ARM ASSEMBLY GUIDELINE

Intro to ARM Assembly Programming

ARM Architecture

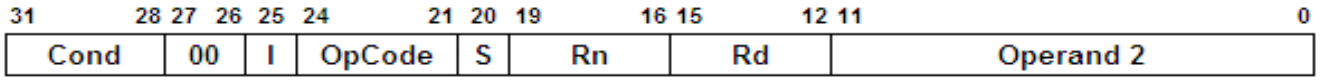
- ARM is a classic **RISC** architecture.
- RISC is known by simple, **fixed length** instruction set.
- CISC has large, complex, and **variable length** instruction set. (Ex. Intel Architecture)
- **Load-Store Architecture**: data operands must first be loaded into the CPU, and then stored back to main memory to save the results.
- ARM7 supports **32-bit ARM instruction set** and **16-bit Thumb instruction set**.
- Thumb instruction set is a subset of ARM instruction set.
- ARM instructions provide **higher performance**, while Thumb instructions reach higher **code density**.
- An ARM procedure can call Thumb procedure with few overhead, and vise versa.
- In the introduction, we only talk about ARM instructions, but not Thumb instructions.

Important Registers

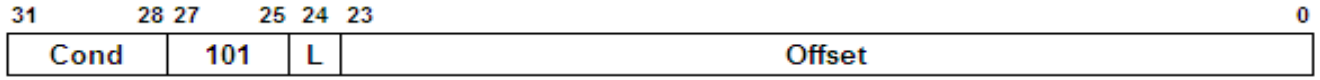
- ARM7 has 16 general purpose registers, R0 to R15.
- **Frame Pointer (FP)** — **R11** is by convention used to hold the **address of a subroutine's frame** on the stack. The frame contains storage for local variables and for registers which need to be saved into memory.
- **Intra-Procedure-call scratch register (IP)** — **R12** is by convention used to **backup the stack pointer** in a procedure call. After a subroutine returns, one can get original SP from IP.
- **Stack Pointer (SP)** — **R13** is by convention used as the Stack Pointer.
- **Link Register (LR)** — **R14** is by convention used for subroutine linkage. It automatically receives the return address when the BL instruction is executed.
- **Program Counter (PC)** — **R15** is always used as the Program Counter.
- Another important register is the **Current Program Status Register (CPSR)**.
- CPSR is set automatically during executing instructions, based on computing results.
- The top 4 bits of CPSR hold following ALU information. They will be set automatically during Compare Instructions and **Arithmetic Instruction** with **S flag**.
  - **Negative bit (N)** is set when the result is negative.
  - **Zero bit (Z)** is set when every bit of the result is zero.
  - **Carry bit (C)** is set when there is a carry out of the operation.
  - **Overflow bit (V)** is set when operation results in an overflow.

ARM Instruction Format

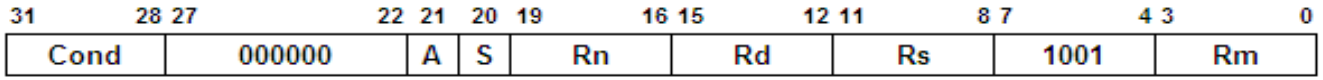
- Arithmetic Instructions



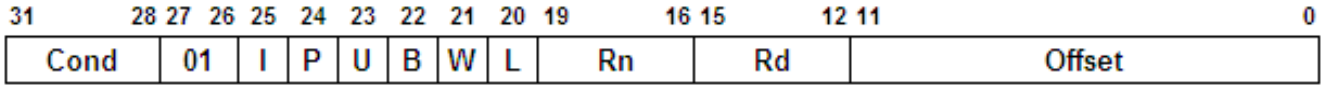
- Branch Instructions

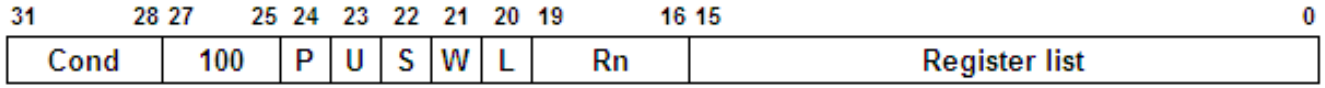


- Multiply Instructions



- Data Transfer Instructions

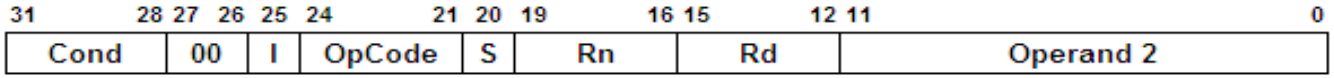




- Software Interrupt

ARM Instruction Format

- We take **Arithmetic Instructions** for example, to explain some important concepts. They will be described in following slides.



- <opcode> {<cond>} {S} <Rd>, <Rn> {, <operand2>}
- **Opcode**
  - LDR, STR, ADD, …
- **Cond (optional)**
  - **Condition of Execution**. Execute or not based on result of last execution (stored in CPSR)
  - EQ, NE, …
- **S (optional)**
  - Specify whether this instruction should affect the CPSR register.
  - **With S, CPSR register will be affected**, otherwise won't.
- **Rd, Rn**
  - Destination Register & First Operand Register
- **Operand2 (optional)**
  - Second operand. Can be an **immed_8r**, a **register**, or a **register with shift operation**.

Condition Code

- In many instructions, one can add an optional condition code to specify in which condition the instruction should be executed.
- When encounters instructions with condition code, CPU **checks CPSR register** and determines whether or not to execute the instruction.
- Examples:

```
ADDEQ    R1, R2, R3          /* If Equal, R1 = R2 + R3 */
SUBNE    R0, R5, #0x0F       /* If Not Equal, R0 = R5 – 0x0F */
BGT      Label1              /* If Greater Than, Branch to Label1 */
MOVLE    R1, #0x10           /* If Less or Equal, R1 = 0x10 */
LDRLT    R0, [R1, #0x10]     /* If Less Than, R0 = [R1 + 0x10] */
MOVGE    PC, LR              /* If Greater ro Equal, return from subroutine */
LDREQ    R3, = Label+30      /* If Equal, load Label address + 30 into R3 */
```

Condition Code

- This table shows all condition code with the corresponding flags in CPSR.

Code	Flag	Condition
EQ	Z = 1	Equal
NE	Z = 0	Not Equal
CS	C = 1	Unsigned Higher or Equal
CC	C = 0	Unsigned Lower Than
MI	N = 1	Negative number
PL	N = 0	Positive number of Zero
VS	V = 1	Overflow
VC	V = 0	No Overflow
HI	C = 1, Z = 0	Unsigned Higher
LS	C = 0, Z = 1	Unsigned Less Than or Equal
GE	N = V	Signed Greater Than or Equal
LT	N != V	Signed Less Than
GT	Z = 0, N = V	Signed Greater Than
LE	Z = 1, N != V	Signed Less Than or Equal
AL	Any	Always execute (default condition)

Immediate Number - Immed_8r

- ARM CPU has fixed instruction length (RISC).
- In most instructions, the Immediate numbers only occupy 12-bit field.
- So how to express a 32-bit number by 12 bits?
- ARM solution: **8 bits store value + 4 bits describe shift amount**
- 4 bits can express up to only 16 (a half of 32), so shift amount can be only even number within 0 to 32.
- A valid immediate number must be converted from a **8 bits value shifting an even number** (i.e. multiply by power of 2).
- This kind of immediate number refers to **immed_8r.**
- **Valid immed_8r:** 0x5A, 0xFF00, 0x1C000, 0x0FF, 0x3FC, 0x03A0
- **Invalid immed_8r:** 0xFFF (more than 8 bits), 0x101 (more than 8 bits), 0x7F8 (can't be converted by shifting an even number)
- 0000 0000 0000 0000 0000 0011 0110 1100 (Valid)
- 0000 0000 0000 0000 0000 0110 1101 1000 (Invalid)
- Surprisingly, **0xFFFFFFFF** is a **Valid** immediate number, because ARM instructions can express it by **~0x00000000**.

Shift and Rotate Operations

- The **operand2** may consists of a **register** and an **shift or rotate operation**.
- Here we explain some offset operations. (C stands for **C flag** in CPSR)
- LSL (Logical Shift Left)
  - MOV R0, R2, **LSL #12** /* (R2 << 12) → R0 */
  - 

- LSR (Logical Shift Right)
  - MOV R0, R2, **LSR R3** /* (R2 >> R3) → R0 */
  - 

- ASR (Arithmetic Shift Right — with **Sign Extension**)
  - ADD R0, R2, R6, **ASR #3** /* R0 = R2 + (R6 arithmetic shift right 3 bits)*/
  - **R6 >= 0**
  - **R6 < 0**
  - 

- ROR (Rotate Right)
  - SUB R1, R3, R0, **ROR #2** /* R1 = R3 - (R0 rotate right 2 bits) */
  - 

ARM Addressing Mode

- ARM CPU supports 9 addressing modes.
- Here we learn 5 of them:
- Register Addressing
  - MOV R1, R2 /* R2 → R1 */
  - SUB R0, R1, R2 /* R1 – R2 → R0 */
- Relative Addressing
  - B FUNC1 /* branch to FUNC1 */
  - BEQ LOOP /* branch to LOOP if equal*/
  - LDR PC, Label /* load address Label into R1 */
- Immediate Addressing
  - SUB R0, R0, #1 /* R0 – 1 → R0 */
  - MOV R0, #0xFF00 /* 0xFF00 → R0 */

ARM Addressing Mode

- Register Indirect Addressing
  - LDR R1, [R2]

    /* Use value in R2 as address; read data from memory; put data into R1 */
  - ADD R0, R1, [R2]

    /* Use R2 as address; read data from memory; add by R1; put into R0 */
- Register Offset Addressing
  - MOV R0, R2, LSL #3 /* (R2 << 3) → R0 */
  - AND R0, R1, R2, LSR R3 /* (R2 >> R3) & R1 → R0 */
- Base and Offset Addressing
  - LDR R2, [R3, #0x0F]

    /* Take value in R3, adding by 0x0F, using it as address, loading data to R2 */
  - STR R1, [R0, #-2]

    /* Take value in R0, adding by -2, using it as address, storing data of R1 there */

About the Assembler

- The Directive **".text"** declares that following codes are in text section.
- A name following by a colon like "_start:", "main:", "loop:" are labels, which mark a block of code.
- Within an instruction, a label represents the **starting address** of a block of code.
- To let another files use a block of code (or subroutine), one need to use **".global"** to declare that subroutine is global.
  - .global main /* declare main as a global subroutine */
- CPU starts the execution from **"_start:"** label.
- Comments can be written in one of the following forms:
  - /* this is comment */
  - // this is comment
  - @ this is comment
- **".data"** declares start portion of the source code where data allocation is declared.
- **".word n"** reserves one word of storage in data section, initialized to n.
- The directive **".equ"** can be used to define a macro (similar with #define in C).
  - .equ PIN7, 0xE01FC1A0 /* define PIN7 as memory location 0xE01FC1A0 */

About the Assembler

- The table shows all ARM assembler directives.

Directive	Usage
.text	Begin a source code area containing instructions
.data	Start portion of the source code where data allocation is declared
.end	This denotes the end of the source code
.global label	Make label externally visible
_start:	Make this address the start address for the Linker
.extern label	declare label as being a storage location defined in another module
.word n	Reserve one word of storage (4 bytes), initialized to n
.byte n	Reserve one byte of storage, initialized to n
.skip k	Reserve k consecutive bytes of memory, uninitialized
.ascii "string"	Place an ASCII string in memory
.asciz "string"	Place a null-terminated ASCII string in memory
.align [n]	Align next item on an address divisible by 2^n. I.e., 0 → byte boundary, 1 → halfword boundary, 2 → word bound-ary. The default is a word boundary if n is omitted.
.equ name, value	Define symbolic label name to represent the constant value.

A Simple Example

- The following code calls a subroutine "sum" to sum up two integers, and then goes into infinite loop.

```
.text
.global  _start
_start:
        MOV     R0, #10     /* R0 = 10 */
        MOV     R1, #20     /* R1 = 20 */
        BL      sum         /* branch to sum, setting LR */
        B       stop        /* branch to stop */
sum:
        ADD     R2, R0, R1  /* R2 = R0 + R1 */
        MOV     PC, R14     /* Return from sum (PC = LR) */
stop:
        B       stop        /* infinite loop */
```

- We have learned almost all concepts we need, now let's start describing some useful instructions in detail.

Arithmetic Instructions

- ADD {cond} {S} Rd, Rn, operand2
  - ADDS R1, R1, #1 /* R1= R1+1; affect CPSR */
  - ADD R3, R1, R2, LSL R3 /* R3 = R1+ (R2<<R3) */
- SUB {cond} {S}Rd, Rn, operand2
  - SUB R6, R7, #0x10 /* R6 = R7 – 0x10 */
  - SUBS R1, R2, R3 /* R1 = R2 – R3; affect CPSR */
- ADC {cond} {S} Rd, Rn, operand2
  - Addition with **C flag** of **CPSR register**
  - Rd = operand2 + Rn + **C flag**
  - ADD R0, R0, R2 /* 64-bits addition */
    ADC R1, R1, R3 /* (R1 R0) = (R1 R0) + (R3 R2) */
- SBC {cond} {S}Rd, Rn, operand2
  - Subtraction with **C flag** of **CPSR register**
  - SUB R0, R0, R2 /* 64-bits subtraction */
    SBC R1, R1, R3 /* (R1 R0) = (R1 R0) - (R3 R2) */

Arithmetic Instructions

- RSB {cond} {S} Rd, Rn, operand2
  - RSB R3, R1, #0xFF0 /* R3 = R1 – #0xFF0 */
  - RSBS R0, R1, #0 /* R0 = -R1; affect CPSR */
- AND {cond} {S} Rd, Rn, operand2
  - AND R0, R0, #1 /* R0 = R0 & 1 (Extract bit 0) */
  - AND R2, R1, R3, LSR #0x0F /* R2 = R1& (R3 >> 0x0F) */
- ORR {cond} {S} Rd, Rn, operand2
  - ORR R0, R0, #0x04 /* R0 = R0 | 0x04 (Set bit 2) */
  - ORRS R1, R2, R6 /* R1 = R2 | R6; affect CPSR */
- EOR {cond} {S} Rd, Rn, operand2
  - EOR R1, R1, #0x0F /* Reverse lowest 4 bits of R1 */
  - EORS R1, R7, R5, LSL #2 /* R1 = R7 ^ (R5 << 2); affect CPSR */

Multiply Instructions

- MUL {cond} {S} Rd, Rm, Rs
  - The format is different from other arithmetic instructions
  - **Rd can't be the same with Rm**
  - MUL R1, R2, R3 /* R1 = R2 * R3 */
  - MULS R2, R0, R1 /* R2 = R0 * R1; update CPSR */

Compare Instructions

- Compare Instructions store results in CPSR register.
- Usually followed by instructions with conditions like EQ, NE, … etc.
- CMP {cond} Rn, operand2
  - Do **Rn – operand2**, setting **CPSR register**, but discard the SUB result.
  - CMP R1, #10 /* Compare R1 and 10, setting CPSR register */
  - CMP R1, R2 /* Compare R1 and R2, setting CPSR register */
- TST {cond} Rn, operand2
  - Do Rn & operand2, setting CPSR register, but discard the AND result.
  - TST R0, #0x08 /* Test if bit 3 of R0 is 1 */
- TEQ {cond} Rn, operand2
  - Do Rn ^ operand, setting CPSR register, but discard the EOR result.
  - TEQ R0, R1 /* Test if R0 = R1 */

Data Transfer Instructions

- MOV {cond} {S} Rd, operand2
  - MOV R1, 0x10 /* R1 = 0x10 */
  - MOV R1, R0 /* R0 = R1 */
  - MOVS R3, R1, LSL #2 /* R3 = R1 << 2; affect CPSR */
  - MOV PC, LR /* **Return from subroutine** */
- MVN {cond} {S} Rd, operand2
  - Rd ← **~opernad2**
  - Can express larger immediate number.
  - MVN R1, #**0xFF** /* R1 = **0xFFFFFF00** */
  - MVN R1, R2 /* R1 = ~R2 */

Load and Store Instructions

- Address can be expressed by one of the addressing modes.
- LDR {cond} Rd, <address>
  - Load a **32-bit word** from **[address]**
  - LDR R1, [R0, #0x12] /* load [R0 + 12] into R1 */
  - LDR R1, [R0] /* load [R0] into R1 */
  - LDR R1, [R0, R2, LSL #2] /* R1 = [R0+R2<<2] */
  - LDR R1, Label /* R1 = [Label], **label must within ±4KB from PC** */
- STR {cond} Rd, <address>
  - Store a **32-bit word** into **[address]**
  - STR R1, [R0, #0x12] /* [R0 + 12] = R1 */
  - STR R1, [R0] /* [R0] = R1 */
  - STR R1, [R0, R2, LSL #2] /* [R0+R2<<2] = R1 */
  - STR R1, Label /* [Label] = R1, **label must within ±4KB from PC** */

Load and Store Instructions

- Other Load Instructions
  - **LDRB** — load a **unsigned byte** from [address] into Rd.
  - **LDRH** — load a **unsigned 16-bit half word** from [address] into Rd.
  - **LDRSB** — load a **signed byte** from [address] into Rd.
  - **LDRSH** — load a **signed 16-bit half word** from [address] into Rd.
  - When load a **unsigned data**, remaining bits of Rd **will be feed in 0**.
  - When load a **signed data**, remaining bits of Rd will be **sign-extended**.
- Other Store Instructions
  - **STRB** — store a **byte** from Rd into [address].
  - **STRH** — store a **16-bit half word** from Rd into [address].

Load and Store Multiple Instructions

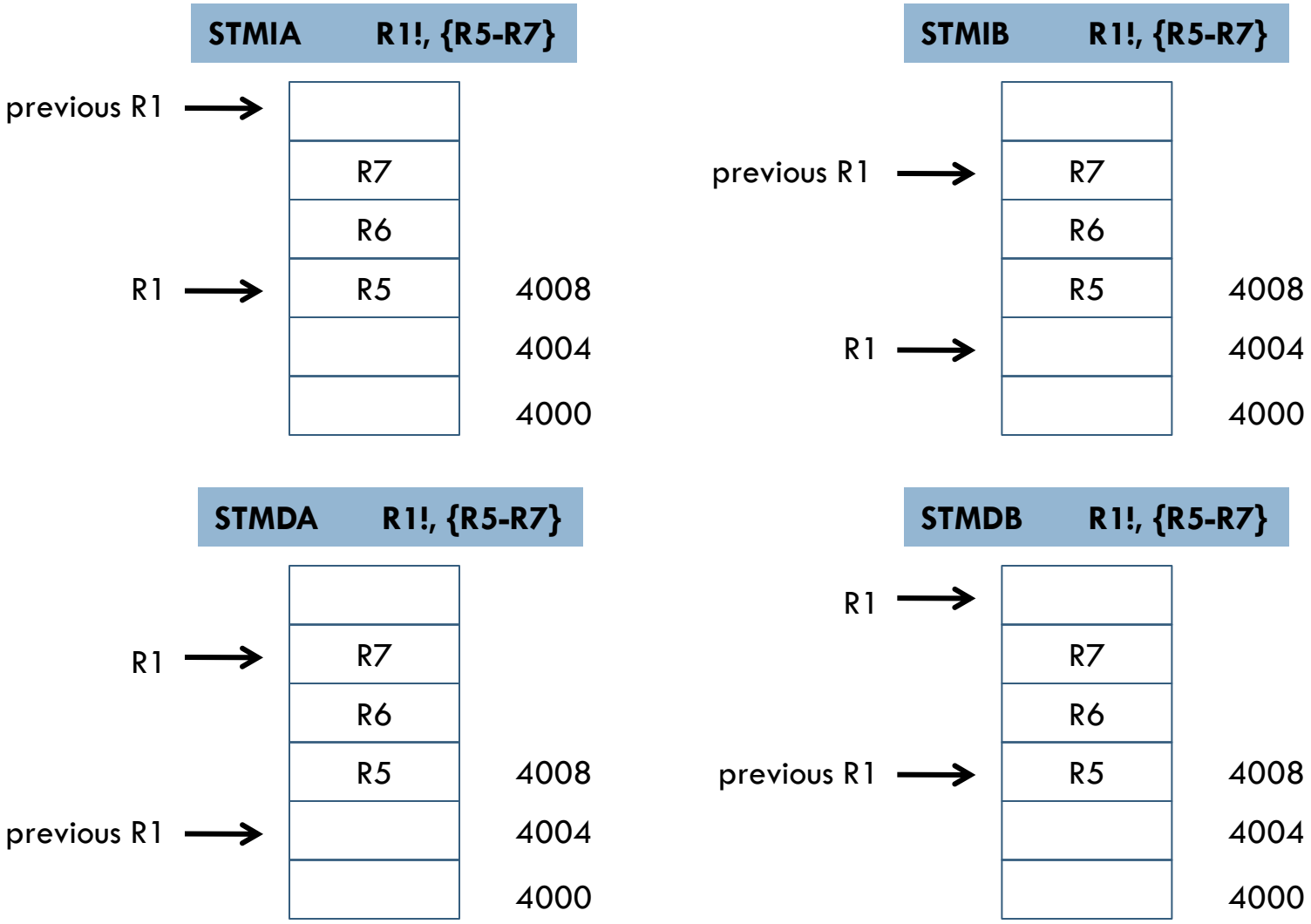
- **Load multiple** and **Store multiple** instructions transfer the content of **multiple registers** between memory and the processor in a single instruction.
- They are useful for saving and restoring context during subroutine calls.
- Also useful for Push to and Pop from the stack.
- LDM {cond} <mode> Rd {!}, {register list}
- STM {cond} <mode> Rd {!}, {register list}
- The **starting address** of the memory area used to store/load the contents of the register from the list is addressed by a **base register, Rd**.
- The base register Rd can be incremented or decremented automatically if the extra **'!'** is added.
- The choices given in <mode> decide two aspects of the execution:
  - Whether the **base address** grows **upwards** or **downwards**.
  - Whether the **base address** is going to be adjusted **before** or **after** the operation occurs.

Load and Store Multiple Instructions

- When they are used to **push to and pop from stack**, the following mode **FD, ED, FA, EA** are used.
- When they are used to **block load/store from a memory area** (not stack), the following mode **IA, IB, DA, DB** are used.
- FD = IA, ED = IB, EA = DB, FA = DA.
- STMIA R9!, {R1-R3,R6} /* store R1-R3 and R6 starting from [R9], updating R9 */
- LDMDB R9!, {R1-R3,R6} /* load R1-R3 and R6 starting from [R9], updating R9 */
- STMFD SP!, {R1-R3} /* Push R1-R3 onto stack */
- LDMFD SP!, {R1-R3} /* Pop from stack into R1-R3 */

Stack		Other Memory Area	
FD	Full Descending	IA	Increment After
ED	Empty Descending	IB	Increment Before
FA	Full Ascending	DA	Decrement After
EA	Empty Ascending	DB	Decrement Before

Load and Store Multiple Instructions



Branch Instructions

- B {cond} <address>
  - **PC ← Label**
  - Can't jump longer than **±32Mb**.
  - B Main /* Jump to starting point of Main */
  - B 0x1234 /* Jump to physical address 0x1234*/
- BL {cond} <address>
  - Store **return address** in **Link Register**, and then jump to Label
  - **LR ← PC – 4, PC ← Label**
  - Can't jump longer than **±32Mb**.
  - BL Function1 /* Store return address, then jump to Function1*/

Other Instructions

- SWI {cond} immed_24
  - Issue a **Software Interrupt**.
  - Immed_24 is a **24-bit immediate number**, ranging from 0 to 16777215.
  - SWI #12 /* Call interrupt service routine No.12 */
  - SWI #500 /* Call interrupt service routine No. 500 */
- MRS {cond} Rd, PSR
  - Read value in CPSR or SPSR register into Rd.
  - SPSR is a register serving as a backup of CPSR when interrupts happen.
  - **Rd can't be R15 (R15 is used as PC)**
  - MRS R1, CPSR /* Read CPSR value into R1 */
  - MRS R7, SPSR /* Read SPSR value into R7 */

Pseudo Instructions

- Here we describe the two most useful pseudo instructions.
- LDR {cond} Register, = <expression>
  - LDR Pseudo instruction contains '=', normal LDR does not.
  - Will be converted to MOV, ADD, MVN…
  - Immediate number in <expression> needn't to add '#'
  - **The expression can express 32-bit constant.**
  - LDR R0, = 0x12345678 /* load 32-bit immediate number into R0 */
  - LDR R3, = Label+30 /* load Label address + 30 into R3 */
  - LDR R1, = IOPIN /* load IOPIN into R1 */
- NOP
  - CPU does nothing and **delays a cycle** (may be translated into MOV R0, R0)
  - Label:

```
        NOP             /* delay 2 cycles */
        NOP
        B       Loop    /* than branch to Loop */
```

Reference

- http://netwinder.osuosl.org/pub/netwinder/docs/arm/ARM7500FEvB_3.pdf
- http://www.heyrick.co.uk/assembler/qfinder.html
- http://ozark.hendrix.edu/~burch/cs/230/arm-ref.pdf
- http://www.peter-cockerell.net/aalp/html/frames.html
- http://www.csc.uvic.ca/courses/csc230/200901/ARMSim/ARM_Manual4.pdf
- http://simplemachines.it/doc/arm_inst.pdf
- http://simplemachines.it/doc/QRC0001H_rvct_v2.1_arm.pdf