

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

Lecture – 8
ARM Instruction Set (Part II)

If you recall in the last lecture we were discussing about the ARM structures with which you can do some arithmetic and logic operations. Now in this lecture we shall be further looking into instructions which allow you to transfer data between the registers and memory. So, the title of this lecture is ARM Instruction Set, the second part.

(Refer Slide Time: 00:44)

CONCEPTS COVERED

- ❑ Data transfer instructions
- ❑ Single register transfer instructions
- ❑ Multiple register transfer instructions
- ❑ Memory-mapped I/O in ARM

00:44

FREE ONLINE EDUCATION
swayam
THINK WITH A DIFFERENCE

Now, in this lecture as I said we shall be primarily talking about the data transfer instructions mainly between registers and memory and you will see that there are instructions which can transfer a single register value and also multiple register value. And we shall at the end talk something about this input output operations in ARM which supports something called memory mapped I O. This is also something shall be briefly talking about at the end.

(Refer Slide Time: 01:18)

(b) Data Transfer Instructions

- ARM instruction set supports three types of data transfers:
 - a) Single register loads and stores
 - Flexible, supports byte, half-word and word transfers
 - b) Multiple register loads and stores
 - Less flexible, multiple words, higher transfer rate
 - c) Single register-memory swap
 - Mainly for system use (for implementing locks)

Diagram: R ↔ M

Diagram: Semaphore

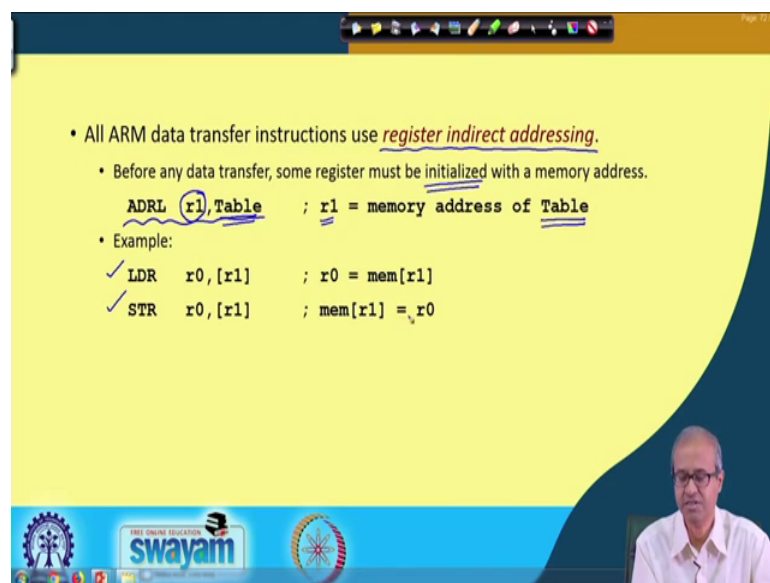
So, let us look at the data transfer instructions that are supported by the ARM architecture. Broadly there are 3 types of data transfer which are allowed. Well, here we are mainly talking about registers and we are talking about memory. So, transfer between registers and memory. So, you can have single register load and store it from memory you can load the value into a single register there is a load instruction; store means from a register you write that result into some memory location that is store.

So, there is a single register variant one value is loaded or stored or you can have an option for multiple registers loaded and stored like you can say that not one, four registers will be loaded or let us say for resistors will be written into memory one after the other. So, you can specify that in the same instruction in a single instruction and also there is a very special kind of instruction where a register memory swap operation is allowed swap means exchanging the value of a register and a memory in a single instruction.

Now this kind of instructions are very useful not in the normal scenario primarily by the operating system, it is used by the operating system mainly for implementing several different kind of locks. There is a very commonly used lock which is used this is called semaphores and these kind of register memory swap instructions are very much useful to implement these semaphores.

But for normal application development, we do not require to use this kind of instructions. So, the other instructions are sufficient.

(Refer Slide Time: 03:32)



- All ARM data transfer instructions use register indirect addressing.
- Before any data transfer, some register must be initialized with a memory address.
`ADRL r1, Table ; r1 = memory address of Table`
- Example:
 - ✓ `LDR r0, [r1] ; r0 = mem[r1]`
 - ✓ `STR r0, [r1] ; mem[r1] = r0`

Well, the first thing to notice that in ARM the data transfer instructions that we are talking about, they use registered indirect addressing that you must use some register to point to memory location and via that register you are doing load and store. So, what I mean an example is shown here. Before you can access memory load and store, you will have to initialize some register with the address of a memory location. So, there is an instruction ADRL Address Register load. Here you specify some register let us say r1 and you specify some memory location, let us say table. Well in assembly language, you do not specify the absolute address you specify some label with respect to the data; let us say table.

So, what will happen? The memory address corresponding to this data item table that memory address will be loaded in r1. But once you have done this, you can use this load register and store register instructions in LDR, STR. LDR r0 within square bracket r1

means memory location whose addresses in r1 will be transferred to r0 in store just the reverse memory location whose address is in r1 will get the value from r0 this is store ok. So, these kinds of load and store operations are supported.

(Refer Slide Time: 05:24)

• **Single register loads and stores**

- The simplest form uses register indirect without any offset:
`LDR r0, [r1] ; r0 = mem[r1]`
`STR r0, [r1] ; mem[r1] = r0`
- An alternate form uses register indirect with offset (limited to 4 Kbytes):
`LDR r0, [r1, #4] ; r0 = mem[r1+4]`
`STR r0, [r1, #12] ; mem[r1+12] = r0`
- We can also use auto-indexing in addition:
`LDR r0, [r1, #4] ; r0 = mem[r1+4], r1 = r1 + 4`
`STR r0, [r1, #12] ; mem[r1+12] = r0, r1 = r1 + 4`

Now, this is what we call single register load and store. So, we are loading and storing one register at a time right like in the example that we have just not shown. This value of r0 was being stored in memory this or this memory location was being loaded into register r0. So, one register transfer at a time by an instruction. Now as I said ARM supports various other options multiple register, we shall come a little later. But before that well even in single register load and store, there are various options we can have. First option is well this is simple register indirect ok.

Now, we can have register indirect with offset like it we can also specify an offset value with respect to the current instruction for this offset is limited to 4 kilobytes because in the instruction this space that is provided is 12 bits and in 12 bit sequence specify 2 to the power 14 or 4 kilobytes of offset. Now none the instruction how do you specify that you specify it like this. Earlier within the square bracket you are specifying only the register, now you are specifying also some immediate data like this; r1 comma hash 4. What does this mean? The value of this register is added to this offset and then memory location corresponding to that address that value is loaded into the register. Similarly for

store; if I write like this, then r1 will be added to 12 like this. This will give you the memory address and r0 will be stored into that memory address right.

So, not only a register indirect, I can also specify an offset which will be added to that register to generate the final memory address from we are loading and storing can be done. There is a third optional also available. Well, you can have auto indexing well in addition to using register indirect with offset. Well, auto indexing means there are many application where you want to load or store data one by one consecutively from a block of memory locations. So, they are in this instruction, what happen? This register which was pointing to some memory location this automatically gets incremented after one transfer is done. And how do I specify this? I specify this by this exclamation symbol. This indicates auto indexing and the meaning is that in the earlier case you see we are doing this right.

So, here we are doing this alright, but after this we are also increasing the value of r1 by 4 and here we are doing this and after that we are increasing the value of r1 by 4. This value is getting incremented by 4 because every memory word is 32 bits which means 4 memory addresses that way the address is incremented by 4. So, that next time when you come this arrow is pointing to the next memory location; next word right ok.

(Refer Slide Time: 09:23)

```
• We can use post indexing:  
LDR r0, [r1], #4 ; r0 = mem[r1], r1 = r1 + 4  
STR r0, [r1], #12 ; mem[r1] = r0, r1 = r1 + 12  
  
• We can specify a byte or half-word to be transferred:  
LDRB r0, [r1] ; r0 = mem8[r1]  
STRB r0, [r1] ; mem8[r1] = r0  
LDRSH r0, [r1] ; r0 = mem16[r1]  
STRSH r0, [r1] ; mem16[r1] = r0
```

There is another variation also this is called post indexing where after this square bracket is specified the r1 you give a comma and this hash value you give after that. Here

meaning is slightly different, you see this means you first load the value as usual and then increase r1 not by 4 always, but whatever opposite you have given yet. If you give hash 4, it will be incremented by 4. If you give hash 12, it will be incremented by 12. This is the main difference ok.

Sometimes you may need to increase the register by some other value than 12, you can use this variation of the instruction in those cases. And the other thing is that whatever we have discussed so far were assuming that the entire 32 bit word has been transferred. But there are many application where you are you really do not need entire 32 bit of data; 8 bits or 16 bits of data may be sufficient. These are called byte or half word. There are some various you can say categories of instruction load and store which work with byte and half word. Byte is indicated by B and half word is indicated by SH. You see load register byte, store register byte, load register half word, store register half word.

So, the way you specify registers is the same the meaning is that for byte thing only 8 bits of data are transferred from memory. That means, only 1 byte is loaded or stored not the whole 32 bits. Similarly for half word 2 bytes are transferred, 16 bits are transferred. So, it depends on whether you are using byte, half word or normal word; it will depend how many by its will be loaded or stored. It can be 1 byte, 2 bytes or 4 bytes all right.

(Refer Slide Time: 11:45)

• **Multiple register loads and stores**

- ARM supports instructions that transfer between several registers and memory.
- Example: $r2-r5$
`LDMIA r1, {r3, r5, r6}` ; $r3 = \text{mem}[r1]$ A: After
; $r5 = \text{mem}[r1+4]$ B: Before
; $r6 = \text{mem}[r1+8]$
- For LDMIA, the addresses will be $r1+4$, $r1+8$, and $r1+12$.
- The list of destination registers may contain any or all of $r0$ to $r15$.
- Block copy addressing
 - Supported with addresses that can increment (I) or decrement (D), before (B) or after (A) each transfer.

Now, let us come to multiple register load store variety. This is interesting. Here we are saying that in ARM instruction set, there are facilities to transfer data between several

registers and memory. Now one thing you recall we also mentioned earlier when you when you are talking about this ARM and RISC architecture. These are some facilities where ARM is deviating from RISC architectures.

Because one philosophy of RISC architecture was that all instructions should be of equal size. They should be very simple, they should be able to execute in a single cycle, but when you are transferring multiple words naturally you cannot finish execution in one cycle; multiple cycle will be required because you are loading or storing multiple memory words right. So, this is where the things are slightly deviating from the pure RISC concept, but this is a very powerful instruction. So, in many application you may want to use it. You see one variation of this instruction is LDMIA well, there is another version LDMIB. Well I will see the difference, I will show you this A and B actually stands for B is before a stands for after means A is actually the acronym for After, B is the acronym for Before.

So, let us see how these instructions are specified. Here normally a load register; there are two operands. But here what I am saying this first one is r1. So, it is assumed that the first operand is containing the memory address. It is that register indirect thing. So, here we are not giving that square bracket that is implied and in the second operand within curly bracket we are specifying a set of registers. Now there are many ways you can specify if the registers are consecutive in number. We can also write like this r2 dash r5 this means r2, r3, r4, r5 all 4 or otherwise you can separate by comma r3, r5, r6 which means this r1 contains the address of a memory location. You load the content of that memory location into r3 then load the next word; that means, r1 plus 4 add 4 to the address load into r5 r1 plus 8 you add 4 again loaded into r6. This A means you first load and then after that increase the address.

There is another variation said LDMIB before. Before means you first increase it by 4 and then load which means if I give LDMIB instead of this then the data will be loaded from memory addresses corresponding to r1 plus 4 r1 plus 8 and r1 plus 12 because first we are increasing it by 4 and then we are loading right ok. And in the list of destination registers, we can use any of the 16 registers as we choose and in addition to this B and A, we can also use this I and D option you see here. This I has been used I stands for increment the addresses are getting incremented. I can also give D; if I give D then the

load and store will be happening in the reverse order addresses will get decremented by 4 ok. So, these all options are available.

(Refer Slide Time: 16:02)

• Examples of addressing modes in multiple-register transfer:

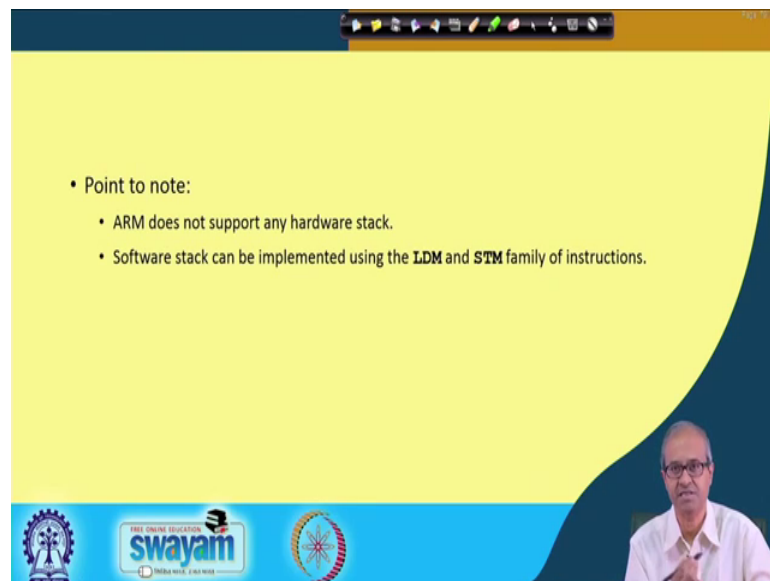
Instruction	Addressing Mode
LDmia, STmia	Increment after
LDmib and STmib	Increment before
LDmda, STmda	Decrement after
LDmdb, STmdb	Decrement before

So, let us see some examples here. The first example I am showing here is store memory location increment after. So, here this r9 here I am giving this exclamation sign means this unlined will get incremented auto indexing auto increment. So, let us see what a blessed assume r9 initially points here. This is memory location; these are the memory addresses I am showing. Let us assume r9 is pointing here to address 100C. So, because it is after r0 r1 r2 store so, the value of r0 will get stored here, then address will be incremented it will be just increased C plus 4 then r1 will be stored here. Again it will increase by 4 then r5 will be stored here.

Now because of use this exclamation sign this r9 will get incremented again and the final value of r9 will be this 1018. So, it will point to the next location after all these 3 store operations are done right. Now if instead if I give a STMIB before, I increment before and then I store then you see if r9 was here nothing is getting stored here. First I increase the address, then I store. Again I increase and store then I increase in store, and then finally I update r9 as whatever was there plus 12. And let us look at some example for decrement if it is STM decrement after then the memory will go in the reverse direction; the address will get decremented r9 first you see r9 was here at zero r1 r5. So, it will get in the reverse order in this way and the final value of r9 line here and similarly before.

So, this first one will be skipped, then decrease. So, first you decrement and then you store decrement store decrement store. So, for the decrement option you note the values are getting stored in the reverse order the rightmost one are getting stored first the leftmost one a little stored last. So, these are the various options and this already I mentioned what are the meaning of this increment after, increment before decrement after and decrement before ok. So, these flexible instruction set options are available in ARM.

(Refer Slide Time: 18:58)



• Point to note:

- ARM does not support any hardware stack.
- Software stack can be implemented using the **LDM** and **STM** family of instructions.

The slide is part of a presentation, as indicated by the navigation icons at the top and the Swamyam logo at the bottom. A small video inset in the bottom right corner shows a man with glasses speaking.

Now, one interesting thing is that there is no stack in ARM architecture which is so much popular in other contemporary architectures. Of course, you can as a user or as a programmer, you can maintain a software stack, but there are no explicit push pop instructions which are normally a part of the instruction set of conventional computer architectures right. So, LDM and STM instructions can be used to implement stack we shall see some examples later.

(Refer Slide Time: 19:38)

An Example

- Copy a block of memory (128 bytes aligned).
- r9: address of the source
- r10: address of the destination
- r11: end address of the source

```

Loop:  LDmia r9!, {r0-r7}
        STmia r10!, {r0-r7}
        CMP  r9, r11
        BNE  Loop
    
```

16

64 instr.

Let us take one example of copying a block of memory locations copy a block of one twenty eight memory locations. Suppose I have an application where in memory I have a block of consecutive 128 bytes ok.

Let us say I want to transfer this entire block into some other memory location. Let us say here I want to transfer it here. Now using single register moves, what I can do? I can write a loop. So, I initialize some register with the starting address and I do it 128 times read one load one number from the initial blocks, store it here; load the next number, store it in the next location; load the third number, store it in the third location and I check whether 128 number of data have been transferred.

Now here I will have to look for 128 times. So, 128 times the entire loop will be executed, but when I use this kind of multiple register transfer instruction let us say, I have 8 registers at my disposal. So, if I load 8 registers together store 8 registers at one time; that means I am transferring 8 bytes in one go. In terms of the loop how many times I have to loop in that case only 16 times 16 into 8 is 128. So, the number of instructions being executed in ARM will be drastically less. This will also reduce the overall execution time because every instruction execution has its own overhead.

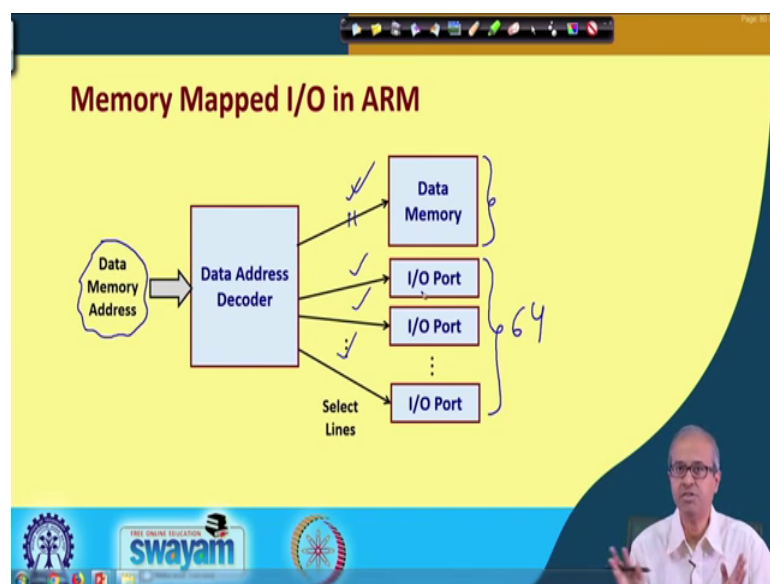
So, if you transfer multiple memory location in one memory in as part of one single instruction, then the total number of clock cycles we will be needing will yes right. So, let us see how the program will look like here let us assume that r9 contains the address of the source; that means, r9 contains the address of this memory location, r 10 contains

the address of the destination that mean this one this is r 10 and r 11 is the address of the end of the source; that means, r11 contains the last address of the source.

So, I can write a program like this a simple program it checks first I am doing LDMIA, this r9 is already initialized. You can see I am transferring eight registers loading 8 registers r0 to r7 with address from r9. So, the first 8 words will be transferred and because I am using this auto indexing mode; this r9 will automatically get incremented by 8 to 432. So, that next time it will be pointing to the next set of numbers and we are doing a store with r10 same way, this is also auto indexing. So, r10 you will also get incremented by 8 and these 8 registers were storing one by one in this single instruction. Then you are comparing whether r9 has reached r11 or not. Well, if it has reached our eleven it means we are done otherwise branch if not equal we repeat this loop. We repeat this to how many times we are transferring eight data at a time. So, this will be 16 times right.

So, this will be much faster because total you see because it is 16 times. So, total number of instructions executing will be 16 instructions, but in the earlier case if you are using single instruction load and store your loop, would go for 128 times. So, it would be just you can imagine here only 6. So, it will be 128 into 4 it would be 512 instructions, but in comparison you are here only requiring 64 instructions to be executed right. So, it is a big gain you are getting.

(Refer Slide Time: 24:25)

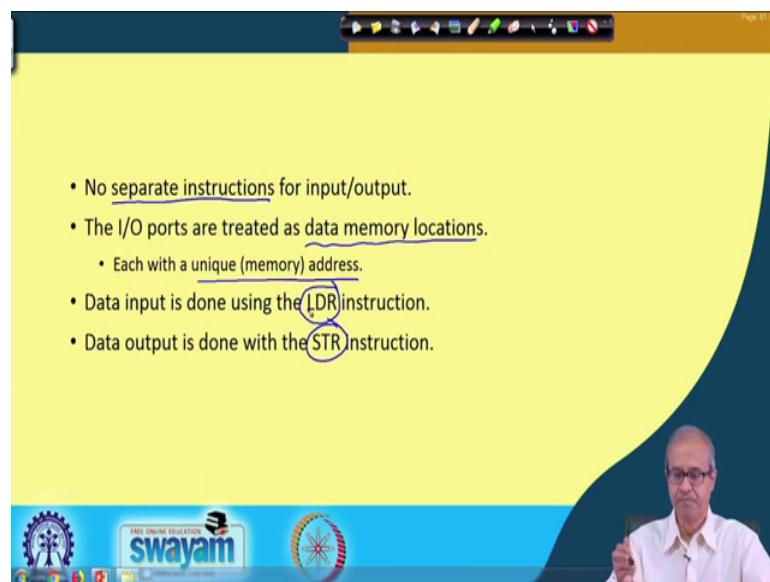


So, lastly one thing we want to discuss, there is a concept called memory mapped I O which is prevalent in many computer systems microcontrollers in particular in ARM also.

So, the basic idea behind memory mapped I/O is this. Normally whenever some data are accessed data memory address is generated by the microcontroller, the processor. There is a decoder which decodes this address and it selects the corresponding memory location. Normally the data memory is accessed, but in memory mapped I/O what we are saying in addition is that the input output which are typically called I/O ports. The I/O ports are also regarded as memory locations. Suppose let us take an example suppose there are 6 I/O ports.

So, we are assuming that there are 64 memory locations also here. So, the same decoder depending on the address you are giving will be selecting either one of the data memory addresses or one of these are your ports. So, this I/O ports are I/O devices are treated as memory locations. This is the basic concept behind memory mapped I/O. There are no separate instruction for doing input and output.

(Refer Slide Time: 26:12)



- No separate instructions for input/output.
- The I/O ports are treated as data memory locations.
 - Each with a unique (memory) address.
- Data input is done using the LDR instruction.
- Data output is done with the STR instruction.

So, the basic thing as I mentioned there are no separate instructions for input and output which is there for some of the conventional processors like let us say for those of you who are familiar with let us say the 8051 microcontroller or the 85 microprocessor there you will find in and out instructions which are meant primarily for input and output; I/O.

The I O ports are treated as data memory location, where each of them will be having a unique address. So, when you want to read some data from an input device, you will be using load instruction when you want to write some data into an output device you will be using store instruction. That is how ARM handles it. So, this provides some added flexibility because, there are so many flexibilities the instruction set offers when you are accessing memory locations. Now if you treat your I O devices also as memory locations then even when accessing the I O devices the same kind of flexibility you can utilize, fine.

So, with this we come to the end of this lecture. In the next lecture we shall be talking something about the control flow instructions that are available in the ARM instruction set.

Thank you.