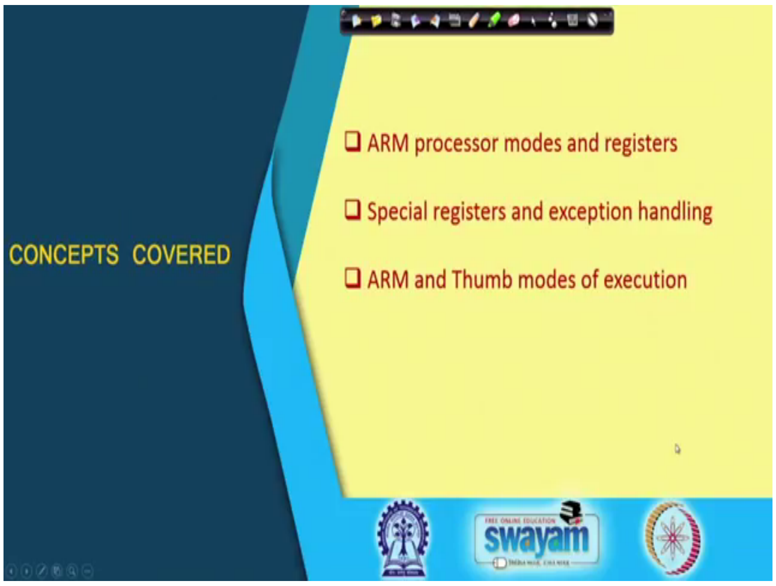
Embedded System Design with ARM

Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 06
Architecture of ARM Microcontroller (Part III)

So, we continue with our discussion. In this lecture we shall be talking about the Architecture of ARM Microcontroller, the third part of it.

(Refer Slide Time: 00:30)

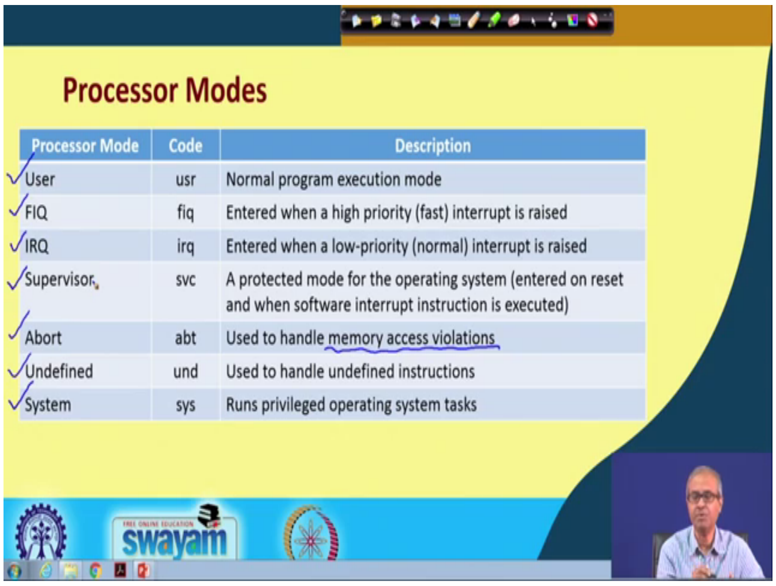


CONCEPTS COVERED

- ARM processor modes and registers
- Special registers and exception handling
- ARM and Thumb modes of execution

Now, in this lecture we shall be mainly talking about the processor modes and registers in the ARM architecture. There are some special registers, we shall be trying to tell you the necessity and roles of this special register; something about exception handling and there is something called thumb mode of execution also very briefly about that. So, again we shall not be going into very much detail of this because this course is primarily meant for a hands on demonstration and training for designing various systems, but learning some basic concepts on the architectural issues will always help you in becoming a better designer. This is the main purpose of trying to give you with some of the initial backgrounds of the embedded system design in the architectural concepts ok, fine.

(Refer Slide Time: 01:33)



Processor Modes

Processor Mode	Code	Description
User	usr	Normal program execution mode
FIQ	fiq	Entered when a high priority (fast) interrupt is raised
IRQ	irq	Entered when a low-priority (normal) interrupt is raised
Supervisor	svc	A protected mode for the operating system (entered on reset and when software interrupt instruction is executed)
Abort	abt	Used to handle memory access violations
Undefined	und	Used to handle undefined instructions
System	sys	Runs privileged operating system tasks

So, let us start to the processor modes. Now in the ARM processor during execution at a given time, the processor can be in one of 7 modes, and this table summarizes these 7 processor modes. The user mode is the normal mode during execution of a program.

When a program is executed normally, we say that the system is in user mode. This user mode the acronym is usr, we sometimes denoted by usr, this is the normal program execution mode. Now you know in processors, we can have interrupts from external devices some interrupt signal might come. If it comes, the program that is executing will be suspended, we will have to go to some interrupt handling routine run that routine and then again come back and resume the interrupted program. Now in ARM two different levels of interrupt processing are permissible or allowed one is called high priority or other is called normal priority.

Now, this FIQ is the high priority mode. This FIQ mode is entered when a high priority interrupt is activated and is acknowledged. Now the point to note is that when a high priority interrupt is being processed, if some lower priority interrupt comes in the meantime; they will not be acknowledged, they will be ignored. So, higher priority interrupt will be having real higher priority over the lower priority ones.

They will be handled first. And IRQ is the low priority or the normal priority interrupts. This IRQ is the processor mode which is the mode you see this FIQ and IRQ are saying is the mode when the corresponding interrupt handler routine is executing that interrupt has come we are handling the interrupts. During that time we say that processor is in either FIQ mode or in IRQ mode well.

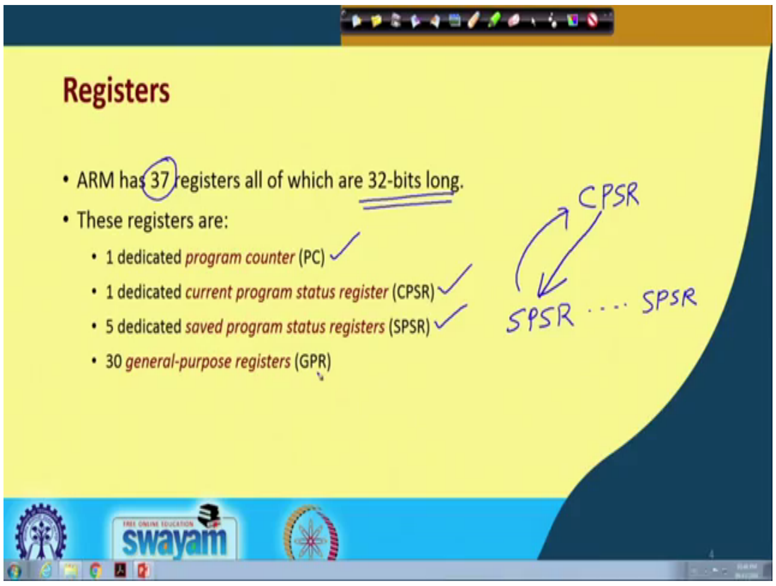
There is a supervisory mode which most of the modern processors they have, there are some instructions like supervisory called trap; these are or you know there is a instruction called SWI software interrupt. There are various names to the syndromes, but the purpose of this instructions is that, these instructions are some ways to transfer control to the operating system. Well in a computer where there is an operating system, these instructions allow the processor mode to be changed from user mode to supervisor mode and then just like a subroutine called control will jump to the supervisor or the operating system right; this is what is done here.

Now, in the supervisory mode is svc. This is as it is mentioned here a protected mode and this mode is required only when there is an operating system in your implementation well. In all embedded system application you will not require this, but in system where there is really an operating system and you need some protection you need to have this mode of execution. And abort is a mode again; this is an optional mode for cases where you want to have memory protection that you want to see that when a program is executing; you are supposed to access only this region of memory. If accidentally your program tries to access any memory location beyond the permissible limits, this abort interrupt will be generated and the corresponding processor mode is called the abort mode; for handling memory access violations the processor goes to the abort mode right.

And undefined is actually something which is kept for future expansion. Well some instruction opcodes have not been defined, they are undefined. So, if by mistake you are trying to execute an instruction whose opcode is undefined, you get an undefined interrupt and the processor mode goes to undefined state und. And there are instances where you want to run some privileged operating system tasks for that you typically go to this system mode.

The system mode and supervisory mode are very much related. So, I am not going into detail of this because for an embedded system design, these are not really required very really we will be requiring these modes in an actual system implementation ok. So ARM supports all these modes. So, ARM is quite flexible in terms of its architectural features copy of registers. ARM is a large set of general purpose registers as I mentioned earlier.

(Refer Slide Time: 07:25)



Registers

- ARM has 37 registers all of which are 32-bits long.
- These registers are:
  - 1 dedicated *program counter* (PC) ✓
  - 1 dedicated *current program status register* (CPSR) ✓
  - 5 dedicated *saved program status registers* (SPSR) ✓
  - 30 *general-purpose registers* (GPR)

So, so in total there are 37 registers. All of these registers are 32 bit long. Talking about dedicated registers, there are 7 registers which have specific purpose; 1 is the program counter PC. It is called there is 1 which is called current program status register. Well for those of you who are familiar with the some micro processors some architecture like 8085 or any other microprocessor, we will be knowing that there is something called condition flags.

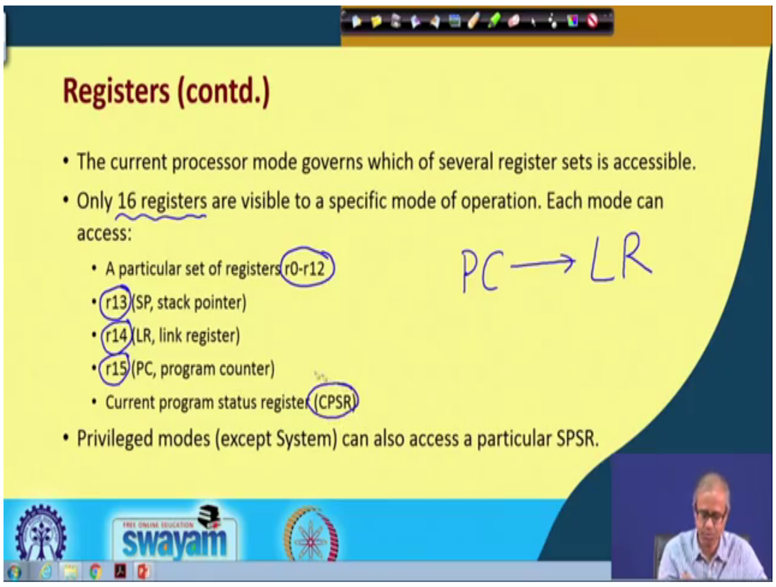
Whenever some instructions are executed this condition flags are set or reset there is 0 flag carry flag overflow flag and so on. So, this CPSR or the current program status register is like a flag register. It consists of several flags which are set depending on the mode of the processor or the instruction execution right.

And there are 5 saved program status register. Now this concept is interesting. There is one CPSR and there are several SPSR, saved program status registers. You see in a conventional processor whenever an interrupt comes, what do we do normally? We save all the registers in the stack that can include also the status registers in the flags. Go to the interrupt handler, finish everything, come back restore all registers and status register from the flag and resume execution, but in ARM there is no support for stack, there is no instruction to push or pop instructions in stack or from stack.

So, pushing and popping the status into stack you cannot do like that. Just for that reason what is done? There is a special CPSR copies which are maintained; suppose a program is running current status is stored in CPSR, there is an interrupt that has come. So, the content of the CPSR will get copied into an SPSR then interrupt handler will run off before coming back the content of the SPSR will be restored back into CPSR. So, it will be restored back.

So, there is no stack as such it is the users responsibility to restore the value from the CPSR and then again move it back and forth right. And of course, there are 30 general purpose registers; these are named typically r 0 to r 29 fine.

(Refer Slide Time: 10:39)



Registers (contd.)

- The current processor mode governs which of several register sets is accessible.
- Only 16 registers are visible to a specific mode of operation. Each mode can access:
  - A particular set of registers r0-r12
  - r13 (SP, stack pointer)
  - r14 (LR, link register)
  - r15 (PC, program counter)
  - Current program status register (CPSR)
- Privileged modes (except System) can also access a particular SPSR.

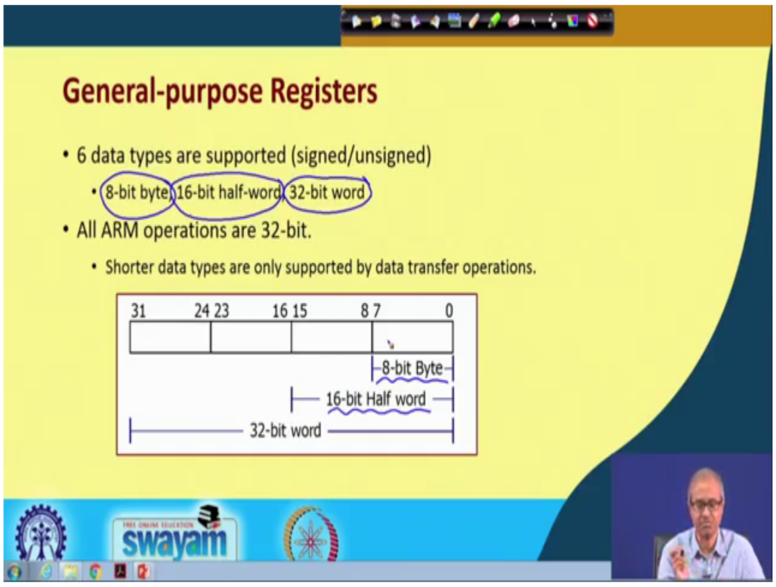
$PC \rightarrow LR$

Now, depending on the processor more, it depends how many of these registers you can actually access. Now 16 registers are typically visible in a specific mode of operation; this 16 registers are as follows.

There are 13 general purpose registers r 0 to r 12 stack. There is no stack I told, but r 13 is your mark as the stack pointer. If you want to implement a stack yourself by software, you can use this r 13 for the purpose and r 14 is a link register. You see for subroutine call instructions let us say for a conventional processor, the value of the program counter is also pushed in the stack, but here there is no stack. So, there is a special register called the link register, the value of the program counter gets copied into the link register. Whenever there is a subroutine call and when you are returning back from the subroutine whatever is then the link register is copied back into program counter.

So, you start from there again right and program counter is accessed as r 15 and of course, current program status register. Now here I shall show a picture later which will show you the different registers, how they exist in different modes.

(Refer Slide Time: 12:29)



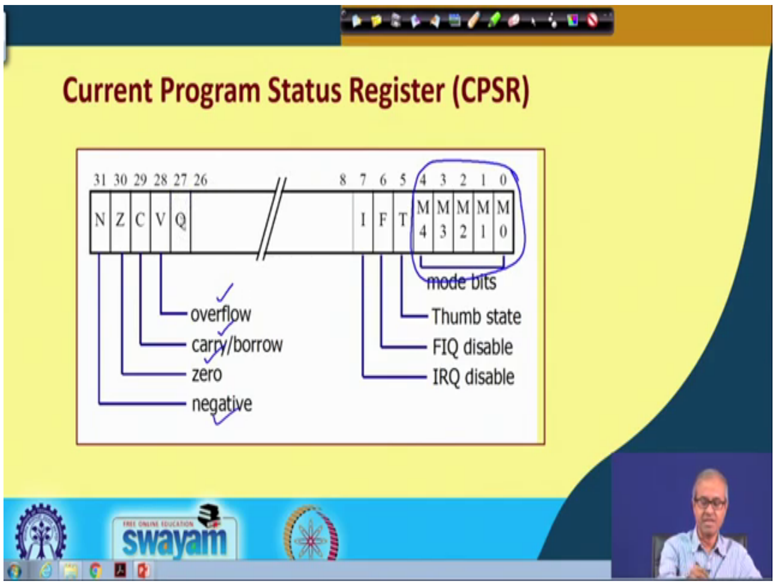
General-purpose Registers

- 6 data types are supported (signed/unsigned)
  - 8-bit byte, 16-bit half-word, 32-bit word
- All ARM operations are 32-bit.
  - Shorter data types are only supported by data transfer operations.

Now, in terms of the general purpose register where you are expected to store some data, I told you the registers are all 32 bits in size. But in terms of the operations which are supported, you can have 8 bit operations 16 bit which are called half word operations or full 32 bit word operations. So, in terms of the 32 bit word, the whole 32 bit is referred to as the word.

And when you are using half word operations the lower 16 bit is used, this is the half world and for byte operations the last 8 bits are used right. So, you should remember this. When you are using a byte operation, the last 8 bits of the register will be used and for half word last 16 bits will be used ok. And these are not used for athletic operations, only for data transfer operations because all arithmetic operations are only 32 bits in size. But when you are transferring data from one place to another, you can transfer 8 bits or 16 bits or 32 bits. From memory you can transfer 8 bits of data into a register, it will go into the last 8 bits, fine.

(Refer Slide Time: 13:58)

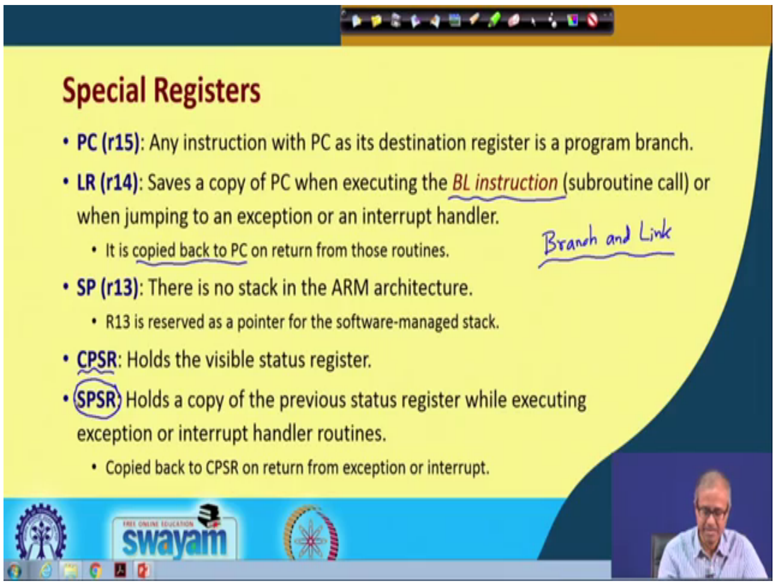


Current Program Status Register (CPSR)

Now, current program status CPSR, so what does it contain? So, it contains as I had set the flags. So, what are the flags? There is a overflow flag denoted by V, there is a carry flag C for subtract operation; you call it the borrow, there is a zero flag it checks whether the result is zero or nonzero and the sign flag negative or positive right. Now in addition the processor can be in many modes.

Well processor I said there are 7 modes, but to keep with future expansion ARM keeps 5 bits to store mode. So, in 5 bits you can support up to 0 to 32; 32 modes can be supported maximum and this bit number 5 6 and 7, they are meant for thumb state. Then 6 and 7 are for enabling and disabling the interrupts the high priority interrupts and the normal prior to interrupt. If these bits are set to 1, these are disabled if there is 0 they are enabled; the other bits in between they are normally not used, they are unused.

(Refer Slide Time: 15:26)



Special Registers

- PC (r15): Any instruction with PC as its destination register is a program branch.
- LR (r14): Saves a copy of PC when executing the BL instruction (subroutine call) or when jumping to an exception or an interrupt handler.
  - It is copied back to PC on return from those routines.
- SP (r13): There is no stack in the ARM architecture.
  - R13 is reserved as a pointer for the software-managed stack.
- CPSR: Holds the visible status register.
- SPSR: Holds a copy of the previous status register while executing exception or interrupt handler routines.
  - Copied back to CPSR on return from exception or interrupt.

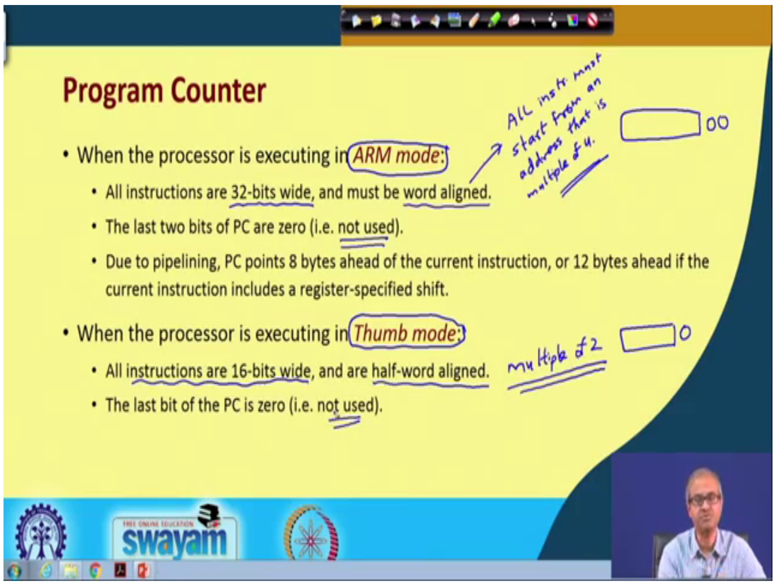
Branch and Link

Now, talking about the special register just a relook again this r 15 is the program counter. So, as you know so program counter always holds the address of the next instruction in memory to be executed right. So, if the destination register is program counter which means you are jumping to there ok. Link register r 14 is here marked as a link register for subroutine call instruction.

Now, in ARM the subroutine call instruction is called BL branch and link this BL stands for branch and link. Branch and link means will in the BL instruction, it will be specifying the address where to branch. Before branching, the current value of the program counter will be saved into LR r14 and then the destination which is specified will be loaded into PC. So, that jump will take place right. So, when you are coming back, the value of LR will be copied back into PC.

So, you resume execution from where you left and r 13 is a you are not from stack of course, there is no stack in ARM, I told you but it is reserved as a pointer. If you want to implement it in software, you can do it. And CPSR as I told you this holds the current status register with respect to the program that has been executed and whenever some exception or interrupt comes, the CPSR gets copied into one of the special or saved program status registers SPSRs. They will hold a copy of the CPSR. So, when you come back from the exception routine that a SPSR has to be copied backed into CPSR before you resume execution; this is how they work.

(Refer Slide Time: 17:45)



Program Counter

- When the processor is executing in ARM mode:
  - All instructions are 32-bits wide, and must be word aligned.
  - The last two bits of PC are zero (i.e. not used).
  - Due to pipelining, PC points 8 bytes ahead of the current instruction, or 12 bytes ahead if the current instruction includes a register-specified shift.
- When the processor is executing in Thumb mode:
  - All instructions are 16-bits wide, and are half-word aligned.
  - The last bit of the PC is zero (i.e. not used).

All instrs must start from an address that is multiple of 4.
00
multiple of 2
0

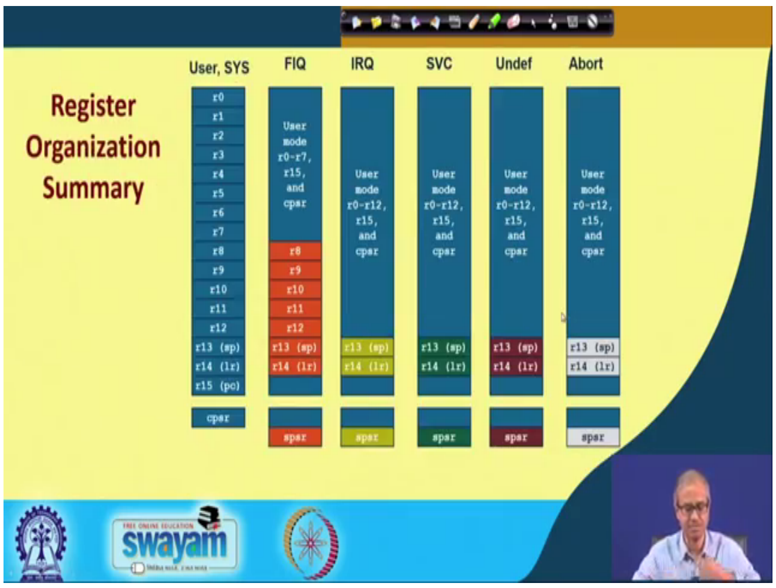
Now, talking about the program counter, there are two modes of execution; one is the ARM mode, one is the Thumb mode. Let us make the distinction clear here. We first talk about the ARM mode of execution which is the default mode of execution. ARM mode says that all instructions are 32 bit wide. There are 32 bit instructions and their word aligned. Word aligned means all instructions must start from an address memory address I mean that is some multiple of 4. This is what we mean by what aligned. They need for doing this is that well again I am not going to detail of this.

When you interface memory with the microcontroller, the processor there is a concept called memory interleaving. If you have 4 way interleaving, then 4 consecutive bytes starting from an address which is a multiple of 4 can be transferred in a single clock cycle. But if it is not so, you will be requiring 2 clock cycles. So, your memory transfer will become slower just to have faster memory access ARM insists on word alignment and multiple of four means for the last two bits are 0. So, any address with the last two bits 0 means it is a multiple of 4. Therefore, when it is in ARM mode, the program counter is expected to hold the address of our instruction right. So, always the last 2 bits will be 0 of the program counter.

So, the last two bits of the program counter are actually not used in the ARM model they will always be 0. So, this already you have mentioned the PC can point to 8 bytes ahead or 12 by depending on the number of stages. But there is another mode of execution whenever you do not need the full power of 32 bit processing maybe we are using the ARM processor in a very simple application, where smaller instructions 16 bit instructions are there a subset instruction subset. Here in the thumb mode, the instructions are 16 bit wide; that means, 2 bytes and they are half word aligned means the instruction addresses are multiple of 2. So, earlier it was multiple of 4, it is multiple of 2 multiple of 2 means the last bit of the address will be 0.

So, in the program counter, the last bit is 0 which is not used right. So, this is the main difference between the ARM mode and a thumb mode. ARM mode is a superset where all instruction encoding a 32 bits thumb mode is a subset of that where you make all the instructions shorter 16 bits because of which may be the total program memory will be requiring too much smaller, your code density will be much higher. For small applications, you can have much cheaper mode of you can say means implementation right. So, thumb mode is used for such cases where you need small systems.

(Refer Slide Time: 21:50)

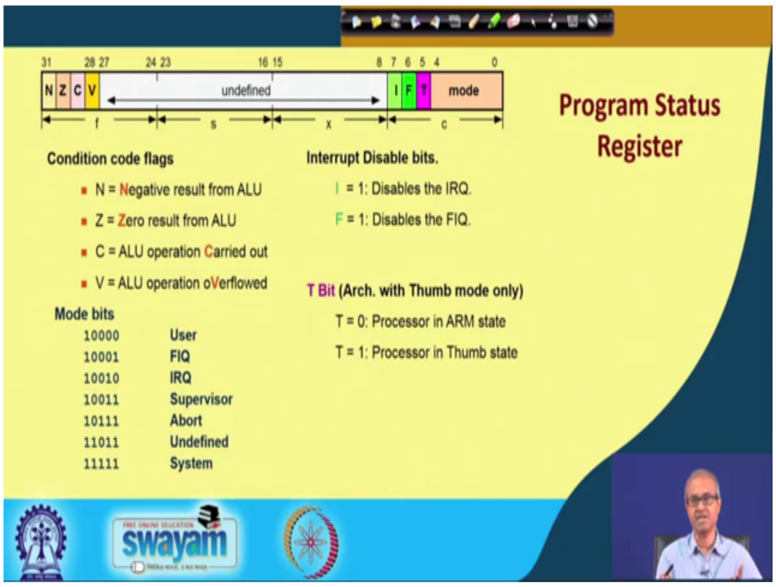

Register Organization Summary
User, SYS
FIQ
IRQ
SVC
Undef
Abort
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
User mode r0-r7, r15, and cpsr
User mode r0-r12, r15, and cpsr
spsr

Now, this diagram gives you an overview about the registers in the different modes I told you. Well in the in the user or the system mode, well you have access as you can see to the 13 registers r 0 to r 13 then r 13 14 15 spv link and PC and the CPSR and the 4 SPSRs which are there, they correspond to the other 5 modes. If I interrupt FIQ comes then CPSR gets copied into this SPSR and this r 0 to r 7 will be there in FIQ, but the other registers will be specific to a FIQ. If you come back, they will again change and in

IRQ r 0 to r 12 the whole thing is there, only r 13 r 14 are specific to FIQ. SVC also and if abort also.

So, this depends which are the registers which can be accessed when you are writing the interrupt handler right like you are when you are writing the first interrupt handler a FIQ interrupt handler, you are allowed to use only the registers r 0 to r 7 in the user mode and these are the specific registers which you can use, they will be automatically used in the FIQ mode. And they will not override the original registers here right, you need not have to save restore them. There is another copy of those registers in the register bank fine.

(Refer Slide Time: 23:36)



Program Status Register

Condition code flags

- N = Negative result from ALU
- Z = Zero result from ALU
- C = ALU operation Carried out
- V = ALU operation oVerflowed

Mode bits

10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

Interrupt Disable bits.

I = 1: Disables the IRQ.

F = 1: Disables the FIQ.

T Bit (Arch. with Thumb mode only)

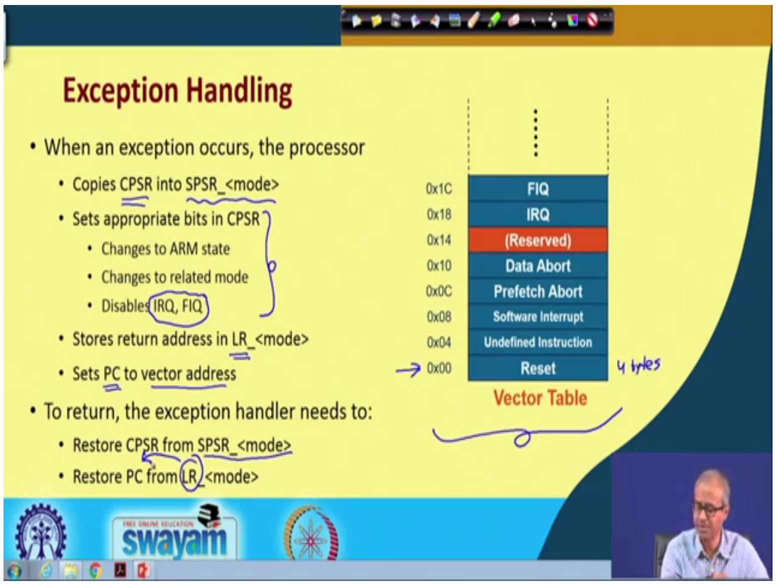
T = 0: Processor in ARM state

T = 1: Processor in Thumb state

Now, talking about the CPSR once more, this is this is again the picture of the CPSR I told earlier. The last mode the 5 bits, now the specific bit combinations are shown here. For these 7 modes, the mode bit combinations are defined like this 1000 the means user mode 1001 is a FIQ mode and so on.

And condition code flag, I already mentioned negative 0 carry and overflow and interrupt disable if it is 1, the corresponding interrupts are disabled and thumb mode this bit indicates whether you are processing in the ARM mode or the thumb mode. If the bit is 0, then you are in a ARM mode. If it is 1, you are in thumb mode. So, in this way it is decided whether which mode of execution you are selecting 32 bit mode or 16 bit mode.

(Refer Slide Time: 24:38)



Exception Handling

- When an exception occurs, the processor
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate bits in CPSR
    - Changes to ARM state
    - Changes to related mode
    - Disables IRQ, FIQ
  - Stores return address in LR_<mode>
  - Sets PC to vector address
- To return, the exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

4 bytes

Now, talking about exception handling, there is a process like this here followed; there is an interrupt vector kind of a table this is called a vector table. This vector table is present from address 0 on 0x00 means 0x means hexadecimal 00; these are hexadecimal numbers 0004080C. So, each of these entries are 4 bytes long this can be seen from the address next one is 04080C10 and so on.

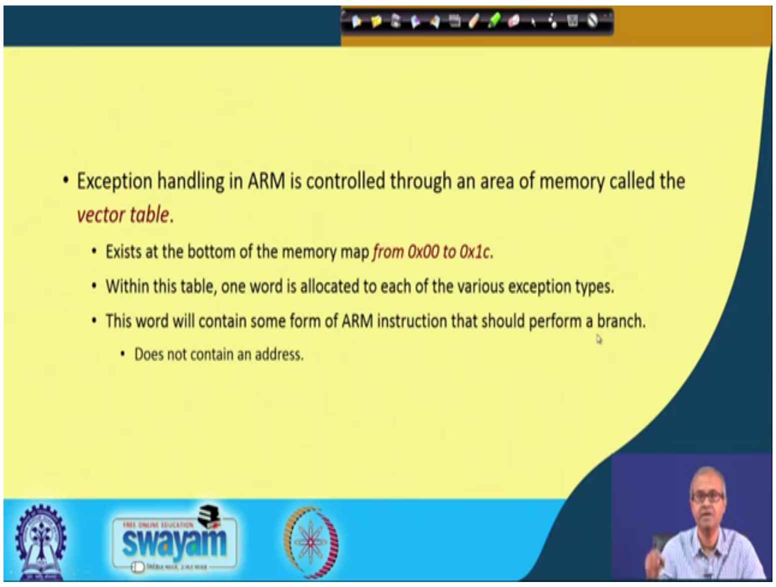
So, whenever any exception occurs exception is interrupt memory where access violation and define instruction all these are exceptions some kind of interrupt or the supervisory call. So, whenever this happens, the CPSR will be copied into the corresponding SPSR. So, you know there are 5 SPSRs depending on the type of exception CPSR will be copied to the correspondings saved program status register.

And the corresponding bits of the CPSR will be set depending on which mode you are in and during interrupt processing; the interrupt bits will be disabled because if they are enabled then another interrupt may come in between also. So, in order to avoid nested interrupt from coming while you are doing the interrupt handling interpret, bits can be disabled. Then before coming back well the return address will have to store in this LR register corresponding to that mode you see, there are separate LR registers for the different modes and PC is loaded with the vector address. If you jump to this for return, the exception handler will do the reverse thing. So, it will be loading back that SPSR into

CPSR equal to copying to a spacer CPSR and LR will be continuing the return address; it will be transferring it to PC ok. So, it will be jumping back.

Now the point to notice that here the entries out here, these are not addresses of this interrupt handler rather these are instructions. Typically these are jump or call kind of instructions. So, when you come here reset so, there will be a jump from here to the routine which is handling reset. Well if it is let us say this FIQ so, if you come to 1C, here there will be an instruction which will be jumping to the interrupt handler for FIQ and so on. So, these are all one word instructions that are stored in the vector table. These are not addresses of the routines which are stored it is the actual instruction some instruction that is stored here right.

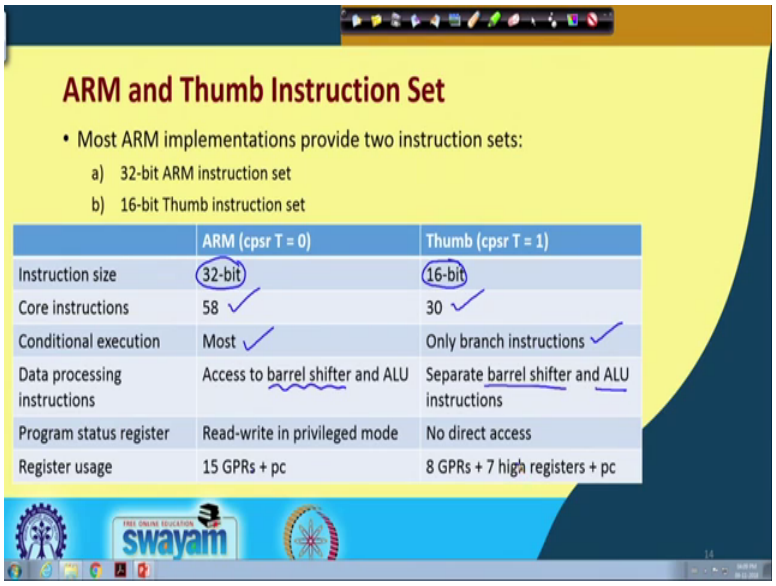
(Refer Slide Time: 28:58)



- Exception handling in ARM is controlled through an area of memory called the *vector table*.
  - Exists at the bottom of the memory map *from 0x00 to 0x1c*.
  - Within this table, one word is allocated to each of the various exception types.
  - This word will contain some form of ARM instruction that should perform a branch.
    - Does not contain an address.

So, this is what I have already mentioned. The vector table is stored from address 0x00 to 0x1c. So, one word allocated for every exception type and as I have said, this word will contain some instruction; some jump some branch instruction ok. It will branch to the corresponding interrupt handler. So, it does not contain any address you should remember this. This already I have mentioned.

(Refer Slide Time: 28:27)



ARM and Thumb Instruction Set

- Most ARM implementations provide two instruction sets:
  a) 32-bit ARM instruction set
  b) 16-bit Thumb instruction set

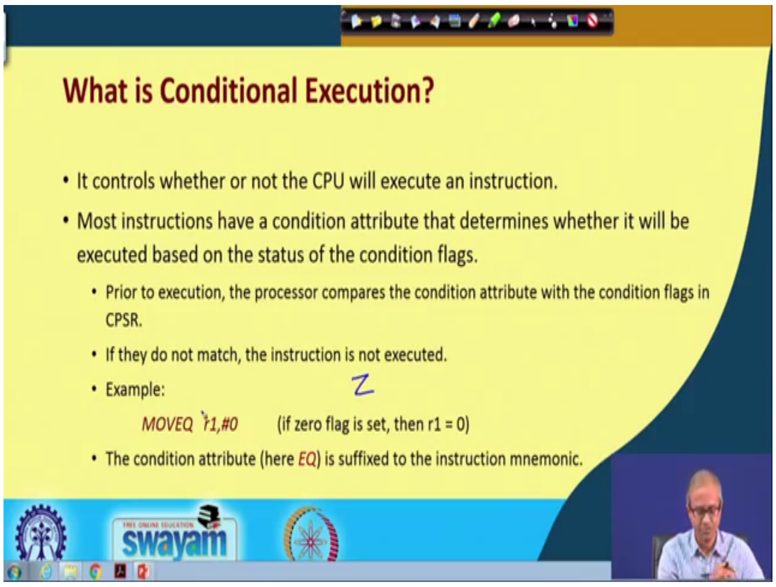
	ARM (cpsr T = 0)	Thumb (cpsr T = 1)
Instruction size	32-bit	16-bit
Core instructions	58 ✓	30 ✓
Conditional execution	Most ✓	Only branch instructions ✓
Data processing instructions	Access to barrel shifter and ALU	Separate barrel shifter and ALU instructions
Program status register	Read-write in privileged mode	No direct access
Register usage	15 GPRs + pc	8 GPRs + 7 high registers + pc

So, I am talking about the ARM and thumb instruction set a quick comparison. So, in ARM mode the T bit in the CPSR is set to 0 and for thumb mode set to 1. The differences are as follows. Instruction size in ARM mode, there are 32 bit instructions; in thumb mode as I said there are 16 bit instructions. The number of main instructions are 58 and 30. Conditional execution I told you, conditional execution is a feature where some instruction will be executed depending on some condition is true or false.

So, in ARM we had almost all the instructions support condition execution, but in thumb only for branch instruction you can check for zero-non zero, carry-no carry, but other instruction do not support conditional execution; just only the branch just the normal case. Data processing instructions in ARM mode, it supports barrel shifting like I told you when you add 2 numbers, you can say you add the second numbered shifted left by 10 positions. This will be done automatically by the barrel shifter, but in thumb such instructions are not there, here there are separate shift instructions and there are separate ALU instructions.

So, there is no single instruction that combines shifting and arrow adding together such instructions are not there in thumb. Program status register in ARM mode, you can do read write in privilege mode, but in thumb mode you cannot access those program status register; those are prevented from being accessed. And for registers in ARM mode, you can use the 15 GPR general purpose registers r0 to r14 and the PC, but in thumb you can use the eight GPRs, 7 high registers and PC. It is the high register means the high order 16 bits of the registers. So, there are some specific registers which can be accessed. So, only a subset can be accessed here that is what is meant.

(Refer Slide Time: 31:01)



What is Conditional Execution?

- It controls whether or not the CPU will execute an instruction.
- Most instructions have a condition attribute that determines whether it will be executed based on the status of the condition flags.
  - Prior to execution, the processor compares the condition attribute with the condition flags in CPSR.
  - If they do not match, the instruction is not executed.
  - Example: Z

    MOVEQ r1,#0 (if zero flag is set, then r1 = 0)
  - The condition attribute (here *EQ*) is suffixed to the instruction mnemonic.

Conditional execution told you, let me take an example here. We shall be discussing some more cases of condition execution later. Conditional execution it controls whether or not CPU will execute the current instruction. Now most instruction in ARM, it allows a condition attribute to be specified. Let me take an example an instruction like this. Normally if you do not use this EQ, just imagine this EQ is not there then move r1 hash 0 means, this r1 is initialized to 0 right. Now if I give this MOVEQ, this means I am checking this zero flag in this CPSR. It says if the zero flag is set which means equal condition is true, then only you do the move otherwise you do not do the move.

So, this instruction will be executed depending on certain condition ok. So, this is how the conditional executions exist. We shall see later that using this kind of condition execution just allows us to prevent branch instructions in many cases. It makes the code density higher less number of instructions.

So, with this ah, we stop with our brief discussion on the ARM architectures. From the next lecture onwards, we shall be moving on to some aspects about the ARM instruction sets and the other features about ARM which will be helpful in the actual experimentation that we shall be showing you we shall be giving a demonstrations on.

And all the demonstrations that we shall be showing you would be based on some ARM modes and also a few experiments, we shall be showing you on the arduino boards ok. This we shall be seeing over the next few weeks.

Thank you.