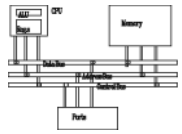
Data transfer instructions

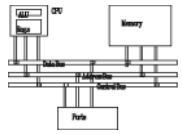


- Move data between registers and memory
- Three basic forms
  - Single register load/store
  - Multiple register load/store
  - Single register swap: **`SWP(B),`** atomic

Cond	01	I	P|U|S|W	L	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

- L – load/store
- Rn – base register
- P – pre/post-indexing
- U – up/down (relative to base register)
- W? S? - find out

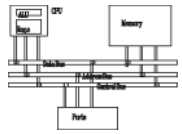
Single register load/store



Syntax: <LDR|STR>{<cond>}{B} Rd,addressing[1]
LDR{<cond>}SB|H|SH Rd, addressing[2]
STR{<cond>}H Rd, addressing[2]

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]*

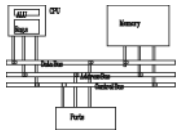
Single register load/store



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])*

No **STRSB/STRSH** since **STRB/STRH** stores both signed/unsigned ones

Single register load/store



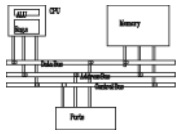
- The data items can be a 8-bit byte, 16-bit half-word or 32-bit word. Addresses must be boundary aligned. (e.g. 4's multiple for **LDR/STR**)

```
LDR  R0, [R1]   @ R0 := mem32[R1]
STR  R0, [R1]   @ mem32[R1] := R0
```

LDR, LDRH, LDRB for 32, 16, 8 bits

STR, STRH, STRB for 32, 16, 8 bits

Addressing modes



- Memory is addressed by a register and an offset.

```
LDR  R0, [R1]  @ mem[R1]
```

- Three ways to specify offsets:
  - Immediate

```
LDR  R0, [R1, #4]   @ mem[R1+4]
```

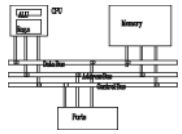
  - Register

```
LDR  R0, [R1, R2]   @ mem[R1+R2]
```

  - Scaled register

```
                    @ mem[R1+4*R2]
LDR  R0, [R1, R2, LSL #2]
```

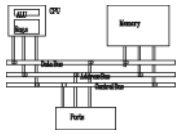
Addressing modes



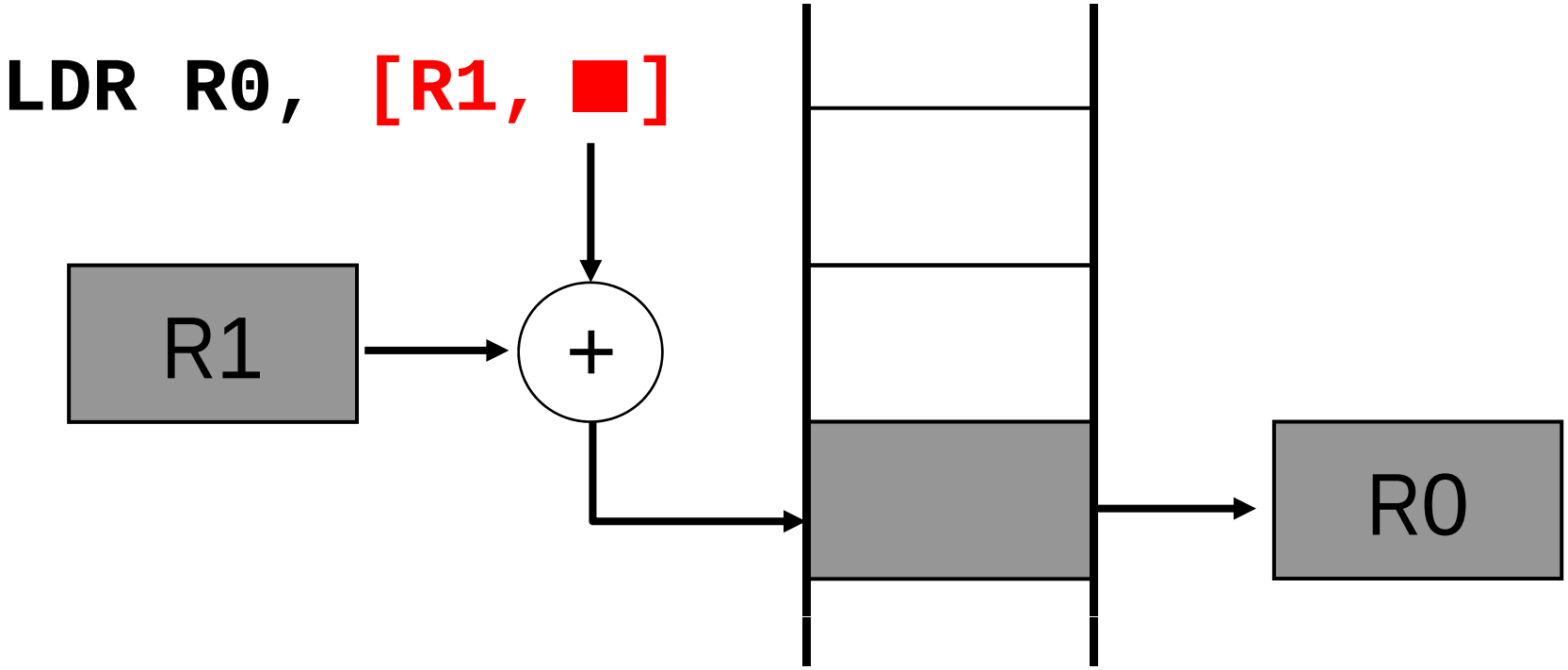
- Pre-index addressing (**`LDR R0, [R1, #4]`**)
  without a writeback
- Auto-indexing addressing (**`LDR R0, [R1, #4]!`**)
  Pre-index with writeback
  calculation before accessing with a writeback
- Post-index addressing (**`LDR R0, [R1], #4`**)
  calculation after accessing with a writeback

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

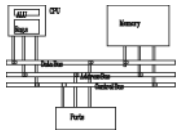
Pre-index addressing



```
LDR  R0, [R1, #4]    @ R0=mem[R1+4]
                     @ R1 unchanged
```



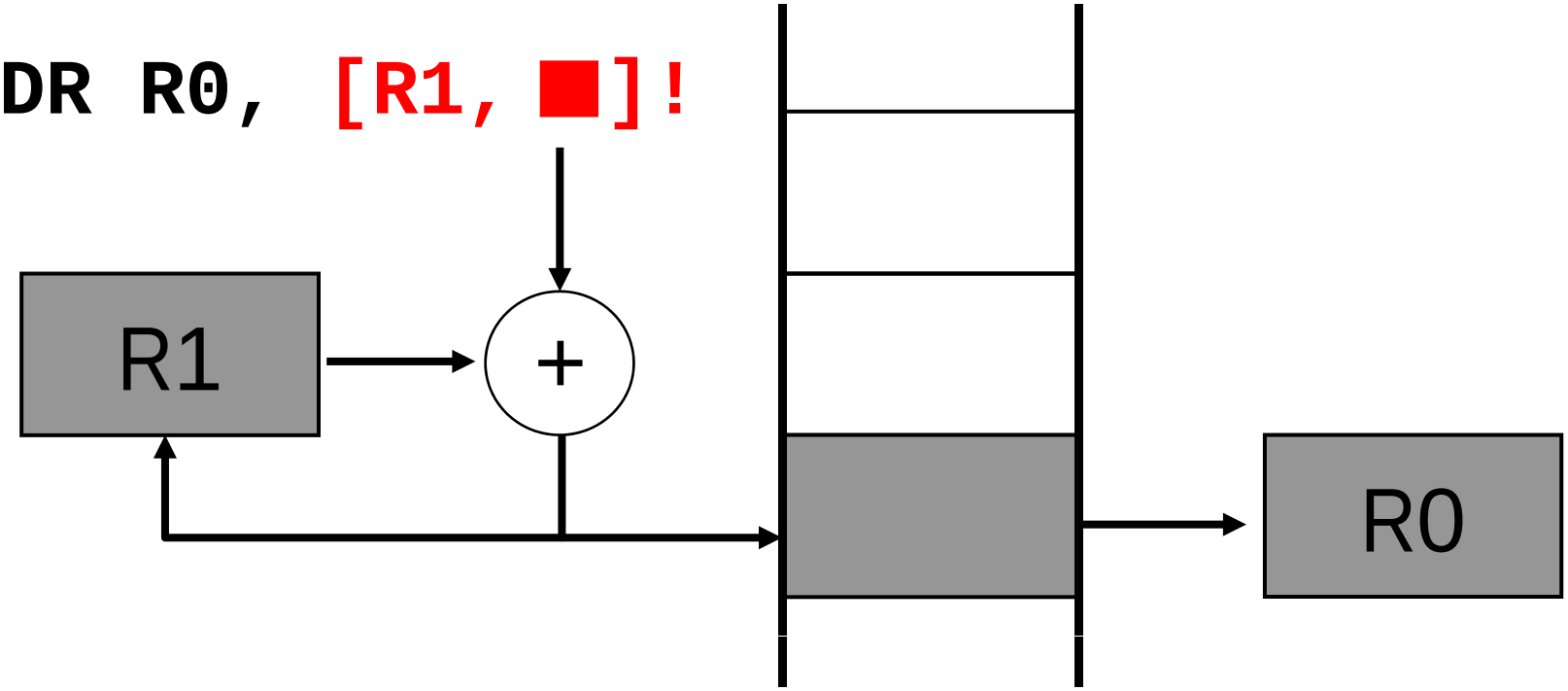
Auto-indexing addressing



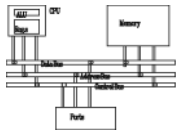
```
LDR  R0, [R1, #4]!   @ R0=mem[R1+4]
                     @ R1=R1+4
```

No extra time; Fast;

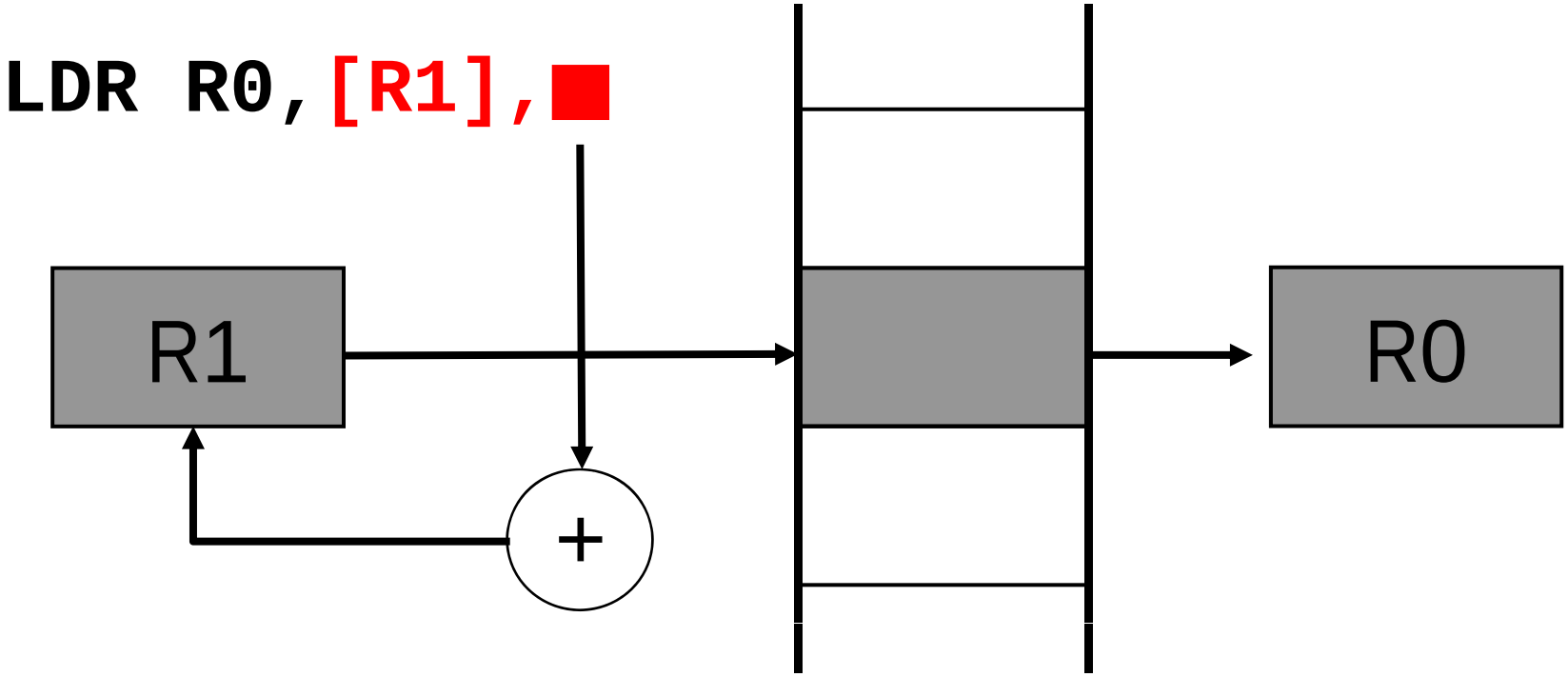
LDR R0, [R1, ■]!



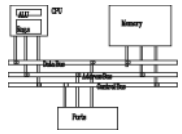
Post-index addressing



```
LDR   R0, R1, #4        @ R0=mem[R1]
                        @ R1=R1+4
```



Comparisons



- Pre-indexed addressing

```
LDR  R0, [R1, R2]  @ R0=mem[R1+R2]
                   @ R1 unchanged
```

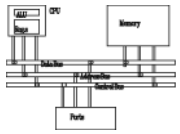
- Auto-indexing addressing

```
LDR  R0, [R1, R2]! @ R0=mem[R1+R2]
                   @ R1=R1+R2
```

- Post-indexed addressing

```
LDR  R0, [R1], R2  @ R0=mem[R1]
                   @ R1=R1+R2
```

Example

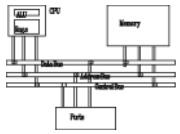


```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR      r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1)   r0 = 0x02020202
          r1 = 0x00009004
```

Example

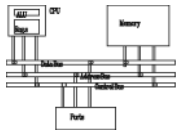


```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR     r0, [r1, #4]
```

Preindexing:

```
POST(2)   r0 = 0x02020202
          r1 = 0x00009000
```

Example

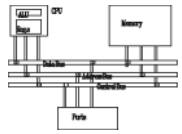


```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR     r0, [r1], #4
```

Postindexing:

```
POST(3)   r0 = 0x01010101
          r1 = 0x00009004
```

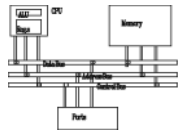
Summary of addressing modes



```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing[1]
        LDR{<cond>}SB|H|SH Rd, addressing[2]
        STR{<cond>}H Rd, addressing[2]
```

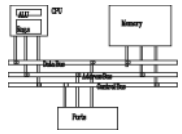
Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback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

Summary of addressing modes



	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

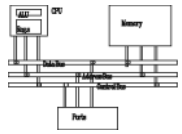
Summary of addressing modes



```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing[1]
        LDR{<cond>}SB|H|SH Rd, addressing[2]
        STR{<cond>}H Rd, addressing[2]
```

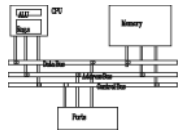
Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register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

Summary of addressing modes



	Instruction	Result	*r1* +=
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Swap instruction

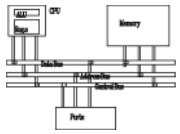


- Swap between memory and register. Atomic operation preventing any other instruction from reading/writing to that location until it completes

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

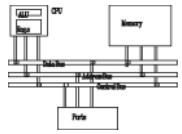
SWP	swap a word between memory and a register	$tmp = mem32[Rn]$ $mem32[Rn] = Rm$ $Rd = tmp$
SWPB	swap a byte between memory and a register	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

Example



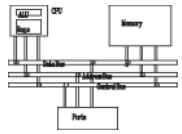
```
PRE     mem32[0x9000] = 0x12345678
        r0 = 0x00000000
        r1 = 0x11112222
        r2 = 0x00009000

        SWP    r0, r1, [r2]

POST    mem32[0x9000] = 0x11112222
        r0 = 0x12345678
        r1 = 0x11112222
        r2 = 0x00009000
```

Application



```
spin
        MOV     r1, =semaphore
        MOV     r2, #1
        SWP     r3, r2, [r1] ; hold the bus until complete
        CMP     r3, #1
        BEQ     spin
```

Software interrupt

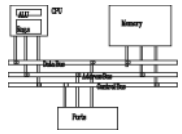


- A software interrupt instruction causes a software interrupt exception, which provides a mechanism for applications to call OS routines.

```
Syntax: SWI{<cond>} SWI_number
```

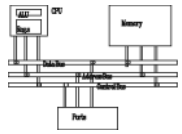
SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

Example



```
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
```

Load constants

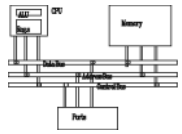


- No ARM instruction loads a 32-bit constant into a register because ARM instructions are 32-bit long. There is a pseudo code for this.

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

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

Load constants



- Assemblers implement this usually with two options depending on the number you try to load.

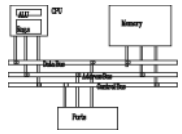
Pseudoinstruction	Actual instruction
LDR r0, =0xff	MOV r0, #0xff
LDR r0, =0x55555555	LDR r0, [pc, #offset_12]

Loading the constant 0xff00ffff

```
        LDR     r0, [pc, #constant_number-8-{PC}]
        :
constant_number
        DCD     0xff00ffff
```

```
MVN     r0, #0x00ff0000
```

Instruction set



Operation Mnemonic	Meaning	Operation Mnemonic	Meaning
`ADC`	Add with Carry	`MVN`	Logical NOT
`ADD`	Add	`ORR`	Logical OR
`AND`	Logical AND	`RSB`	Reverse Subtract
`BAL`	Unconditional Branch	`RSC`	Reverse Subtract with Carry
`B`$\langle cc \rangle$	Branch on Condition	`SBC`	Subtract with Carry
`BIC`	Bit Clear	`SMLAL`	Mult Accum Signed Long
`BLAL`	Unconditional Branch and Link	`SMULL`	Multiply Signed Long
`BL`$\langle cc \rangle$	Conditional Branch and Link	`STM`	Store Multiple
`CMP`	Compare	`STR`	Store Register (Word)
`EOR`	Exclusive OR	`STRB`	Store Register (Byte)
`LDM`	Load Multiple	`SUB`	Subtract
`LDR`	Load Register (Word)	`SWI`	Software Interrupt
`LDRB`	Load Register (Byte)	`SWP`	Swap Word Value
`MLA`	Multiply Accumulate	`SWPB`	Swap Byte Value
`MOV`	Move	`TEQ`	Test Equivalence
`MRS`	Load SPSR or CPSR	`TST`	Test
`MSR`	Store to SPSR or CPSR	`UMLAL`	Mult Accum Unsigned Long
`MUL`	Multiply	`UMULL`	Multiply Unsigned Long