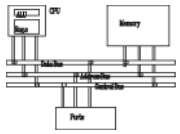
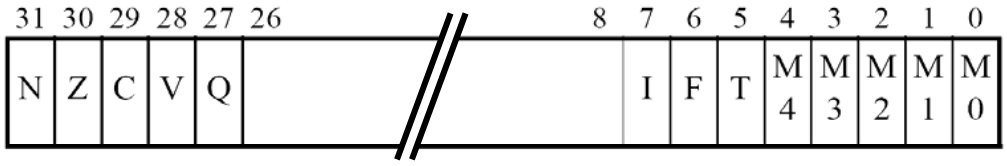
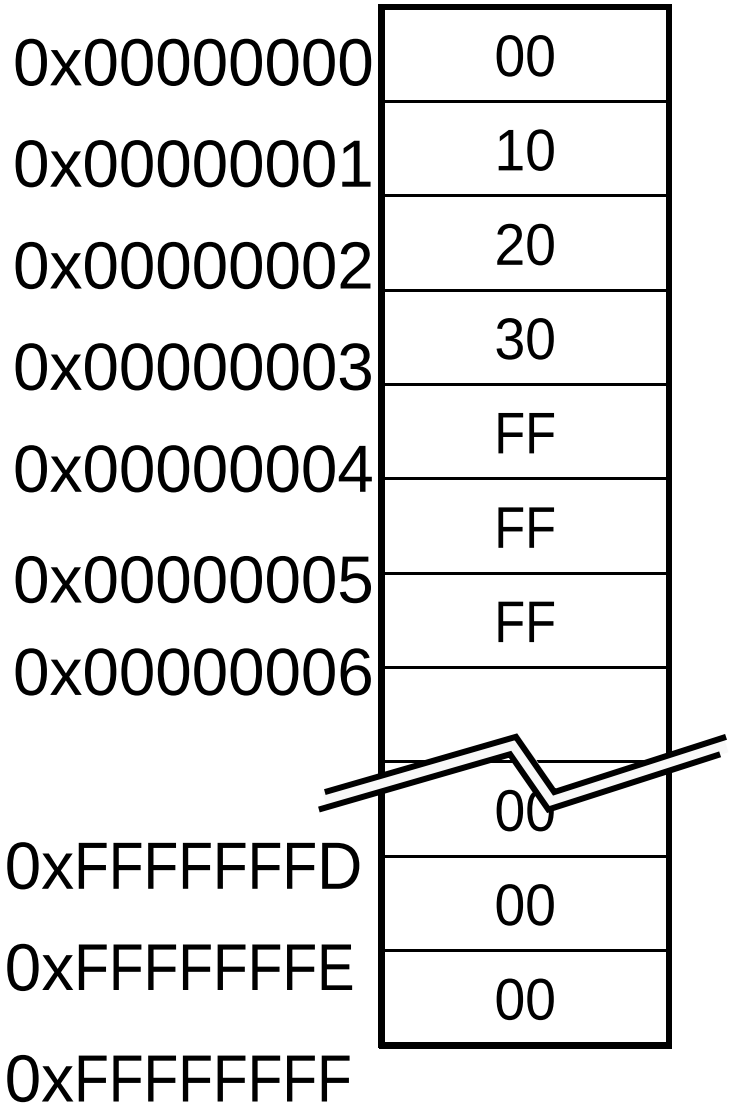
ARM programmer model

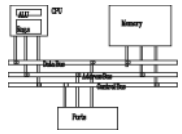


R0	R1	R2	R3
R4	R5	R6	R7
R8	R9	R10	R11
R12	R13	R14	PC





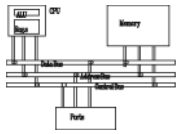
Instruction Encoding



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

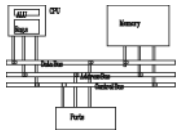
	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
Data processing immediate shift	cond [1]	0 0 0	opcode	S	Rn	Rd	shift amount	shift	0	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0 0 0	1 0 x x	0	x x x x x x x x	x x x x	x x x	x x	0	x x x x
Data processing register 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	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	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	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	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	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 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	x x x x

Instruction Classification



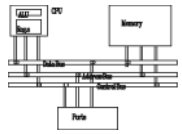
- Data processing
- Data transfer
- Flow control

Conditional execution



```
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

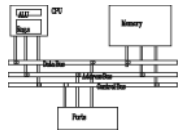


- Almost all ARM instructions have a condition field which allows it to be executed conditionally.

```
movcs R0, R1
```

Mnemonic	Condition	Mnemonic	Condition
CS	*C*arry *S*et	CC	*C*arry *C*lear
EQ	*Eq*ual (Zero Set)	NE	*N*ot *E*qual (Zero Clear)
VS	O*v*erflow *S*et	VC	O*v*erflow *C*lear
GT	*G*reater *T*han	LT	*L*ess *T*han
GE	*G*reater Than or *E*qual	LE	*L*ess Than or *E*qual
PL	*Pl*us (Positive)	MI	*Mi*nus (Negative)
HI	*Hi*gher Than	LO	*Lo*wer Than (aka CC)
HS	*H*igher or *S*ame (aka CS)	LS	*L*ower or *S*ame

Variants of an instruction



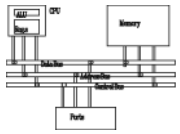
MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

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

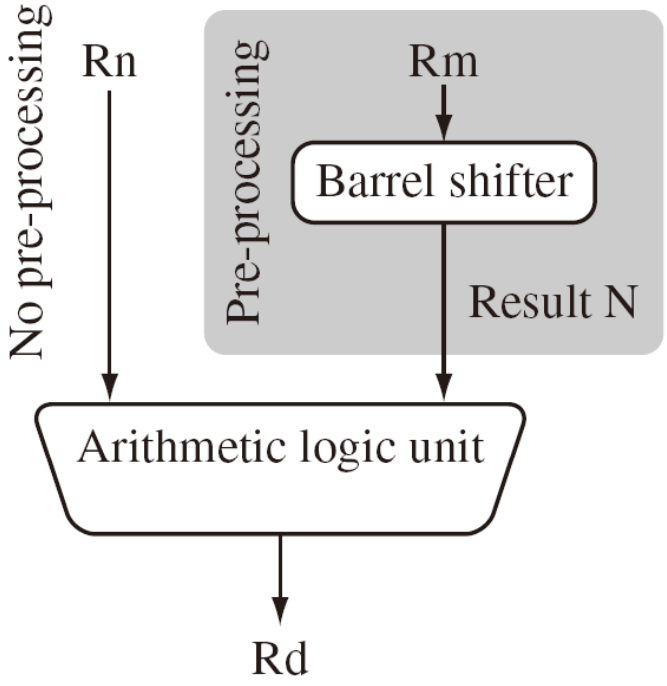
move negated

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

Shifted register operands

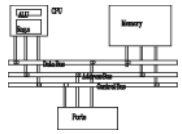


- One operand to ALU is routed through the Barrel shifter. Thus, the operand can be modified before it is used. Useful for fast multipliation and dealing with lists, table and other complex data structure. (similar to the displacement addressing mode in CISC.)



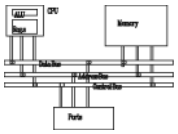
- Some instructions (e.g. **`MUL, CLZ, QADD`**) do not read barrel shifter.

Shifted register operands



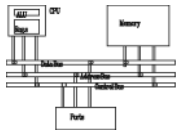
Mnemonic	Description	Shift	Result
LSL	logical shift left	*x*LSL *y*	$x \ll y$
LSR	logical shift right	*x*LSR *y*	$(\text{unsigned})x \gg y$
ASR	arithmetic right shift	*x*ASR *y*	$(\text{signed})x \gg y$
ROR	rotate right	*x*ROR *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$
RRX	rotate right extended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

Logical shift left



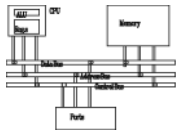

```
MOV  R0, R2, LSL #2 @ R0:=R2<<2
                    @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x000000C0
       R2=0x00000030
```

Logical shift right



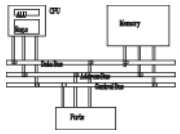

```
MOV  R0, R2, LSR #2 @ R0:=R2>>2
                    @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x0000000C
       R2=0x00000030
```

Arithmetic shift right



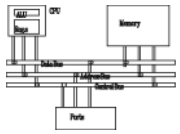

```
MOV   R0, R2, ASR #2 @ R0:=R2>>2
                     @ R2 unchanged
Example: 1010 0…0 0011 0000
Before R2=0xA0000030
After  R0=0xE800000C
       R2=0xA0000030
```

Rotate right



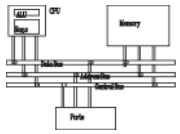

```
MOV  R0, R2, ROR #2 @ R0:=R2 rotate
                    @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031
After  R0=0x4000000C
       R2=0x00000031
```

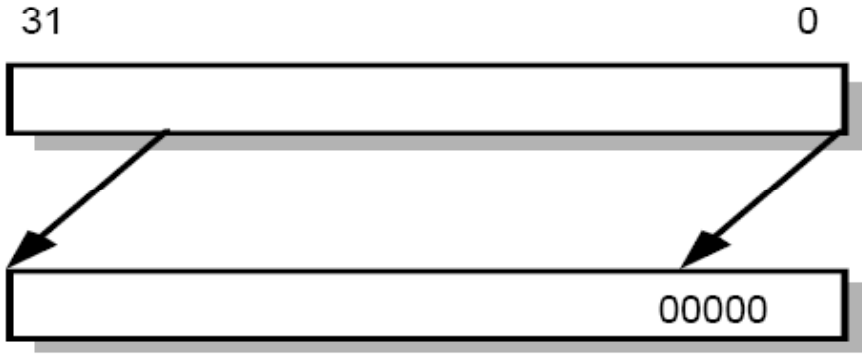
Rotate right extended



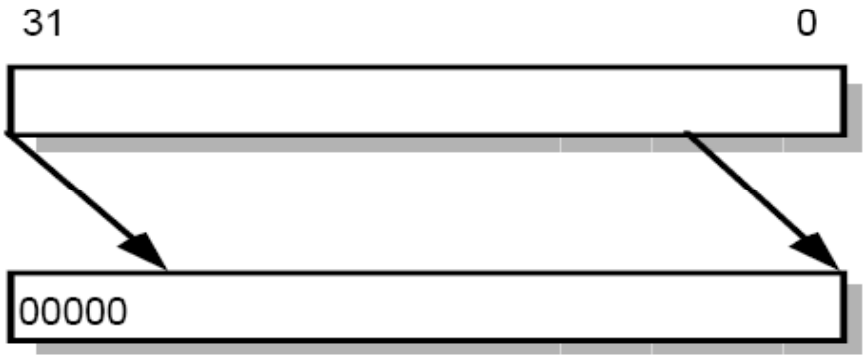

```
MOV  R0, R2, RRX      @ R0:=R2 rotate
                      @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031, C=1
After  R0=0x80000018, C=1
       R2=0x00000031
```

Shifted register operands

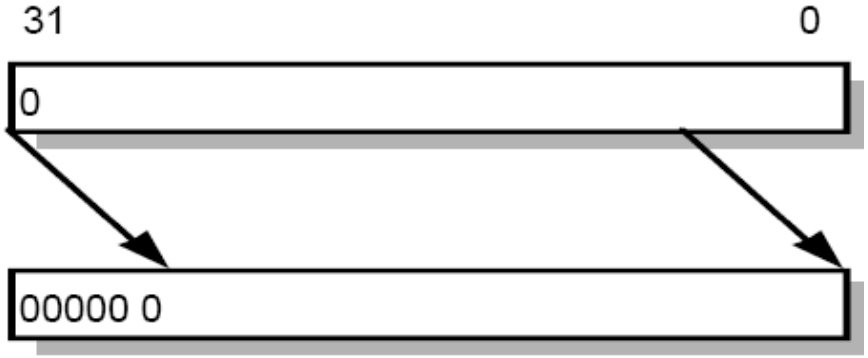




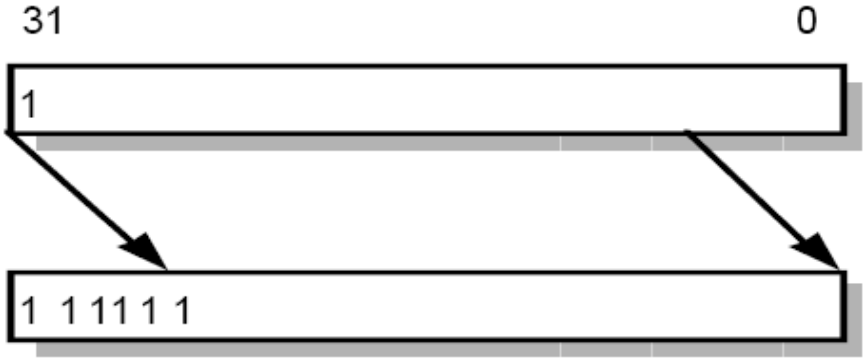
LSL #5



LSR #5

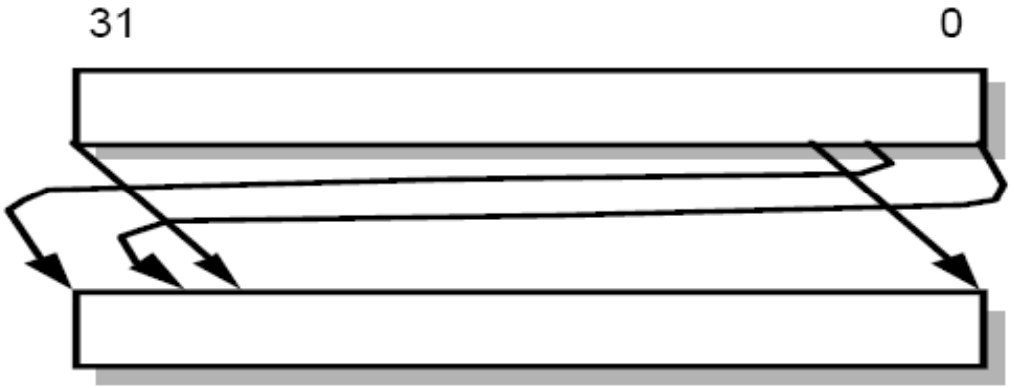


ASR #5 , positive operand

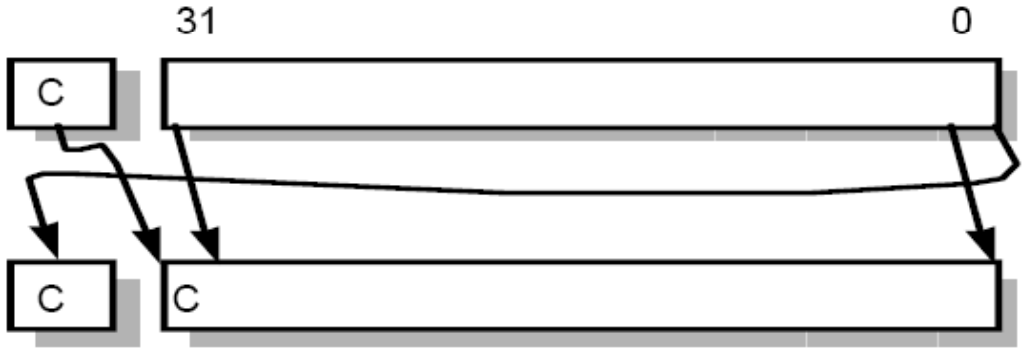


ASR #5 , negative operand

Shifted register operands

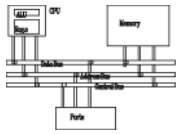


ROR #5



RRX

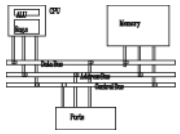
Shifted register operands



- 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 using Shifted Operands



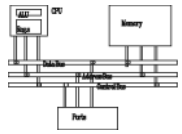
```
MOV   R1, #35
MUL   R2, R0, R1
```

R2=35xR0

or

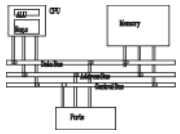
```
ADD   R0, R0, R0, LSL #2   @ R0'=5xR0
RSB   R2, R0, R0, LSL #3   @ R2 =7xR0'
```

Shifted register operands

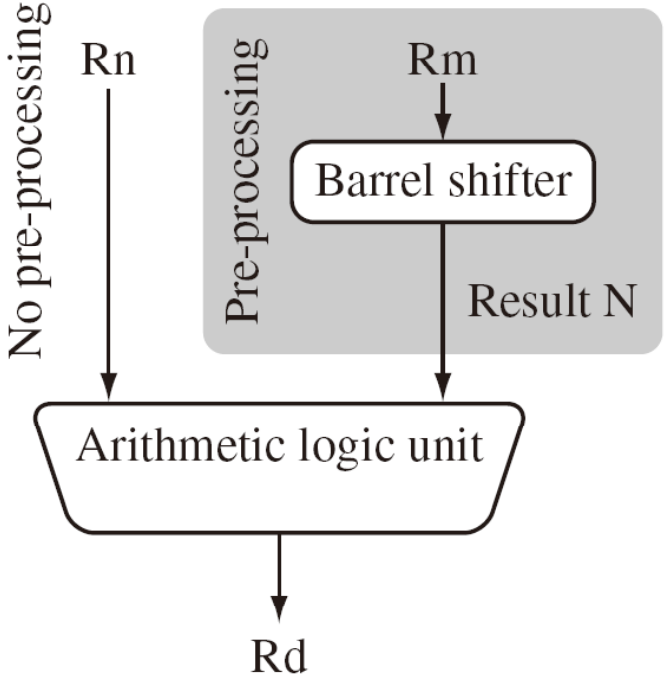


N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm,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

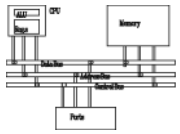
Data processing

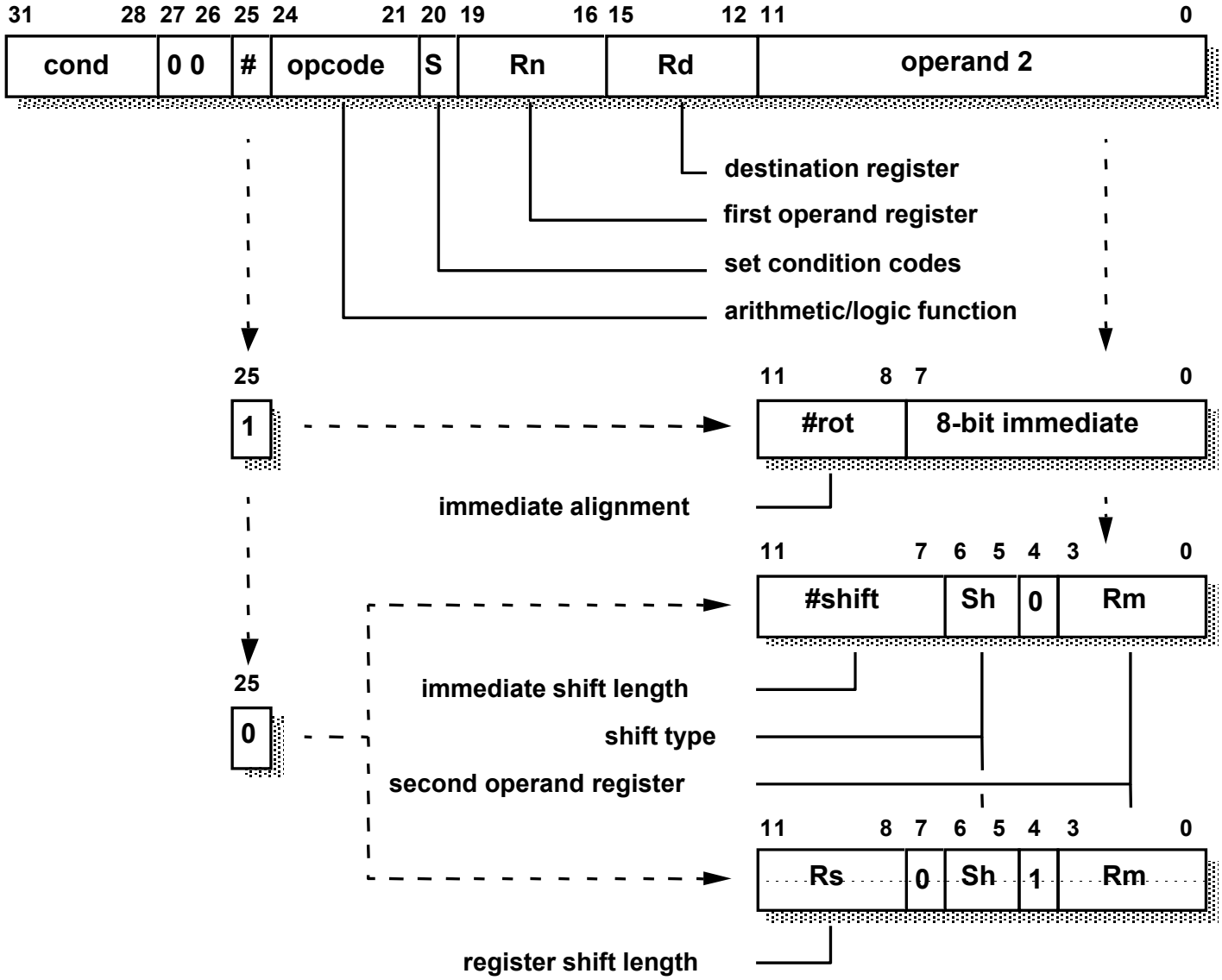


- They are move, arithmetic, logical, comparison and multiply instructions.
- Most data processing instructions can process one of their operands using the barrel shifter.
- General rules:
  - All operands are 32-bit, coming from registers or literals.
  - The result, if any, is 32-bit and placed in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

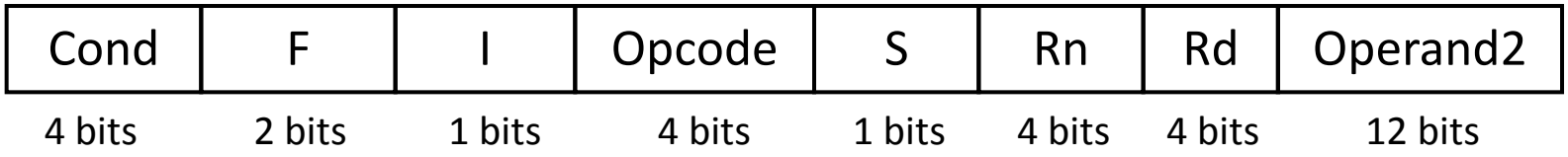


Encoding data processing instructions

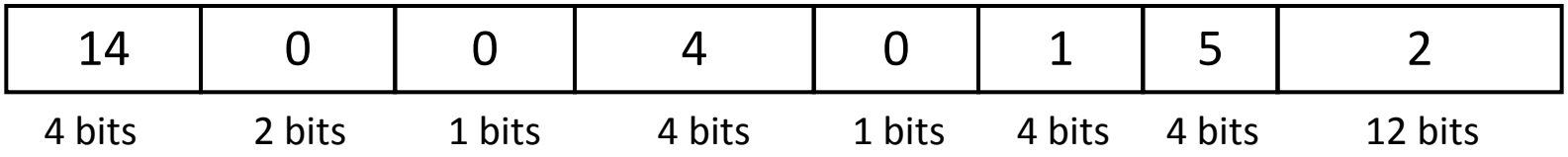




DP Instruction Example

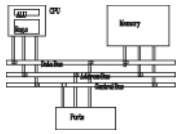



```
ADD r5,r1,r2 ; r5 = r1 + r2
```



$$11100000100000010101000000000010_2$$

Arithmetic

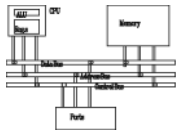


- Add and subtraction

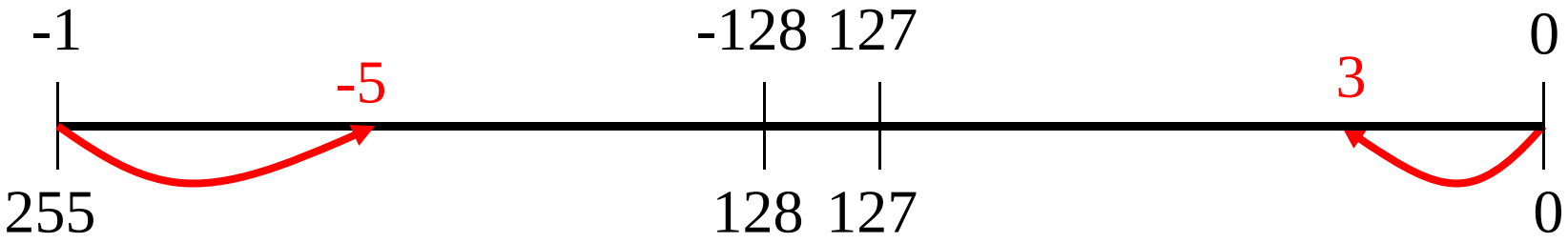
Syntax: <instruction>{<cond>}{S} Rd, Rn, N

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



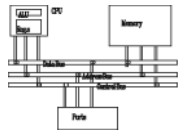
- **ADD R0, R1, R2 @ R0 = R1+R2**
- **ADC R0, R1, R2 @ R0 = R1+R2+C**
- **SUB R0, R1, R2 @ R0 = R1-R2**
- **SBC R0, R1, R2 @ R0 = R1-R2-!C**
- **RSB R0, R1, R2 @ R0 = R2-R1**
- **RSC R0, R1, R2 @ R0 = R2-R1-!C**



3-5=3+(-5) ⟶ sum<=255 ⟶ C=0 ⟶ borrow

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

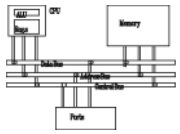
Arithmetic



```
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
```

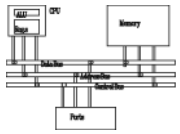
Arithmetic



```
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



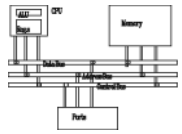
- Any data processing instruction can set the condition codes if the programmers wish it to

64-bit addition

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

	R1	R0
+	R3	R2
	R3	R2

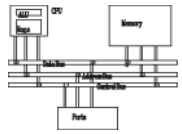
Logical



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)`**

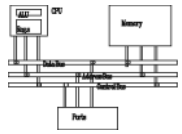
bit clear: **R2** is a mask identifying which bits of **R1** will be cleared to zero

`R1=0x11111111` **`R2=0x01100101`**

`BIC R0, R1, R2`

`R0=0x10011010`

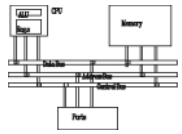
Logical



```
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

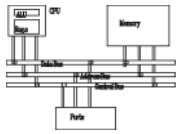


- These instructions do not generate a result, but set condition code bits (N, Z, C, V) in CPSR. Often, a branch operation follows to change the program flow.

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 $Rn \ \hat{} \ N$
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`**

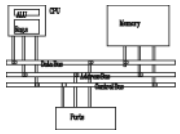
bit test

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

test equal

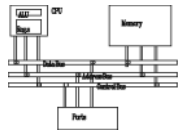
- **`TEQ  R1, R2      @ set cc on R1 xor R2`**

Comparison



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

        CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

Multiplication



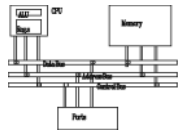
```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

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	$[RdHi, RdLo] = Rm^* Rs$

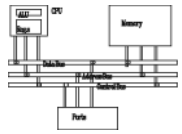
Multiplication



- **`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
```

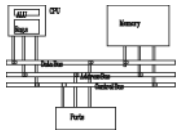
- Multiply with a constant can often be more efficiently implemented using shifted register operand

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

or

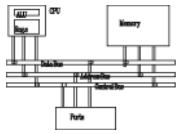
```
ADD  R0, R0, R0, LSL #2  @ R0'=5xR0
RSB  R2, R0, R0, LSL #3  @ R2 =7xR0'
```

Multiplication



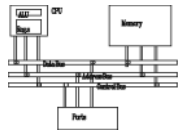
```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        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

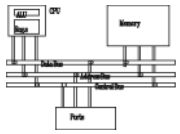


- Determine the instruction to be executed next

```
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



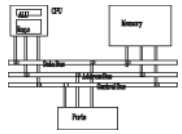
- Branch instruction

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

- Conditional branches

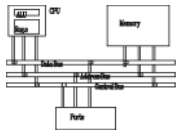
```
          MOV   R0, #0
loop:          …
          ADD   R0, R0, #1
          CMP   R0, #10
          BNE   loop
```

Branch conditions



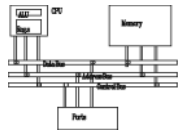
Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater 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

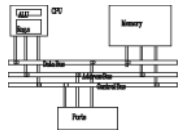
Branch and link



- **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



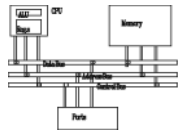
```
        BL    sub1      @ call sub1
        …
```

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



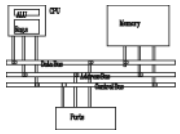
```
            CMP   R0, #5
            BEQ   bypass      @ if (R0!=5) {
            ADD   R1, R1, R0  @   R1=R1+R0-R2
            SUB   R1, R1, R2  @ }
bypass:     …
```

```
            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
```