ARM Instruction Set

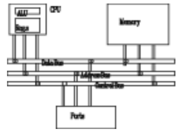
Computer Organization and Assembly Languages
Yung-Yu Chuang

with slides by Peng-Sheng Chen

Introduction

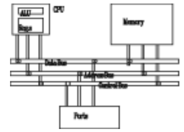
- The ARM processor is easy to program at the assembly level. (It is a RISC)
- We will learn ARM assembly programming at the user level and run it on a GBA emulator.

ARM programmer model

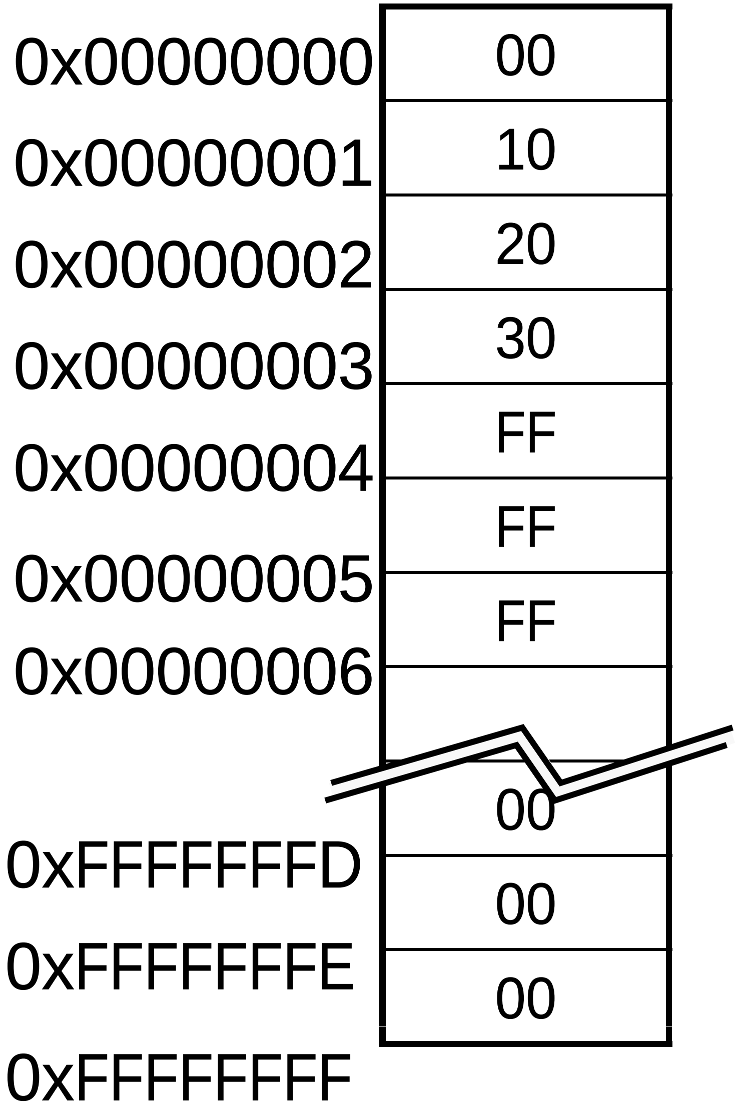


- The state of an ARM system is determined by the content of visible registers and memory.
- A user-mode program can see 15 32-bit general-purpose registers (R0-R14), program counter (PC) and CPSR.
- Instruction set defines the operations that can change the state.

Memory system



- Memory is a linear array of bytes addressed from 0 to 2^{32}-1
- Word, half-word, byte
- Little-endian



Byte ordering

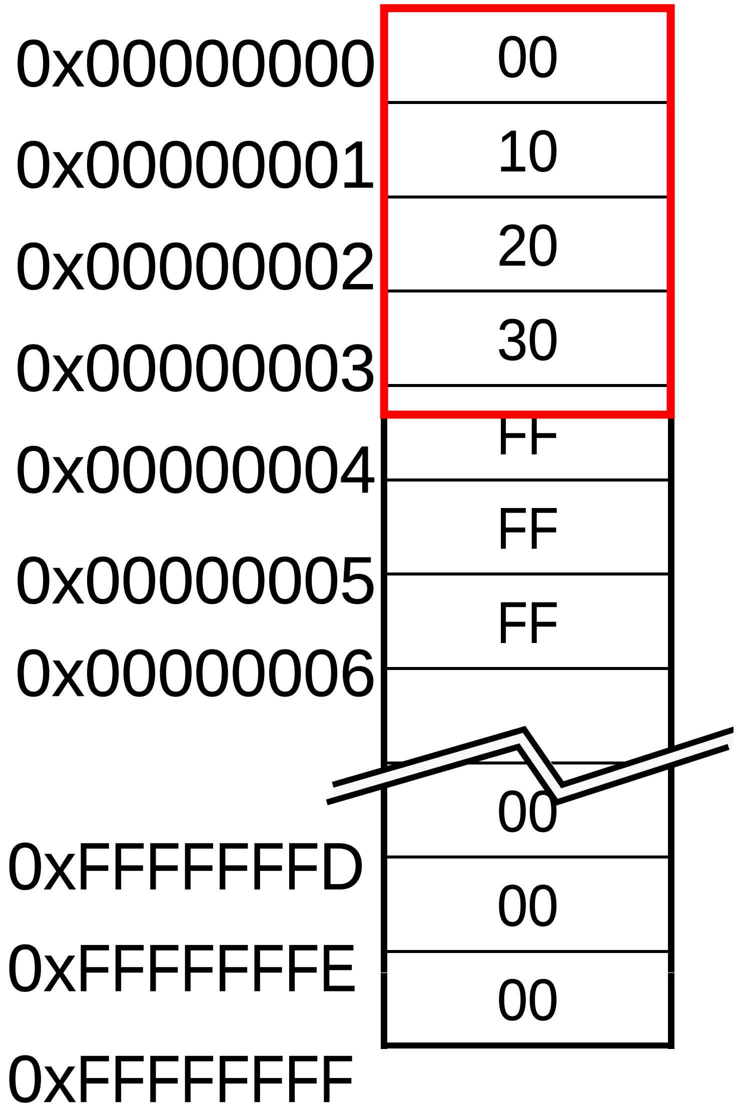
- Big Endian
  - Least significant byte has highest address

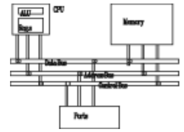
  Word address 0x00000000

  Value: 00102030
- Little Endian
  - Least significant byte has lowest address

  Word address 0x00000000

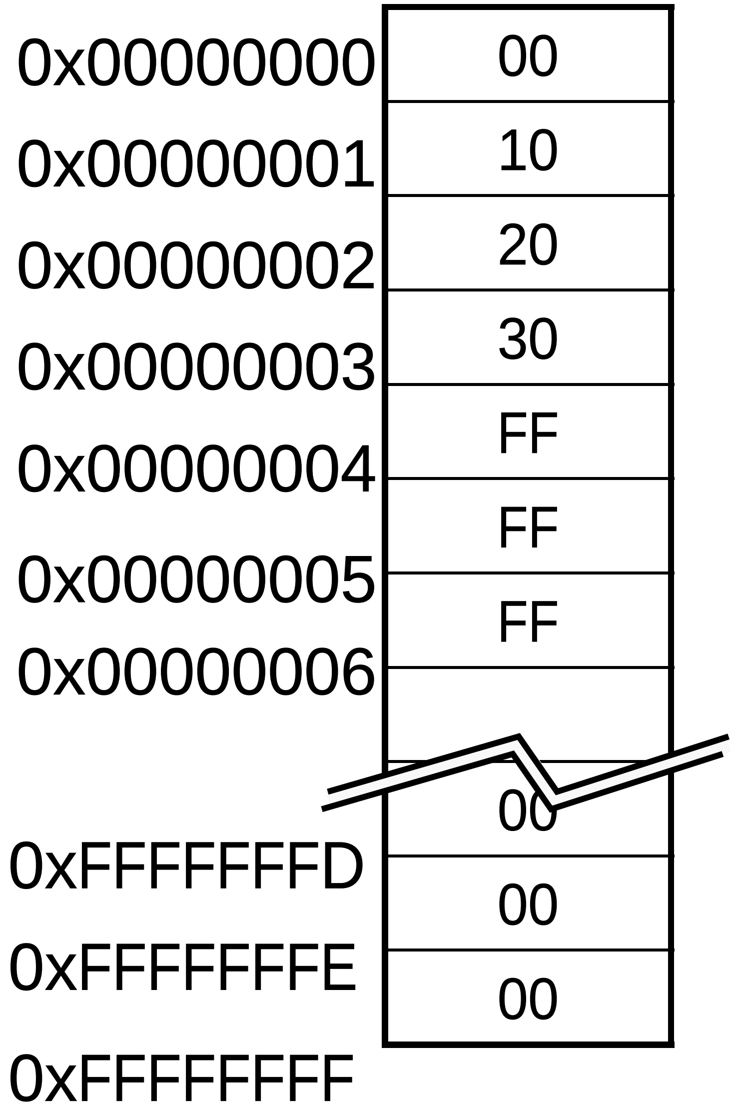
  Value: 30201000

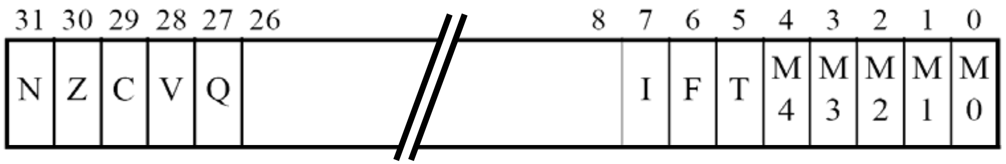




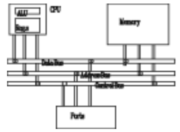
ARM programmer model

R0	R1	R2	R3
R4	R5	R6	R7
R8	R9	R10	R11
R12	R13	R14	PC





Instruction set



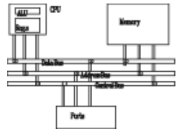
ARM instructions are all 32-bit long (except for Thumb mode). There are 2^{32} possible machine instructions. Fortunately, they are structured.

	31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
Data processing immediate shift	cond [1]	0	0	0	opcode				S	Rn	Rd	shift amount		shift		0	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x x x x	x x x x	x x x x	x	x	x	0	x x x x
Data processing register shift [2]	cond [1]	0	0	0	opcode				S	Rn	Rd	Rs	0	shift		1	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x x x x	x x x x	x x x x	0	x	x	1	x x x x
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0	0	0	x	x	x	x	x	x x x x	x x x x	x x x x	1	x	x	1	x x x x
Data processing immediate [2]	cond [1]	0	0	1	opcode				S	Rn	Rd	rotate	immediate				
Undefined instruction [3]	cond [1]	0	0	1	1	0	x	0	0	x x x x	x x x x	x x x x	x	x	x	x	x x x x
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask	SBO	rotate	immediate				
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn	Rd	immediate					
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn	Rd	shift amount		shift		0	Rm
Undefined instruction	cond [1]	0	1	1	x	x	x	x	x	x x x x	x x x x	x x x x	x	x	x	1	x x x x
Undefined instruction [4,7]	1 1 1 1	0	x	x	x	x	x	x	x	x x x x	x x x x	x x x x	x	x	x	x	x x x x
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn	register list						
Undefined instruction [4]	1 1 1 1	1	0	0	x	x	x	x	x	x x x x	x x x x	x x x x	x	x	x	x	x x x x
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset											
Branch and branch with link and change to Thumb [4]	1 1 1 1	1	0	1	H	24-bit offset											
Coprocessor load/store and double register transfers [6]	cond [5]	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	8-bit offset				
Coprocessor data processing	cond [5]	1	1	1	0	opcode1				CRn	CRd	cp_num	opcode2			0	CRm
Coprocessor register transfers	cond [5]	1	1	1	0	opcode1			L	CRn	Rd	cp_num	opcode2			1	CRm
Software interrupt	cond [1]	1	1	1	1	swi number											
Undefined instruction [4]	1 1 1 1	1	1	1	1	x	x	x	x	x x x x	x x x x	x x x x	x	x	x	x	x x x x

Features of ARM instruction set

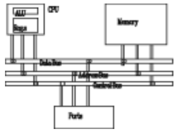
- Load-store architecture
- 3-address instructions
- Conditional execution of every instruction
- Possible to load/store multiple registers at once
- Possible to combine shift and ALU operations in a single instruction

Instruction set

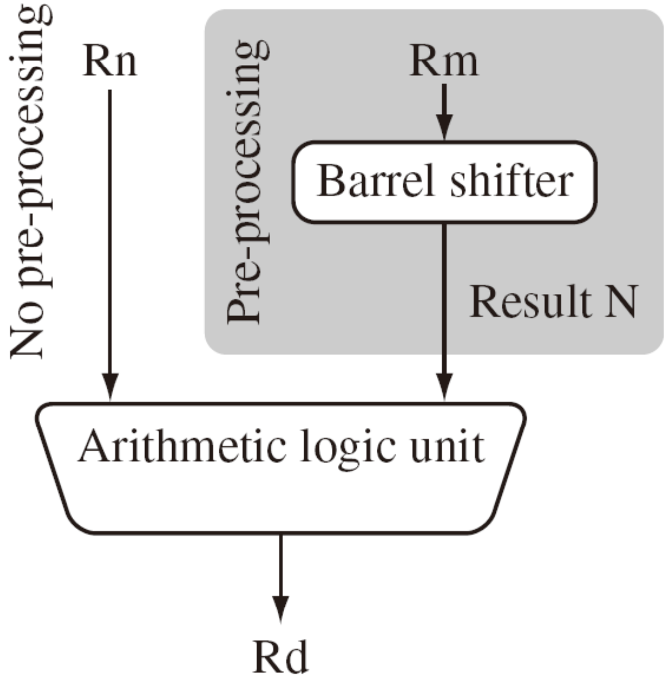


- Data processing
- Data movement
- Flow control

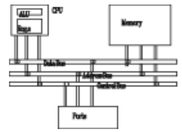
Data processing



- They are move, arithmetic, logical, comparison and multiply instructions.
- Most data processing instructions can process one of their operands using the barrel shifter.
- General rules:
  - All operands are 32-bit, coming from registers or literals.
  - The result, if any, is 32-bit and placed in a register (with the exception for long multiply which produces a 64-bit result)
  - 3-address format



Instruction set



```
MOV<cc><S>  Rd, <operands>

MOVCS R0, R1 @ if carry is set
             @ then R0:=R1

MOVS  R0, #0 @ R0:=0
             @ Z=1, N=0
             @ C, V unaffected
```

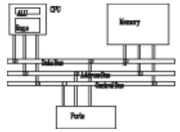
Conditional execution

- Almost all ARM instructions have a condition field which allows it to be executed conditionally.

```
movcs R0, R1
```

Mnemonic	Condition	Mnemonic	Condition
CS	Carry Set	CC	Carry Clear
EQ	Equal (Zero Set)	NE	Not Equal (Zero Clear)
VS	Overflow Set	VC	Overflow Clear
GT	Greater Than	LT	Less Than
GE	Greater Than or Equal	LE	Less Than or Equal
PL	Plus (Positive)	MI	Minus (Negative)
HI	Higher Than	LO	Lower Than (aka CC)
HS	Higher or Same (aka CS)	LS	Lower or Same

Register movement



Syntax: <instruction>{<cond>}{S} Rd, N   immediate,register,shift

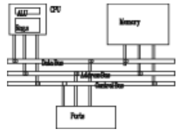
MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

- **MOV R0, R2 @ R0 = R2**
- **MVN R0, R2 @ R0 = ~R2**

move negated

```
PRE   r5 = 5
      r7 = 8
      MOV    r7, r5    ; let r7 = r5
POST  r5 = 5
      r7 = 5
```

Addressing modes



- Register operands

```
ADD  R0, R1, R2
```

- Immediate operands

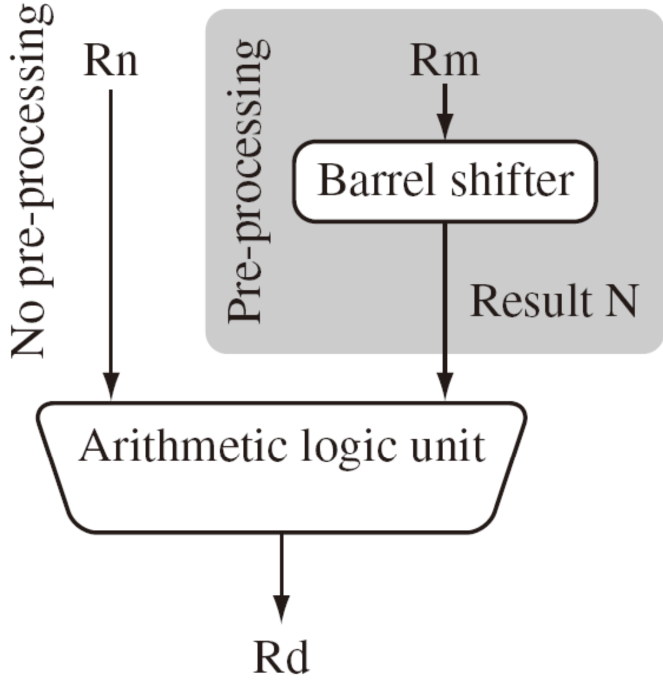
a literal; most can be represented by (0..255)x2^{2n} 0<n<12

```
ADD  R3, R3, #1    @ R3:=R3+1
AND  R8, R7, #0xff @ R8=R7[7:0]
```

a hexadecimal literal
This is assembler dependent syntax.

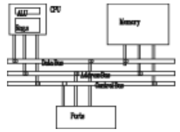
Shifted register operands

- One operand to ALU is routed through the Barrel shifter. Thus, the operand can be modified before it is used. Useful for fast multipliation and dealing with lists, table and other complex data structure. (similar to the displacement addressing mode in CISC.)



- Some instructions (e.g. **`MUL, CLZ, QADD`**) do not read barrel shifter.

Shifted register operands



Mnemonic	Description	Shift	Result
LSL	logical shift left	x LSL y	$x \ll y$
LSR	logical shift right	x LSR y	$(\text{unsigned})x \gg y$
ASR	arithmetic right shift	x ASR y	$(\text{signed})x \gg y$
ROR	rotate right	x ROR y	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$
RRX	rotate right extended	x RRX	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$

Logical shift left


```
MOV  R0, R2, LSL #2 @ R0:=R2<<2
                    @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x000000C0
       R2=0x00000030
```

Logical shift right


```
MOV  R0, R2, LSR #2 @ R0:=R2>>2
                    @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x0000000C
       R2=0x00000030
```

Arithmetic shift right


```
MOV   R0, R2, ASR #2 @ R0:=R2>>2
                     @ R2 unchanged
Example: 1010 0…0 0011 0000
Before R2=0xA0000030
After  R0=0xE800000C
       R2=0xA0000030
```

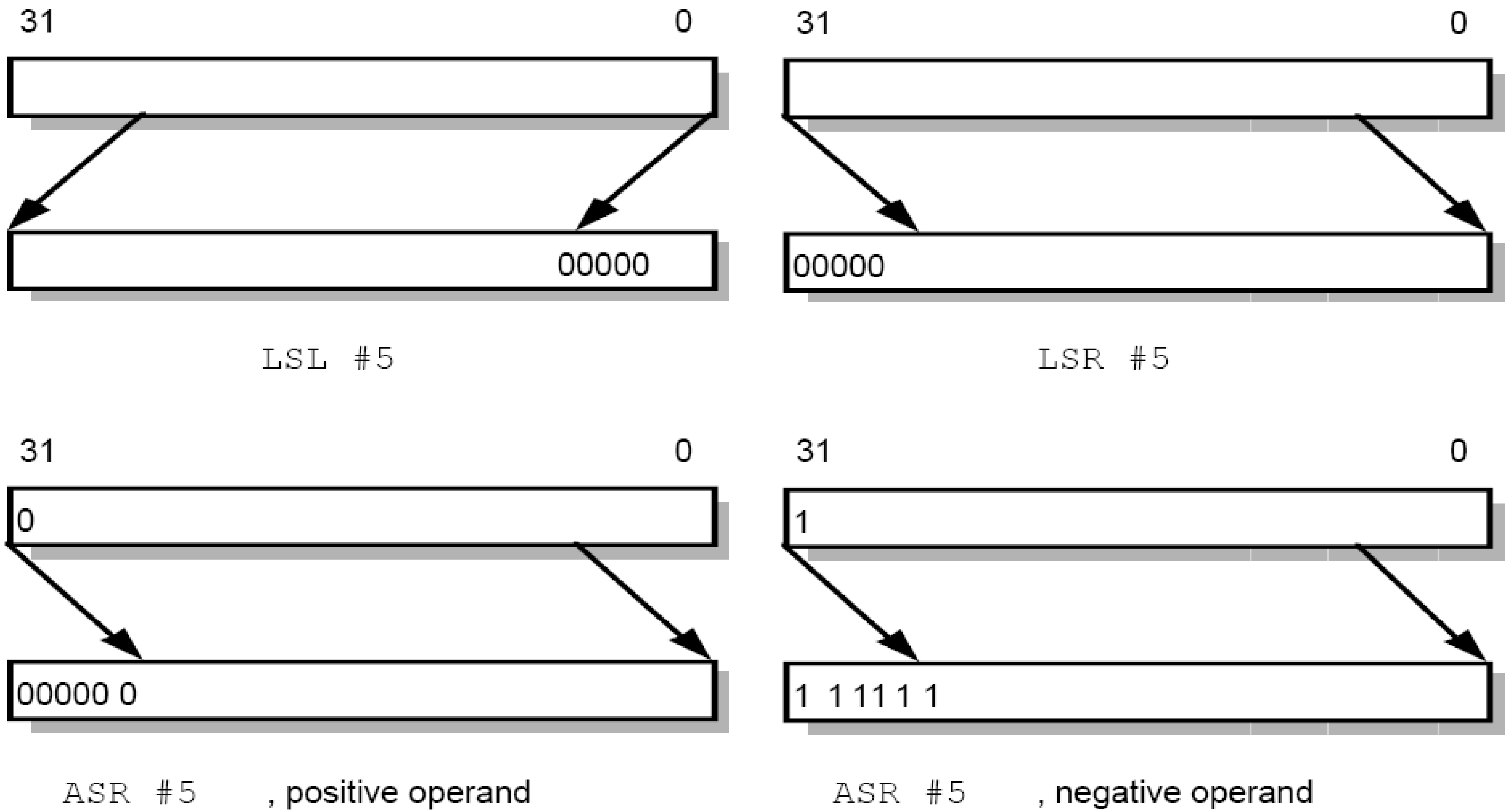
Rotate right


```
MOV  R0, R2, ROR #2 @ R0:=R2 rotate
                    @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031
After  R0=0x4000000C
       R2=0x00000031
```

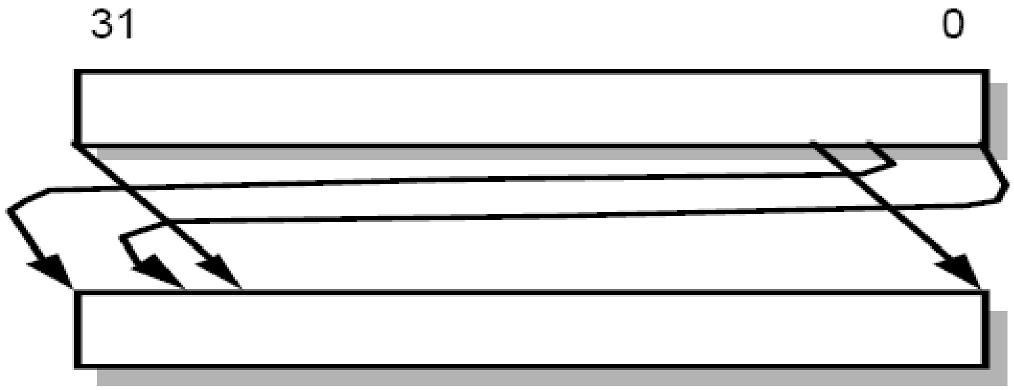
Rotate right extended


```
MOV   R0, R2, RRX        @ R0:=R2 rotate
                         @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031, C=1
After  R0=0x80000018, C=1
       R2=0x00000031
```

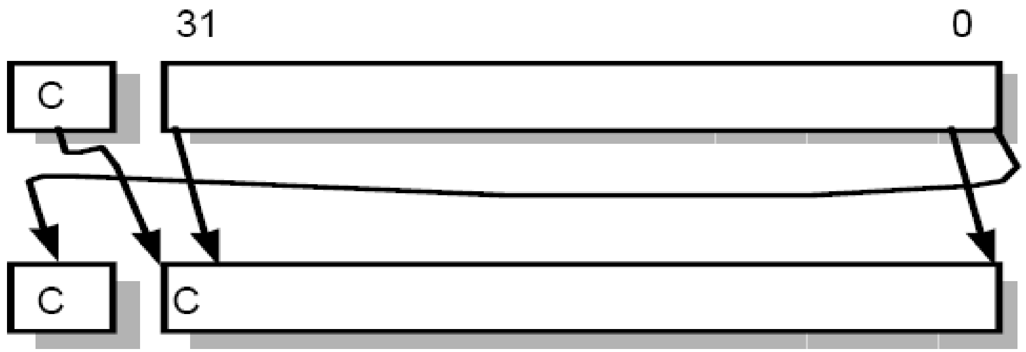
Shifted register operands



Shifted register operands



ROR #5



RRX

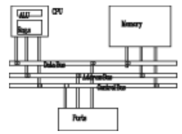
Shifted register operands

- It is possible to use a register to specify the number of bits to be shifted; only the bottom 8 bits of the register are significant.

```
@ array index calculation
ADD  R0, R1, R2, LSL R3  @ R0:=R1+R2*2^R3
```

```
@ fast multiply R2=35xR0
ADD  R0, R0, R0, LSL #2  @ R0’=5xR0
RSB  R2, R0, R0, LSL #3  @ R2 =7xR0’
```

Multiplication



```
MOV  R1, #35
MUL  R2, R0, R1
```

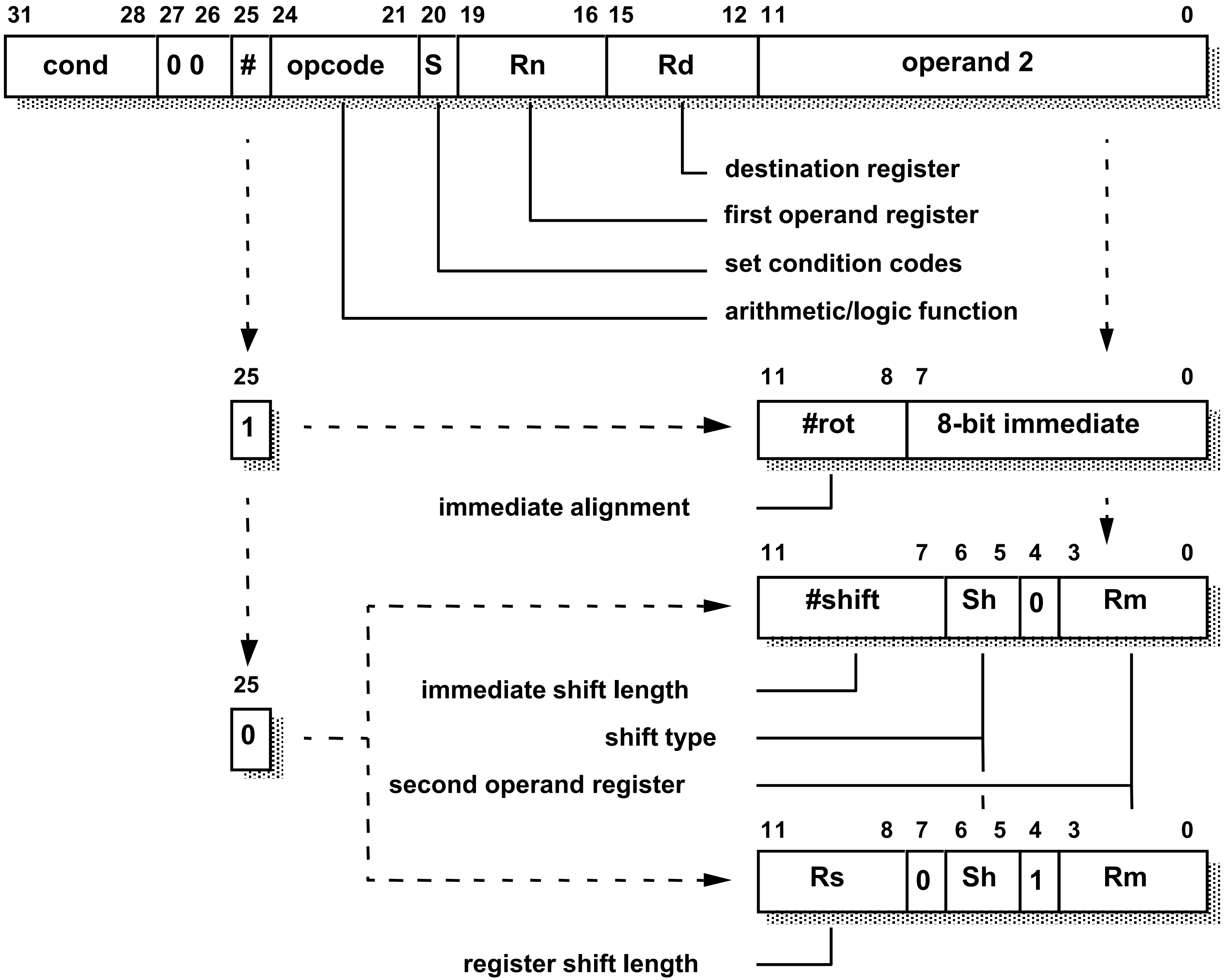
or

```
ADD  R0, R0, R0, LSL #2  @ R0'=5xR0
RSB  R2, R0, R0, LSL #3  @ R2 =7xR0'
```

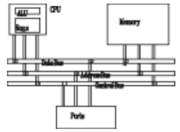
Shifted register operands

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

Encoding data processing instructions



Arithmetic



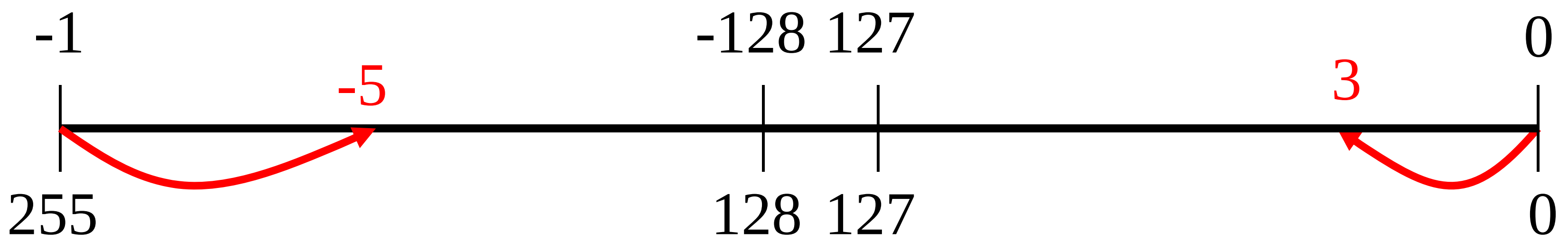
- Add and subtraction

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

Arithmetic

```
• ADD  R0, R1, R2       @ R0 = R1+R2
• ADC  R0, R1, R2       @ R0 = R1+R2+C
• SUB  R0, R1, R2       @ R0 = R1-R2
• SBC  R0, R1, R2       @ R0 = R1-R2-!C
• RSB  R0, R1, R2       @ R0 = R2-R1
• RSC  R0, R1, R2       @ R0 = R2-R1-!C
```



3-5=3+(-5) ⟶ sum<=255 ⟶ C=0 ⟶ borrow

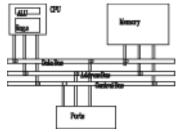
5-3=5+(-3) ⟶ sum > 255 ⟶ C=1 ⟶ no borrow

Arithmetic

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000001

        SUB r0, r1, r2

POST    r0 = 0x00000001
```

```
PRE     r0 = 0x00000000
        r1 = 0x00000077

        RSB r0, r1, #0       ; Rd = 0x0 - r1

POST    r0 = -r1 = 0xffffff89
```

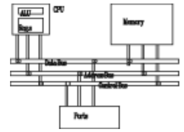
Arithmetic



```
PRE     cpsr = nzcvqiFt_USER
        r1 = 0x00000001

        SUBS r1, r1, #1
POST    cpsr = nZCvqiFt_USER
        r1 = 0x00000000
```

```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

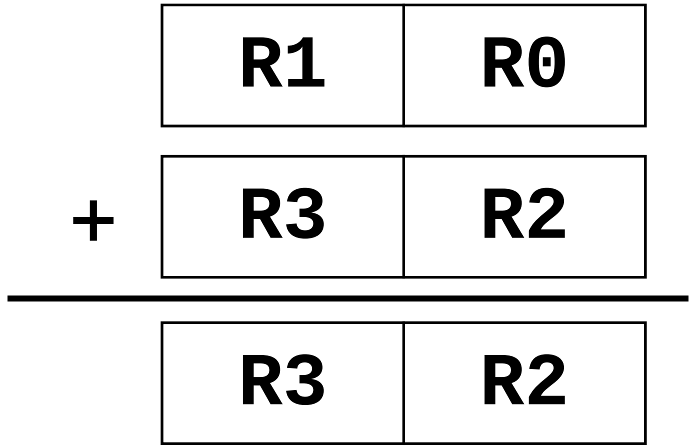
Setting the condition codes



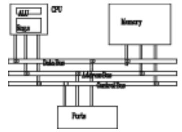
- Any data processing instruction can set the condition codes if the programmers wish it to

64-bit addition

```
ADDS   R2, R2, R0
ADC    R3, R3, R1
```



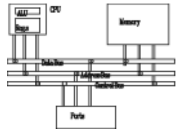
Logical



Syntax: <instruction>{<cond>}{S} Rd, Rn, N

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \,\&\, N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \,^\wedge\, N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \,\&\, \sim N$

Logical



- `AND  R0, R1, R2   @ R0 = R1 and R2`
- `ORR  R0, R1, R2   @ R0 = R1 or  R2`
- `EOR  R0, R1, R2   @ R0 = R1 xor R2`
- `BIC  R0, R1, R2   @ R0 = R1 and (~R2)`

bit clear: **R2** is a mask identifying which bits of **R1** will be cleared to zero

```
R1=0x11111111    R2=0x01100101

BIC R0, R1, R2

R0=0x10011010
```

Logical

```
PRE     r0 = 0x00000000
        r1 = 0x02040608
        r2 = 0x10305070

        ORR   r0, r1, r2

POST    r0 = 0x12345678
```

```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

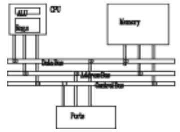
Comparison

- These instructions do not generate a result, but set condition code bits (N, Z, C, V) in CPSR. Often, a branch operation follows to change the program flow.

Syntax: <instruction>{<cond>} Rn, N

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of Rn ^ N
TST	test bits of a 32-bit value	flags set as a result of Rn & N

Comparison



compare

- **CMP R1, R2 @ set cc on R1-R2**

compare negated

- **CMN R1, R2 @ set cc on R1+R2**

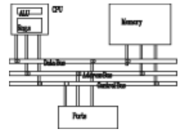
bit test

- **TST R1, R2 @ set cc on R1 and R2**

test equal

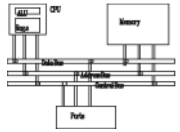
- **TEQ R1, R2 @ set cc on R1 xor R2**

Comparison



```
PRE     cpsr = nzcvqiFt_USER
        r0 = 4
        r9 = 4

        CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

Multiplication



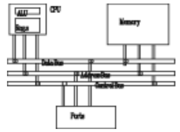
```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm^* Rs) + Rn$
MUL	multiply	$Rd = Rm^* Rs$

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm^* Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm^* Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm^* Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm^* Rs$

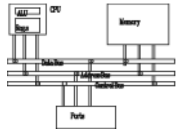
Multiplication



- **MUL R0, R1, R2 @ R0 = (R1xR2)$_{[31:0]}$**

- Features:
  - Second operand can't be immediate
  - The result register must be different from the first operand
  - Cycles depends on core type
  - If S bit is set, C flag is meaningless
- See the reference manual (4.1.33)

Multiplication



- Multiply-accumulate (2D array indexing)

```
MLA  R4, R3, R2, R1  @ R4 = R3xR2+R1
```

- Multiply with a constant can often be more efficiently implemented using shifted register operand

```
MOV  R1, #35
MUL  R2, R0, R1
```

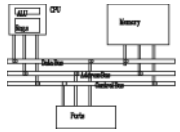
or

```
ADD  R0, R0, R0, LSL #2  @ R0’=5xR0
RSB  R2, R0, R0, LSL #3  @ R2 =7xR0’
```

Multiplication

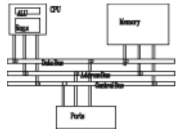
```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        MUL   r0, r1, r2    ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

Multiplication



```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3   ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

Flow control instructions

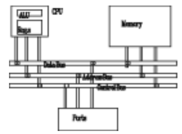


- Determine the instruction to be executed next

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

B	branch	*pc* = *label* pc-relative offset within 32MB
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

Flow control instructions



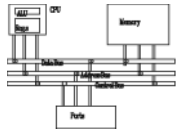
- Branch instruction

```
        B   label
        …
label:  …
```

- Conditional branches

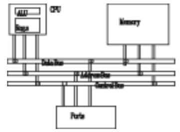
```
        MOV  R0, #0
loop:        …
        ADD  R0, R0, #1
        CMP  R0, #10
        BNE  loop
```

Branch conditions



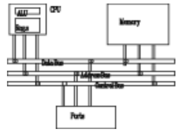
Mnemonic	Name	Condition flags
`EQ`	equal	Z
`NE`	not equal	z
`CS HS`	carry set/unsigned higher or same	C
`CC LO`	carry clear/unsigned lower	c
`MI`	minus/negative	N
`PL`	plus/positive or zero	n
`VS`	overflow	V
`VC`	no overflow	v
`HI`	unsigned higher	zC
`LS`	unsigned lower or same	Z or c
`GE`	signed greater than or equal	NV or nv
`LT`	signed less than	Nv or nV
`GT`	signed greater than	NzV or nzv
`LE`	signed less than or equal	Z or Nv or nV
`AL`	always (unconditional)	ignored

Branches



Branch	Interpretation	Normal uses
B BAL	Unconditional Always	Always take this branch Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

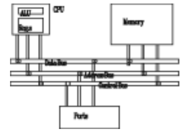
Branch and link



- **BL** instruction save the return address to **R14** (lr)

```
       BL     sub      @ call sub
       CMP    R1, #5   @ return to here
       MOVEQ  R1, #0
       …
sub:   …               @ sub entry point
       …
       MOV    PC, LR   @ return
```

Branch and link



```
        BL      sub1        @ call sub1
        …
```

use stack to save/restore the return address and registers

```
sub1:   STMFD R13!, {R0-R2,R14}
        BL      sub2
        …
        LDMFD R13!, {R0-R2,PC}
```

```
sub2:   …
        …
        MOV     PC, LR
```

Conditional execution

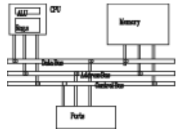
```
        CMP  R0, #5
        BEQ  bypass     @ if (R0!=5) {
        ADD  R1, R1, R0 @  R1=R1+R0-R2
        SUB  R1, R1, R2 @ }
bypass: …
```

```
        CMP   R0, #5
        ADDNE R1, R1, R0
        SUBNE R1, R1, R2
```

smaller and faster

Rule of thumb: if the conditional sequence is three instructions or less, it is better to use conditional execution than a branch.

Conditional execution

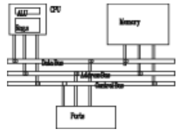


```
    if ((R0==R1) && (R2==R3)) R4++
```

```
         CMP   R0, R1
         BNE   skip
         CMP   R2, R3
         BNE   skip
         ADD   R4, R4, #1
skip:    …
```

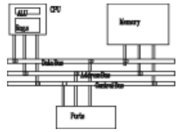
```
         CMP   R0, R1
         CMPEQ R2, R3
         ADDEQ R4, R4, #1
```

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 instruction for semaphore

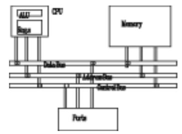
Single register load/store



```
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]*

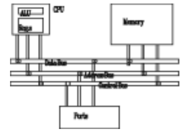
Single register load/store



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**)

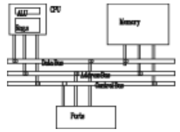
LDR R0, [R1]   **@ R0 := mem_{32}[R1]**

STR R0, [R1]   **@ mem_{32}[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]
```

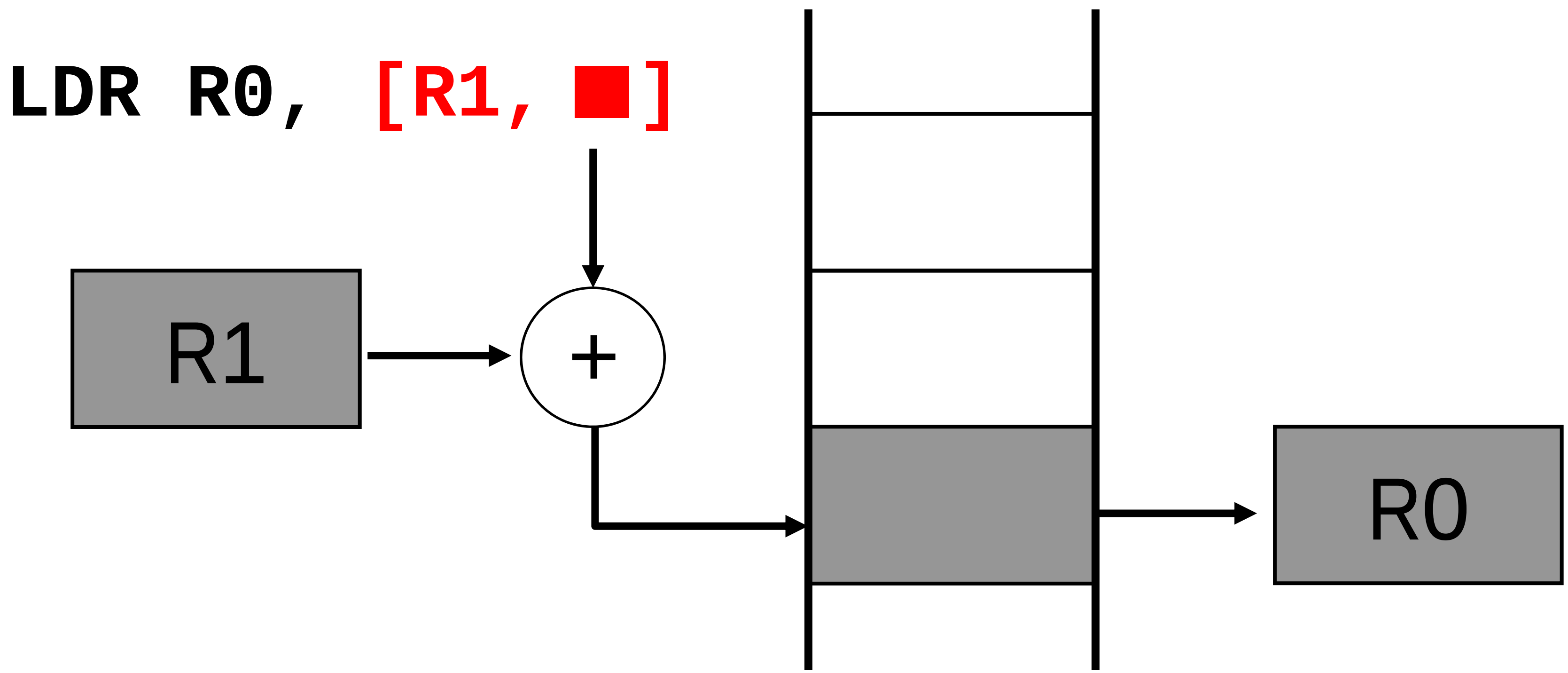
Addressing modes

- 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

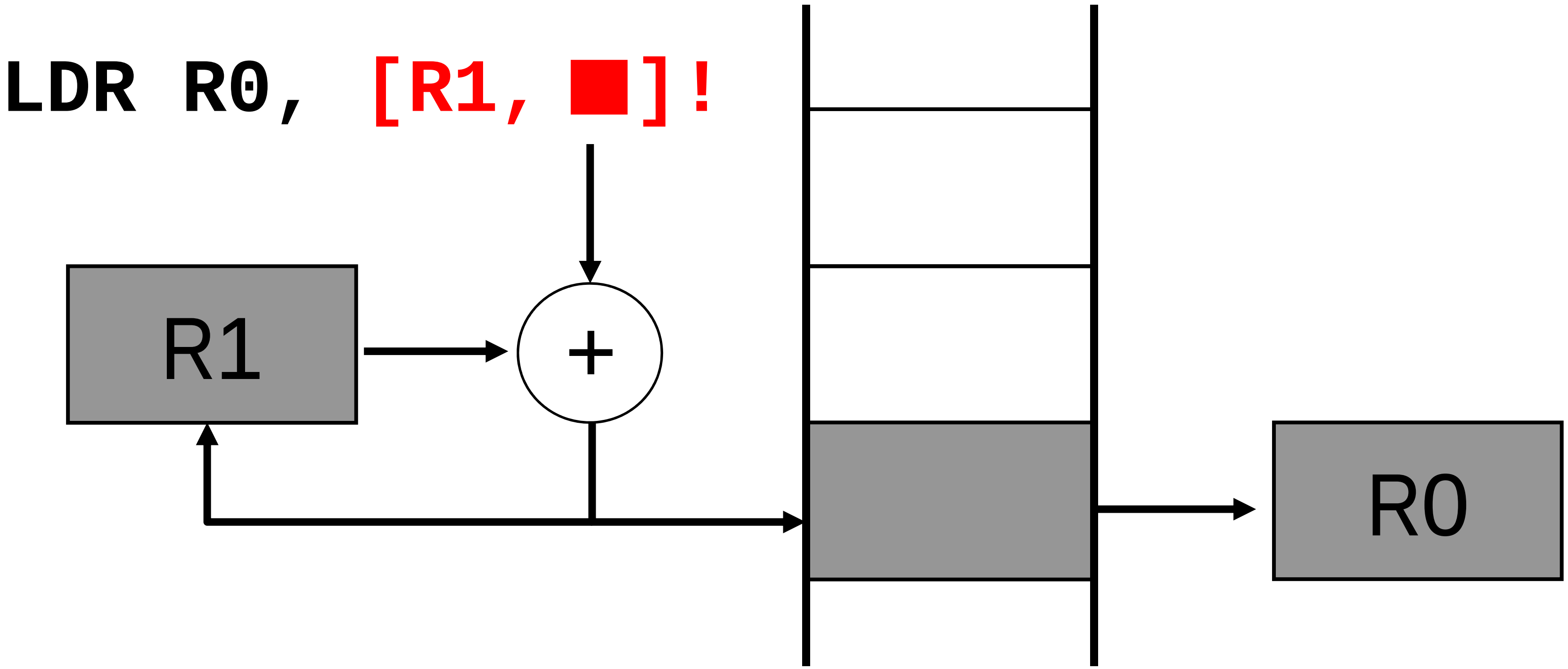
```
LDR  R0, [R1, #4]    @ R0=mem[R1+4]
                     @ R1 unchanged
```



Auto-indexing addressing

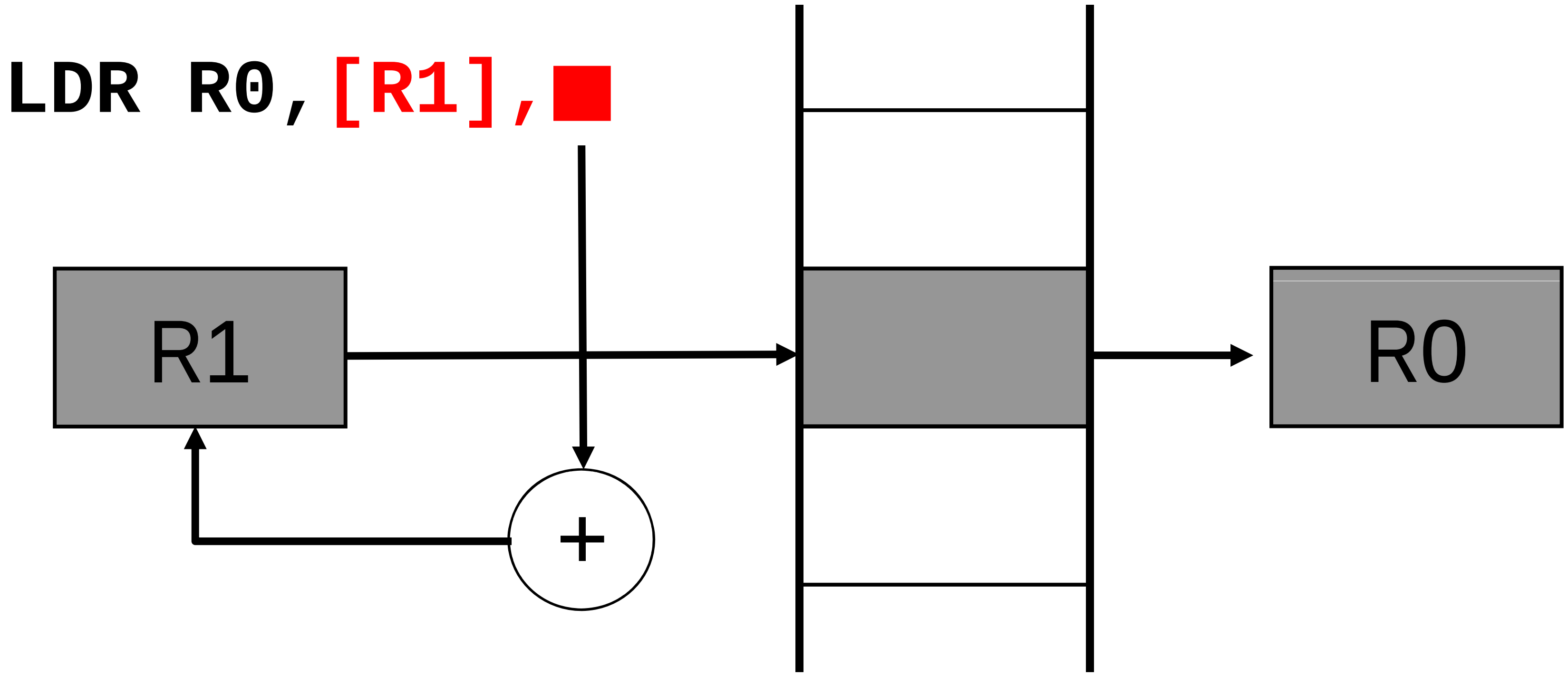
```
LDR  R0, [R1, #4]!  @ R0=mem[R1+4]
                    @ R1=R1+4
```

No extra time; Fast;

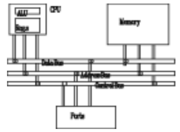


Post-index addressing

```
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
```

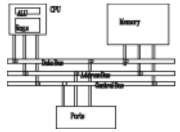
Example

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

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

Postindexing:

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

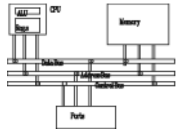
Summary of addressing modes



```
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



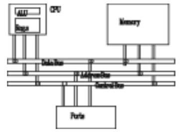
	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*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

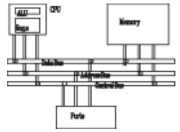
Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register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

Load an address into a register



- Note that all addressing modes are register-offseted. Can we issue **`LDR R0, Table`**? The pseudo instruction **`ADR`** loads a register with an address

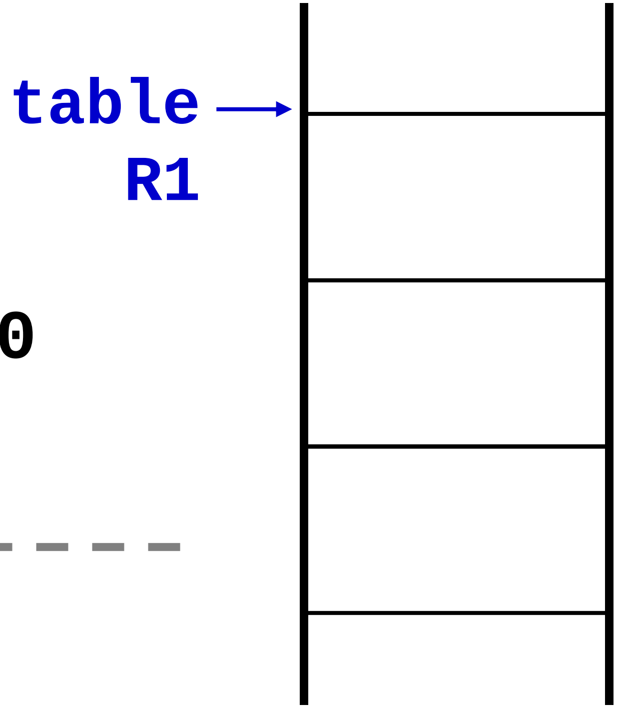
```
table:    .word    10
…
          ADR  R0, table
```

- Assembler transfer pseudo instruction into a sequence of appropriate instructions

```
sub    r0, pc, #12
```

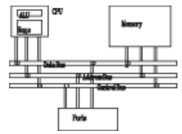
Application

```
        ADR R1, table
loop:   LDR R0, [R1]
        ADD R1, R1, #4
        @ operations on R0
        …
```




```
        ADR R1, table
loop:   LDR R0, [R1], #4

        @ operations on R0
        …
```

Multiple register load/store



- Transfer a block of data more efficiently.
- Used for procedure entry and exit for saving and restoring workspace registers and the return address
- For ARM7, $2+Nt$ cycles (N:#words, t:time for a word for sequential access). Increase interrupt latency since it can't be interrupted.

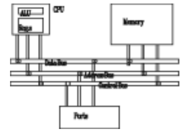
registers are arranged an in increasing order; see manual

```
LDMIA  R1, {R0, R2, R5} @ R0 = mem[R1]
                        @ R2 = mem[r1+4]
                        @ R5 = mem[r1+8]
```

Multiple load/store register

```
LDM   load multiple registers
STM   store multiple registers

suffix      meaning
  IA    increase after
  IB    increase before
  DA    decrease after
  DB    decrease before
```

Addressing modes

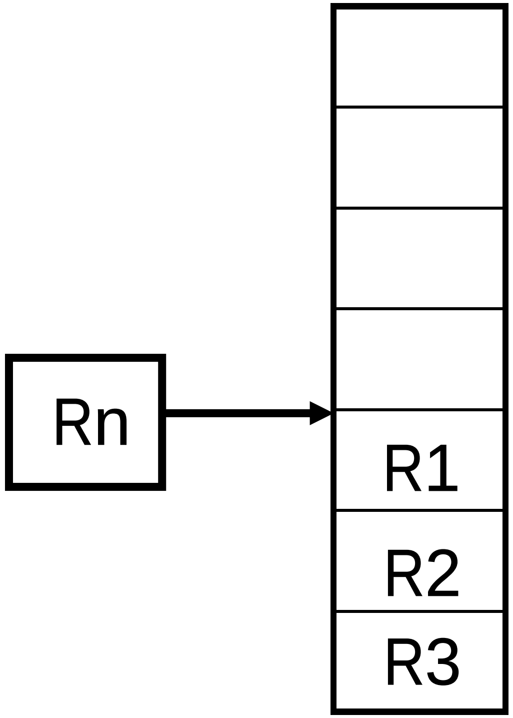


Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	Rn	$Rn + 4^*N - 4$	$Rn + 4^*N$
IB	increment before	$Rn + 4$	$Rn + 4^*N$	$Rn + 4^*N$
DA	decrement after	$Rn - 4^*N + 4$	Rn	$Rn - 4^*N$
DB	decrement before	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

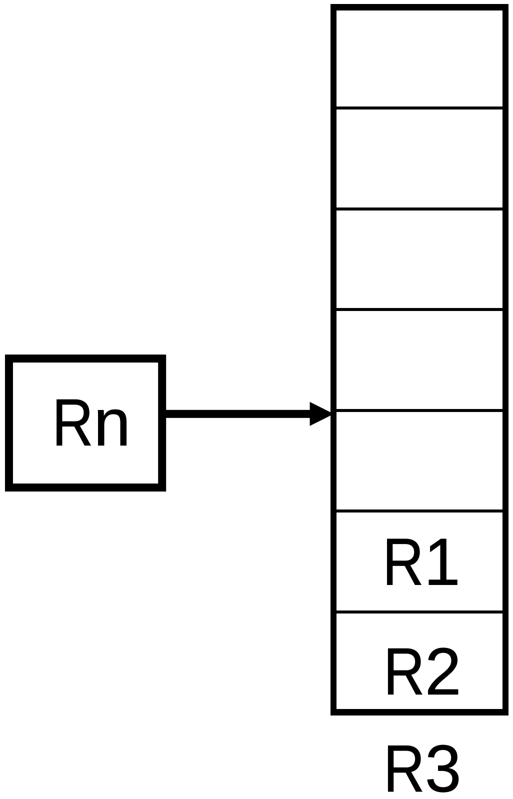
Multiple load/store register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



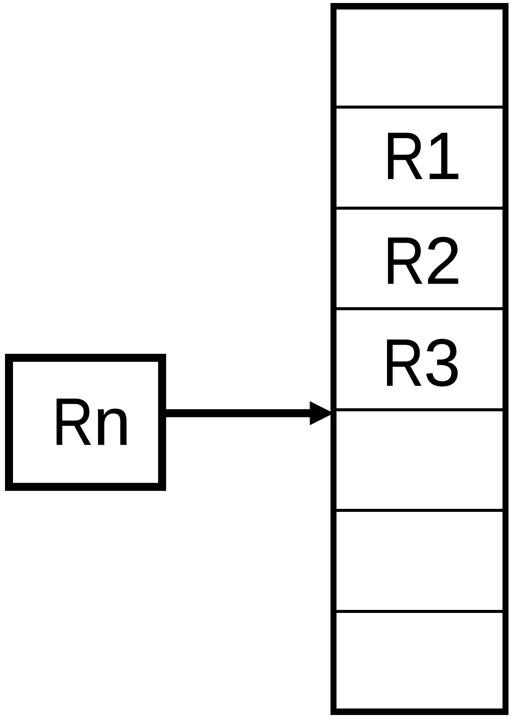
Multiple load/store register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



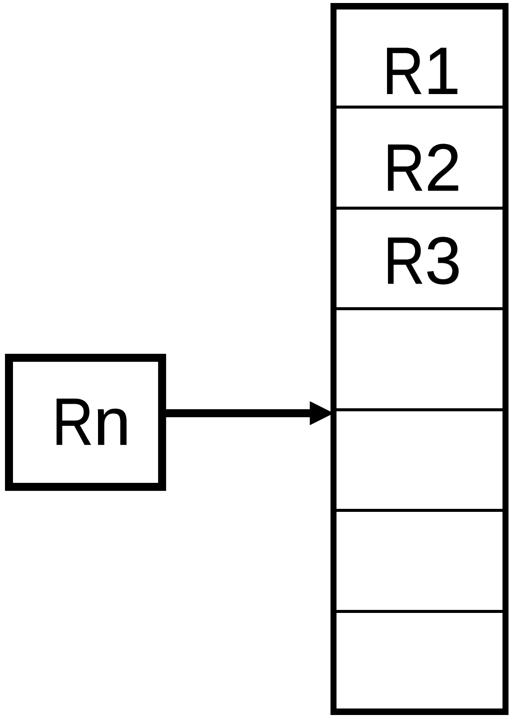
Multiple load/store register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



Multiple load/store register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



Multiple load/store register

LDMIA R0, {R1,R2,R3}

or

LDMIA R0, {R1-R3}

R1: 10

R2: 20

R3: 30

R0: 0x10

R0 →

addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Multiple load/store register

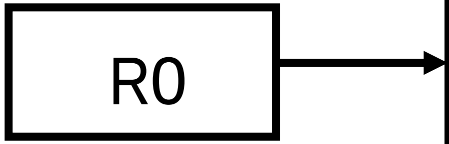
LDMIA R0!, {R1,R2,R3}

R1: 10

R2: 20

R3: 30

R0: 0x01C



addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Multiple load/store register

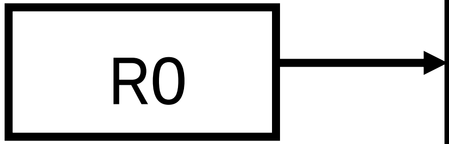
`LDMIB R0!, {R1,R2,R3}`

R1: 20

R2: 30

R3: 40

R0: 0x01C



addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Multiple load/store register

`LDMDA R0!, {R1,R2,R3}`

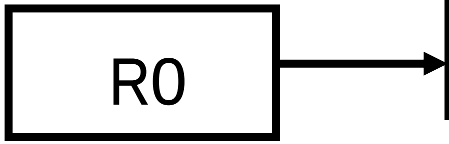
R1: 40

R2: 50

R3: 60

R0: 0x018

addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60



Multiple load/store register

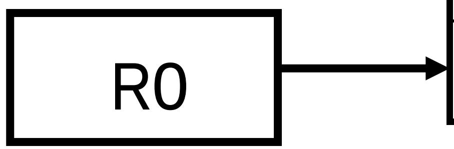
LDMDB R0!, {R1,R2,R3}

R1: 30

R2: 40

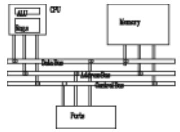
R3: 50

R0: 0x018



addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Example



```
PRE    mem32[0x80018] = 0x03
       mem32[0x80014] = 0x02
       mem32[0x80010] = 0x01
       r0 = 0x00080010
       r1 = 0x00000000
       r2 = 0x00000000
       r3 = 0x00000000

       LDMIA   r0!, {r1-r3}
```

Example

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
r0 **= 0x80010** →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

```
LDMIA  r0!, {r1-r3}
```

Address pointer	Memory address	Data	
	0x80020	0x00000005	
r0 **= 0x8001c** →	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000003
	0x80014	0x00000002	*r2* = 0x00000002
	0x80010	0x00000001	*r1* = 0x00000001
	0x8000c	0x00000000	

Example

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
***r0* = 0x80010** →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

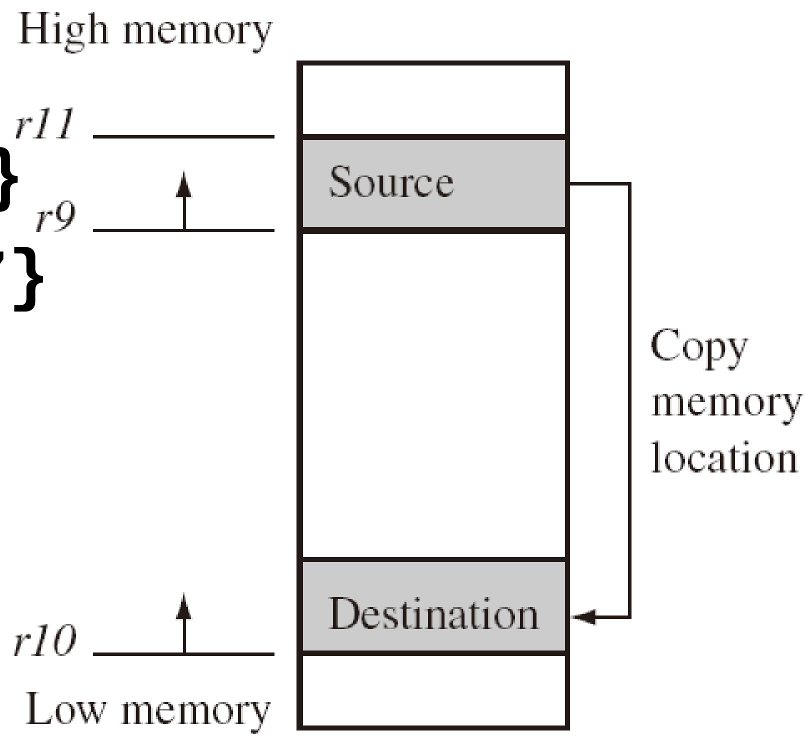
```
LDMIB  r0!, {r1-r3}
```

Address pointer	Memory address	Data	
	0x80020	0x00000005	
***r0* = 0x8001c** →	0x8001c	0x00000004	*r3* = 0x00000004
	0x80018	0x00000003	*r2* = 0x00000003
	0x80014	0x00000002	*r1* = 0x00000002
	0x80010	0x00000001	
	0x8000c	0x00000000	

Application

- Copy a block of memory
  - R9: address of the source
  - R10: address of the destination
  - R11: end address of the source

```
loop: LDMIA R9!, {R0-R7}
      STMIA R10!, {R0-R7}
      CMP   R9, R11
      BNE   loop
```



Application

- Stack (full: pointing to the last used; ascending: grow towards increasing memory addresses)

mode	POP	=LDM	PUSH	=STM
Full ascending (**FA**)	**LDMFA**	**LDMDA**	**STMFA**	**STMIB**
Full descending (**FD**)	**LDMFD**	**LDMIA**	**STMFD**	**STMDB**
Empty ascending (**EA**)	**LDMEA**	**LDMDB**	**STMEA**	**STMIA**
Empty descending (**ED**)	**LDMED**	**LDMIB**	**STMED**	**STMDA**

```
LDMFD R13!, {R2-R9} @ used for ATPCS
… @ modify R2-R9
STMFD R13!, {R2-R9}
```

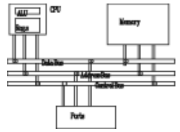
Example

PRE	Address	Data
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	*Empty*
	0x8000c	*Empty*

```
STMFD   sp!, {r1,r4}
```

POST	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002

Swap instruction

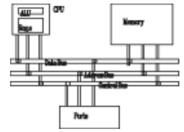


- Swap between memory and register. Atomic operation preventing any other instruction from reading/writing to that location until it completes

```
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	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

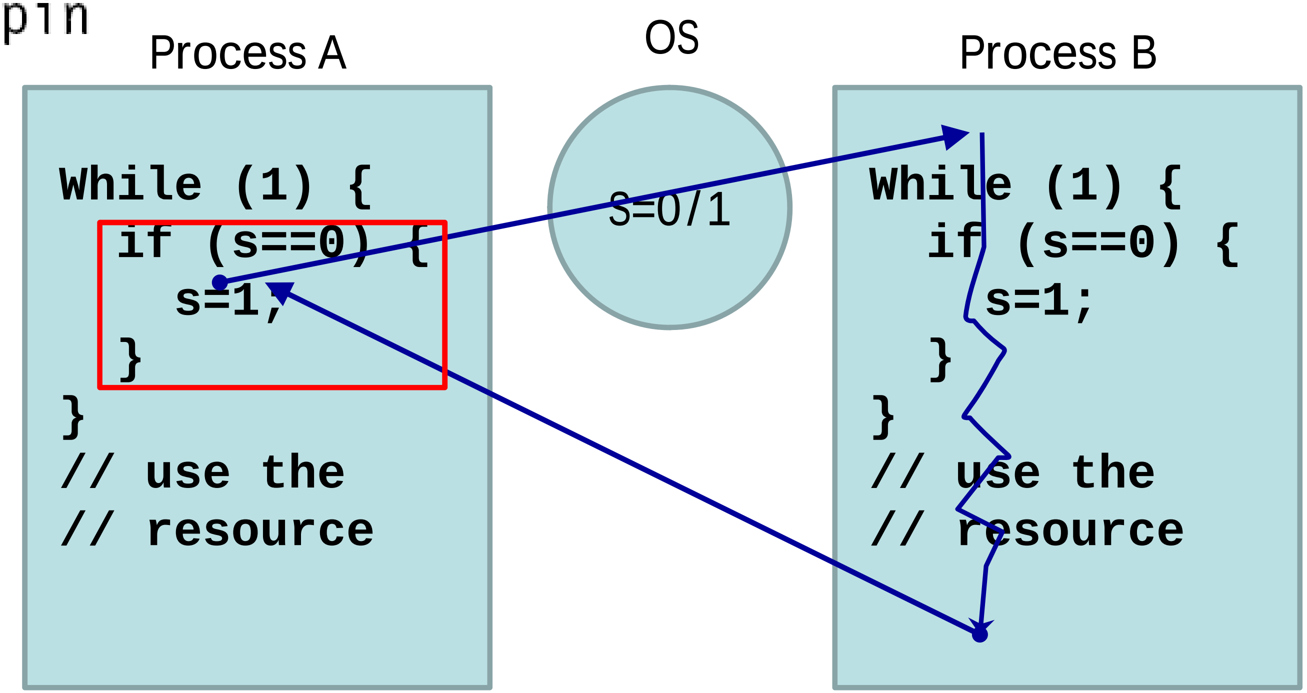
Example



```
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.

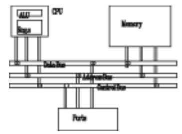
```
Syntax: SWI{<cond>} SWI_number
```

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

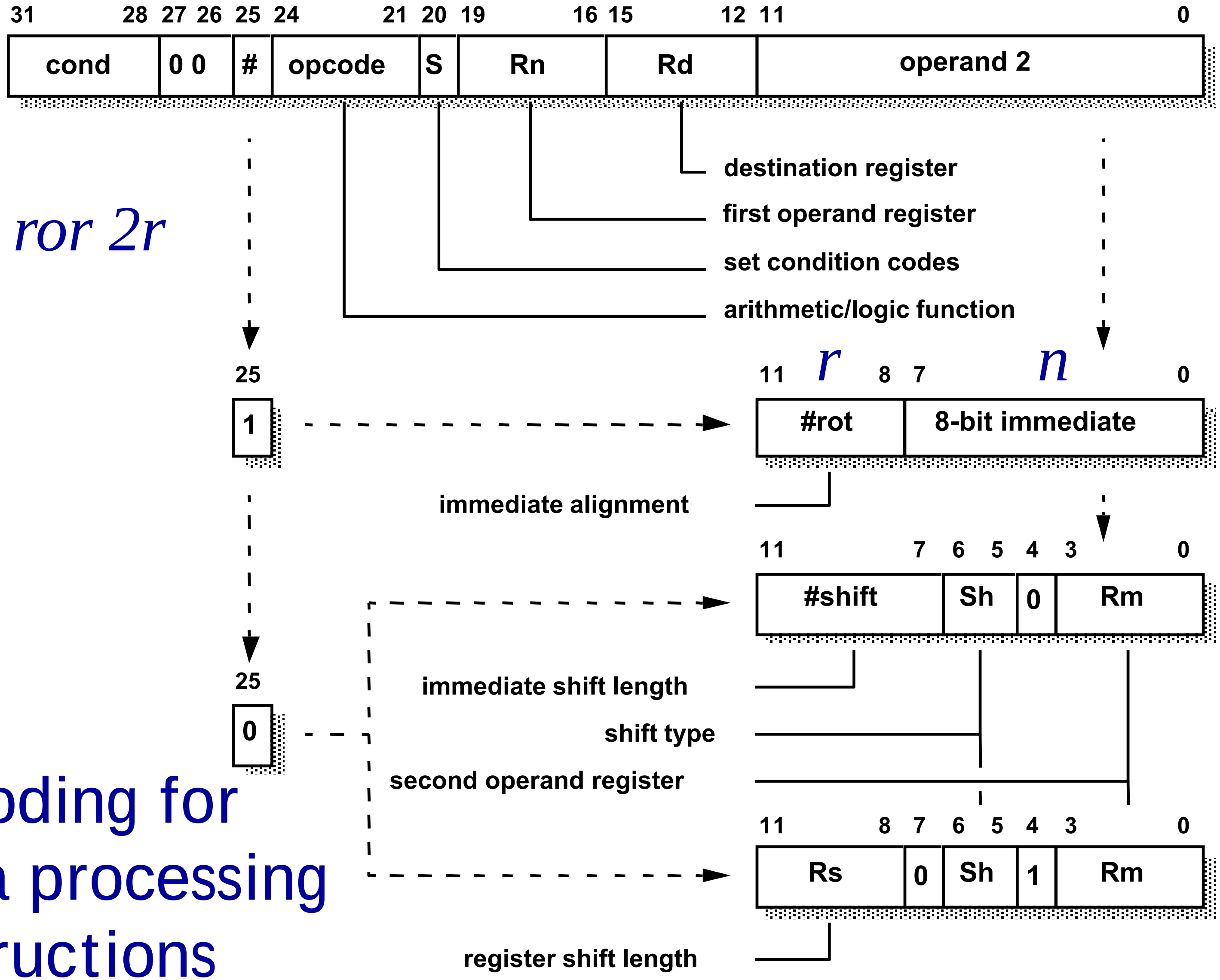


- No ARM instruction loads a 32-bit constant into a register because ARM instructions are 32-bit long. There is a pseudo code for this.

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

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

Immediate numbers



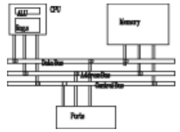
v=n ror 2r

r

n

encoding for data processing instructions

Load constants



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

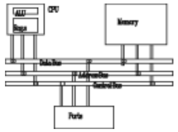
Pseudoinstruction	Actual instruction
`LDR r0, =0xff`	`MOV r0, #0xff`
`LDR r0, =0x55555555`	`LDR r0, [pc, #offset_12]`

Loading the constant `0xff00ffff`

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

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

Load constants



- Assume that you want to load 511 into R0
  - Construct in multiple instructions

```
mov  r0, #256
add  r0, #255
```

  - Load from memory; declare **`L511  .word 511`**

```
ldr  r0, L511  →  ldr r0, [pc, #0]
```

- Guideline: if you can construct it in two instructions, do it; otherwise, load it.
- The assembler decides for you

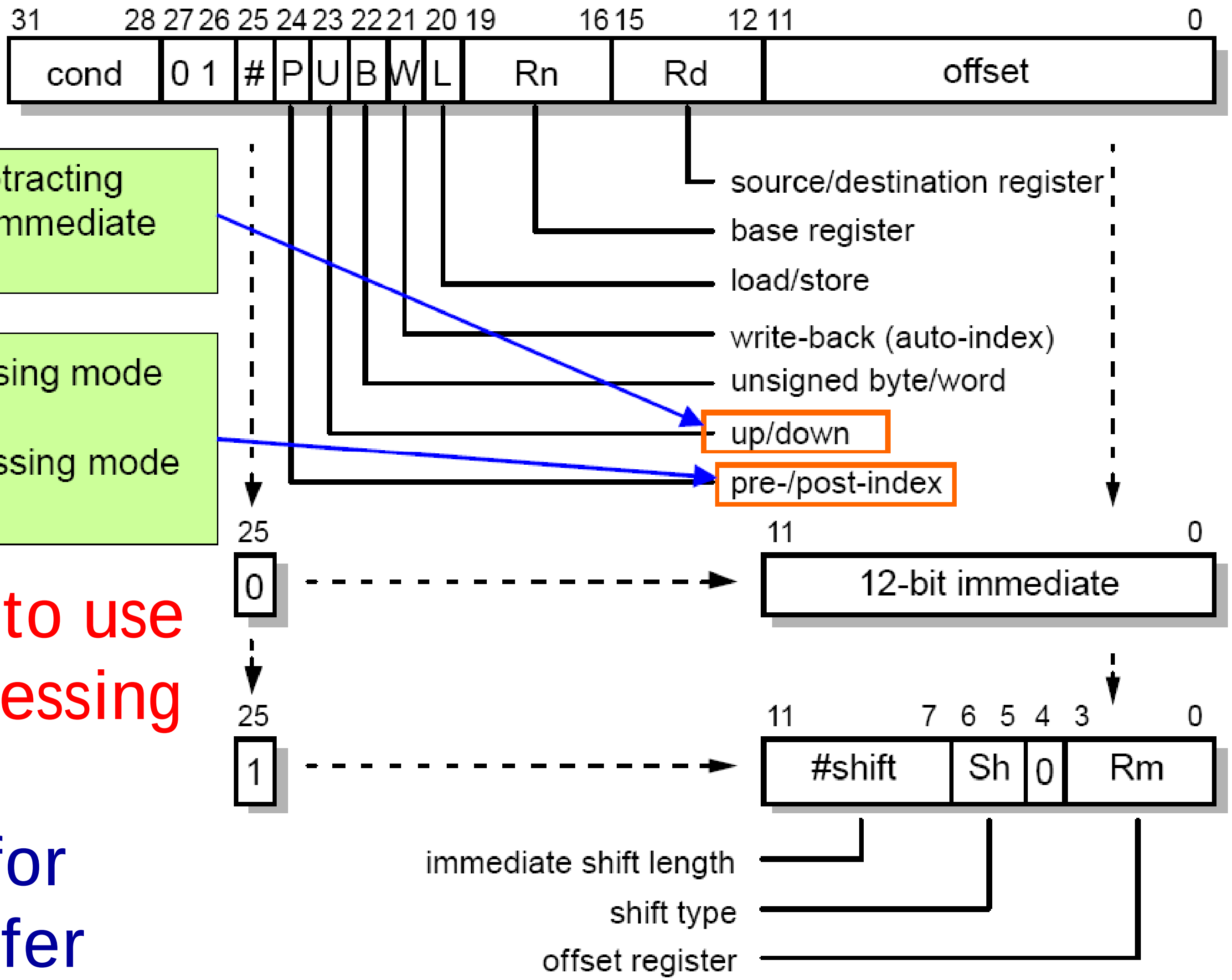
```
ldr r0, =255  →  mov r0, 255
ldr r0, =511  →  ldr r0, [pc, #4]
```

PC-relative modes

Adding (U=1) or subtracting (U=0) an unsigned immediate or register offset

Pre-indexed addressing mode (P=1)
Post-indexed addressing mode (P=0)

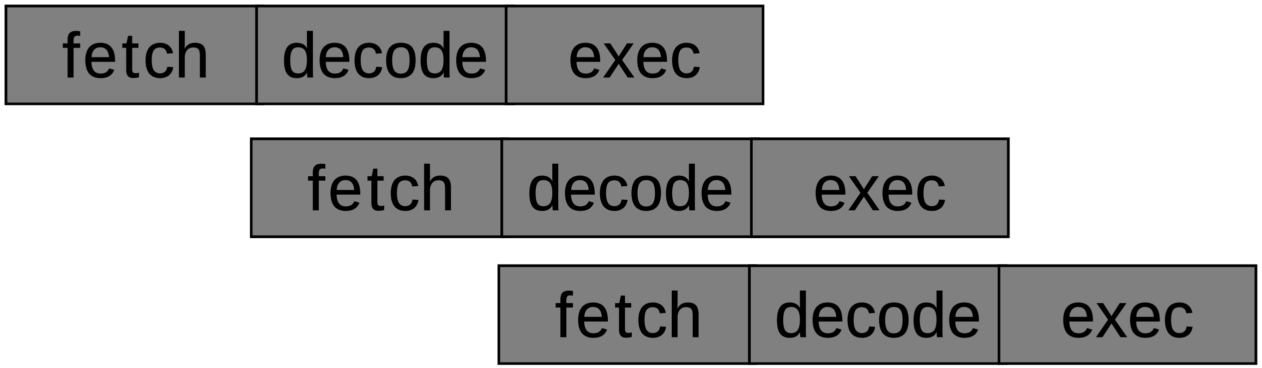
Impossible to use direct addressing



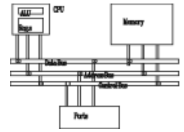
encoding for data transfer instructions

PC-relative addressing

```
main:
        MOV   R0, #0
        ADR   R1, a          @ add  r1, pc, #4
        STR   R0, [R1]
PC ---> SWI #11
a:      .word  100
        .end
```



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