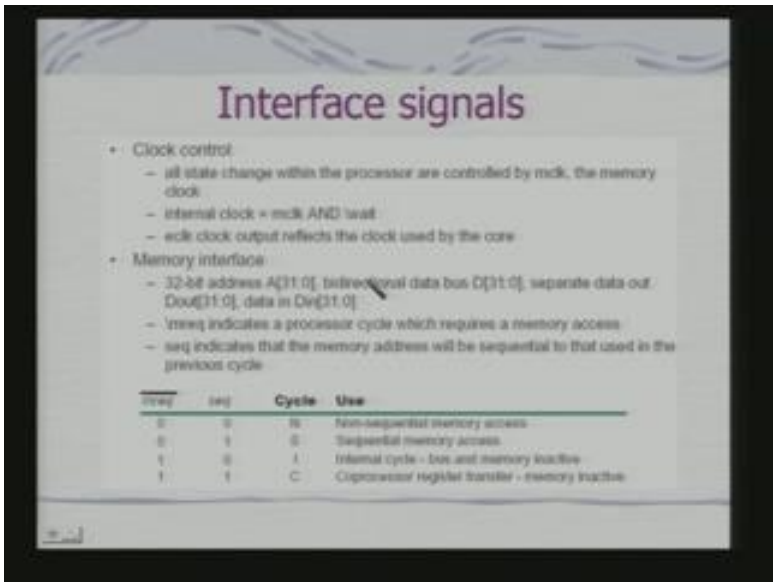


So, these are the different set of signals, we did not bother about each one of them right now. What is important is to know the groups, you can see that there is the set of signals which are enabling the debug operations. There are set of signals for external coprocessor interface.

These are the power signals, this are for the JTAG controls, JTAG I told you is the port through which I can actually inject instructions onto the core. So, I need to have a JTAG control signals. Then, you have got the memory management unit interface, you have got this memory interface. That is how exactly the memory is to be connected, external memory is to be connected.

These are your interrupts signals, out of each you can recognize that FIQ and IRQ, which are basic interrupt signals. This is the configuration signal, which indicate whether it is being configure in a big-endian or a small-Indian form. And these are your width control your clock control signals.

(Refer Slide Time: 40:17)



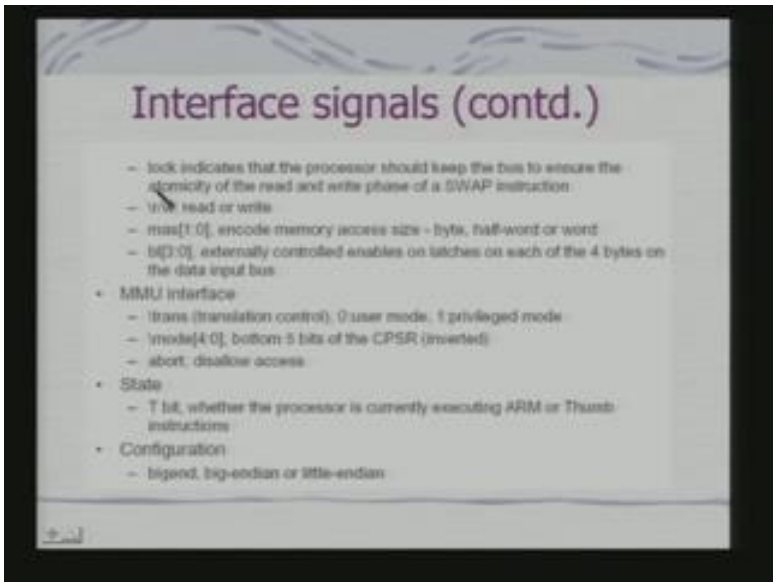
So, this is a kind of a description of the interface signals. What is important and interesting to note the memory interface. It provides 32 bit address, it has got bidirectional data bus and separate data out and data in. So, you have got in fact, these 3 buses. So, to you can have a faster data transfer. And there are various signals, which indicates the kind of processor cycle, which is going on.

Now, here we have given set. So, what you tells you is that, in request indicates a processor cycle, which request a memory access. It is not that all processor cycle request memory access. So, external devices has to be told that, whether access is required or not. And also it indicates, whether it is a sequential or non-sequential.

That means, you are accessing a memory location, which will be sequential to that use in previous cycle or not. So, that enables design of the hardware appropriate design of the hardware. And the cycle's which are defined you will find non-sequential memory access, sequential memory access, internal cycle.

So, bus and memory in active. So, this bus now can be used by some other bus controller. Then, this is coprocessor register transfer. So, now the data transfer is to the coprocessor register and not to the memory locations. So, these are kind of bus status signals which are provided by the ARM 7.

(Refer Slide Time: 42:04)



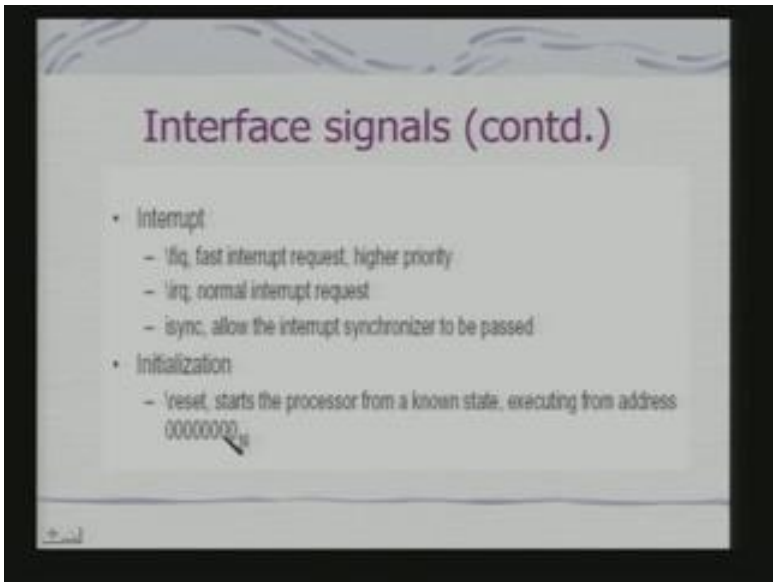
Then, you have got lock. Lock indicate the process should keep the bus to ensure the atomicity of read and write phase of swap instruction. Swap means, a read as well as write. So, I require bus and bus should not be released in between. Because, otherwise the data integrity maybe at state. So, that is the ensure by the lock signal.

This read and write, this is the standard read and write indicator. And this signals indicate, whether what is the memory access size? Are you accessing the byte, half-word or word . And this MMU interface, this is the Memory Management Unit, this memory management unit actually manages the memory.

And it takes here of the fact, that if it is an a privileged mode your, if you are in user mode you are not accessing the memory area, which is allocated to a software, which is expected to be executed in a privileged modes. So, this memory pack partitions, virtual memory. All this things are managed by the MMU and these are the interface signals.

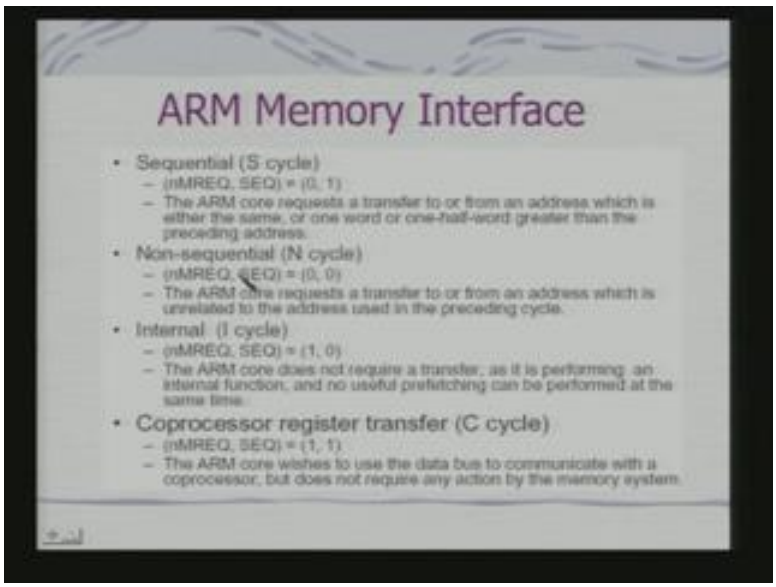
Because, MMU you needs to know, the mode. Because, whether it is a privileged or non-privileged mode, it also needs to know how to do the address translation. Because, if it is a virtual memory, appropriate address translation has to be done. Also it has to the abort signal, that whether this location is disallowed access or not. Then, where is this state which is indicated by the T bit, whether your processor is in thumb state or not. And configuration I already told you.

(Refer Slide Time: 43:51)



These are all interrupts and this initialization is the reset signal, which forces the processor to start from an known state and executing from this location.

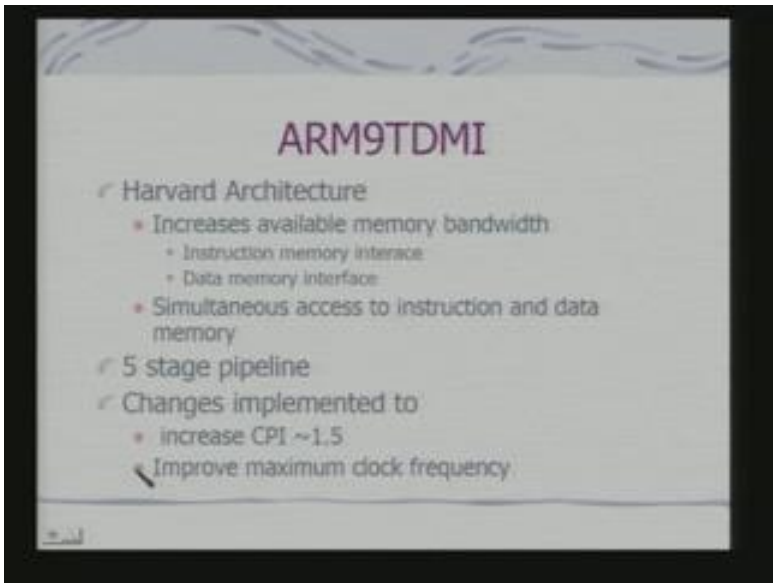
(Refer Slide Time: 44:06)



The ARM memory interface has got I have already shown you the cycles. And this is exactly the definition of the details of the cycles. So, it says the ARM code now request the transfer to or from an address, which is at the same or one word or one half word greater than the preceding address.

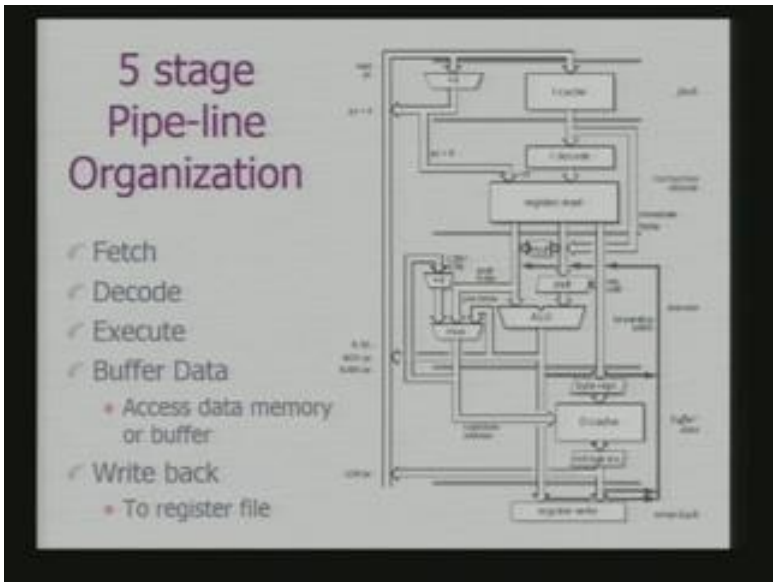
This is exactly the meaning of sequential access. This is non-sequential access, where it is not related to the previous address. And this is corresponding to coprocessor register transfer.

(Refer Slide Time: 44:46)



Next, we look at ARM 9 which is an enhancement over ARM 7. Now, ARM 7 was a Von Neumann architecture, this is Harvard architecture. Obviously, the Harvard architecture ensures increased data transferring. Because, you have got a separate instruction memory interface and a separate data memory interface. So obviously, the memory bandwidth is more.

(Refer Slide Time: 45:35)



It implements not a 3 stage, but 5 stage pipeline and this changes what implemented to achieve a kind of a CPI of the order 1.5. And to improve the maximum clock frequency. This 5 stage pipeline has got the following stages, fetch, decode, execute, buffer data, write back. In fact, these are the two modes, which are last to other two modes which are not their originally.

Here, you can buffer the data or access data memory, which is different from instruction memory. And in the organization we are showing basically the different stages. When, you fetch the instruction, this instruction goes into what is call instruction cache, I-cache. So, this is where your fetch pipeline stages is executed,. Then, you do an instruction decode.

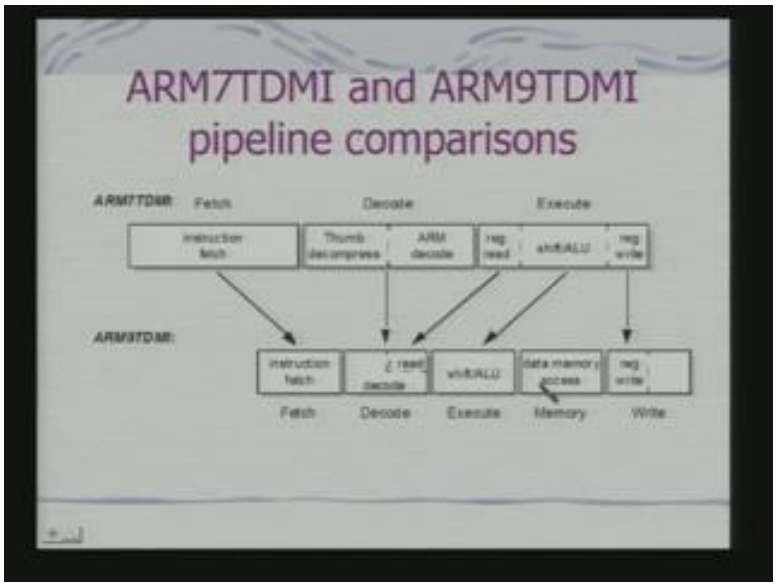
Here, you do the decode as well as you select the registers. So, that you can actually get the data enable, the operands to come out of the registers. Next is actually your execute stage, where you actually have got your ALU. This is the same Barrel register that shift operation that I talked about, there would be a multiplier block. And these are all operations to do with address manipulations.

So, pre indexing and other things if it is to be done, then this multiplexer comes into the play, this operation takes place and it is goes back. After the execution takes place, the results go to data cache. And this is what we call the buffering or the data memory access

stage. After following this, you go to register I state and from here it is actually return back to the register bank.

So, these are the 5 stages of pipeline, which is implemented in ARM 9. And you can realize that we are talking about I-cache and D-cache. In fact, this is the reflection of the flag, that we are talking about the separate instruction memory and separate data memory.

(Refer Slide Time: 47:47)

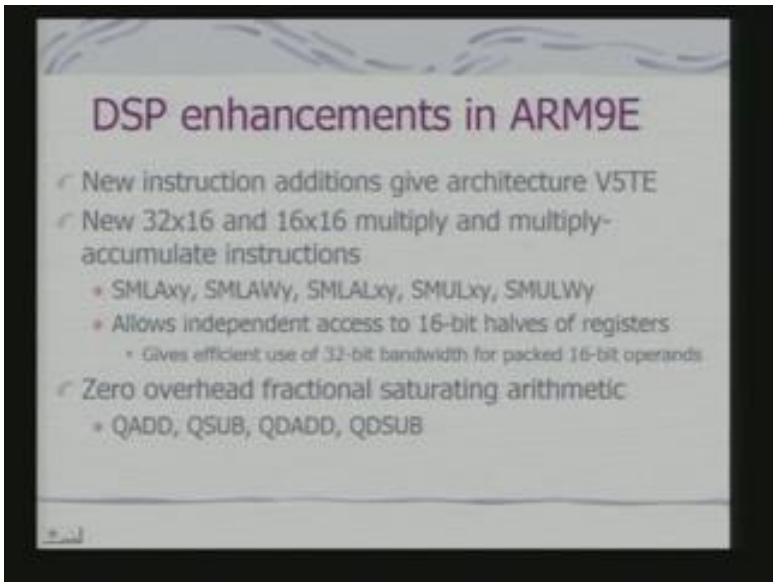


And this is a kind of a comparison between ARM 7 and ARM 9 pipelines. ARM 7 had a 3 stage ARM 9 has got a 5 stage. In fact, what we are showing here is that, thumb decompress and ARM decode is actually everything put together to a decode. Because ,this is TDMI; that means, I have got a thumb mode as well takes place here.

And in this case of a decode, you have got the complete decode here coming into this block. As well as register read is also enabled here. I had already shown that in the block diagram. Because, I can enable the registers once I have decode then instructions. Next, I do that ALU operations shift and ALU, which is basically execute. And in fact, in the execute stage I had the register write as well in ARM 7.

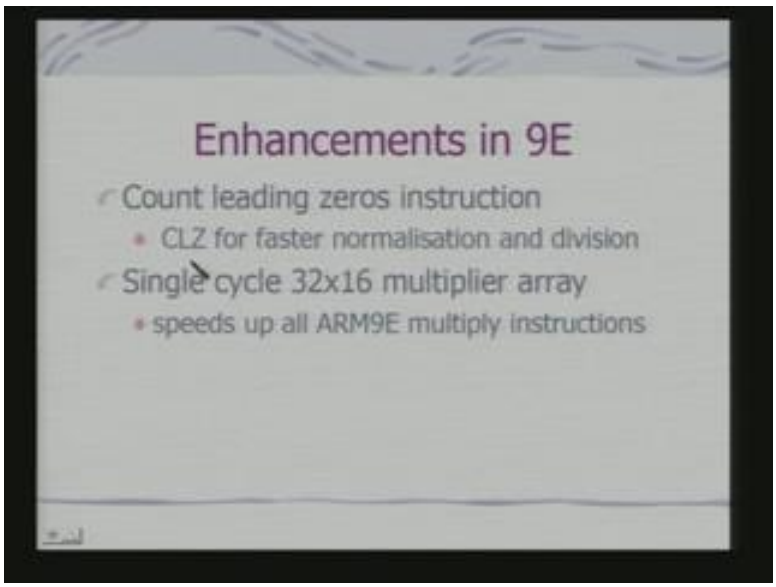
Now, that has been split, I have put in and intermediate data memory access of the buffering stage and register write stage. Because, in many cases I shall be writing the data on to the data memory, which is again distinct from my instruction mode.

(Refer Slide Time: 48:56)



Now, 9 E is a DSP enhancement. In fact, we had already discussed instruction sets. And architecturally you will find that, what is pin change is your ISA, the Instruction Set Architecture. It effectively means that, there enhancement in the ALU of the processor. So, you have got a saturation arithmetic, you have got this multiplication and accumulate operations.

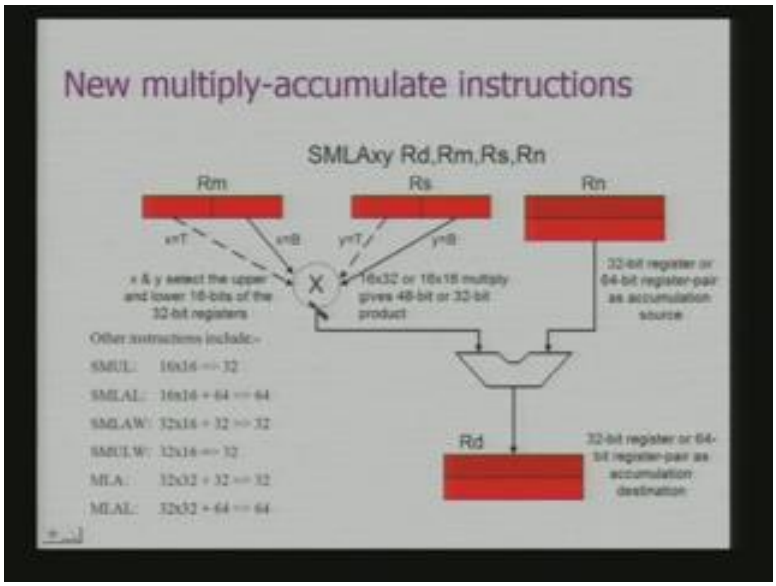
(Refer Slide Time: 49:31)



So, let us see other things is interestingly there is a CLZ count leading zero instruction. So, if this is required for first normalization and division. Because, if I have a floating

point number if I need to normalize, I need to find out how many leading zeros are there. There is a special hardware block to implement this. Because, this cannot be done using a sting standard ALU. And it has got a single cycle multiplier array, which is splits up all ARM 9E multiply instructions.

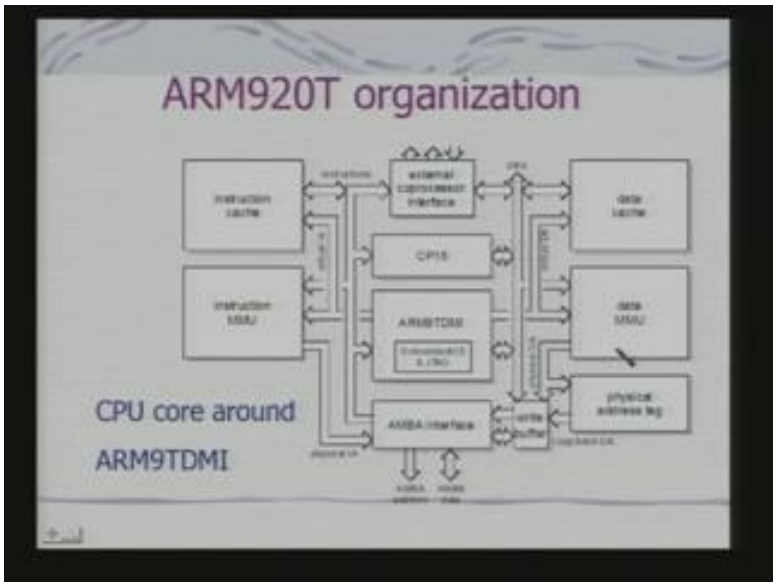
(Refer Slide Time: 50:01)



We shall just look at this enhancement, this is just a part of the data path again of ARM 9E. In fact, 9E was; obviously, when we are talking about multiply an accumulate. And this is an instruction required for primarily convolution operation, hence an enhancement for DSP application. So, what you find is, this is a basic data path enhancement. So, you have got this multiplication taking place here, you get 48 bit or 32 bit product, which is added with a 64 bit register or a 64 bit register p r accumulation shows.

So, these data can be added with the result of this multiplication and stored here. So, this is how the data path get enhanced incase of any. And this is basically your multiplier block. Because, this is between the multiplication operation, which is built in the hardware itself. So, these are the other combination of this operand which are possible.

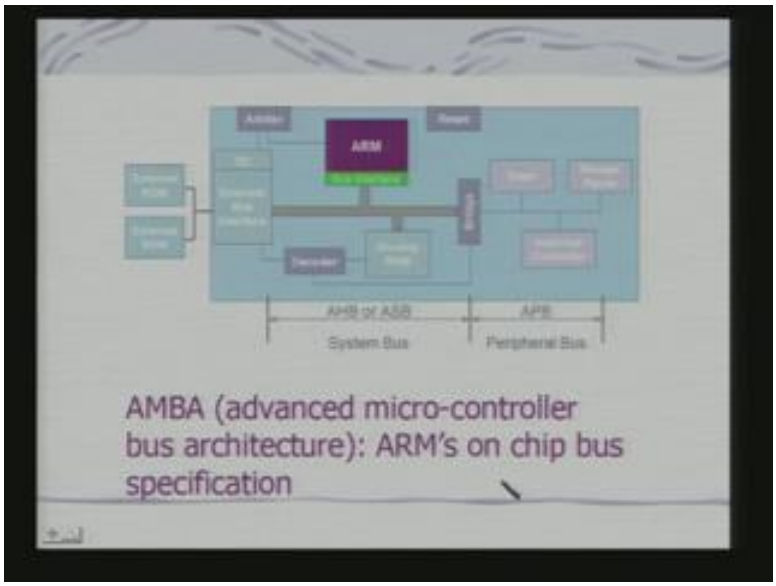
(Refer Slide Time: 50:59)



This ARM920T is basically using this as a processor core, this is as a CPU core. This is what you get the complete processor core. So, you have got an instruction cache A, instruction MMU this is the memory management unit for your instruction memory. This is than the memory management unit for your data memory. So, you got to a two MMU's.

In ARM 7 there was only one MMU, because I was having Von Neumann architecture. And this is your external coprocessor interface and this is AMBA interface. AMBA is an interface protocol, but definition of a bus.

(Refer Slide Time: 51:43)

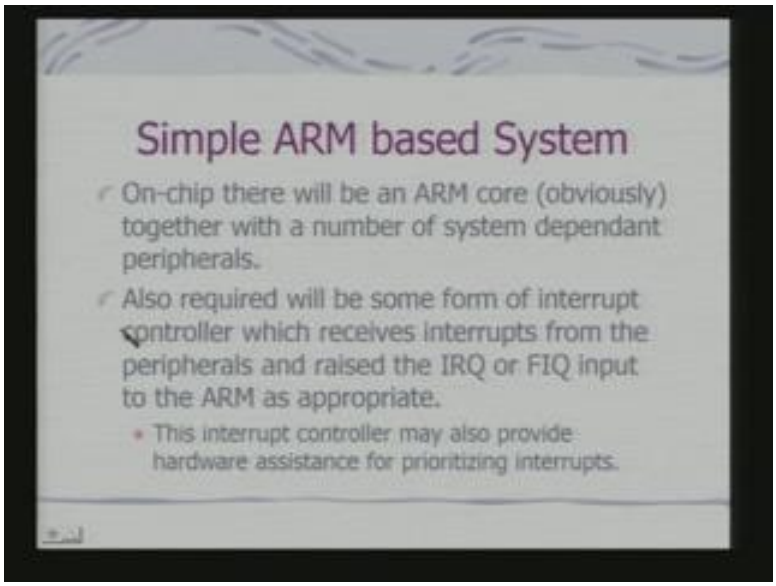


In fact, AMBA is advanced micro-controller bus architecture, which is ARM's one chip bus specification. In fact, just like on your PC if you have got a PCI bus definition. So, this is AMBA is a 1 chip bus definition. In fact, this is basically your bus structure and this is your bus interface with the ARM. And these are the other one chip peripherals, which can be connected.

So, on the main bus itself you have got a 1 chip RAM. Because, at will be first enough, the external bus interface block is also connected to the same bus. And there is a breach, breach through breach it can be connected to a peripheral bus. Because, peripheral bus can be possibly of slow bus compare to that of a system bus.

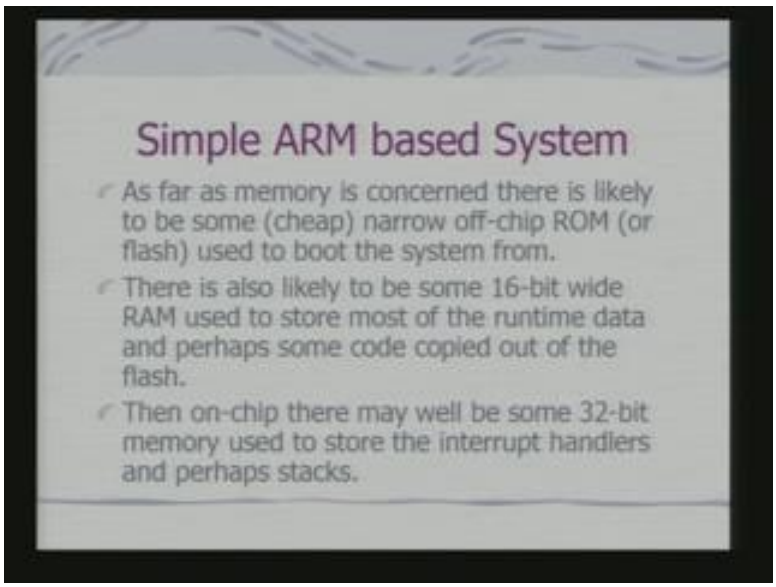
So, it is kind of a hierarchical bus organization. So, on a peripheral bus you have got your timer, you have got your interrupt controller. Because, interrupt controller will be again interface to the devices. And this is your arbiter if there are multiple bus controllers and they are trying to access the bus I have to do an arbitration. So, I need a arbiter block. So, this is very broad overview of AMBA bus.

(Refer Slide Time: 52:59)



So, a simple AMBA system what should be there. On-chip, there will be an ARM core with a number of system dependant peripherals. Also required will be some form of interrupt controller, which receives interrupts from the peripherals. And raised IRQ or FIQ input to the ARM as appropriate. So, an interrupt controller may also provide the hardware assistance for prioritizing the interrupts.

(Refer Slide Time: 53:29)

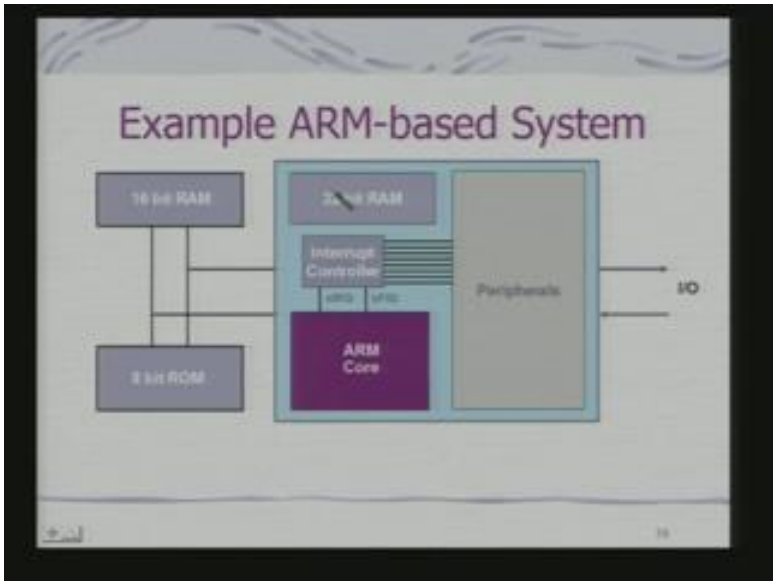


So, it is becomes the part of the hardware as for as memory is concerned there is likely to be some narrow off-chip ROM or flash, which will be used to boot the system from.

Because, during develop in face, you will be using the code from the core system maybe to test on the chip using your JTAG port. But, actually your code has to going to the flash or a EPROM in and for an embedded system.

There is also likely to be some 16 bit wide RAM used to store most of the run time data and part of some code copied out of the flash. Then, on-chip there may will be some 32 bit memory used to store the interrupt handlers. Interrupt handlers are basically the servicing for the interrupt, as well as the vectors tables and also the stacks. So, typically the picture would be something like this.

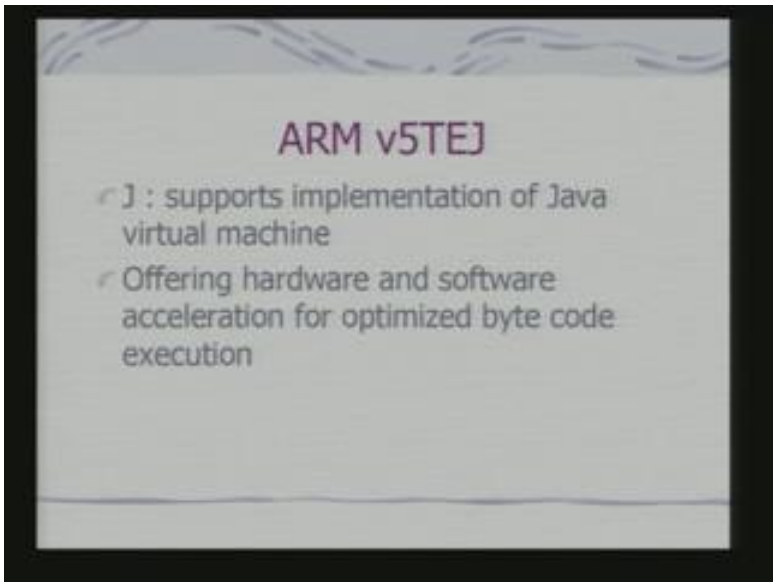
(Refer Slide Time: 54:22)



So, you have a interrupt this is what goes into the complete thing, you have got a ARM core. You have got the interrupt controller to which the peripherals are connected. Via this peripherals, you actually connect the I/O devices. And this peripherals generates this signals for the ARM core. This is the one chip 32 bit RAM, which may have the device handlers. That is the your device service routines, as well as your stack, as well as your vector. And these are the external memory.

So, external memory you can use of different kinds. These are 8 bit ROM is a 16 bit RAM. But, they have to be organize such that they provide the 32 bit address reference. So, this the very simple AMBA system. In fact, this is a generic architecture are of all ARM based embedded applications, including many of your mobile phones. In fact, Nokia actually use ARM based a source is in that implementations.

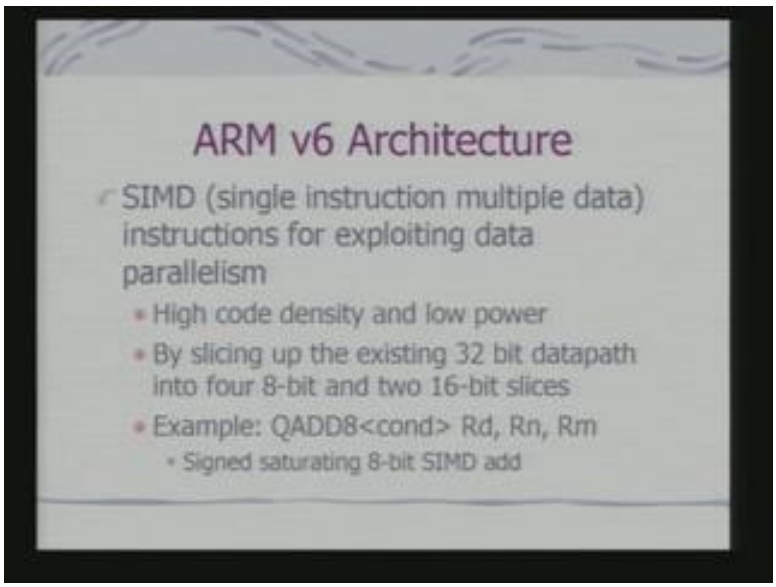
(Refer Slide Time: 55:17)



The another enhancement of this ARM 5 architecture. Since, I am talking about mobile phones. And we talk about java enable mobile phone today in the market. So, you have got an extension which is call J TEJ. J to support java virtual machine for execution of the java byte code. Because, you know java one compilation generates a byte code, which is to be interpreted by the processor.

So, that byte code interpretation request some hardware features, enabling accelerator execution. And that is put in the architecture in the J mode.

(Refer Slide Time: 56:00)

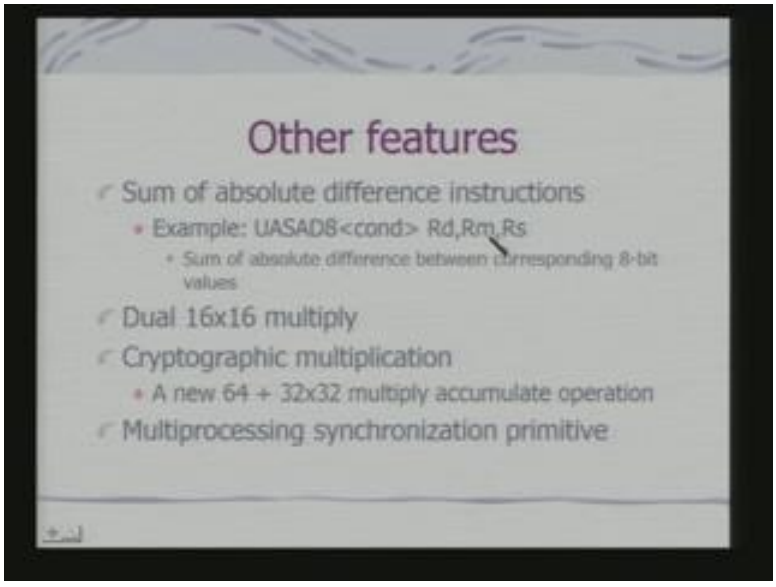


The other enhancement, which has coming with ARM version 6, which is again an enhancement for DSP target DSP application. Because, if it is going for mobile another application, you will find that your speech, your camera, your video. These are all signals which are to be processed.

So, one of the interesting feature is ARM 6 is assigned instruction set. This provides high code density and low power, what is assigned instruction is single instruction multiple data. That, it usage of single instruction to execute on multiple data. And so it exploit what we call data parallelism. A simple example is this is QADD8 now what does; that means, it is registers these are operands and they will contain the 32 bit registers. 32 bit registers means, they will have 4 8 bit data value.

So, when I am talking about QADD8, it means I am not talking about the 32 bit addition I am talking about 4 8 bit addition. And that is why, this is an example of signed instruction. So, corresponding bytes gets added and I use saturation arithmetic.

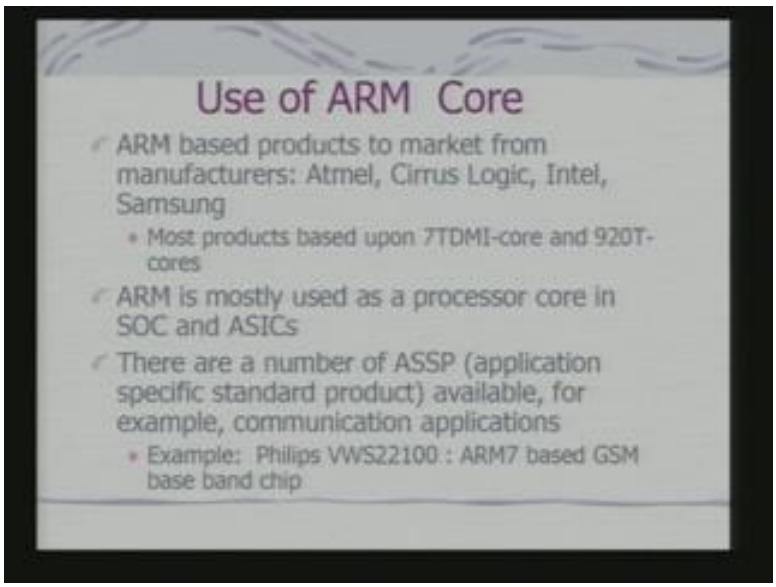
(Refer Slide Time: 57:15)



There are other features are some of obsolete different instruction. This again an example of an UASAD here I have got 32 bits. So, what I shall be doing? I shall be subtracting individual bytes, the corresponding bytes of the registers. And the absolute different would be some down. And you can consider, that if I am trying to do a compute different between two small image areas consisting of pixels.

What I shall do? I shall subtract the value of the pixels and add it up. And that would give me the different over a region. So, in this case I can use one instruction to compute different over as set of 4 pixels in the image. If I using 8 bits for pixel. It also support for Cryptography multiplication. A large multiplication, because many of the coding request large multiplications, it also has most sophisticated interface from multiprocessing.

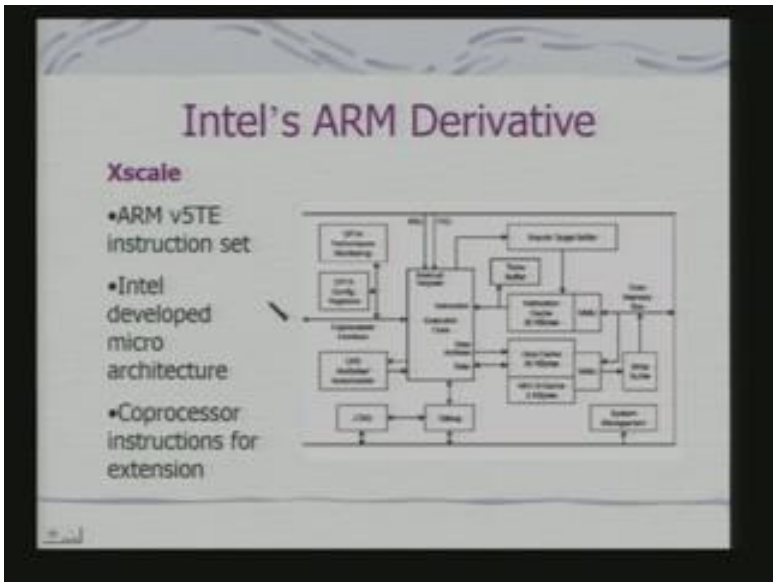
(Refer Slide Time: 58:23)



So, what we have got is variety of features in the ARM core, which enables it is use for an number of application. In fact, ARM core or ARM processor architecture has been licensed to a number of users. And they have used Atmel, Cirrus logic, Intel, Samsung there are may products out of each. And ARM is mostly used as a processor core in SOC and ASICs as well.

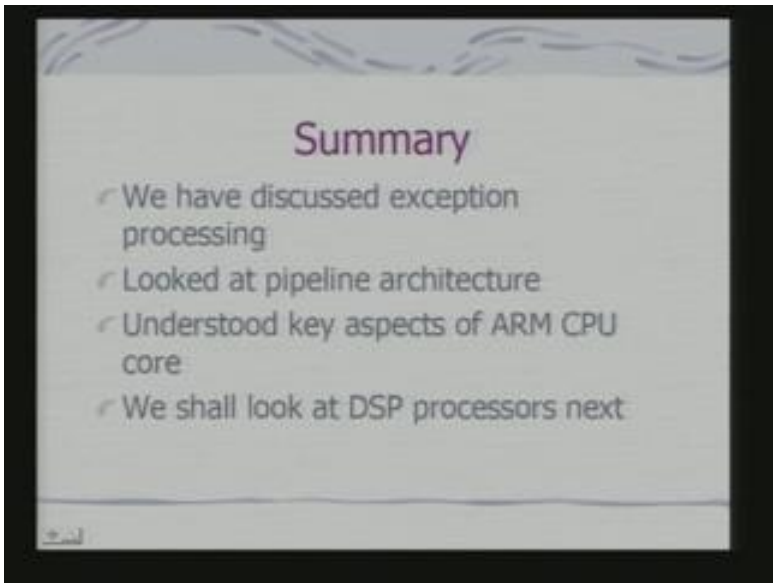
And there are example communication chips build by Philips, which is got a GSM add-on to the basic processor. So, you will find all switch such variants coming from a number of manufactures.

(Refer Slide Time: 58:57)



And Intel has made one such enhancement, which is called X scale architecture, which is built around ARM 5TE extension. And this architecture is different, Intel develop its own micro architecture and coprocessor extensions.

(Refer Slide Time: 59:15)



So therefore, what we have done by now is your understand, you have understood the instruction set architecture completely. We have understood the basic organization of processor and the CPU. And we have also understood how the complete exception processing gets handled. So, this finishes our discussion on ARM. In the next class we

shall discuss more about other DSP processors, which are used in many cases with ARM are as a SOC.