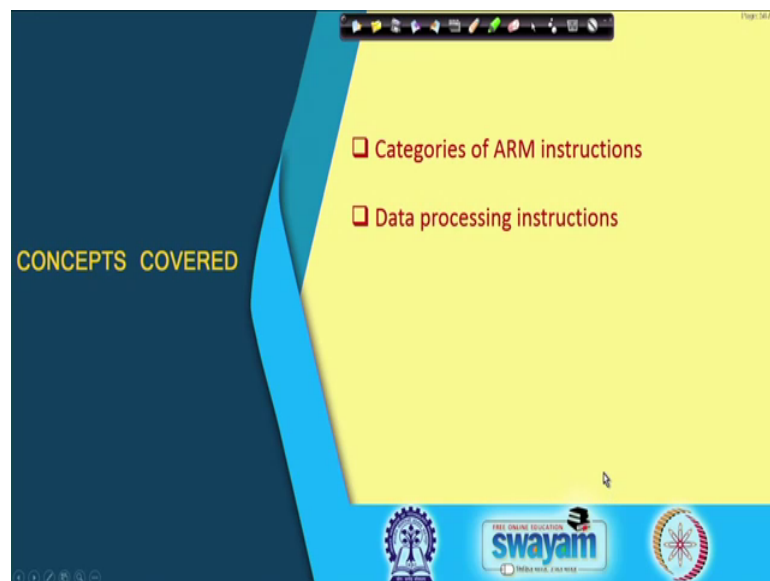


Embedded System Design with ARM
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 07
ARM Instruction Set (Part I)

We shall now start a very short discussion on the ARM instruction set. Now, you shall see later as part of this course, we shall be showing you lot of demonstration on some ARM based boards and also some Arduino boards. There we shall be writing our programs in a higher level language, but here in this and a couple of other lectures we shall be discussing about the low level features of the ARM processor and for that we need to have a basic idea about the assembly language features that ARM architecture provides. So, the title of this lecture is ARM Instruction Set, the first part of it.

(Refer Slide Time: 01:07)



Now, here we shall be broadly talking about the various categories of ARM instructions and in particular the data processing instructions that are provided by the ARM instruction set architecture, ok.

(Refer Slide Time: 01:27)

The ARM Instruction Set

- ARM instruction can be categorized into three groups:
 - a) Data processing instructions
 - Operate on values in registers
 - b) Data transfer instructions
 - Move values between registers and memory
 - c) Control flow instructions
 - Change the value of the program counter (PC)

The diagram illustrates the Harvard architecture of the ARM processor. It shows a central box containing 'Registers' and 'PC' (Program Counter). To the right, there are two separate memory blocks: 'Program Memory' and 'Data Memory'. Arrows indicate the flow of data: from Program Memory to Registers, from Registers to Data Memory, and from Data Memory to Registers. The PC is also connected to the Program Memory. The slide includes a Swamyam logo at the bottom left and a small video inset of a man speaking at the bottom right.

Talking about the ARM instruction set, you see as a developer you need to develop or design an embedded system, you will be having a hardware platform, a microcontroller based system and you have to write a software for it.

Today most of us develop the software in some high level language either in C or in Python or in Java in some language like this, but it is always good to know the lower level features of the processor in order to have an informed decision that which processor may be better for a particular application, ok. So, it is with this motivation we are giving you a brief introduction to the assembly language features of the ARM processor which is a reflection of the hardware features that are provided by the ARM platform, ok.

So, broadly speaking well any instruction set not necessarily the ARM instruction set they can be categorized into various groups or broad categories. So, in ARM let us define these categories as data processing, data transfer and control flow. So, the instructions which are there they can be categorized or classified into one of these three types. Let us try to understand.

Now, in a microcontroller I have said most of the modern microcontrollers not the first generation ARMs the subsequent ones they are based on the Harvard architecture. So, you will be having a separate program memory and a separate data memory. So, the instructions which constitute the program that will be stored in the program memory, while all temporary data which will be manipulating on will be stored in the data

memory. So, among the registers this program counter is one register which will be pointing to the address of the next instruction in the program memory, right. So, whenever a new instruction is brought or fetched from memory it will be fetched from the address which is stored in the program counter.

So, when I am talking about the data processing instructions we are talking about carrying out various arithmetic and logic operations on data. Now, ARM is based on the risk philosophy of architectures most of the ARM features are risk based reduced instruction set computer based. Now, here all data seeing instructions they operate only on registers, they work only on register; that means, when I say add we will be adding the contents of two registers and storing the result back into another register. So, memory is not coming into the picture here at all, ok.

Now, when we talk about data transfer here there are options we can transfer from one register to another or we can transfer from register to data memory or data memory to register. So, these options are all there these will constitute data transfer instructions. And, control flow instructions are those which will alter the sequence of execution of a program. Normally as I said program counter will be pointing to the next instruction to be executed, now, as the instructions are executing the value of the program counter gets incremented consecutively. But, whenever you have some instructions like jump or a subroutine call suddenly that sequence will be disturbed you will be going to some other address and from there you will be starting to fetch your instructions, right.

So, this is what is meant by change of control flow and such instructions are termed as control flow instructions. So, essentially control flow instructions are those which will be altering the value of program counter in some way so that the normal sequence of instruction execution will be altered, ok fine.

(Refer Slide Time: 06:17)

(a) Data Processing Instructions

- All operands are 32-bits in size:
 - Either registers
 - Or literals (immediate values) specified as part of the instruction
- The result, if any, is also 32-bit in size and goes into a specified register.
 - One exception: long multiply, that generates 64-bit results.
- All operand and result registers are independently specified as part of the instruction.

So, we first talked about the data processing instructions that are present in the ARM instruction set. Now, in the ARM instruction set the first thing to notice that the registers are all 32-bit in size, the arithmetic logic unit which is inside the processor that also has the capability of operating on 32-bit numbers, ok. So, all operands that are being operated on are 32-bits in size this is something fixed in ARM and this operands can be either registers. So, you may carry out these operations on registers or some of the operands may be constants these are called literals or immediate values. I may say add the value 10 to the content of a register. That 10 is like a constant that is called a literal or an immediate value, that value 10 is specified as part of the instruction that is called immediate operand right.

And, the result after the calculation is also a 32-bit result and this will be stored in some particular register there is of course, one exception there is a special multiply instruction where the result is stored as a 64-bit number. Now, you know whenever we multiply 2 n bit numbers the result can be twice n bits the result can become double in size. So, when you multiply two 32-bit numbers the result can be maximum 64-bit in size. So, there is a version of multiply instruction where the result can be stored in two registers 64-bits, this is an exception, and the other thing is that all operands and the result registers that we use in an instruction will be specified as part of the instruction.

Now, in these lectures we shall not be discussing about the instruction encoding; that means, exactly how the bits of the instruction word specify the registers; we shall not be going into that detail of the architecture, but essentially what kind of instructions are supported those we shall be discussing, ok.

(Refer Slide Time: 08:55)

• **Arithmetic instructions:**

ADD r0, r1, r2 ; r0 = r1 + r2

ADC r0, r1, r2 ; r0 = r1 + r2 + C (C is carry bit)

SUB r0, r1, r2 ; r0 = r1 - r2

SBC r0, r1, r2 ; r0 = r1 - r2 + C - 1

RSB r0, r1, r2 ; r0 = r2 - r1

RSC r0, r1, r2 ; r0 = r2 - r1 + C - 1

• All operations can be viewed as either unsigned or 2's complement signed.

• Means the same thing.

Handwritten note: SUB r0, r2, r1

First let us talk about the arithmetic instructions. Well, arithmetic instruction the most basic instructions are addition and subtraction. So, ARM provides with various addition and subtraction instruction alternatives. Let us see what kind of instructions are there. First is a simple add instruction, well here as an example I have shown to three registers r 0, r 1, r 2, but you can have any registers here not necessarily only r 0, r 1, r 2. The first one represents the destination, the second and third indicates the two source operands.

So, when I write add r 0, r 1, r 2 the values of r 1, r 2 will be added and the result will be stored in r 0, right. There is a version add with carry which is normally used to handle multi precision arithmetic. So, when you are adding two 64-bit numbers you add first two 32-bit numbers, if there is a carry the next 32 bit numbers you would be adding with that carry. So, you should have an add with carry version of instruction this is add with carry. Here what it does same thing it adds r 1 and r 2 with carry if the carry bit is 1 then an additional 1 will be added, if the carry is 0 then nothing is added.

Then there is a normal subtract instruction this is r 1 minus r 2 result is going into r 0 and there is a similar borrow concept in subtraction for multiplication arithmetic again. So,

there is an subtract with borrow version as we see it is called where the carry flag is used. Now, the actual calculation is done like this $r_1 \text{ minus } r_2 \text{ plus } C \text{ minus } 1$. This takes care of the borrow during subtraction operation, ok. Now, there are two variation of the subtract instructions, where the role of the operands are reversed. Like here sub instruction we are saying $r_1 \text{ minus } r_2$. Now, if I say RSB this stands for reverse subtract this means $r_2 \text{ minus } r_1$. Similarly, there is a version with borrow reverse subtract with carry. So, it is $r_2 \text{ minus } r_1 \text{ plus } C \text{ minus } 1$.

Now, you may ask why you need two kinds of subtract instruction. I can always write this instruction as let us say subtract r_0, r_2, r_1 . I can always write like this why I need another RSB instruction. The need is that we shall see later I have not talked about it. This ARM instruction allows this second operand to specify to be specified in more flexible ways like this second operand r_2 , this can be specified in multiple ways. So, if you have a reverse version of subtract then that flexible operand can be subtracted from or the normal data can be subtracted from that flexible operand both ways you can have. So, we shall be seeing what kind of flexibility the second operand provides you, but this reverse subtract allows you to add a normal operand from a flexible operand or from a flexible operand you can subtract a normal operand both options are provided.

And, in these subtract instructions you are viewing these 32-bit numbers as either unsigned or as 2's complement signed numbers. Now, with respect to addition and subtraction in binary these two make no difference. The way the arithmetic is carried out is exactly identical, ok. So, they actually mean the same thing, all right.

(Refer Slide Time: 13:21)

• Bit-wise logical instructions:

- ✓ AND $r0, r1, r2$; $r0 = r1 \text{ and } r2$
- ✓ ORR $r0, r1, r2$; $r0 = r1 \text{ or } r2$
- ✓ EOR $r0, r1, r2$; $r0 = r1 \text{ xor } r2$
- ✓ BIC $r0, r1, r2$; $r0 = r1 \text{ and not } r2$

• BIC is the acronym for "bit clear"

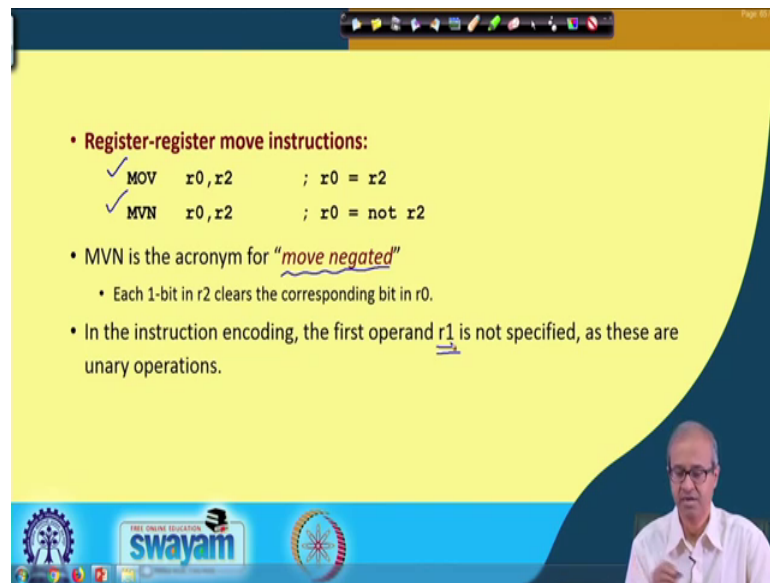
- Each 1-bit in $r2$ clears the corresponding bit in $r1$.

So, after this addition and subtract instructions there are some bitwise logical instructions there are instructions like AND, ORR and Exclusive ORR. So, when I say AND, let us say similar to add $r0, r1$ and $r2$, what happens $r1$ and $r2$ are the two operands. So, they are all 32-bit numbers, ok. Let us say this is let us say this is $r1$, this is $r2$. So, the last bit of $r1$ and $r2$ are bitwise ANDed together and the result is stored in $r0$ in that corresponding bit here.

Similarly, the next two bits are ANDed together the result will be stored here. Similarly this bits and this bit are ANDed together result is stored here, this is called bitwise logical operation. So, I can do AND operation OR operation or I can do Exclusive OR operation, these three instructions are supported. Now, there is another instruction also which is supported at the bitwise level this is called bit clear instruction in short BIC. So, the actual meaning is the register $r1$ is ANDed with the complement of $r2$; you take a not operation and then do an AND. So, that in $r2$ whichever bit is 1, if you do or not those bits will become 0 if you do I and with $r1$ those corresponding bits of $r1$ will become 0 because and anything AND 0 is 0. So, actually because of that this is called a bit clear instruction.

So, wherever in $r2$ you have one those corresponding bits in $r1$ will be made 0; that means, those will be cleared that. So, this is also called a bit clear instruction, ok, fine.

(Refer Slide Time: 15:33)



• Register-register move instructions:

- ✓ `MOV r0, r2 ; r0 = r2`
- ✓ `MVN r0, r2 ; r0 = not r2`

• MVN is the acronym for "move negated"

- Each 1-bit in r2 clears the corresponding bit in r0.

• In the instruction encoding, the first operand r1 is not specified, as these are unary operations.

Then you have some register to register move instruction, ok. This also we are defining under the data manipulation instruction category. Now, here there are two kind of register to register move that are supported one is a simple register to register move. You see when I am saying move register to register I need to specify only two operands, just like the other instruction we have seen where three operands are required here we need only two, ok. So, when I say move r 0 comma 2 it means the value of r 2 is copied into r 0, ok. And, there is another version move negated; that means, if I say MVN r 0 and r 2 here this r 2 will be complimented the not of r 2 will be moved into r 0, right.

So, there are many applications where this negative value of our 32-bit number is required. So, there you can use this MVN instruction move negative, right. Now, here in the encoding as I said that you do not need three registers. This r 1 which we are mentioning earlier, the middle operand that is not required here only two operands are required.

(Refer Slide Time: 17:13)

Comparison instructions:

- ✓ `CMP r1,r2 ; set cc on (r1 - r2)`
- ✓ `CMN r1,r2 ; set cc on (r1 + r2)`
- ✓ `TST r1,r2 ; set cc on (r1 and r2)`
- ✓ `TEQ r1,r2 ; set cc on (r1 xor r2)`

• All these instructions affect the condition codes (N, Z, C, V) in the current program status register (CPSR).

• These instructions do not produce result in any register (r0).

Handwritten notes on the right side of the slide:

Z
S
V

$0 \oplus 0 = 0$
 $1 \oplus 1 = 0$

$0 \oplus 1 = 1$
 $1 \oplus 0 = 1$

Now, there are a variety of compare instructions. Now, you recall in the program status what current and the saved program status words that we discussed in our previous lectures there are some condition flags you recall there were zero flag, there were a sign flag there was an overflow flag and so on. So, these flags actually keep track of the results of some arithmetic or logic operation. When you add two numbers these flags will keep track of the fact whether the result was 0, whether the result was positive or negative or whether there was an overflow that took place because of that operation, ok.

Now, sometimes you just need to compare two numbers and take a decision. You do not you are actually not doing addition and subtraction and storing the results somewhere just you need to compare two values. So, there are a host of compare instructions available. Like you can say simple comparison; you compare r 1 and r 2 and; that means, internally you are actually doing r 1 minus r 2 you are subtracting and you are checking whether there is 0, negative or positive, so, accordingly the flags will be set.

There is another version compare negative; actually it means you do r 1 plus r 2 and then set the flags. Result of addition of the two numbers, but you are not storing the result in here this is the difference between a normal add and subtract instruction and the compare instruction. Here these instructions do not produce any result in a register; this is what you should remember. It only sets the condition flag so that you can use that result later depending on the condition flag you can take some decision, right.

Similarly, there is test kind of a comparison. This comparison means you are doing logical and bitwise and of r1 and r2 then you are checking the result is 0 or nonzero whatever then accordingly set the flags. Similarly, you test for equality, test equal. Equality means exclusive or you say exclusive are you not own property if you take the exclusive OR of 0 and 0 the result is 0, if you take exclusive OR of 1 and 1, that is also 0, but if you take exclusive OR of 0 and 1 or 1 and 0, this is 1.

So, the result of an exclusive OR will tell you whether the bits are equal or they are not equal right. So, if you take XOR of entire 32-bit numbers and see the result is 0, this means that the two numbers are equal, all the bits are giving XOR value of 0, that is why the result is 0, ok. So, these instructions as I told you they affect the condition codes carrying 0, this sign or negative flag, overflow these are stored in the current program status register.

(Refer Slide Time: 20:53)

• Specifying immediate operands:

```
ADD r1, r2, #2 ; r1 = r2 + 2
SUB r3, r3, #1 ; r3 = r3 - 1
AND r6, r4, #&0f ; r6 = r4[3:0]
```

• Notations:

- # indicates immediate value
- & indicates hexadecimal notation

• Allowed immediate values:

- 0 to 255 (8 bits), rotated by any number of bit positions that is multiple of 2.

There are ways of specifying immediate operands. Like as I said you can specify some immediate data, like I can write like this with this hash symbol I can use the second operand as an immediate open I can write r2 comma hash 2 this means r2 will be added with the constant 2. I can write subtract r3 hash 1; that means, r3 minus 1, ok. Like this I can add an immediate operand as my second operand of an arithmetic or logic operations. Like for example, if I say AND of this, what does this mean? Ok, hash as I

told you hash indicates immediate operand and this ampersand indicates that the number I am specifying is in hexadecimal.

So, when I do and with 0 f what does that mean? 0 f in 32-bit means 0 0 0 F and F means what? Four 1's; so, if you do AND only the last four bits of the number will remain all the others will become 0. So, actually what you are doing, this are for the last four bits I am showing like this bit number 0 to 3, these 4 bits are copied to r 6 the remaining bits are all made 0, right. So, with respect to the immediate values well there is of course, some restriction of the range maximum value of this number you can represent.

So, the range is typically 0 to 255, you can also regard it as a 2's complement number you can give a negative value also and there is another facility there are additional 4 bits in the instruction where you can specify that these operand value will be rotated and means within that 32-bit range these 8 bits can be positioned either here or here or here or here. So, you can make your number as per your requirement and then you can do add subtract and you want. So, that bit position is some multiple of 2 I am not going to detail, but it has a flexibility in that you can specify that where that immediate data will fit in inside that 32-bit number.

(Refer Slide Time: 23:33)

• **Shifted register operands:**

- The second source operand may be shifted either by a constant number of bit positions, or by a register-specified number of bit positions.

```
ADD r1, r2, r3, LSL #3 ; r1 = r2 + (r3 << 3)
ADD r1, r2, r3, LSL r5 ; r1 = r2 + (r3 << r5)
```

• Various shift and rotate options:

- LSL: logical shift left
- LSR: logical shift right
- ROR: rotate right
- RRX: rotate right extended by 1 bit
- ASL: arithmetic shift left
- ASR: arithmetic shift right

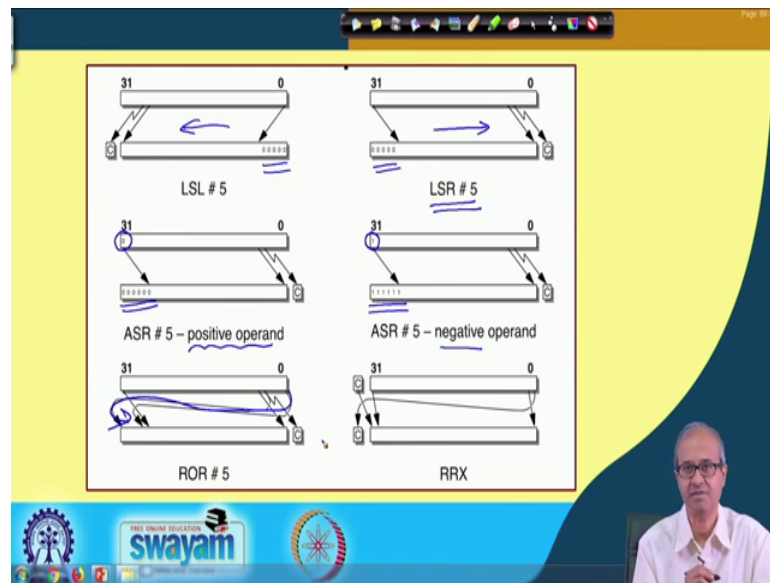
Then, I talked about some flexibility in the second operand you see here this shifted register operand comes in. You recall in the architecture I told in ARM there is a barrel shifter. The second operand optionally goes through the barrel shifter you can shift it and

then you can do some arithmetic or logic operations. So, in the assembly language level also you can specify like this you can specify add r 1, r 2, r 3 and in the fourth parameter you can say logical shift left hash 3; that means, three position this actually means the second operand is shifted left by three positions and then added to r 2, ok. So, you can specify this as a constant how many bits want to shift or you can also specify some register logical shift left r 5; whatever is the value in r 5 that many bits will be shifted.

Let us say if r 5 is 20, then it will be a 20 bit shift. You see here like this r 3 is shifted left by whatever is the value of r 5 that many positions. Now, you see here we have a facility of specifying the second operand in some flexible way. Either in the normal form or in some shifted form that is why this reverse subtract operation we recall there I can subtract a register from the shifted value of another register, that is why this reverse subtract facility is also there.

And, not only logical shift left there are various shift and rotate instructions or options available; logical shift left, logical shift right, rotate right, rotate right means when you shift right the last bit coming out will again get inside the register, that is rotate right. Rotate right extended means the register you are rotating right then there will be your carry flag. This bit will go into this carry flag and carry flag will get rotated in it, that is why this is something like a 33-bit rotation this is called extended by 1 bit and arithmetic shift left is same as logical shift left, no difference. And, arithmetic shift right means if it is a negative number when you shift right, 1 will be added to the most significant bit side if it is positive number 0 will be added, these are the various options.

(Refer Slide Time: 26:17)



So, in this diagram some of these are shown. You see here if you say logical shift left 5, the original register will be shifted left by 5 positions and 0's will be added on the right. If you say logical shift right by 5 position similar you shift right and 0's get added.

Now, if I say arithmetic shift right, but the number is positive which means the MSB was 0 then 0's will be added, but if the number was negative which means most significant bit was one then 1's will be added on the left side. And, rotate as I said whatever goes out rotate right whatever goes out of the register this will be getting inside here. So, you are rotating and rotate extended means along with the carry flag as that said it will be a 33-bit rotation. These are the various options.

(Refer Slide Time: 27:23)

• **Multiplication instruction**

```
MUL r1, r2, r3 ; r1 = (r2 x r3) [31:0]
```

- Only the least significant 32-bits are returned.
- Immediate operands are not supported.

• **Multiply-accumulate instruction:**

```
MLA r1, r2, r3, r4 ; r1 = (r2 x r3 + r4) [31:0]
```

- Required in digital signal processing (DSP) applications.
- Multiplication with 64-bit results is also supported.

Now, there are some multiplication instructions also. Multiplication in this simple form is $r1, r2, r3$ like this only. So, you multiply the two numbers $r2, r3$ and you take only 32-bits of the result because multiplication result can be 64-bit, but you ignore the high order bits you assume that the result is fit the result will fit within 32-bits and you take only the last 32-bits, and in multiplication immediate operations cannot be used to only register operands.

And, there is another instruction which is very much useful for digital signal processing or DSP applications, this is called multiply and accumulate. There you need to continuously multiply a number with some other number and add to another value continuously. So, here this multiply accumulate instruction will have four register operands and the meaning is you multiply $r2$ and $r3$ and add it to $r4$. So, not only multiplication also an addition and you take the last 32-bits of the result and as I said a 64-bit multiplication version is also available which you are not discussing right now here ok, that is also available.

So, with this we come to the end of this lecture where we have talked about some of the arithmetic and logical instructions which are available and supported by the ARM instruction set architecture. In the next lecture, we shall be talking about the data transfer instructions; what are the various kinds of data transfer instructions that can be used to transfer data between registers and memory.

Thank you.