ARM9TDMI

“La veritá, ma non tutta la veritá.”

Microelectronic Systems
2017

Barberis Enrico – s239564
Cipolletta Antonio - s235687

Table of Contents

1 Data Processing instructions 4
1.1 Data operation with PC not as destination register 4
1.1.1 Opcode and description 4
1.1.2 Shift and rotate opcodes 5
1.1.3 Opcode specification 6
1.1.4 Timing 7

2 Data operation with register PC (R15) as Rd (register destination) 8
2.1.1 Opcode and description 8
2.1.2 Timing 8

3 Data transfer operations 9
3.1 Single data transfer operations 9
3.1.1 Encoding 9
3.1.2 Important notes 11
3.1.3 Load timings and details 11
3.1.4 Pipeline stage details 12
3.1.5 Store timing and details 12
3.2 Multiple Load/Store 13
3.2.1 Important notes 13
3.2.2 Encoding 13
3.2.3 Multiple register transfer addressing modes 13
3.2.4 Timing 15
3.3 Swap 17
3.3.1 Encoding 17
3.3.2 Description 17
3.3.3 Timing 18

4 Branch 19
4.1 Branch and Branch with link 19
4.1.1 Encoding 19
4.1.2 Timing 19
4.2 BX 20
4.2.1 Encoding and description 20

5 Miscellaneous instructions 20
5.1 SWI 20
5.1.1 Encoding and description 20
5.1.2 Timing 21
5.2 MRS 22
5.2.1 Encoding and description 22
5.2.2 Timing 22
5.3 MSR 23
5.3.1 Encoding and description 23
5.3.2 Timing 23

6 Multiplication 24
6.1 Opcodes and description 24
6.1.1 MUL - Multiply 24
6.1.2 MLA – Multiply and accumulate 24
6.1.3 UMULL - Unsigned Multiply Long 24

6.1.4 UMLAL - Unsigned Multiply Accumulate Long 25
6.1.5 SMULL - Signed Multiply Long 25
6.1.6 SMLAL - Signed Multiply Accumulate Long 25
6.2 Timing 25

7 Exceptions 26
7.1 SWI, UNDEFINED 27
7.2 IRQ, FIQ, PREFETCH 27
7.3 DATA ABORT 28
7.4 Exception vector table 28

8 CPSR/SPSR Management 29

9 Appendix 31
9.1 Conditional codes 31
9.2 Register file view 32

1 Data Processing instructions

1.1 Data operation with PC not as destination register

1.1.1 Opcode and description

32-bit immediate

31 28	27	26	25	24 21	20	19 16	15 12	11 8	7 0
cond	0	0	1	opcode	S	Rn	Rd	rotate_imm	immed_8

Immediate shifts

31 28	27	26	25	24 21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	shift	0	Rm

Register shifts

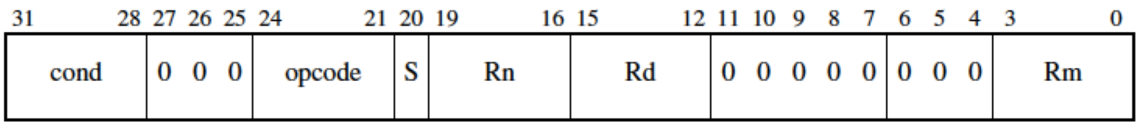
31 28	27	26	25	24 21	20	19 16	15 12	11 8	7	6 5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm

- **Opcode**: specifies the operation of the instruction.
- **S:** indicates that the instruction updates the condition codes (only if Rd is not R15).
- **Rd:** specifies the destination register.
- **Rn:** specifies the first source operand register.
- **Bits[11:0]:** the fields within bits[11:0] are collectively called a **shifter operand**
- **Bit[25]:** is referred to as the I bit, and is used to distinguish between an immediate shifter operand and a register-based shifter operand.

  Shifter operands:
- 32 bit Immediate can be generated only by the following operations:
  - 4 bit to specify the rotate amount that is multiplied by 2 internally (only 0,2,4,8...26,28,30 amounts are possible)
  - 8 bit for constant value to be rotated
  - Examples of some valid constants are:
    0xFF,0x104,0xFF0,0xFF00,0xFF000,0xFF000000,0xF000000F
    Some invalid constants are:
    0x101,0x102,0xFF1,0xFF04,0xFF003,0xFFFFFFFF,0xF000001F
- Immediate shifts
  - See following instruction encoding
- Shift amount register
  - See following instruction encoding

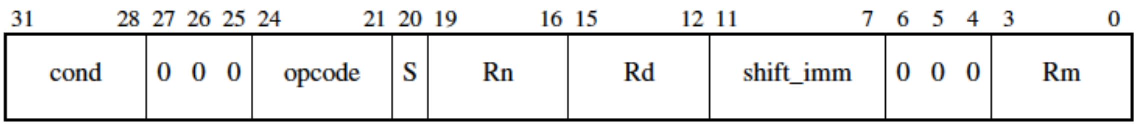
1.1.2 Shift and rotate opcodes

A5.1.4 Data-processing operands - Register



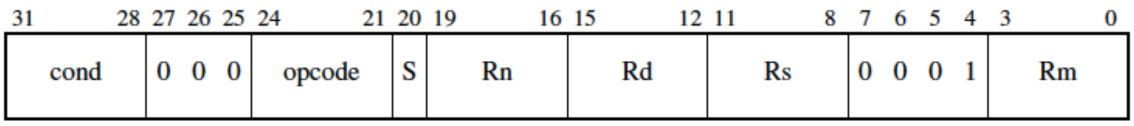
31 28	27	26	25	24 21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	0	0	0	0	0	0	0	0	Rm

A5.1.5 Data-processing operands - Logical shift left by immediate



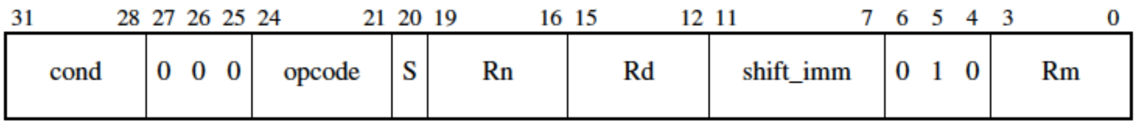
31 28	27	26	25	24 21	20	19 16	15 12	11 7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	0	0	0	Rm

A5.1.6 Data-processing operands - Logical shift left by register



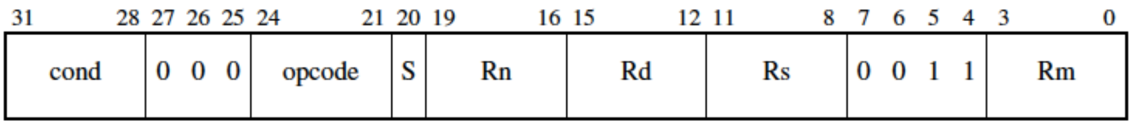
31 28	27	26	25	24 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	0	0	1	Rm

A5.1.7 Data-processing operands - Logical shift right by immediate



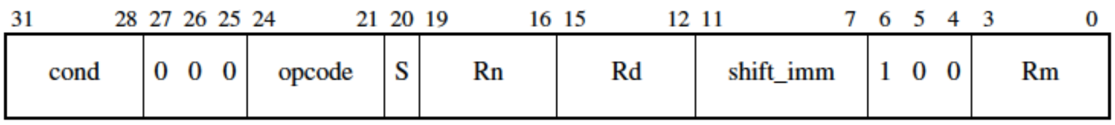
31 28	27	26	25	24 21	20	19 16	15 12	11 7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	0	1	0	Rm

A5.1.8 Data-processing operands - Logical shift right by register



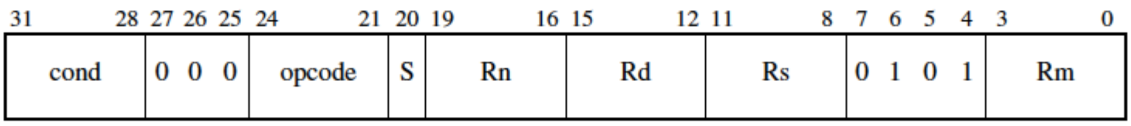
31 28	27	26	25	24 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	0	1	1	Rm

A5.1.9 Data-processing operands - Arithmetic shift right by immediate



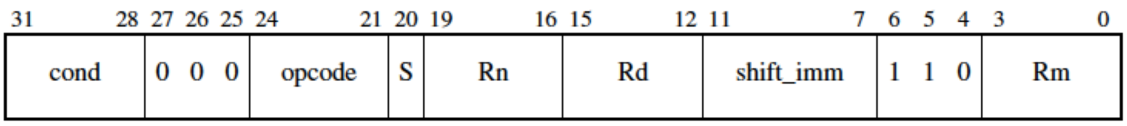
31 28	27	26	25	24 21	20	19 16	15 12	11 7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	1	0	0	Rm

A5.1.10 Data-processing operands - Arithmetic shift right by register



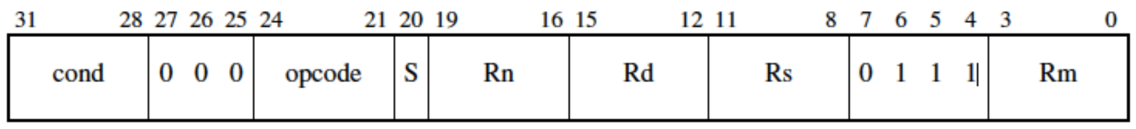
31 28	27	26	25	24 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	1	0	1	Rm

A5.1.11 Data-processing operands - Rotate right by immediate



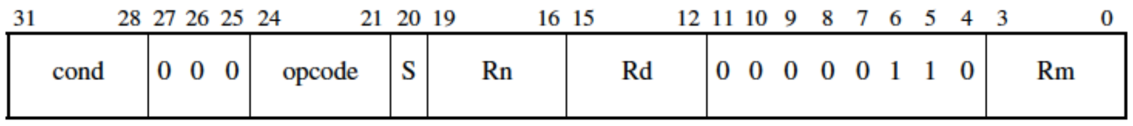
31 28	27	26	25	24 21	20	19 16	15 12	11 7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	1	1	0	Rm

A5.1.12 Data-processing operands - Rotate right by register



31–28	27	26	25	24–21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	1	1	1	Rm

A5.1.13 Data-processing operands - Rotate right with extend



31–28	27	26	25	24–21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
cond	0	0	0	opcode	S	Rn	Rd	0	0	0	0	0	1	1	0	Rm

NB: ROR 0 = RRX

RRX = rotate right of one position with carry. If S=1 it performs a rotation on 33 bits.

1.1.3 Opcode specification

Opcode (decimal base)	Performed operation	Operation type
0	AND	Logical
1	EOR (XOR)	Logical
2	SUB (Rn - B)(1)	Arithmetic
3	RSB (Reverse sub: B - Rn)	Arithmetic
4	ADD	Arithmetic
5	ADC	Arithmetic
6	SBC (Subtract with borrow)	Arithmetic
7	RSC (Reverse sub with borrow)	Arithmetic
8	TST (Bitwise AND without destination register, update always Condition flags)	Logical
9	TEQ (Bitwise OR without destination register, update always Condition flags)	Logical
10	CMP (Subtraction without destination register, update always Condition flags)	Arithmetic
11	CMN (Addition without destination register, update always Condition flags)	Arithmetic
12	ORR (OR)	Logical
13	MOV	Logical
14	BIC (Bit Clear) instruction performs an AND operation on the bits in Rn with the complements of the corresponding bits in the value of Operand2.	Logical
15	MVN (Move register after a bitwise logical NOT)	Logical

(1): B is the second operand specified, i.e. constant value or Rm

Condition flag update details:

- Carry bit:
  - The arithmetic operations set the C bit through the ALU.
  - The logical operations set the C bit through the shifter only if a shift is performed.
- Overflow bit:
  - Only arithmetic operations can set the V bits
  - Logical operations preserve the V bit
- Negative and Zero bit:
  - Updated through the ALU for both operation type

1.1.4 Timing

Not register controlled shift:
Instructions bus: 1S
Data bus: 1I

Register controlled shift:
Instructions bus: 1S + 1I
Data bus: 2I

```
FDEMW
 FDDEMW   (register controlled shift)
  FSDEMW
```

- Fetch
- Decode
  - 1 cycle when no register controlled shift (read 1 immediate and 1 register or 2 registers in 1 clock cycle)
  - 2 cycle of decode to read 3 registers when a register controlled shift is needed. Even if the register file as 3 read ports only 2 of them are available in the decode since a read port is dedicated for the store operation. See store chapter for further details.
    - **1st cycle** read Rs
    - **2nd cycle** read Rn & Rm
- Execute
  - If the instruction has not to be executed -> disable writeback and disable the update of Condition flags
  - If bit S=1 update CPSR flags bits
  - If bit S=0 preserve CPSR flags bits

- Memory (Save value in WB pipeline register)
- Write-back

2 Data operation with register PC (R15) as Rd (register destination)

2.1.1 Opcode and description

Same as Data operation explained before, but being the program counter the destination register a (conditional) jump is performed.

If S=1 and Rd=R15, in an atomic way the jump is performed and also the SPSR is restored. This must be used only if the current mode has the SPSR, otherwise unpredictable results.

2.1.2 Timing

Not register controlled shift:
Instructions bus: 2S + 1N
Data bus: 3I

Register controlled shift:
Instructions bus: 2S + 1N + 1I
Data bus: 4I

- Fetch
- Decode
    - 1 cycle when no register controlled shift (read 1 immediate and 1 register or 2 registers in 1 clock cycle)
    - 2 cycle of decode to read 3 registers when a register controlled shift is needed. Even if the register file as 3 read ports only 2 of them are available in the decode since a read port is dedicated for the store operation. See store chapter for further details.
        - **1st cycle** read Rs
        - **2nd cycle** read Rn & Rm
- Execute
    - If the instruction has not to be executed disable write of the IA (instruction address) register from the ALU output. In this way the sequential flow of the program is preserved. *The write of the new PC is never performed directly into the register file but it is always updated indirectly by the IA register*
    - The computed NEXT_PC is directly sent to the IA register
    - Control Unit must perform the flush of the pipe of the next 2 instructions
- Memory
    - This instruction doesn't use the memory. In this stage the fetch of instruction located at NEXT_PC is performed
- "Write-back
    - Never performed

(1) Example: MOVS PC, R14

```
MOVS    FDEMW
         FDNNN
          FNNNN
           FDEMW
```

For more details about the CPSR/SPSR restoration in atomic way please refer to the chapter “CPSR/SPSR Management”.

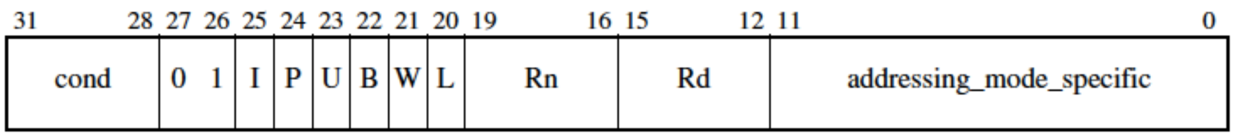
3 Data transfer operations

3.1 Single data transfer operations

3.1.1 Encoding

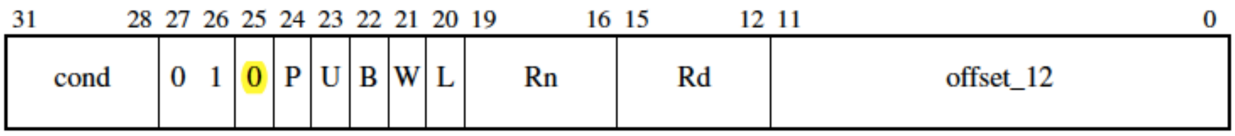
3.1.1.1 Single Word or unsigned byte data transfer opcode

```
LDR|STR{<cond>}{B}{T} Rd, <addressing_mode>
```



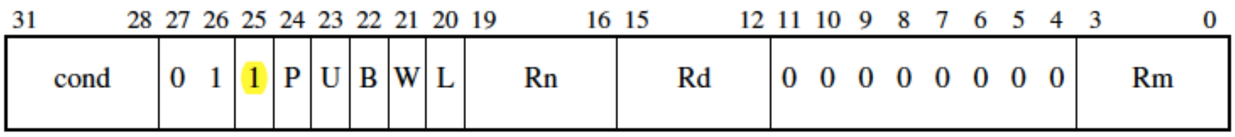
31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	B	W	L	Rn	Rd	addressing_mode_specific

Immediate offset/index



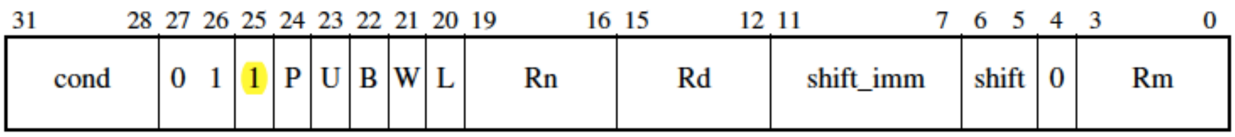
31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	0	P	U	B	W	L	Rn	Rd	offset_12

Register offset/index



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	0	1	1	P	U	B	W	L	Rn	Rd	0	0	0	0	0	0	0	0	Rm

Scaled register offset/index



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0	1	1	P	U	B	W	L	Rn	Rd	shift_imm	shift	0	Rm

- **Rn** Specifies the base register used by <addressing_mode>.
- **Rd** Specifies the register whose contents are to be loaded or stored.
- **Rm** Specifies the offset
- **I, P, U, W** Are bits that distinguish between different types of <addressing_mode>.
  - **I:** Specify addressing mode: immediate or register offset
  - **U:** Indicates whether the offset is added to the base (U == 1) or is subtracted from the base (U == 0). **The offset must be considered as unsigned**
  - **P, W:** A pre-indexed (**P = 1**) addressing mode uses the computed address for the load or store operation. If write-back is requested (**W = 1**), updates the base register to the computed value.
    A post-indexed (**P = 0**) addressing mode uses the unmodified base register for

the transfer and then updates the base register with the computed address independently from the W bit
- **L** bit Distinguishes between a Load (L==1) and a Store instruction (L==0).
- **B** bit Distinguishes between an unsigned byte (B==1) and a word (B==0) access.

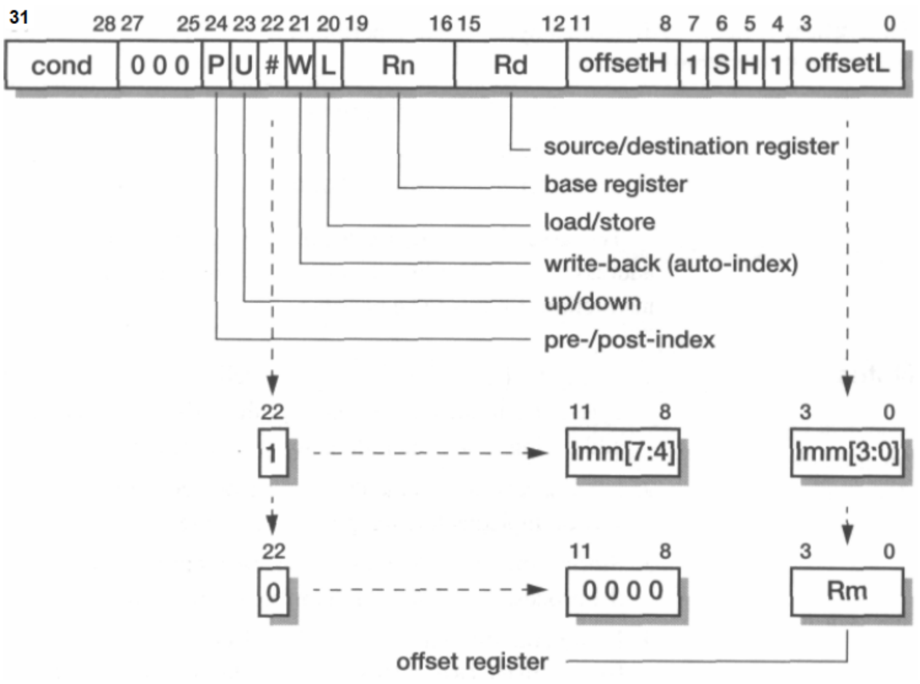
Notes:

- When a word is **read** and the **address is not aligned**, the access in memory is made with the aligned address. The alignment unit detects this event and generates the correct control signals for the Byte/Word Repl. Unit in order to rotate the read word to have the *addressed byte as the least significant byte*. The remaining data is 0-extended.
- When a **word write** is performed **on an unaligned address** the bottom 2 bits of the address are set to 0. So, the access is performed to the aligned word.
- When an unsigned **byte load** is performed the entire word is read from memory but only the addressed byte is read and the upper 24 bits are set to 0.
- When Rd is R15 a **jump** is performed to the word that has been read from memory. So the write-back in this case must not be performed since the value is directly sent to the IA. Since the end of the memory stage must be waited in order to know the jump address, a flush of the pipeline is needed after the EXE stage. Also additional stalls are needed, please refer to Load section for timing diagrams.

Architectural notes:

Independently from the Post/Pre-indexed mode if the W=1 (update the base address) this is always done using the ALU output. This is true for all other ISA instruction. For this reason, one write port of the register file is connected only to the output of the ALU properly skewed to respect the pipeline timing.

3.1.1.2 Half-Word or signed byte data transfer opcode



Similarly to before the offset must be considered as unsigned.
The offset can be specified in 2 different ways depending on Bit[22] value:

- Bit[22] = 1: an immediate offset Imm[11:7] & Imm[3:0] (8 bit immediate)
- Bit[22] = 0: a register (Rm)

SH bits have the following meaning:

- 00: Must be avoided
- 01: unsigned half word
- 10: signed byte
- 11: signed half word

Bit **S**: sign extension must be performed
Bit **H**: tell if it is a half word or not

This instruction requires the same control signal as word load/store with the addition of proper control signals for the byte/word repl. Unit.

3.1.2 Important notes

Storing the PC to memory both for Single and multiple load/store should be avoided since the result is processor dependent
Use PC as base register when W=1 (write-back) or P=0 (post-indexed) must be avoided since this cause unpredictable results (in our implantation no jump is performed)

3.1.3 Load timings and details

Note: **In general Rd, Rn should be distinct registers, even though loading into the base register (Rd = Rn) is acceptable provided auto-indexing is not used in the same instruction.** In our implementation the conflicts is managed by giving priority to the write port dedicated to the memory.

When the following instruction wrt the load has a data dependency with the current loaded value, 1 or 2 additional clock cycle are needed:

1. 1 clock cycle to perform the inter-lock. Notice how in the next clock cycle the forwarding path EXE/MEM -> DEC is used.

```
FDEMW
 FDSEMW
```

2. 2 clock cycles are needed when half-word or unaligned word is read. This is to avoid strict timing requirements for the memory system. In this way the data dependancy is solved by a 2 clock cycle inter-lock in a such a way the forwarding is no more needed because the RAW policy of register file used.

```
FDEMW
 FDSSEMW
```

If Rd=R15 in the exe stage the processor check if the instruction has to be performed or not. If this is true a flush of the pipeline is performed. Until write-back no fetch is performed because the value of the new PC comes from the memory.

```
FDEMW
 FDNNN
  FNNNN
   NNNNN
    NNNNN
     FDEMW
```

3.1.4 Pipeline stage details

3. Fetch
4. Decode
  4.1. Read Rn and Rm if needed
5. Execute
  5.1. Compute address
  5.2. If pre-index is specified, write in the EXE-MEM pipeline register the computed value, that will be used in the write-back stage on porta A if W=1
  5.3. If post index is specified, write in the EXE-MEM pipeline register the content of the base register obtained by the ALU bypass.
6. Memory
7. Write-back
  7.1. Write the read data from memory on write port B and eventually modified by the byte/rot sign. extension unit
  7.2. If W=1 (write-back) or P=0 (post index) also on port write A is updated the Rn register.

3.1.5 Store timing and details

Note: The read of the Rd register is always made during the MEM stage accessing directly to the register file from read port C. In this way an additional forwarding path and control logic is avoided since, even if the store instruction has a data dependency on previous instruction, the requested data is correctly read thanks to the RAW policy of the register file.
The Rd operand must never be extracted by a forwarding path but only by the register file.
Rn Rm may be dependent on previous instruction but it can be solved with a forwarding path.

- Fetch
- Decode
- Read Rn, and Rm if needed
- Execute
- Compute address
- If pre-index is specified, write in the EXE-MEM pipeline register the computed value, that will be used in the write-back stage on porta A if W=1
- If post index is specified, write in the EXE-MEM pipeline register the content of the base register obtained by the ALU bypass.
- Memory
- Only at this stage the Rd register is read and eventually the Byte/Word replication is performed

- Write-back
- The write port A is used if W=1 or P=0 (post index)
- The write port B port is unused

3.2 Multiple Load/Store

3.2.1 Important notes

- **Rn should not be on the register list when W=1 (write-back)**

3.2.2 Encoding

31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	P	U	S	W	L	Rn	register list

- **Rn**: base register
- **Register list**: bitmap of register to be loaded/store.
- **P**: P=1 pre-indexed, P=0 post indexed
- **W**: write-back of auto-indexing
- **U**: U=1 up (add offset), U=0 down (subtract offset)
- **L**: L=1 load, L=0 Store
- **S**: For a LDM that load the PC, the **S bit indicates that the CPSR is loaded from the SPSR after all the registers have been loaded**. ***For all STMs, and LDMs that do not load the PC and S=1, it indicates that when the processor is in a privileged mode, the registers of user mode are transferred and not the registers of the current mode.***
  Example: processor is not in a user mode and there is the need of a load/store on user mode register file. This is needed for example when returning from FIQ to NORMAL mode since to restore the previous state the register bank must be viewed like a normal mode and no more like in FIQ mode.
  Notice: this requires that register file must be readable from port C and writeable on port B in user mode, while the write port A must be in actual mode of the processor. In this way the write back of base register is done in the correct register of the actual mode of the processor.

3.2.3 Multiple register transfer addressing modes

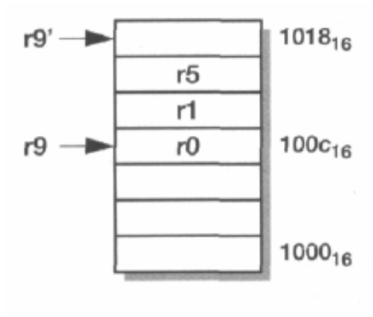
In a multiple load operation R15 (PC) can be on the register list. This cause a branch after all the requires registers are loaded from the memory. In order to simplify the controller, the registers are always read in the order R0→R15. This causes different start address and end address depending on the U and P bit as specified here:

- IA (Increment After) U=1 P=0
  - **Start address** = Reg[Rn]
  - **End address** = Reg[Rn] + (# registers << 2) - 4
  - If W=1 final Reg[Rn] ← **End address + 4** (Reg[Rn] **+** (# registers << 2))

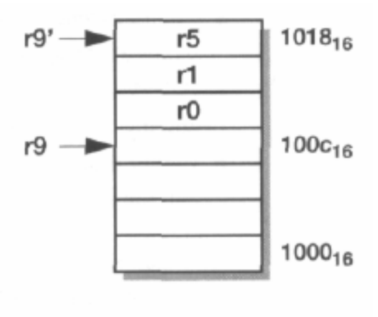
- IB (Increment Before) U=1 P=1
  - **Start address** = Reg[Rn] + 4
  - **End address** = Reg[Rn] + (# registers << 2)
  - If W=1 final Reg[Rn] ← **End address** (Reg[Rn] **+** (# registers << 2))

- DA (Decrement After) U=0 P=0
    - **Start address** = Reg[Rn] - (# registers << 2) + 4
    - **End address** = Reg[Rn]
    - If W=1 final Reg[Rn] ← **Start address – 4** (Reg[Rn] **-** (# registers << 2))

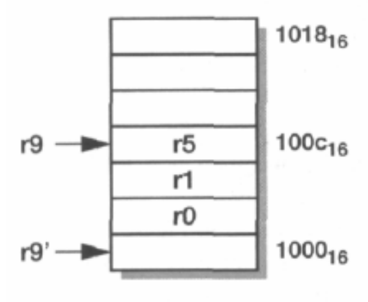
- DB (Decrement Before) U=0 P=1
    - **Start address** = Reg[Rn] - (# registers << 2)
    - **End address** = Reg[Rn] - 4
    - If W=1 final Reg[Rn] ← **Start address** (Reg[Rn] **-** (# registers << 2))



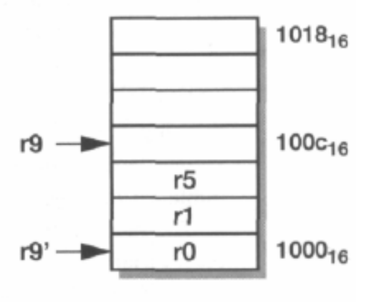
`STMIA r9!, {r0,r1,r5}`



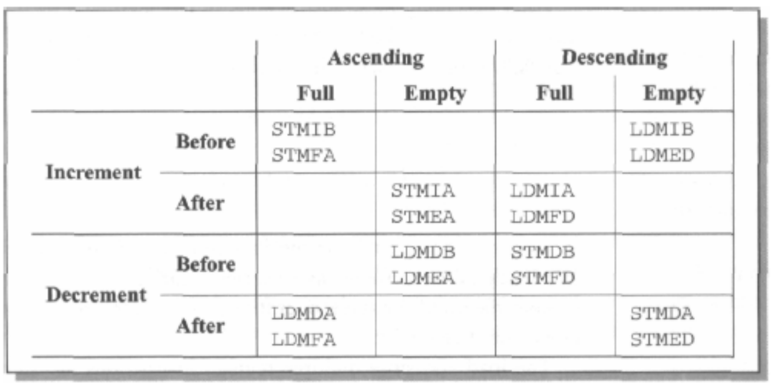
`STMIB r9!, {r0,r1,r5}`



`STMDA r9!, {r0,r1,r5}`



`STMDB r9!, {r0,r1,r5}`



		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

3.2.3.1 Additional notes
If a multiple load is performed, and R15 is on the registers list:

- If S=1
    - If the current mode has SPSR (Not in user mode)
      When performing the branch the CPSR also need to be overwritten with the value stored in SPSR of the actual mode
    - If the current mode has not the SPSR (In user mode) UNPREDICTABLE results
- If S=0
  A branch is performed and the CPSR is maintained

3.2.4 Timing
3.2.4.1 LDM: loading 1 register, not the PC
Instruction bus: 1S + 1I
Data bus: 1S + 1I

Instruction flow is equal to the N>1 case for the exception of the timing: here the additional clock is needed to perform the Rn update as shown here:

```
FDEMW        (LDM here exe compute start address)
   EMW       (Rn updated when W=1 or post-inedx)
 FDSEMW      (Next instruction wrt LDM)
```

3.2.4.2 LDM: Loading N registers, N > 1, not loading the PC
Instruction bus: 1S + (n-1)I
Data bus: 1N + (n-1)S

- Fetch
- Decode
    - Rn value on read port A
    - Generate *0/1/(#registers)/(#registers-1)*
    - compute first register to write with NextRegLogic
- Execute
    - Compute start address (add/sub)
    - IF S=1 (case in which PC is not listed and S=1 -> LDM with user-mode register)
      Starting from the next cycle until the last write-back force the register file to be in the user mode for all the read port C (for store) and write port B (for load).
    - Compute second register to write (physically performed in the DEC stage)
- Memory-0
    - Read from memory
    - DINC compute next memory address
    - Compute Reg[Rn] +/- (#registers << 2) (End address)
    - Compute third register to write with NextRegLogic (physically performed in the DEC stage)
- Memory-1
    - Read from memory with incremented address
    - Write-back of the read data during Memory-0 into correct register computed with NextRegLogic during the decode stage
    - Compute next register to write with NextRegLogic

        - DINC compute next address
        - Propagate ALU out to the writeback-stage in order to save during the next cycle Rn (Reg[Rn] +/- (#registers << 2))

    - Memory N-1 (**NB Exe stage is busy due to the next instruction wrt LDM [(1)]**)
        - Read from memory with incremented address
        - Write-back of the read data during Memory-(N-2) into correct register

    - Write-back
        - Write-back of the read data during Memory-(N-1) into correct register

```
(1) N = 3 example:
   012
FDEMMMW
   EMW         (Rn updated when W=1 or post-inedx)
 FDSSEMW       (Following instruction wrt LDM)

N = 2 example:
   01
FDEMMW
   EMW         (Rn updated when W=1 or post-inedx)
 FDSEMW
```

3.2.4.3 LDM: Loading N registers including the PC, N > 0

Instruction bus: 2S + 1N + (n+1)I
Data bus: 1N + (n-1)S + 4I

```
FDEMMMW
 FDSSNNN
  FSSNNNN
       FDEMW
        FDEMW
```

1. Fetch
2. Decode
    a. As before
3. Execute
    a. As before
4. Memory
    a. As before
5. Memory-(N-2)
    a. During this instant the NextRegLogic compute the R15 address. This is reported to the control unit for next stages in such a way the PC will not be loaded in R15 but directly to the Instruction Address register.
6. Memory-(N-1)
    a. During this stage the value of next program counter is read from the memory.
7. Write-back
    a. During this stage the Instruction Address register is written with the correct value, so the next clock cycle can perform the correct fetch

b. If S=1 and not in user mode:
   CPSR is updated with the current mode SPSR

3.3 Swap

3.3.1 Encoding

SWP (Word swap)

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	0	0	Rn	Rd	SBZ	1	0	0	1	Rm

SWPB (Byte swap)

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	0	0	Rn	Rd	SBZ	1	0	0	1	Rm

- **<Rd>** Specifies the destination register for the instruction.
- **<Rm>** Contains the value that is stored to memory.
- **<Rn>** Contains the memory address to load from.
- **SBZ** is a constant: 0000

3.3.2 Description

SWP (Swap) swaps a word between registers and memory. SWP loads a word from the memory address given by the value of register <Rn>. The value of register <Rm> is then stored to the memory address given by the value of <Rn>, and the original loaded value is written to register <Rd>. If the same register is specified for <Rd> and <Rm>, this instruction swaps the value of the register and the value at the memory address.
The LOCK signal for Data memory is set to 1 during the 2 memory phases in order to guarantee the atomicity of the instruction.

Pseudocode:

SWPB
```
temp = Memory[RF[Rn],4]
Memory[RF[Rn],4] = Rm
Rd = temp
```

SWPB
```
temp = Memory[RF[Rn],1]
Memory[RF[Rn],1] = Rm[7:0]
Rd = temp
```

Notes:
- If R15 is specified for <Rd>, <Rn>, or <Rm>, the result is UNPREDICTABLE.
- If the same register is specified as <Rn> and <Rm>, or <Rn> and <Rd>, the result is UNPREDICTABLE.

- When Rd=Rm and a store is performed sequentially after the load the just read value would be written in memory. This is not the intended behaviour of the Swap operation. In order to solve this problem a skewing register is added in order to preserve the correct value to be stored. See datapath "Delayed RC" register.

3.3.3 Timing

Timing for word SWP or SWPB when following instruction doesn't use the loaded byte
Instructions bus: 1S + 1I
Data bus: 2N

```
SWP     F   D   E   M_L M_S W
                        W_L
            F   D   S   E   M   W
```

Timing for SWPB when following instruction use the loaded byte
Instructions bus: 1S + 2I
Data bus: 2N + 1I

```
SWPB    F   D   E   M_L M_S W
                        W_L
            F   D   S   S   E   M   W
```

The red stall is needed to a slowdown of the exe stage due to the forwarding path through Byte Rot/Sign extension. The stall is in the decode stage and it's able to read the correct value due to the RAW policy.

Timing for SWPB when following instruction use the loaded word
Instructions bus: 1S + 1I
Data bus: 2N

```
SWPB    F   D   E   M_L M_S W
                        W_L
            F   D   S   E   M   W
```

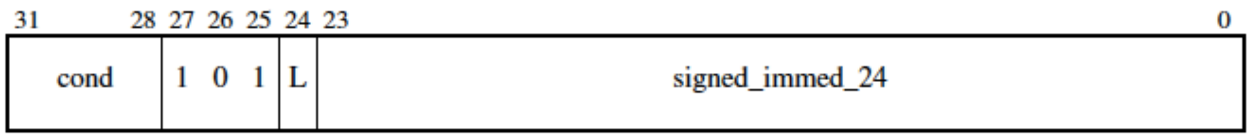
- Fetch
- Decode
  - Read Rn
  - Set immediate to 0
  - Set Rm as address for read port C (need to be propagated to Memory stage)
  - Set Rd as address for write port B (need to be propagated to Memory stage)
- Execute
  - Add operation (compute address)
- 1st Memory
  - Read from memory -> Rd
  - Read from port C register (Rm) destination and enable the skewed store register
- 2nd Memory
  - Write to memory the delayed stored register with the same address using same address.
  - Writeback of Rd

- Writeback
  - Nothing

4 Branch

4.1 Branch and Branch with link

4.1.1 Encoding



- **L:** L=1 Branch with link. L=0 Branch
  If the instruction is a BL, on R14 it will be saved PC-4 which is the following instruction wrt the current branch instruction. This is used to resume the program flow.
- **Immediate:** the **immediate is sign extended and shifted left of 2 bits** and then added to PC (+/- 32 MegaBytes)

4.1.2 Timing

Branch instruction flow:

- Fetch
- Decode
  - Read the program counter (NB PC+8 where PC is the address of the actual branch instruction)
- Execute
  - Check if the instruction must be executed
    - If it has not to be executed transform it into a NOP
    - If it must be executed
      - Compute the branch address (PC + (imm << 2))
      - **Flush the pipe without loading the decode/exe pipeline** register in order to keep the PC of the branch in the exe stage. This is needed to compute the PC correction for the link phase
- Memory
- Write-back

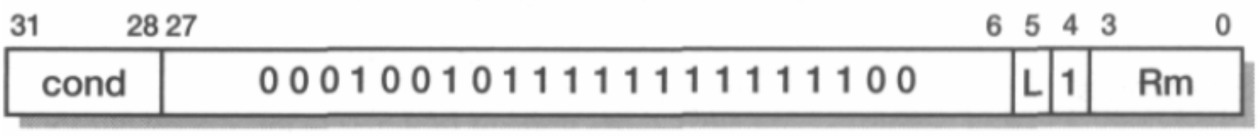
Flow of the instruction below the branch:

- Fetch
  - normal
- Decode
  - normal
- Execute
  - At this point the processor know that this instruction must not be performed
  - **NB only for BL:** the exe stage of this instruction, that must not be executed, is transformed into a PC-4 computation in order to compute the PC correction for the link phase. This operation is done contemporary with the memory stage of the branch.
- Memory
- Write-back
  - **NB only for BL**: Write the computed PC-4 into R14 if the instruction is a BL

```
BL    FDEMW
       FDEMW    (PC+8)-4 computation
        FNNNN   (NOP)
         FDEMW  (correct instruction to be fetched)
```

4.2 BX

4.2.1 Encoding and description



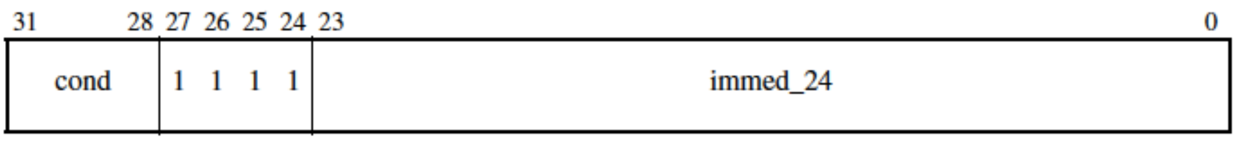
- **NB in ARMv4T L is always equal to 0**
- This operation performs a branch into the address saved into Rm. The Rm[0] bit is used to set the Thumb flag of CPSR. (Last bit is “useless” since fetch is always aligned on 4 byte address for non-thumb and 2 bytes for thumb)
- As reported in the following pipeline diagram the 2 instructions after the BX must be transformed into a NOP. The fetch of the correct instruction can be performed only after the BX exe stage (need to verify that the BX must be executed using CPSR bits). In order to fetch correctly the new instruction, the CPSR T bits must be already set accordingly during the fetch phase, but in the same clock cycle the instruction previous to the BX is performing the Write-back stage. Since Thumb mode and non-Thumb mode have different view of the register file these two stages must be performed concurrently with the T bit with different value. To solve this problem a register “Next CPSR” is used to anticipate the Fetch with the correct mode, while the actual CPSR still use the previous T bit. (see datapath)

```
FDEMW       (previous instruction)
 FDEMW      (BX)
  FDNNN
   FNNNN
    FDEMW   (Correctly Thumb/Non-Thumb instruction to be fetched)
```

5 Miscellaneous instructions

5.1 SWI

5.1.1 Encoding and description



- **Immediate:** Is a 24-bit immediate value that is put into bits[23:0] of the instruction. This value is ignored by the ARM processor, but can be used by an operating system SWI exception handler to determine what operating system service is being requested
- SWI is used as an operating system service call. The method used to select which operating system service is required is specified by the operating system, and the SWI exception handler

for the operating system determines and provides the requested service. The 24-bit immediate in the instruction specifies which service is required, and any parameters needed by the selected service are passed in general-purpose registers.

- This instruction apparently doesn't use the 24 bit immediate. Usually an OS when enters the SWI handlers read from the Instruction memory the SWI instruction located at R14_svc-4 to understand which is the required system call.

```
R14_svc = address of next instruction after the SWI instruction
SPSR_svc = CPSR
CPSR[4:0] = 0b10011 /* Enter Supervisor mode */
CPSR[5] = 0 /* Execute in ARM state */
CPSR[7] = 1 /* Disable normal interrupts */
CPSR[9] = CP15_reg1_EEbit
PC = 0x00000008
```

5.1.2 Timing

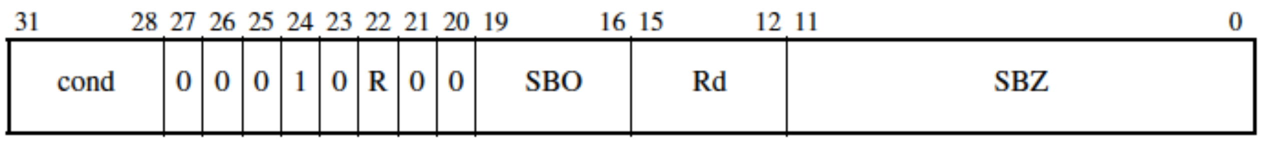
Instructions bus: 2S + 1N
Data bus: 3I

```
PREVIOUS  FDEMW
SWI        FDEMW
            FDNNN
             FNNNN
SWI (B HANDL) FDEMW
HANDLER          FDEMW
```

- Fetch
- Decode
  - Read program counter (for link)
  - Set immediate to 4
- Execute
  - Compute PC-4
  - Set next Instruction address register to the vector_table_out (`0x00000008 or 0xFFFF0008`)
  - Set Next CPSR in order to set supervisor mode for the next fetch
- Memory
  - At the end of the memory CPSR is written using the Next CPSR. In this way in the next clock cycle the Fetch has already the correct CPSR
  - At the end of the memory CPSR is copied into temp CPSR in order to be able to write it in the SPSR_svc in the next clock cycle (i.e. during the writeback).
- Writeback
  - Set R14 to the computed PC-4
  - SPSR_svc = temp CPSR (the CPSR before the update, see CPSR/SPSR management chapter)

5.2 MRS

5.2.1 Encoding and description



MRS (Move PSR to general-purpose register) moves the value of the CPSR or the SPSR of the current mode into a general-purpose register. In the general-purpose register, the value can be examined or manipulated with normal data-processing instructions.

- **<Rd>** Specifies the destination register. If R15 is specified for <Rd>, the result is UNPREDICTABLE.
- **R** if R=1 SPSR , R=0 CPSR
- **SBZ** = all 0s
- **SBO** = all1s

Code:

```
if R == 1 then
Rd = SPSR
else
Rd = CPSR
```

Notes: Accessing the SPSR when in User mode or System mode is UNPREDICTABLE

5.2.2 Timing

Instructions bus: 1S
Data bus: 1T

- Fetch
- Decode
  - Generate correct mux selector to read CPSR/SPSR
- Execute
  - Or 0 (can use 12 bit of immediate set to 0)
- Memory
  - Nothing
- Writeback
  - Writeback <Rd> with SPSR/CPSR

5.3 MSR

5.3.1 Encoding and description

Immediate operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 0
cond	0 0 1 1 0	R	1 0	field_mask	SBO	rotate_imm	8_bit_immediate

Register operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0	R	1 0	field_mask	SBO	SBZ	0 0 0 0	Rm

MSR (Move to Status Register from Register) transfers the value of a general-purpose register or an immediate constant to the CPSR or the SPSR of the current mode.

- **<fields>** Is a sequence of one or more of the following:
  - **C** sets the control field mask bit (**bit 16**)
  - **X** UNUSED (sets the extension field mask bit (**bit 17**))
  - **S** UNUSED (sets the status field mask bit (**bit 18**))
  - **F** sets the flags field mask bit (**bit 19**).

  These bits are used to set independently upper/lower 8 bits.
- **<immediate>** Is the immediate value to be transferred to the CPSR or SPSR. Allowed immediate values are 8-bit immediates (in the range 0x00 to 0xFF) and values that can be obtained by rotating them right by an even amount in the range 0 to 30.
- **<Rm>** Is the general-purpose register to be transferred to the CPSR or SPSR.
- **<R>** Bit[22] of the instruction is 0 if the CPSR is to be written and 1 if the SPSR is to be written

5.3.2 Timing

If any bits other than just the flags are updated (all masks other than mask_f)
Instructions bus: 1S + 2I
Data bus: 3I

```
MSR     FDEMW
         FDSSEMW
          FSSDEMW
```

In red is reported when the CPSR of the processor is updated with its new value.

If only the flags are updated, so this is the only operation allowed in user mode
Instructions bus: 1S
Data bus: 1I

```
MSR     FDEMW
         FDEMW
          FDEMW
```

Notice: the flag bits of the CPSR are updated immediately after the EXE stage in this way the next instruction will be executed based on the new flags.

6 Multiplication

6.1 Opcodes and description

- **<S>** Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication.
  If S is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
- **<Rd>** Specifies the destination register for the instruction.
- **<Rm>** Specifies the register that contains the first value to be multiplied.
- **<Rs>** Holds the value to be multiplied with the value of <Rm>.
- **<RdLo>** Stores the lower 32 bits of the result.
- **<RdHi>** Stores the upper 32 bits of the result.

MUL can optionally update the condition code flags, based on the result

6.1.1 MUL - Multiply

MUL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	0	0	0	S	Rd	SBZ	Rs	1	0	0	1	Rm

```
Rd = (Rm * Rs)[31:0]
```

6.1.2 MLA – Multiply and accumulate

MLA

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	0	0	1	S	Rd	Rn	Rs	1	0	0	1	Rm

```
Rd = (Rm * Rs + Rn)[31:0]
```

6.1.3 UMULL - Unsigned Multiply Long

UMULL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	1	0	0	S	RdHi	RdLo	Rs	1	0	0	1	Rm

UMULL multiplies the unsigned value of register <Rm> with the unsigned value of register <Rs> to produce a 64-bit result.

```
RdHi = (Rm * Rs)[63:32] /* Unsigned multiplication */
RdLo = (Rm * Rs)[31:0]
```

6.1.4 UMLAL - Unsigned Multiply Accumulate Long

UMLAL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	1	0	1	S	RdHi	RdLo	Rs	1	0	0	1	Rm

UMLAL multiplies the unsigned value of register <Rm> with the unsigned value of register <Rs> to produce a 64-bit product. This product is added to the 64-bit value held in <RdHi> and <RdLo>, and the sum is written back to <RdHi> and <RdLo>. The condition code flags are optionally updated, based on the result.

```
RdLo = (Rm * Rs)[31:0] + RdLo /* Unsigned multiplication */
RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
```

6.1.5 SMULL - Signed Multiply Long

SMULL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	1	1	0	S	RdHi	RdLo	Rs	1	0	0	1	Rm

SMULL multiplies two 32-bit signed values to produce a 64-bit result.

```
RdHi = (Rm * Rs)[63:32] /* Signed multiplication */
RdLo = (Rm * Rs)[31:0]
```

6.1.6 SMLAL - Signed Multiply Accumulate Long

SMLAL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	1	1	1	S	RdHi	RdLo	Rs	1	0	0	1	Rm

```
RdLo = (Rm * Rs)[31:0] + RdLo /* Signed multiplication */
RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
```

6.2 Timing

MUL, MLA:

    Instructions bus: 1S + (1+m)I
    Data bus: (2 + m)I

An additional clock is need to perform final addition (ADD and ADC)
SMULL, UMULL, SMLAL, UMLAL

    Instructions 1S + (2 + m)I
    Data bus: (3+m)I

Where m is in the range [1,4] depending on early termination, i.e. stop multiplication if no more useful operations will be performed:

For MUL, MLA, SMULL, SMLAL, and Thumb MUL, m is:

- 1 if bits [31:8] of the multiplier operand are all zero or one
- 2 if bits [31:16] of the multiplier operand are all zero or one
- 3 if bits [31:24] of the multiplier operand are all zero or all one
- 4 otherwise.

For UMULL, UMLAL, m is:

- 1 if bits [31:8] of the multiplier operand are all zero
- 2 if bits [31:16] of the multiplier operand are all zero
- 3 if bits [31:24] of the multiplier operand are all zero
- 4 otherwise.

Example with m=4

```
PREV     FDEMW
MUL       FDDEEEEMW
NEXT       FSDSSSEMW
```

- Fetch
- Decode 1st
  - The Rn register if is a MAC operation is read. Otherwise porta A is set to 0. The multiplier stores it internally
- Decode 2nd
  - Rm and Rs registers are read and stored internally into the multiplier
- Execute
  - This stage may last from 2 to 5 cycle.
- Memory
- Writebak

7 Exceptions

Exceptions priority:

1. reset (highest priority);
2. Data abort;
3. FIQ;
4. IRQ;
5. Prefetch abort;
6. SWI;

Exception entry:

- It changes to the operating mode corresponding to the particular exception.
- It saves the address of the instruction following the exception entry instruction in r14 of the new mode.
- It saves the old value of the CPSR in the SPSR of the new mode.

- It disables IRQs by setting bit 7 of the CPSR and, if the exception is a fast interrupt, disables further fast interrupts by setting bit 6 of the CPSR.
- It forces the PC to begin executing at the vector address related to the specific exception.

Exception return:

- Any modified user registers must be restored from the handler's stack.
- The CPSR must be restored from the appropriate SPSR.
- The PC must be changed back to the following instruction address in the user instruction stream.

Note that the last two of these steps cannot be carried out independently since the have different view of register file. ARM provides two mechanisms which cause both steps to happen atomically as part of a single instruction

• To return from a SWI or undefined instruction trap use:
    MOVS pc, r14
• To return from an IRQ, FIQ or prefetch abort use:
    SUBS pc, r14, #4
• To return from a data abort to retry the data access use:
    SUBS pc, r14, #8

These PC correction are explained in the following sub-sections

7.1 SWI, UNDEFINED

To return from **a SWI, UNDEFINED** (return to next instruction)
    MOVS PC, R14

Example UNDEFINED:

```
PC    (R15=PC+8)   FDEMW     (Undefined operation)
PC+4                FDNNN
                      FNNNN
                       FDEMW (Fetch with address 0x4)
```

During the decode the control unit detect an undefined instruction, so perform a PC correction in order to save into R14 = R15+8-4 = PC+4 (Same operation done in a SWI).

7.2 IRQ, FIQ, PREFETCH

To return from **IRQ, FIQ, PREFETCH ABORT** (return to usurped instruction)
    MOVS PC, R14, **#4**

IRQ example:

```
PC     (R15=PC+8)             FDEMW
PC+4   (R15=PC+12)             FNNNN
PC+8   (R15=PC+16)              NNNNN
0x18 = B IRQ_handler             FDEMW
                                  FDNNN
```

```
                                     FNNNN
1st  instruction IRQ_handler          FDEMW
```

During the decode the control unit detect an IRQ instruction and read R15 = PC+8. In the exe R15-4 is performed, so R14=PC+4, then after in the IRQ handler R14 is used after a correction of -4 PC= (R14-4=PC+4-4=PC) in order to execute the interrupted instruction.

FIQ example:

```
PC     (R15=PC+8)              FDEMW
PC+4   (R15=PC+12)              FNNNN
PC+8   (R15=PC+16)               NNNNN
0x1C   1 inst. FIQ_handler        FDEMW
```

7.3 DATA ABORT

To return from a DATA ABORT (to retry data access)

MOVS PC, R14, **#8**

```
PC     (R15=PC+8)              FDEMW
PC+4   (R15=PC+12)              FDEMW
PC+8   (R15=PC+16)               FDEMW
PC+12  (R15=PC+20)                FDEMW
```

The data abort is generated by the memory during the Memory stage. This is detected during the decode stage of the PC+12 instruction. In order to return from a data abort handler and retry the instruction which cause the data abort, in R14 must be saved PC+8 (PC+8-8=PC).
Due to the fact that this is detected when R15 = PC+20 in the execution stage a correction is need to be made. R14=R15-12=PC+8.
The following two instructions after the one that cause data abort must be flushed.

7.4 Exception vector table

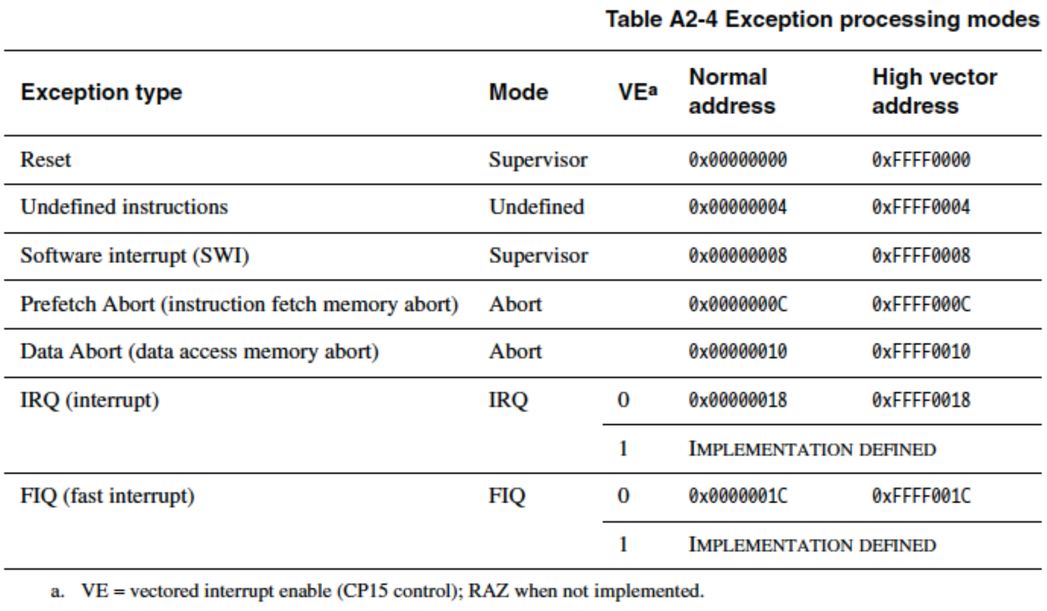
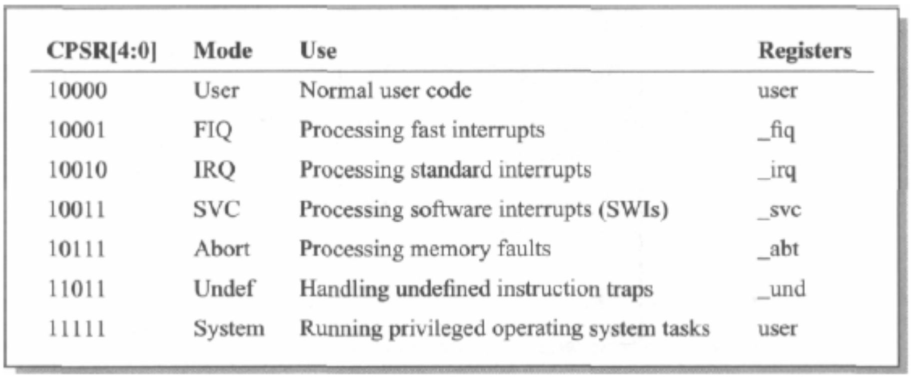


Table A2-4 Exception processing modes

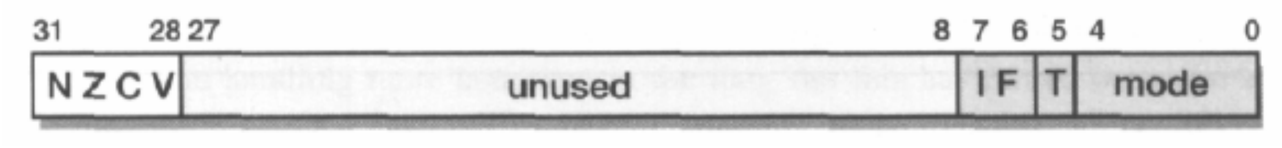
Exception type	Mode	VE[a]	Normal address	High vector address
Reset	Supervisor		0x00000000	0xFFFF0000
Undefined instructions	Undefined		0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor		0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort		0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort		0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0	0x00000018	0xFFFF0018
		1	IMPLEMENTATION DEFINED	
FIQ (fast interrupt)	FIQ	0	0x0000001C	0xFFFF001C
		1	IMPLEMENTATION DEFINED	

a. VE = vectored interrupt enable (CP15 control); RAZ when not implemented.



CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

8 CPSR/SPSR Management

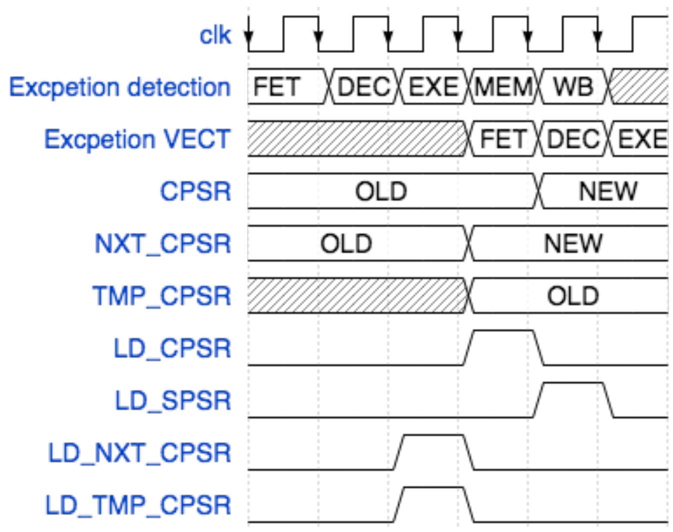


Which are the instructions that interacts with the CPSR/SPSR?

- **Data processing operations**
  - If Rd != R15
    - If S = ‘1’ (update flags)
      CFlags (NCVZ) are updated using ALU/MUL/SHIFTER flags
    - If S = ‘0’ CPSR/SPSR are unchanged
  - If Rd = R15 (IA ⇐ Alu result)
    - If S=’1’
      CPSR ⇐ SPSR of current mode (<u>jump and CPSR update must be done in **atomic** way</u>) Here a NextCPSR register is needed to Fetch correctly the next instruction is needed
- **LDM**
  - S=’1’ and R15 is in the register list
    CPSR ⇐ SPSR of current mode
- **BX**
  - Can set the Thumb bit of CPSR
- **SWI**
  - SPSR_svc = CPSR (save CPSR to restore it)
  - Next_CPSR is modified to fetch like Supervisor (Mode, Thumb and Interrupt bits)
- **MSR**
  - Can edit CPSR by 8 bits groups using ALU output as source
- **Exceptions**: IRQ, FIQ, PREFETCH ABORT, DATA ABORT, UNDEFINED
  - SPSR_current_mode = CPSR (Like SWI)
  - Next_CPSR = Desired mode + IRQ/FIQ flag masked (example while entering in FIQ F is set to 1 to mask other fast interrupts)

In order to correctly enter an exception the following timing diagram must be respected:

- Exception detection is the instruction that detect the exception (or it is a SWI/UNDEFINED)
- Exception vect is the instruction placed at the correct position of the Exception vector address
- CPSR is the value of the CPSR (OLD = previous exception, NEW = CPSR with correct lower 8 bits set)
- NXT_CPSR is the CPSR needed to fetch exception VECT with correct mode and also to maintain in CPSR the old one so the WB stage is still performed on the right register file
- TMP_CPSR is needed by the exception handler to store in the SPSR the old value of the CPSR

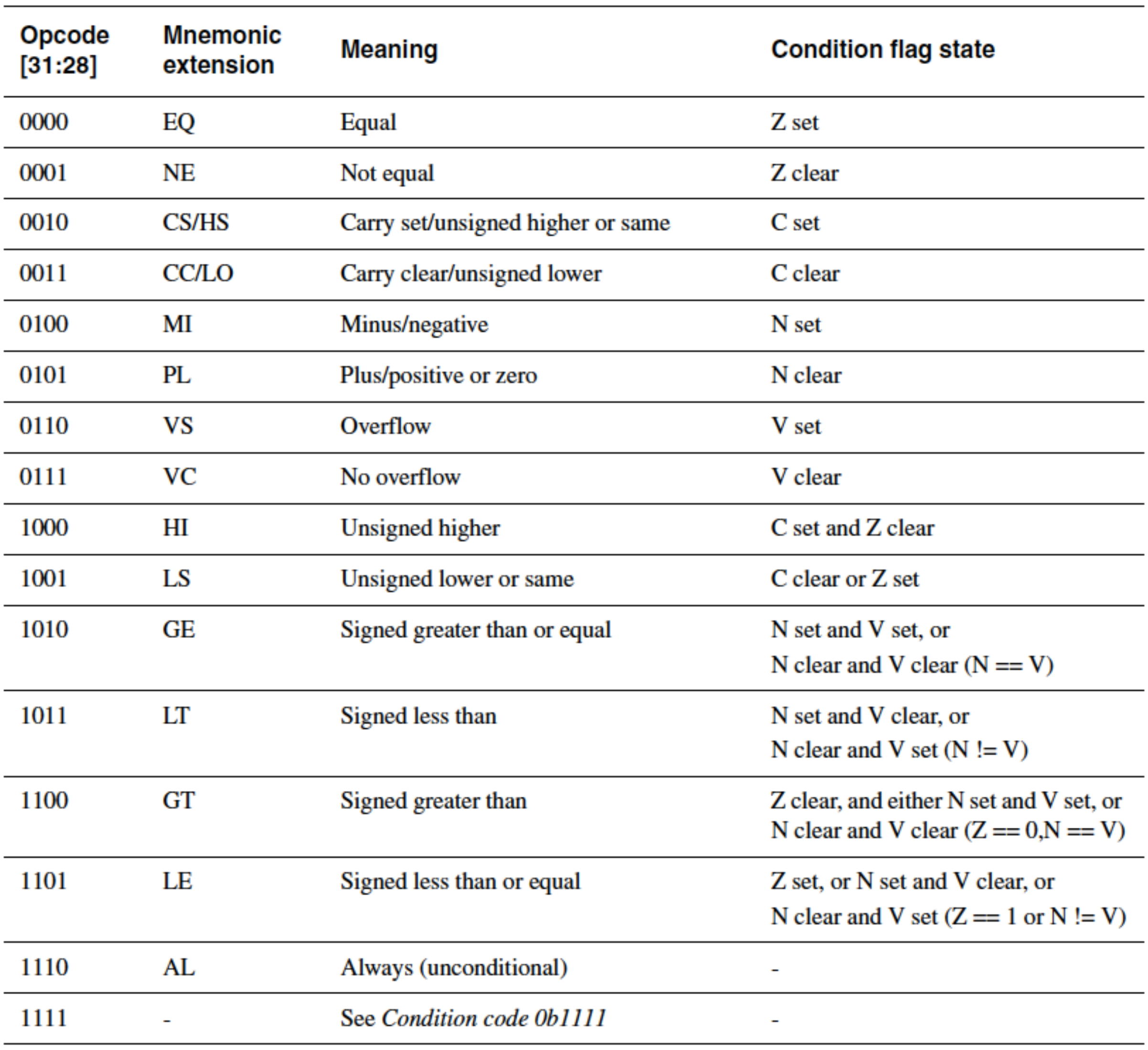


After these considerations CPSR bits can be set by:

- **N**: by ALU
- **Z**: by ALU or MUL
- **C**: by ALU or SHIFTER
- **V**: by ALU
- **I**: by exception
- **F**: by exception
- **T**: by BX
- **Mode**: by exception
- All these bits by **MSR** (Alu output)

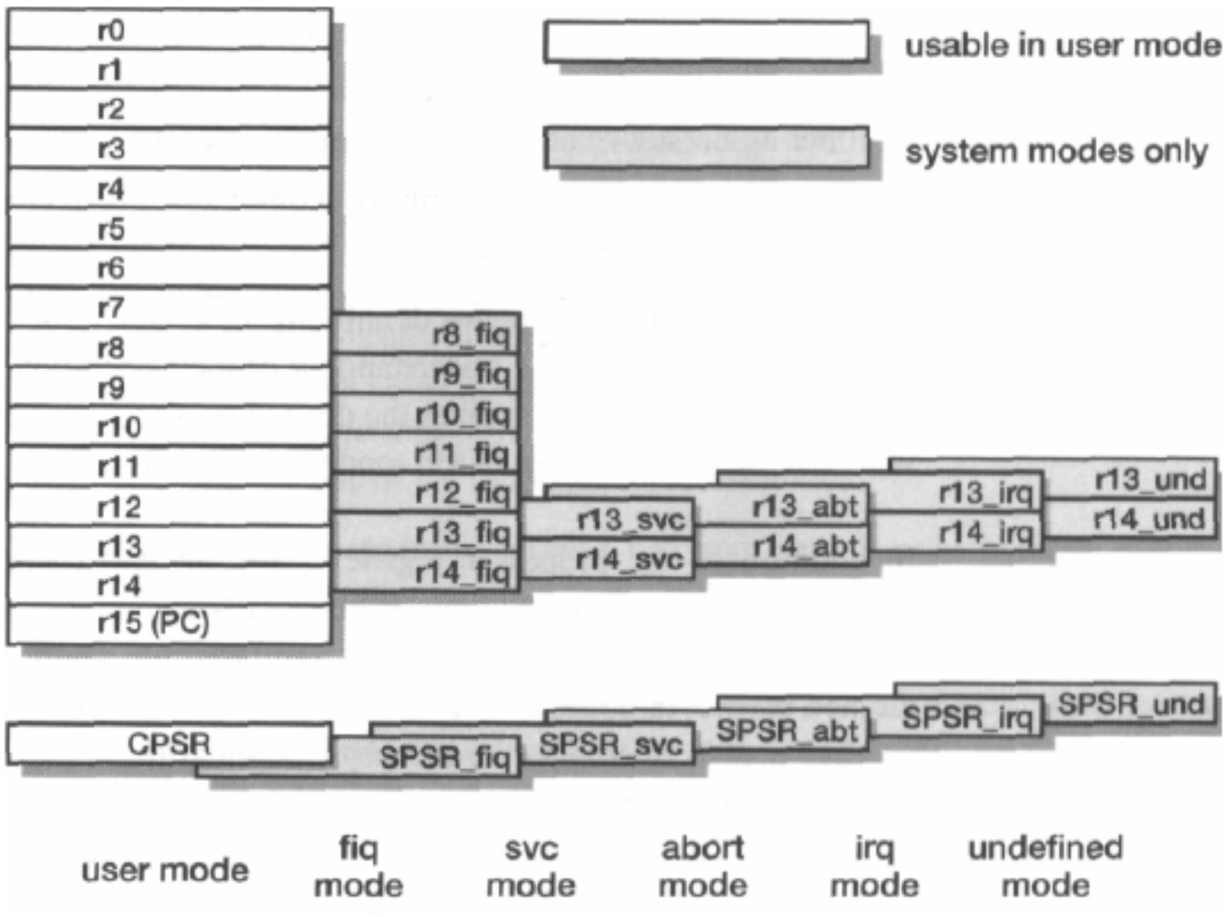
9 Appendix

9.1 Conditional codes



Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See *Condition code 0b1111*	-

9.2 Register file view


r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15 (PC)
usable in user mode
system modes only
r8_fiq
r9_fiq
r10_fiq
r11_fiq
r12_fiq
r13_fiq
r14_fiq
r13_svc
r14_svc
r13_abt
r14_abt
r13_irq
r14_irq
r13_und
r14_und
CPSR
SPSR_fiq
SPSR_svc
SPSR_abt
SPSR_irq
SPSR_und
user mode
fiq mode
svc mode
abort mode
irq mode
undefined mode