18CS44: IAT1 Scheme and solution

Faculty: Dr. Imtiyaz Ahmed & Prof.Preethi A

SCHEME OF EEVALUATION

Q NO	Question	Marks	
1	**Draw and Explain the ARM core Dataflow Model.**		10
	Data Flow Diagram	4	
	Explanation on each unit	6	
2	**Discuss the significance of CPSR? Explain each relevant bit in detail**		10
	Significance	2	
	Diagram	2	
	Flag field Bits	3	
	Control field Bits	3	
3	**Explain in detail the processor modes available for ARM7**		10
	List Different Modes	2	
	Privileged modes explanation	6	
	User mode explanation	2	
4	**Define Core Extension. Explain any two types of core extension with neat diagram**		10
	Requirement of core extension	2	
	Any two types in details	2*4=8	
5	**Closely study the code snippet and write the contents of R1, R2, R3, R4, R5,R6 after the execution of the code snippet.** **PRE:** **R1=0x02, All other register's content is zero** **MVN R2, R1** **MOVS R3, R2, LSL #2** **ADDMI R4,R1,#0x10** **ADDEQ R5, R1, R3** **BIC R6,R2,#0b1111**		10
	R1, R2, R3, R4, R5,R6	1+2+2+1.5+1.5+2	
6	**With proper syntax and examples explain the following mnemonics SUB b.CMN c.BIC d.LSR**	4*2.5	10
7	**With proper syntax and examples explain the following mnemonics (2.5*4=10)** **a. ADC b.RSB c.MOV d.RRX**	4*2.5	10

1. ARM 7 data Flow Model

A programmer can think of an ARM core as functional units connected by data buses, as shown in Figure 1.1, where, the arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area. This model is called data flow model or programmers view of architecture.

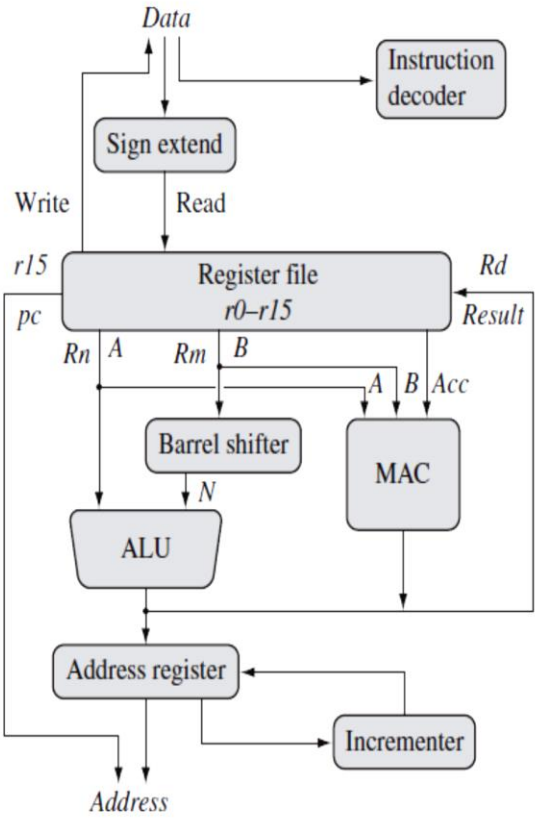

Fig 1.1 ARM core data flow model (Von Neumann Model)

- Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item. The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.
- The ARM processor uses a *load-store architecture*. This means it has two instruction types for transferring data in and out of the processor: load instructions copy data from memory to registers in the core, and conversely the store instructions copy data from registers to memory.
- There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out solely in registers.
- Data items are placed in the register file—a storage bank made up of 32-bit registers.
- Since the ARM7 core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.
- ARM instructions typically have two source registers, Rn and Rm, and a single result or destination register, Rd. Source operands are read from the register file using the internal buses A and B, respectively.

- The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values Rn and Rm from the A and B buses and computes a result. Data processing instructions write the result in Rd directly to the register file.
- Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the Address bus.
- One important feature of the ARM is that register Rm alternatively can be pre-processed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.
- After passing through the functional units, the result in Rd is written back to the register file using the Result bus.
- For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location.

Registers

General-purpose registers are identified with the letter r prefixed to the register number. For example, register 1is given the label r1.There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as r0 to r15.

Three registers are assigned with special function:

- Register r13 is traditionally used as the stack pointer (sp) and stores the head of the stack in the current processor mode.
- Register r14 is called the link register (lr) and is where the core puts the return address whenever it calls a subroutine.
- Register r15 is the program counter (pc) and contains the address of the next instruction to be fetched by the processor.
- In addition to the 16 data registers, there are two program status registers: cpsr (current PSR) and spsr (Saved PSR).
- The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor.

The registers r0 to r13 are orthogonal—any instruction that you can apply to r0 you can equally well apply to any of the other registers. However, there are instructions that treat r14 and r15 in a special way.

2. ***Discuss the significance of CPSR? Explain each relevant bit in detail***

- The CPSR is a dedicated 32-bit register which resides in the register file.
- The ARM core uses the CPSR to monitor and control internal operations.

The Figure 2. shows the CPSR layout.

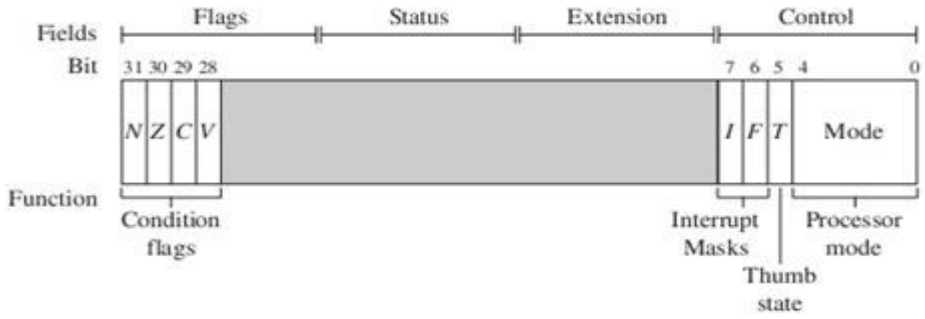

Fig 2. CPSR Layout

The CPSR is divided into four fields, each 8bits wide: **flags, status, extension and control**. In current designs the extension and status fields are reserved for future use.

Control Field: The control field contains the processor mode, state, and interrupt mask bits.

Processor Modes: The processor mode determines which registers are active and the access rights to the CPSR register itself. The least significant 5 bits in the CPSR determines the mode.

Each processor mode is either privileged or non-privileged.

- A privileged mode allows full read-write access to the CPSR. Conversely, a non-privileged mode only allows read access to the control field in the CPSR but still allows read-write access to the condition flags.
- There are seven processor modes in total: six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system, and undefined) and one non-privileged mode (user).
- The processor can change mode by either writing directly in to the control field (b0-b4) when it is in a privileged mode or when exceptions or interrupts happens.

State and Instruction Sets: The state of the core determines which instruction set is being executed. There are three instruction sets: ARM, Thumb, and Jazelle.

- The ARM instruction set is only active when the processor is in ARM state. Similarly the Thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions. Jazelle executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java bytecodes.
- The Jazelle J (bit 24, which falls in flag field) and Thumb T bits in the CPSR reflect the state of the processor.
- When both J and T bits are 0, the processor is in ARM state and executes ARM instructions. If J=1 then the core is in Jazelle state and T=1 then the core is in Thumb state.

Interrupt masks: ARM7 entertains two kinds of hardware interrupts interrupt request (IRQ) and fast interrupt request (FIQ). Bit 6 and Bit 7 of CPSR is used to mask these interrupt requests.

- If I=1 then IRQ is disabled and if F=1 FRQ is disabled.
- When processor mode changes the exception or interrupt handler makes IRQ bit 1 to disable further interrupt requests.

Flags: The flags field contains the condition flags.

Some ARM processor cores have extra bits allocated. For example, the J bit (24), which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8 bit java code.

The bits are described as given below along with condition to set the bits.

V-oVerflow: the result causes a signed overflow

C-Carry: the result causes an unsigned carry

Z- Zero: the result is zero, frequently used to indicate equality

N- Negative: bit 31 of the result is a binary 1

Q (bit 27)-Saturation: the result causes an overflow and/or saturation when extended instructions are used. eg: QADD

3. Explain in detail the processor modes available for ARM7.

Processor Modes: The processor mode determines which registers are active and the access rights to the CPSR register itself. The least significant 5 bits in the CPSR determines the mode. Refer table 3.1

Each processor mode is either privileged or non-privileged.

- A privileged mode allows full read-write access to the CPSR. Conversely, a non-privileged mode only allows read access to the control field in the CPSR but still allows read-write access to the condition flags.
- There are seven processor modes in total: six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system, and undefined) and one non-privileged mode (user).
- The processor can change mode by either writing directly in to the control field (b0-b4) when it is in a privileged mode or when exceptions or interrupts happens.
- The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*. Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

Table 3.1 Processor mode selection bits

The processor enters

- *abort* mode when there is a failed attempt to access memory.
- *Fast interrupt request* and *interrupt request* modes correspond to the two interrupt levels available.

- *Supervisor* mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
- *System* mode is a special version of *user* mode that allows full read-write access to the *cpsr*.
- *Undefined* mode when the processor encounters an instruction that is undefined or not supported by the implementation.
- *User* mode is used for programs and applications.

Banked Registers

All processor modes except *system* mode have a set of associated banked registers that are a subset of the main 16 registers. A banked register maps one-to one onto a *user* mode register. If the processor mode changes change processor mode, a banked register from the new mode will replace an existing register.

Banked registers of a particular mode are denoted by an underline character post-fixed to the mode Mnemonic or *_mode*.

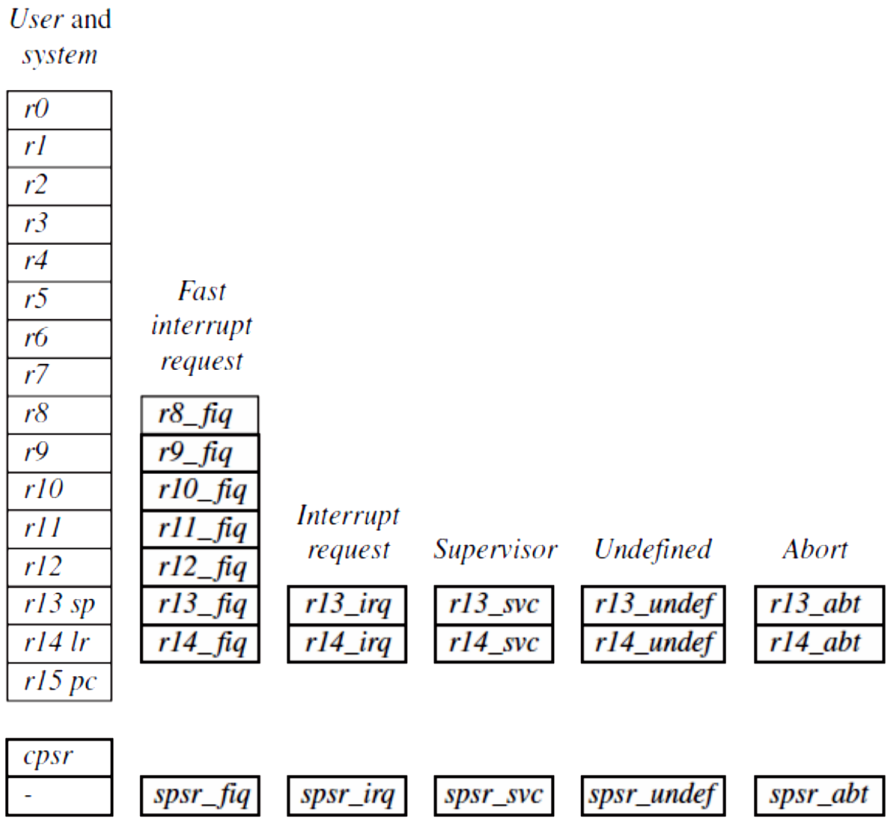

Fig 3.1 banked registers

Figure 3.1 shows all 37 registers in the register file. Of those, 20 registers are hidden from a program at different times. These registers are called *banked registers*. They are available only when the processor is in a particular mode.
For example, when the processor is in the interrupt request mode, the instructions user execute still access registers named r13 and r14. However, these registers are the banked registers r13_irq and r14_irq. The user mode registers r13_usr and r14_usr are not affected by the instruction referencing these registers. A program still has normal access to the other registers r0 to r12.

- The r14_irq contains the return address and r13_irq contains the stack pointer for interrupt request mode, the *cpsr_usr* will be copied into *spsr_irq*.

- To return back to user mode, a special return instruction is used that instructs the core to restore the original cpsr from the spsr_irq and bank in the user registers r13 and r14.

- Another important feature is that the cpsr is not copied into the spsr when a mode change is forced due to a program writing directly to the cpsr. The saving of the cpsr only occurs when an exception or interrupt is raised.

4. *ARM core extension*

There are three main core extensions wrap around ARM processor: cache and tightly coupled memory, memory management and the coprocessor interface.

C. Cache and tightly coupled memory: The cache is a block of fast memory placed between main memory and the core. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.

- ARM has two forms of cache. The first found attached to the Von Neumann-style cores. It combines both data and instruction into a single unified cache as shown in the figure 1 below.

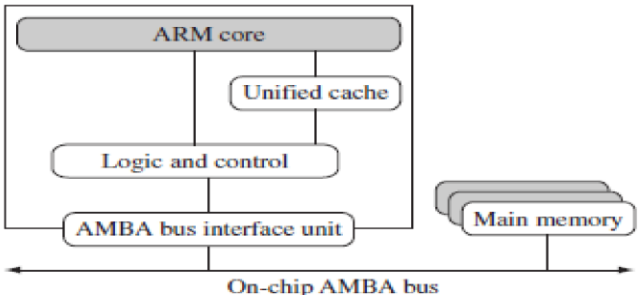

Figure 1: A simplified Von Neumann architecture with cache.

- The second form, attached to the Harvard-style cores, has separate cache for data and instruction as shown figure 2

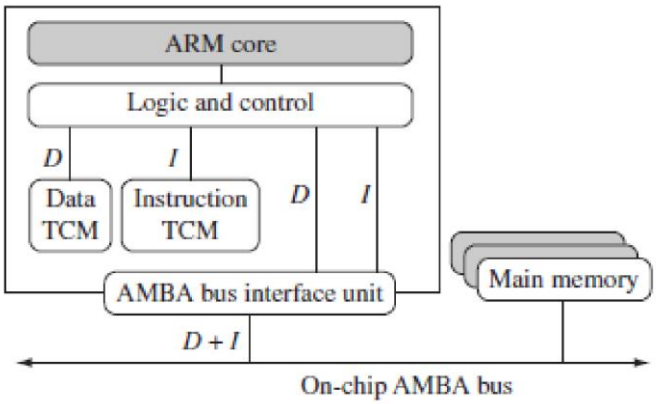

Figure 2: A simplified Harvard architecture with TCMs.

- A cache provides an overall increase in performance but will not give predictable execution.
- But for real-time systems it is paramount that code execution is *deterministic.*
- This is achieved using a form of memory called *tightly coupled memory (TCM).*
- TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data.

- By combining both technologies, ARM processors can behave both improved performance and predictable real-time response. The following diagram shows an example of core with a combination of caches and TCMs as shown in figure 3

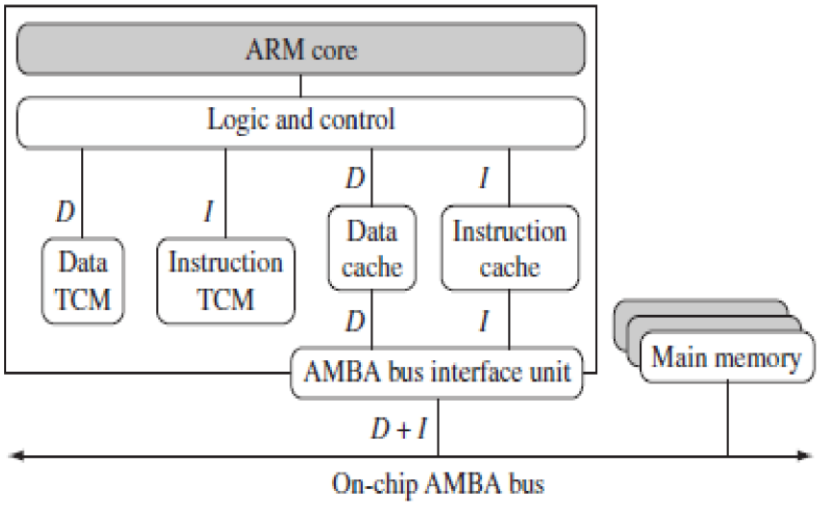

Figure 3: combining both technologies

B. Memory management:

- Embedded systems often use multiple memory devices. It is usually necessary to have a method to help organize these devices and protect the system from applications trying to make appropriate accesses to hardware.
- This is achieved with the assistance of memory management hardware.
- ARM cores have three different types of memory management hardware- no extensions provide no protection, a memory protection unit (MPU) providing limited protection and a memory management unit (MMU) providing full protection.
    - **Nonprotected memory** is fixed and provides very little flexibility. It normally used for small, simple embedded systems that require no protection from rogue applications.
    - **Memory protection unit (MPU)** employs a simple system that uses a limited number of memory regions. These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permission but don't have a complex memory map.
    - **Memory management unit (MMU)**are the most comprehensive memory management hardware available on the ARM. The MMU uses a set of translation tables to provide fine-grained control over memory.
        - These tables are stored in main memory and provide virtual to physical address map as well as access permission. MMU designed for more sophisticated system that supports multitasking.

C. **Coprocessors:**

- A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers.
- More than one coprocessor can be added to the ARM core via the coprocessor interface.
- The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface.
- The coprocessor can also extend the instruction set by providing a specialized instructions that can be added to standard ARM instruction set to process vector floating-point (VFP) operations.
- These new instructions are processed in the decode stage of the ARM pipeline. If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor.
- But, if the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception.

5. ***Closely study the code snippet and write the contents of R1, R2, R3, R4, R5,R6 after the execution of the code snippet.***

PRE:

R1=0x02, All other register's content is zero

MVN R2, R1

MOVS R3, R2, LSL #2

ADDMI R4,R1,#0x10

ADDEQ R5, R1, R3

BIC R6,R2,#0b1111

Solution:

MVN R2, R1

R2=~R1 (one's complement of R1)

R2= 0xffffffff-0x00000002 =0xfffffffd

MOVS R3, R2, LSL #2

R3 = logically shift left by 2 position oxfffffffd and update flags

Shift	Carry	B31	B30	B29	B28…B4	B5	B4	B3	B2	B1	B0
0	0	1	1	1	1……..1	1	1	1	1	0	1
1	1	1	1	1	1……..1	1	1	1	0	1	0
2	1	1	1	1	1……..1	1	1	0	1	0	0

R3= 0xfffffff4, Carry set, Negative set CN, rest all flags not set

ADDMI R4,R1,#0x10

Since the negative flag is set before MI is true so this instruction will be executed

R4=R1+0x10 =2+0x10 =0x12

ADDEQ R5, R1, R3

Since zero flag is not set this instruction wont be executed. R5=0

BIC R6,R2,#0b1111

R6=r2 not(0b1111)

The lower nibble of R2 will be cleared, other nibbles remains unchanged

R6= 0xfffffff0

R2=0xfffffffd

R3= 0xfffffff4

R4 =0x12

R5=0x0
R6=0xfffffff0

6. ***With proper syntax and examples explain the following mnemonics***
a.SUB b.CMN c.BIC d.LSR

a. SUB

Syntax

SUB{cond}{S} Rd, Rn, N ; SUB subtract two 32-bit values and store in Rd, *Rd = Rn-N*

- *N can be register, Immediate data or Barrel shifted register*

PRE
r0 = 0x00000000
r1 = 0x00000077
SUB r0, r1, #2 ; r0= r1-2
POST:
r0 = 0x00000075

r1 = 0x00000077

b. CMN

Syntax

CMN{cond}{S} Rn, N ;perform Rn+N and update the flag, both Rn and N remains unchanged

- *N can be register, Immediate data or Barrel shifted register*

PRE
cpsr = nzcvqIFt_USR
r0 = 0xFFFFFFFF
r1 = 0x00000001
CMN r0, r1 ; update flags based on r0+r1
POST:

cpsr = nZCvqIFt_USR
r0 = 0xFFFFFFFF
r1 = 0x00000001

c. BIC

Syntax

BIC{cond}{S} Rd, Rn, N ; ***BIC*** bit wise clear the Rn bits based on corresponding bits of N and store in Rd, *Rd = Rn&not(N)*

- *N can be register, Immediate data or Barrel shifted register*

PRE
r0 = 0x00000011
r1 = 0x00000077
r2=0x0
BIC r2, r1, r0 ; r2=r1& not(r0)
POST:
r0 = 0x00000011

r1 = 0x00000077
r2=0x00000066

d. LSR

Syntax

LSR{cond}{S} Rd, Rn, N ; ***LSR shift right Rn by the number of bit position specified in N*** and store in Rd

- *N can be register, Immediate data*
- *Equivalent to divided by 2^N*

PRE
r0 = 0x00000004
r1 = 0x00000001
r2=0x0
LSR r2, r0,r1 ; r2= logically shift right r0 by 1 bit positions
POST:
r0 = 0x00000001

r1 = 0x00000001
r2= 0x00000002

7. *With proper syntax and examples explain the following mnemonics a.ADC b.RSB c.MOV d.RRX*

a. ADC

Syntax

ADC{cond}{S} Rd, Rn, N ; ADC add with carry two 32-bit values *Rd = N+ Rn+ carry*

- *N can be register, Immediate data or Barrel shifted register*

PRE
cpsr = nzCvqIFt_USR
r0 = 0x00000000
r1 = 0x00000077
ADC r0, r1, #2 ; r0= r1+2+Carry
POST:

cpsr = nzCvqIFt_USR
r0 = 0x0000007A

r1 = 0x00000077

b. RSB

Syntax

RSB{cond}{S} Rd, Rn, N ; RSB reverse subtract of two 32-bit values *Rd = N – Rn*

- *N can be register, Immediate data or Barrel shifted register*

PRE
r0 = 0x00000000
r1 = 0x00000077
RSB r0, r1, #0 ; Rd = 0x0 - r1
POST

r0 = -r1 = 0xffffff89

c. MOV

Syntax

MOV{cond}{S} Rd, N ; copy 32 bit values *Rd = N*

- *N can be register, Immediate data or Barrel shifted register*

PRE
r0 = 0x00000000
r1 = 0x00000077
MOV r0, r1 ; r0= r1
POST:
r0 = 0x00000077
r1 = 0x00000077

d. RRx

Syntax
Rm, RRX

Rotate right extended by 1 bit. This is a 33 bit rotate, where the 33rd bit is the PSR C flag.

PRE
cpsr = nzCvqIFt_USR
r0 = 0x00000000
r1 = 0x00000000
MOV r0, r1,RRX ; r0= r1,RRX
POST

cpsr = nzcvqIFt_USR
r0 = 0x10000000