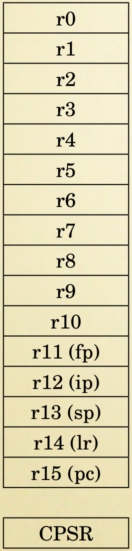
Modern Assembly Language Programming with the ARM processor

Chapter 4: Data Processing and Special Instructions

ARM User Program Registers



- Thirteen general-purpose registers (`r0-r12`)
- The stack pointer (`r13` or `sp`)
- The link register (`r14` or `lr`)
- The program counter (`r15` or `pc`)
- Current Program Status Register (`CPSR`)

Instruction Categories

- Load/Store Instructions
- Data processing Instructions
    - Arithmetic Operations
    - Logical Operations
    - Comparison Operations
    - Data Movement Operations
    - Multiplication Operations
- Branch Instructions
    - Branch with Link (subroutine call)
    - Conditional Branches
- Special Instructions
- Pseudo-Instructions

Hardware-Related Register Use

- All instructions can access `r0-r14` directly.
- Most instructions also allow use of the program counter (`r15`).
- Specific instructions to allow access to `CPSR`.
- `r14`, `r15`, and `CPSR` are “hardware special”.

CPSR

- The CPSR contains four “flag” bits (bits 28-31).
- Data Processing Instructions can affect the flag bits (if the programmer wants them to).
- The meaning of the flags depends on the type of instruction that set them.
- These bits can be used by subsequent instructions to control execution and branching.
- *Most* instructions can have a condition attached, to control whether or not they are actually executed.

CPSR Condition Bits

The following table shows the meaning of the four bits depending on the type of instruction that set or cleared them.

Name	Logical Instruction	Arithmetic Instruction
N	No meaning	Bit 31 of the result has been set. Indicates a negative number in signed operations.
Z	Result is all zeroes	Result of operation was zero
C	After Shift operation ‘1’ was left in carry flag	Result was greater than 32 bits
V	No meaning	Result was greater than 31 bits. Indicates a possible corruption of the sign bit in signed numbers.

Conditional Execution

```
op{<cond>}  operands
```

<cond>	English meaning
al	always (this is the default <cond>
eq	Z set (=)
ne	Z clear (≠)
ge	N set and V set, or N clear and V clear (≥)
lt	N set and V clear, or N clear and V set (<)
gt	Z clear, and either N set and V set, or N clear and V set (>)
le	Z set, or N set and V clear,or N clear and V set (≤)
hi	C set and Z clear (unsigned >)
ls	C clear or Z (unsigned ≤)
hs	C set (unsigned ≥)
cs	Alternate name for HS
lo	C clear (unsigned <)
cc	Alternate name for LO
mi	N set (result < 0)
pl	N clear (result ≥ 0)
vs	V set (overflow)
vc	V clear (no overflow)

Example Data Processing Instruction

- Syntax:
  <Operation>{<cond>}{s|S}   Rd, Rn, Operand2

- Examples:

```
add     r0, r1, r2  @ r0=r1+r2
add     r8, r5, r7  @ r8=r5+r7
addne   r9, r1, r3  @ if(ne) then r9=r1+r3
adds    r4, r6, r7  @ r4=r6+r7 and set SPSR flags
addeqs  r6, r8, r2  @ if(eq) r6=r8+r2 and set SPSR flags
adds    r0, r1, r2  @ r0=r1+r2 AND the condition flags
                    @ are updated
addcs   r3, r4, r5  @ If the carry bit is set,
                    @ then r3=r4+r5,
                    @ else ignore this instruction
adcs    r2,r4,r6    @ Add r4, r6, and the carry bit,
                    @ store the result in r2, and set
                    @ the SPSR flags
```

Operand2

Most of the data processing instructions use Operand2 as the second operand.

Operand2 can be one of three things:

- A register (`r0–r15`)
- A shifted or rotated register
- A 32-bit immediate value that can be constructed by shifting, rotating, and/or complementing an 8-bit immediate value

Shifting Operand2

There are five mnemonics that can be used to specify an arithmetic or logical <shift_op>, or a rotation

`lsl`	Logical Shift Left
`lsr`	Logical Shift Right
`asr`	Arithmetic Shift Right
`ror`	Rotate Right

Note: An arithmetic shift left is equivalent to a logical shift left.

There is also a mnemonic for an extended rotation

`rrx`	Rotate Right with Extend

The RRX operation rotates one place to the right but the Carry flag is used along with Rx to provide a 33 bit quantity to be rotated.

Forms for Operand2

1. #<immediate> (For 32-bit immediate values that can be constructed by shifting and complementing an 8 bit value.)
2. Rm (Any of the 16 registers r0-r15)
3. Rm, <shift_op> #<shift_imm>
4. Rm, <shift_op> Rs
5. Rm, RRX

More on Immediate values

The assembler must be able to construct the value using only 8 bits of data and a shift or rotate, and/or a complement.

Examples:

#32	Ok because it is between 0 and 255.
#1021	Illegal because the number cannot be created from an 8-bit value using shift/rotate/complement.
#1024	Ok because it is 1 shifted left 10 bits.
#-1	Ok because it is the one's complement of 0
#0xFFFFFFFE	Ok because it is the one's complement of 1
#0xEFFFFFFF	Ok because it is the one's complement of 1 shifted left 31 bits

For immediate values that can cannot be constructed by shifting and complementing an 8 bit value, we have to use

```
ldr Rn,=<immediate|symbol>
```

Then `Rn` can be used as Operand2.

Arithmetic Operations

- Operations:

ADD	Rn + operand2	@ Add
ADC	Rn + operand2 + carry	@ Add with carry
SUB	Rn – operand2	@ Subtract
SBC	Rn – operand2 + carry – 1	@ Subtract with carry (borrow)
RSB	operand2 – Rn	@ Reverse subtract
RSC	operand2 – Rn + carry – 1	@ Reverse subtract with carry

- Syntax:
  <Operation>{<cond>}{S}   Rd, Rn, Operand2

  The optional S specifies whether or not the instruction should affect the bits in the CPSR.
- Examples

```
add     r0, r1, r2  @ r0=r1+r2 and don't set CPSR flags
subgt   r3, r3, #1  @ if (GT) then r3=r3-1 and don't set
                    @ CPSR flags\\
rsbles r4, r5, #5   @ if (LE) then r4=5-r5 and set CPSR
                    @ flags
```

Logical Operations

- Operations:

AND	Rn & operand2	@ AND
EOR	Rn ˆ operand2	@ Exclusive OR
ORR	Rn | operand2	@ OR
ORN	!(Rn | operand2)	@ NOR
BIC	Rn & !operand2	@ AND NOT (Bit Clear)

- Syntax:
  <Operation>{<cond>}{S}   Rd, Rn, Operand2

  The optional S specifies whether or not the instruction should affect the bits in the CPSR.
- Examples

```
and      r0, r1, r2    @ r0=r1&r2 and don't set CPSR flags
biceq    r3, r3, #1    @ if (EQ) then r3=r3&!0x00000001 and
                       @ don't set CPSR flags
eorles   r4, r5, #5    @ if (LE) then r4=R5^0x00000005 and
                       @ set CPSR flags
```

Comparison Operations

Comparison operations update the CPSR flags, but have no other effect.

- Operations:

CMP	Rn − operand2	@ Compare
CMN	Rn + operand2	@ Compare Negative
TST	Rn & operand2	@ Test
TEQ	Rn ˆ operand2	@ Test equivalence

- Syntax:
  <Operation>{<cond>}   Rn, Operand2
- Examples

```
cmp     r0, r1    @ Compare r0 to r1 and set CPSR flags
tsteq   r2, #5    @ if (EQ) Compare r2 to 5
                  @ and set CPSR flags
```

Data Movement Operations

- Operations:

MOV	Rd, operand2	@ Copy operand2
MVN	Rd, !operand2	@ Copy 1's complement of operand2

- Syntax:
  <Operation>{<cond>}{S} Rd, Operand2
- Examples

```
mov    r0, r1          @ r0 = r1
movs   r2, #10         @ r2 = 10
mvneq  r1, #1          @ if (EQ) then r1 = -1
movles r2, r2, ASR #1  @ if (LE) then r2 = r2 / 2
                       @ and set CPSR flags
```

Multiply Operations with 32-bit Results

- Operation:
  MUL   Rd = Rm × Rs   @ Multiply with 32-bit result
- Syntax:
  MUL{<cond>}{S}   Rd, Rm, Rs
- Operation:
  MLA Rd = Rm × Rs + Rn   @ Multiply-accumulate with 32-bit result
- Syntax:
  MLA{<cond>}{S}   Rd, Rm, Rs, Rn
- Examples

```
mul    r0, r1, r2
mla    r0, r1, r2, r3
muleq  r0, r1, r2
mlas   r0, r1, r2, r3
mulnes r0, r1, r2
mlalts r0, r1, r2, r3
```

Multiply Operations with 64-bit Results

- Operations:

SMULL	RdHi:RdLo = Rm × Rs	@ Signed multiply with 64-bit result
UMULL	RdHi:RdLo = Rm × Rs	@ Unsigned multiply with 64-bit result
SMLAL	RdHi:RdLo = Rm × Rs + RdHi:RdLo	@ Signed multiply-accumulate with 64-bit result
UMLAL	RdHi:RdLo = Rm × Rs + RdHi:RdLo	@ Unsigned multiply-accumulate with 64-bit result

- Syntax:
  <Operation>{<cond>}{S}   RdLo, RdHi, Rm, Rs
- Examples

```
smull    r0, r1, r3, r4
smulls   r0, r1, r3, r4
umlaleq  r0, r1, r3, r4
```

Divide Instructions

The divide is available on most ARMv7 processors (Cortex M0 and M1 do not have hardware divide).

- Operation:

  SDIV Rd = Rn ÷ Rm @ Signed divide

  UDIV Rd = Rn ÷ Rm @ Unsigned divide

- Syntax:
  SDIV|UDIV{<cond>} Rd, Rm, Rn
- Example

```
div  r0, r1, r2
```

Accessing the CPSR and SPSR

- Operations:

MRS	@ Move from Status Register
MSR	@ Move to Status Register

- Syntax:
  MRS{<cond>}   Rd, CPSR{_<fields>}
  MSR{<cond>}   SPSR{_<fields>}, Rd
  <fields> is any combination of:

c	control field
x	extension field
s	status field
f	flags field

- Example Usage:

```
mrs   R0, CPSR              @ Read the CPSR into r0
bic   R0,R0, #0xF0000000    @ Clear all of the flags
msr   CPSR_f, R0            @ Write the flags field to CPSR
```

Operating System Calls

- Operation:
  SWI   perform software interrupt
- Syntax:
  SWI   <syscall_number>
- In Linux, the <syscall_number> is ignored. The actual system call number is passed in register r7.
- Example

```
        mov  r0, #1      @ fd -> stdout
        ldr  r1, =msg    @ buf -> msg
        ldr  r2, =len    @ count -> len(msg)
        mov  r7, #4      @ write is syscall #4
        swi  #0          @ invoke syscall
```

Thumb Mode

The ARM processor has an alternate mode where it executes a 16-bit instruction set known as Thumb. This instruction allows us to change the mode and branch to Thumb code.

- Operation:

BX	Like BL, but also change to Thumb mode.

- Syntax:
  BX{<cond>}   <target_address>
- Example

```
bx  my_thumb_code
```

Chicken and Egg Problem

- You have to learn the register set, instruction set and assembler directives before you can write assembly.
- You have to write assembly in order to learn the register set, instruction set, and assembler directives.

You may feel unsure of what you are doing and not understand everything at first.

That's ok. Ask questions if you get stuck.

Instruction Summary

ADC	Add with carry	MRS	Move from CPSR or SPSR register
ADD	Add	MSR	Move to CPSR or SPSR register
AND	AND	MUL	Multiply
B	Branch	MVN	Move Negative
BIC	Bit Clear	ORR	OR
BL	Branch and Link	RSB	Reverse Subtract
BX	Branch and Exchange	RSC	Reverse Subtract with Carry
CDP	Coprocessor Data Processing *	SBC	Subtract with Carry
CMN	Compare Negative	SDIV	Signed integer division
CMP	Compare	STC	Store Coprocessor register *
EOR	Exclusive OR	STM	Store Multiple
LDC	Load Coprocessor Register *	STR	Store Register
LDM	Load Multiple Registers	STREX	Store Register Exclusive
LDR	Load Register	SUB	Subtract
LDREX	Load Register Exclusive	SWI	Software Interrupt
MCR	Move to Coprocessor Register *	SWP	Swap register with memory
MLA	Multiply Accumulate	TEQ	Test bitwise equality
MOV	Move Register or Constant	TST	Test bits
MRC	Move from Coprocessor Register *	UDIV	unsigned integer division

* Not covered yet.