SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with ‘A’ Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

ARM-Instruction Set(Part 2)

Dr.G.Arthy
Assistant Professor
Department of EEE
SNS College of Engineering

Classification of Instruction

1. Data processing instructions.
2. Branch instructions.
3. Load store instructions.
4. Software interrupt instructions.
5. Program status register instructions.
6. Loading constants.
7. Conditional Execution.

Branch Instructions

- A branch instruction changes the flow of execution or is used to call a routine.
- This type of instruction allows programs to have subroutines, *if-then-else structures, and loops*.
- The change of execution flow forces the program counter *pc to point to a new address*.
- The ARMv5E instruction set includes four different branch instructions

Syntax:

```
BL{<cond>} label
B{<cond>} label
BX{<cond>} Rm
BLX{<cond>} label | R
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

- The address *label is stored in the instruction as a signed pc-relative offset and must be* within approximately 32 MB of the branch instruction.
- *T refers to the Thumb bit in the cpsr. When instructions set T, the ARM switches to Thumb state.*

Example

- This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions.
- The forward branch skips three instructions. The backward branch creates an infinite loop.

```
        B       forward
        ADD     r1, r2, #4
        ADD     r0, r6, #2
        ADD     r3, r7, #4
forward
        SUB     r1, r2, #4
-----------------------------------
backward
        ADD     r1, r2, #4
        SUB     r1, r2, #4
        ADD     r4, r6, r7
        B       backward
```

The branch labels are placed at the beginning In this example, *forward and backward are the labels.* of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.

Load-Store Instructions

- Load-store instructions transfer data between memory and processor registers.
- There are three types of load-store instructions:
- Single-Register Transfer
- Multiple-Register Transfer
- Swap Instruction

Single-Register Transfer

- These instructions are used for moving a single data item in and out of a register.
- The data types supported are signed and unsigned words (32-bit), half words (16-bit), and bytes.
- Various load-store single-register transfer instructions are
- Syntax:

  <LDR|STR>{< cond >}{B} Rd,addressing1

  LDR{< cond >}SB|H|SH Rd, addressing2

  STR{ < cond > } H Rd, addressing2

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

; load register r0 with the contents of the memory address
;pointed to by register r1.

```
LDR r0, [r1] ; = LDR r0, [r1, #0]
```

; store the contents of register r0 to the memory address
;pointed to by register r1.

```
STR r0, [r1] ; = STR r0, [r1, #0]
```

- The first instruction loads a word from the address stored in register *r1 and places it into* register *r0. The second instruction goes the other way by storing the contents of register r0 to the address contained in register r1. The offset from register r1 is zero. Register r1 is* called *the base address register.*

Single-register load-store addressing, word or unsigned byte.

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	[Rn, #+/-offset_12]
Preindex with register offset	[Rn, +/-Rm]
Preindex with scaled register offset	[Rn, +/-Rm, shift #shift_imm]
Preindex writeback with immediate offset	[Rn, #+/-offset_12]!
Preindex writeback with register offset	[Rn, +/-Rm]!
Preindex writeback with scaled register offset	[Rn, +/-Rm, shift #shift_imm]!
Immediate postindexed	[Rn], #+/-offset_12
Register postindex	[Rn], +/-Rm
Scaled register postindex	[Rn], +/-Rm, shift #shift_imm

Single-register load-store addressing, halfword, signed halfword, signed byte, and doubleword.

Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	[Rn, #+/-offset_8]
Preindex register offset	[Rn, +/-Rm]
Preindex writeback immediate offset	[Rn, #+/-offset_8]!
Preindex writeback register offset	[Rn, +/-Rm]!
Immediate postindexed	[Rn], #+/-offset_8
Register postindexed	[Rn], +/-Rm

- LDR and STR instructions can load and store data on a boundary alignment that is the same as the data type size being loaded or stored.
- For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on.
- This example shows a load from a memory address contained in register *r1, followed by a store back to the same* address in memory.

- The first instruction loads a word from the address stored in register *r1 and places it into*

  register *r0.*

  The second instruction goes the other way by storing the contents of register r0 to the address contained in register r1.

  The offset from register r1 is zero. Register r1 is called the *base address register.*

Swap Instruction

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register. This instruction is an *atomic operation—it reads* and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

SWP	swap a word between memory and a register	$tmp = mem32[Rn]$ $mem32[Rn] = Rm$ $Rd = tmp$
SWPB	swap a byte between memory and a register	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

The swap instruction loads a word from memory into register *r0 and overwrites the memory* with register *r1.*

```
PRE mem32[0x9000] = 0x12345678
        r0 = 0x00000000
        r1 = 0x11112222
        r2 = 0x00009000
                SWP r0, r1, [r2]
POST mem32[0x9000] = 0x11112222
        r0 = 0x12345678
        r1 = 0x11112222
        r2 = 0x00009000
```

This instruction is particularly useful when implementing semaphores and mutual
exclusion in an operating system. You can see from the syntax that this instruction can also
have a byte size qualifier B, so this instruction allows for both a word and a byte swap.

SOFTWARE INTERRUPT INSTRUCTION

- A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.
- Syntax: SWI{<cond>} SWI_number

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

- When the processor executes an SWI instruction, it sets the program counter *pc to the* offset 0x8 in the vector table.
- The instruction also forces the processor mode to *SVC, which* allows an operating system routine to be called in a privileged mode.
- Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

EXAMPLE

- Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

- **PRE** cpsr = nzcVqift_USER
  pc = 0x00008000
  lr = 0x003fffff; lr = r14
  r0 = 0x12

  0x00008000 SWI 0x123456

- **POST** cpsr = nzcVqIft_SVC
  spsr = nzcVqift_USER
  pc = 0x00000008
  lr = 0x00008004
  r0 = 0x12

- Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register *r0 is used to* pass the parameter 0x12.
- The return values are also passed back via registers. Code called the *SWI handler is required to process the SWI call. The handler obtains* the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.

- The SWI number is determined by

  SWI_Number = <SWI instruction> **AND NOT(0xff000000)**
- Here the *SWI instruction is the actual 32-bit SWI instruction executed by the processor.*

Program Status Register Instructions

- The ARM instruction set provides two instructions to directly control a program status register (*psr).*
- *The MRS instruction transfers the contents of either the cpsr or spsr into* a register; in the reverse direction, the MSR instruction transfers the contents of a register into the *cpsr or spsr. Together these instructions are used to read and write the cpsr and spsr.*
- In the syntax you can see a label called *fields. This can be any combination of control* (*c), extension (x), status (s), and flags (f). These fields relate to particular byte regions in a psr.*
- The *c field controls the interrupt masks, Thumb state, and processor mode.*

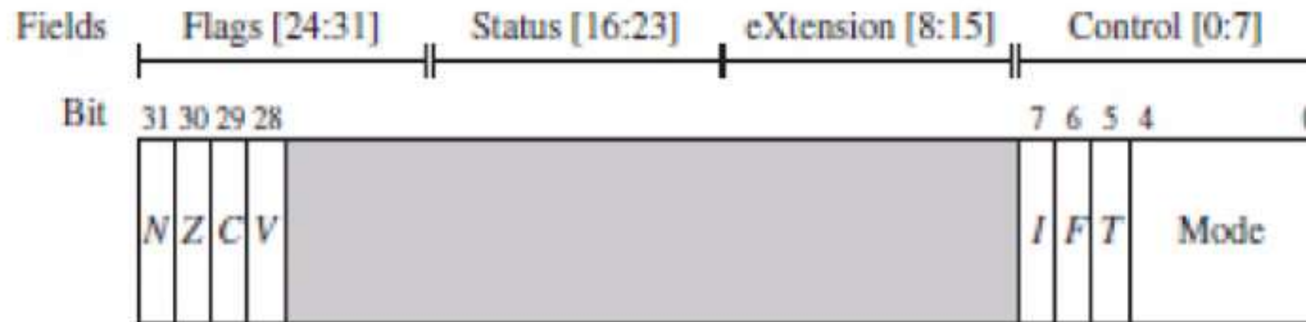
SYNTAX:

MRS{<COND>} RD,<CPSR | SPSR>

MSR{<COND>} <CPSR | SPSR>_<FIELDS>,RM

MSR{<COND>} <CPSR | SPSR>_<FIELDS>,#IMMEDIATE

psr byte fields



MRS	copy program status register to a general-purpose register	*Rd = psr*
MSR	move a general-purpose register to a program status register	*psr[field] = Rm*
MSR	move an immediate value to a program status register	*psr[field] = immediate*

Loading constants

- You might have noticed that there is no ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant.
- To aid programming there are two pseudo instructions to move a 32-bit value into a register

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

The first pseudo instruction writes a 32 bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.

The second pseudo instruction writes a relative address into a register, which will be encoded using a pc-relative expression.

Syntax:

```
LDR Rd, =constant
ADR Rd, label
```

Arm Instruction Set Advantages

- **All instructions are 32 bits long.**
- **Most instructions are executed in one single cycle.**
- **Every instructions can be conditionally executed.**
- **A load/store architecture**
  - Data processing instructions act only on registers
    - Three operand format
    - Combined ALU and shifter for high speed bit manipulation
  - Specific memory access instructions with powerful auto-indexing addressing modes
  - 32 bit ,16 bit and 8 bit data types
  - Flexible multiple register load and store instructions

ASSESSMENT

1) How many modes of operation is available in ARM7?

2)How many registers are available in ARM7?

Thank you