ARM ARCHITECTURE AND INSTRUCTION SET

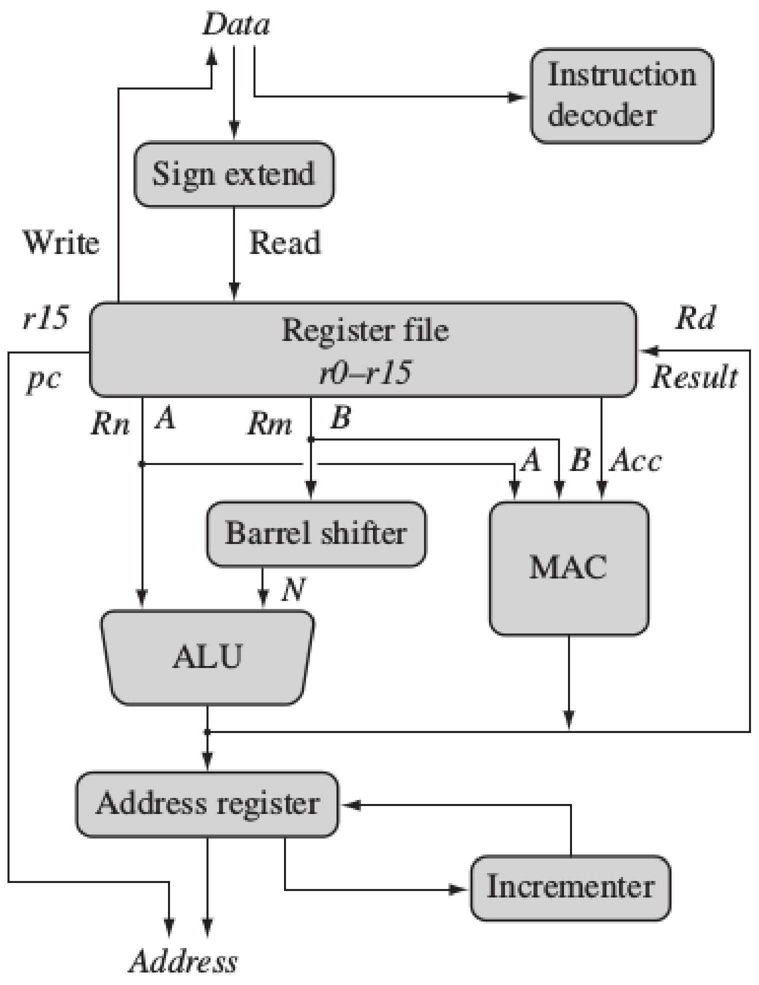
EL-GY 6483 Real Time Embedded Systems

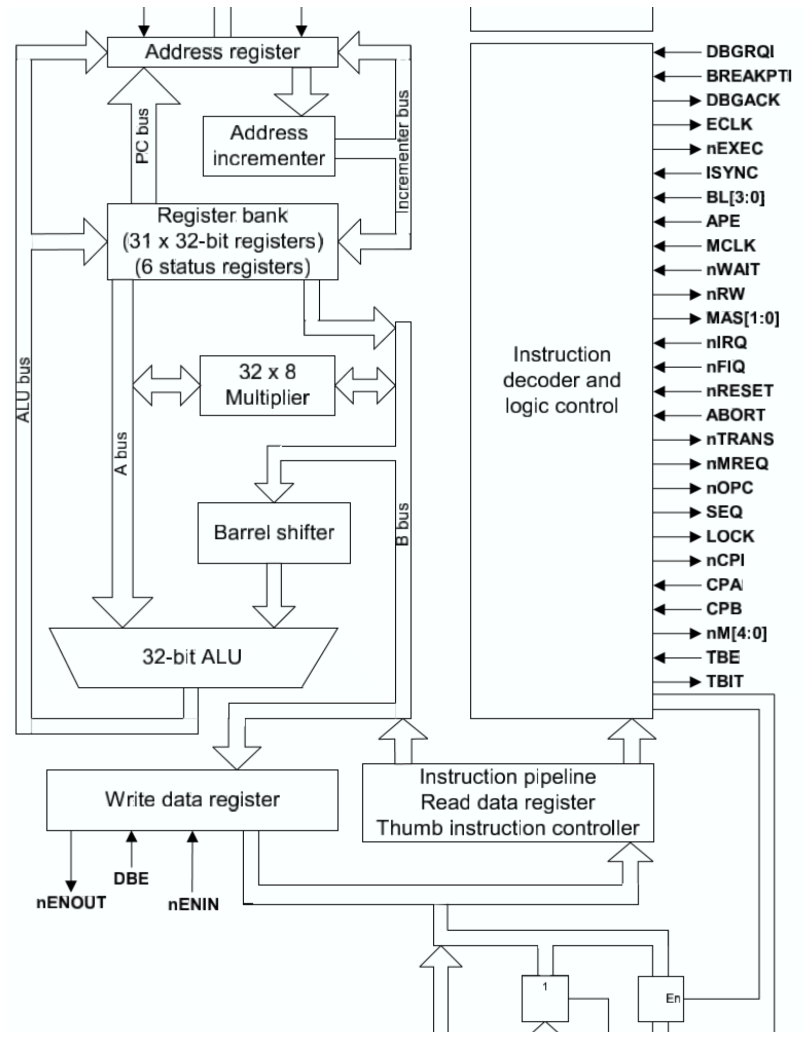
MAIN FEATURES

- 32-bit instructions
- Most executed in single cycle
- Most can be conditionally executed
- Load/store

Also has a compressed 16-bit instruction set (Thumb)

DATAFLOW


Data
Instruction decoder
Sign extend
Write
Read
r15
Register file r0–r15
Rd
pc
Result
Rn
A
Rm
B
A
B
Acc
Barrel shifter
MAC
N
ALU
Address register
Incrementer
Address


Address register
PC bus
Address incrementer
Incrementer bus
Register bank (31 x 32-bit registers) (6 status registers)
ALU bus
A bus
32 x 8 Multiplier
B bus
Barrel shifter
32-bit ALU
Write data register
Instruction pipeline Read data register Thumb instruction controller
Instruction decoder and logic control
nENOUT
DBE
nENIN
DBGRQI
BREAKPTI
DBGACK
ECLK
nEXEC
ISYNC
BL[3:0]
APE
MCLK
nWAIT
nRW
MAS[1:0]
nIRQ
nFIQ
nRESET
ABORT
nTRANS
nMREQ
nOPC
SEQ
LOCK
nCPI
CPA
CPB
nM[4:0]
TBE
TBIT
En

PROCESSOR MODES

Six+ operating modes, each with own registers

- Most tasks run in user mode
- FIQ – entered on high-priority interrupt
- IRQ – entered on a normal interrupt
- Supervisor (reset), Abort (memory access violations), Undef (undefined instructions)
- System mode added in ARMv4, gives privileged access to same registers as user mode

REGISTERS

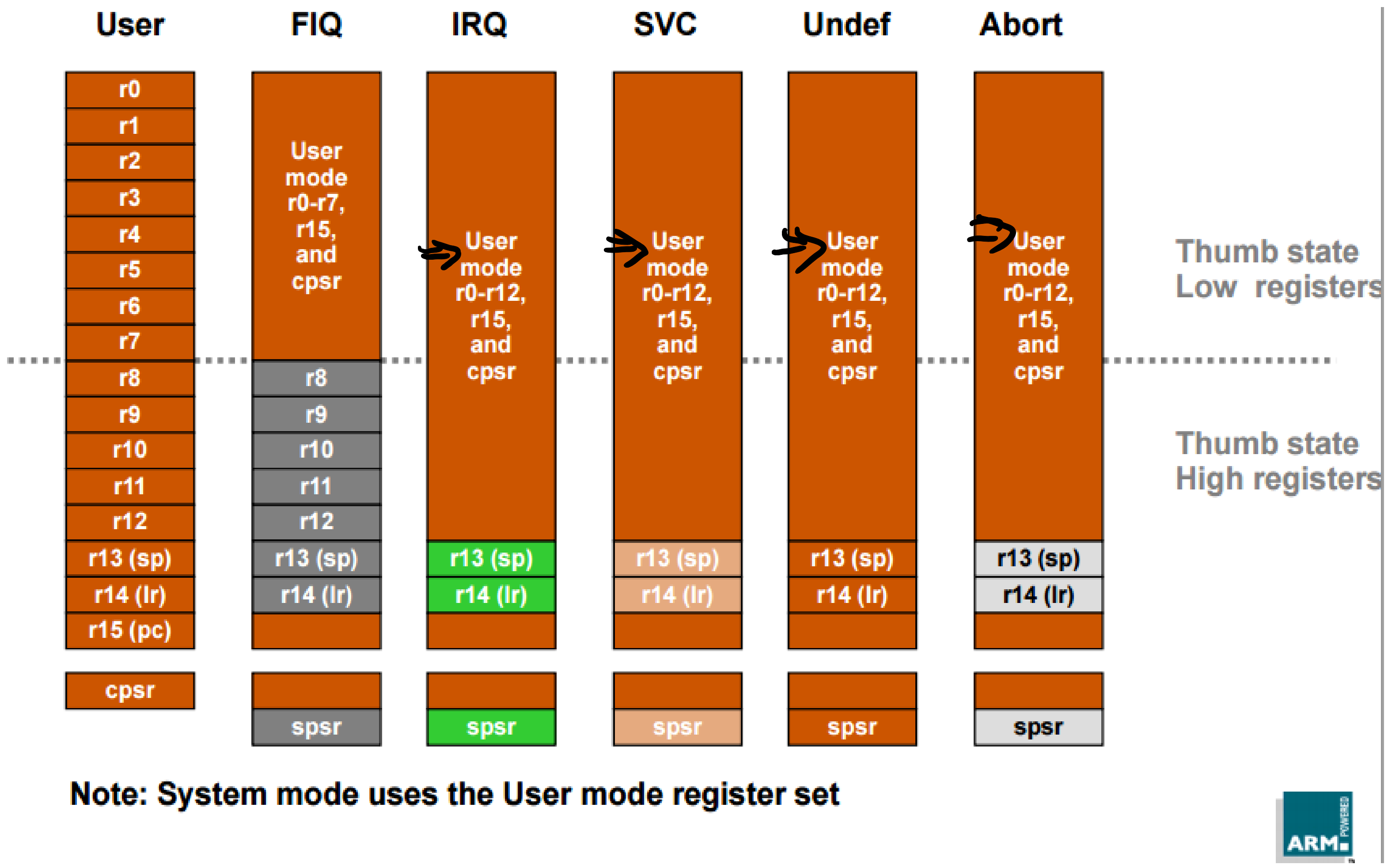
ARM: 37 registers, each 32 bits long

- Each mode can access:
  - A specific set of r0-r12 registers
  - A specific r13 (SP) and r14 (LR)
  - R15 (PC)
  - CPSR
- Privileged modes can also access a SPSR
- Banked registers?

Thumb

- Thumb is a 16-bit instruction set
  - Optimized for code density from C code
  - Improved performance form narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has two execution states –ARM and Thumb
  - Switch between them using BX instruction
- Thumb has characteristic features:
  - Most Thumb instruction are executed unconditionally
  - Many Thumb data process instruction use a 2-address format
  - Thumb instruction formats are less regular than ARM instruction formats, as a result of the dense encoding.

REGISTERS



User
FIQ
IRQ
SVC
Undef
Abort
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
User mode r0-r7, r15, and cpsr
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
Thumb state Low registers
Thumb state High registers

Note: System mode uses the User mode register set

Processor Modes

- The ARM has six operating modes:
  - User (unprivileged mode under which most tasks run)
  - FIQ (entered when a high priority (fast) interrupt is raised)
  - IRQ (entered when a low priority (normal) interrupt is raised)
  - Supervisor (entered on reset and when a Software Interrupt instruction is executed)
  - Abort (used to handle memory access violations)
  - Undef (used to handle undefined instructions)
- ARM Architecture Version 4 adds a seventh mode:
  - System (privileged mode using the same registers as user mode)

Accessing Registers using ARM Instructions

- All instructions can access r0-r14 directly.
- Most instructions also allow use of the PC. r15

- Specific instructions to allow access to CPSR and SPSR.

- Note : When in a privileged mode, it is also possible to load-store the (banked out) user mode registers to or from memory

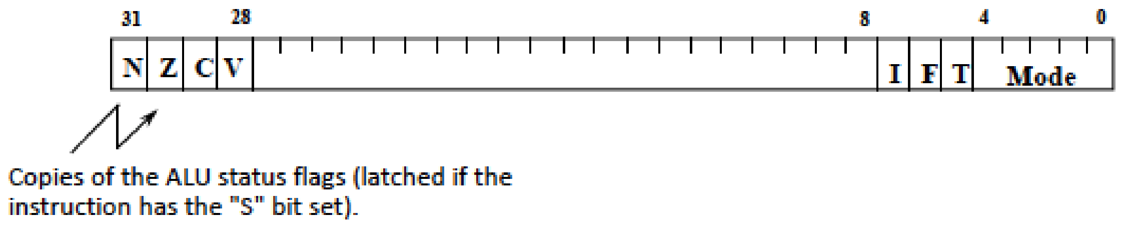
PC

The Program Counter (R15)

- When the processor is executing in ARM state:
    - All instructions are 32 bits in length
    - All instructions must be word aligned
    - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).
- R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link operations are performed, calculated from the PC.
- Thus to return from a linked branch:

```
MOV r15, r14
   or
MOV pc, lr
```

The Program Status Registers (CPSR and SPSR)



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

- **Condition Code Flags**

N = **N**egative result from ALU flag
Z = **Z**ero Result from ALU flag
C = ALU operation **C**arried out
V = ALU operation o**V**erflowed

- **Interrupt Disable Bits**

I = 1, disables the IRQ
F = 1, disables the FIQ

- **Mode Bits**

M[4:0] define the processor mode.

- **T bit (Architecture v4T only)**

T = 0, Processor in ARM state
T = 1, Processor in Thumb state

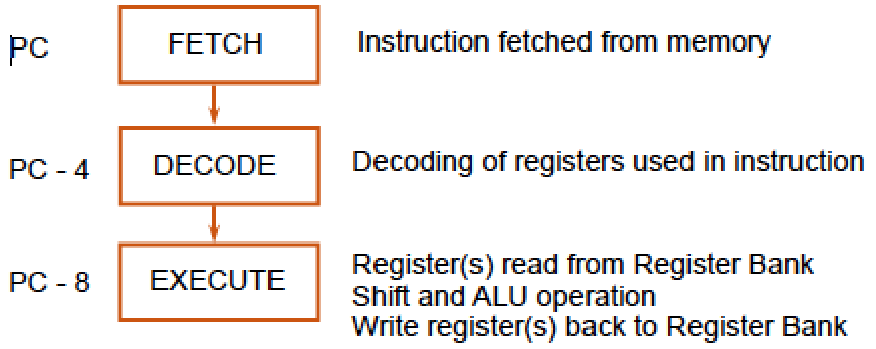
Condition Flags

	Logical Instruction	**Arithmetic Instruction**
Flag		
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater than 32 bits
oVerflow (V='1')	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers

Updated by explicit comparison instructions (eg. CMP) and those that use the optional S to specify if the condition code flags must be updated (e.g. ADDS).

ARM Pipeline

- The ARM uses a pipeline in order to increase the speed of the flow of instructions to the processor.
  - Allows several operations to be undertaken simultaneously, rather than serially.


PC
FETCH
Instruction fetched from memory
PC - 4
DECODE
Decoding of registers used in instruction
PC - 8
EXECUTE
Register(s) read from Register Bank
Shift and ALU operation
Write register(s) back to Register Bank

- Rather than pointing to the instruction being executed, the PC points to the instruction being fetched.

INSTRUCTION SYNTAX

```
<operation> cond flags Rd,Rn,Operand2
```

Basic syntax followed by many instructions. From left to right:

- A three-letter mnemonic, e.g. MOV or ADD.
- An optional two-letter condition code, e.g. EQ or CS.
- An optional additional flag
- The destination register
- First operand register
- Second (more flexible) operand register

ARM Instruction Set Format

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	Instruction Type
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2												Data processing
Condition				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				Multiply
Condition				0	0	0	0	1	U	A	S	Rd HIGH				Rd LOW				Rs				1	0	0	1	Rm				Long Multiply
Condition				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				Swap
Condition				0	1	I	P	U	B	W	L	Rn				Rd				OFFSET												Load/Store - Byte/Word
Condition				1	0	0	P	U	B	W	L	Rn				REGISTER LIST																Load/Store Multiple
Condition				0	0	0	P	U	1	W	L	Rn				Rd				OFFSET 1				1	S	H	1	OFFSET 2				Halfword Transfer Imm Off
Condition				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Transfer Reg Off
Condition				1	0	1	L	BRANCH OFFSET																								Branch
Condition				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				Branch Exchange
Condition				1	1	0	P	U	N	W	L	Rn				CRd				CPNum				OFFSET								COPROCESSOR DATA XFER
Condition				1	1	1	0	Op-1				CRn				CRd				CPNum				OP-2			0	CRm				COPROCESSOR DATA OP
Condition								OP-1			L	CRn				Rd				CPNum				OP-2			1	CRm				COPROCESSOR REG XFER
Condition				1	1	1	1	SWI NUMBER																								Software Interrupt

N 当用两个补码表示的带符号数进行运算时，N=1表示运算的结果为负数；N=0表示运算的结果为正数或零.

Z Z=1表示运算的结果为零，Z=0表示运算的结果非零。

C 可以有4种方法设置C的值：

加法运算（包括CMN）：当运算结果产生了进位时（无符号数溢出），C=1，否则C=0。

减法运算（包括CMP）：当运算时产生了借位时（无符号数溢出），C=0，否则C=1。

对于包含移位操作的非加/减运算指令，C为移出值的最后一位。

对于其它的非加/减运算指令，C的值通常不会改变。

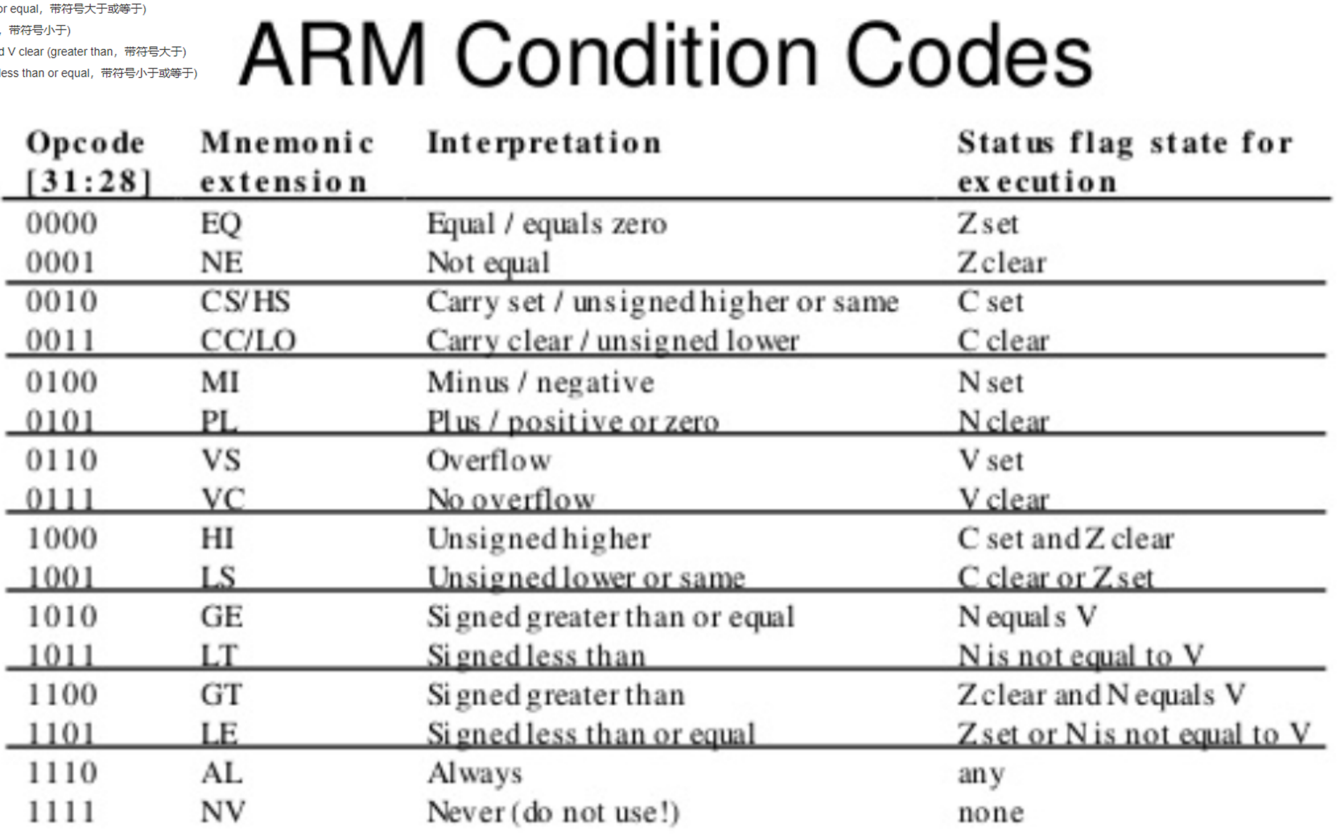
V 可以有2种方法设置V的值：

对于加减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1表示符号位溢出

对于其它的非加/减运算指令，V的值通常不会改变。

0000 = EQ - Z set (equal，相等)
0001 = NE - Z clear (not equal，不相等)
0010 = CS - C set (unsigned higher or same，无符号大于或等于)
0011 = CC - C clear (unsigned lower，无符号小于)
0100 = MI - N set (negative，负数)
0101 = PL - N clear (positive or zero，正数或零)
0110 = VS - V set (overflow，溢出)
0111 = VC - V clear (no overflow，未溢出)
1000 = HI - C set and Z clear (unsigned higher，无符号大于)
1001 = LS - C clear or Z set (unsigned lower or same，无符号小于或等于)
1010 = GE - N set and V set, or N clear and V clear (greater or equal，带符号大于或等于)
1011 = LT - N set and V clear, or N clear and V set (less than，带符号小于)
1100 = GT - Z clear, and either N set and V set, or N clear and V clear (greater than，带符号大于)
1101 = LE - Z set, or N set and V clear, or N clear and V set (less than or equal，带符号小于或等于)
1110 = AL - always
1111 = NV - never

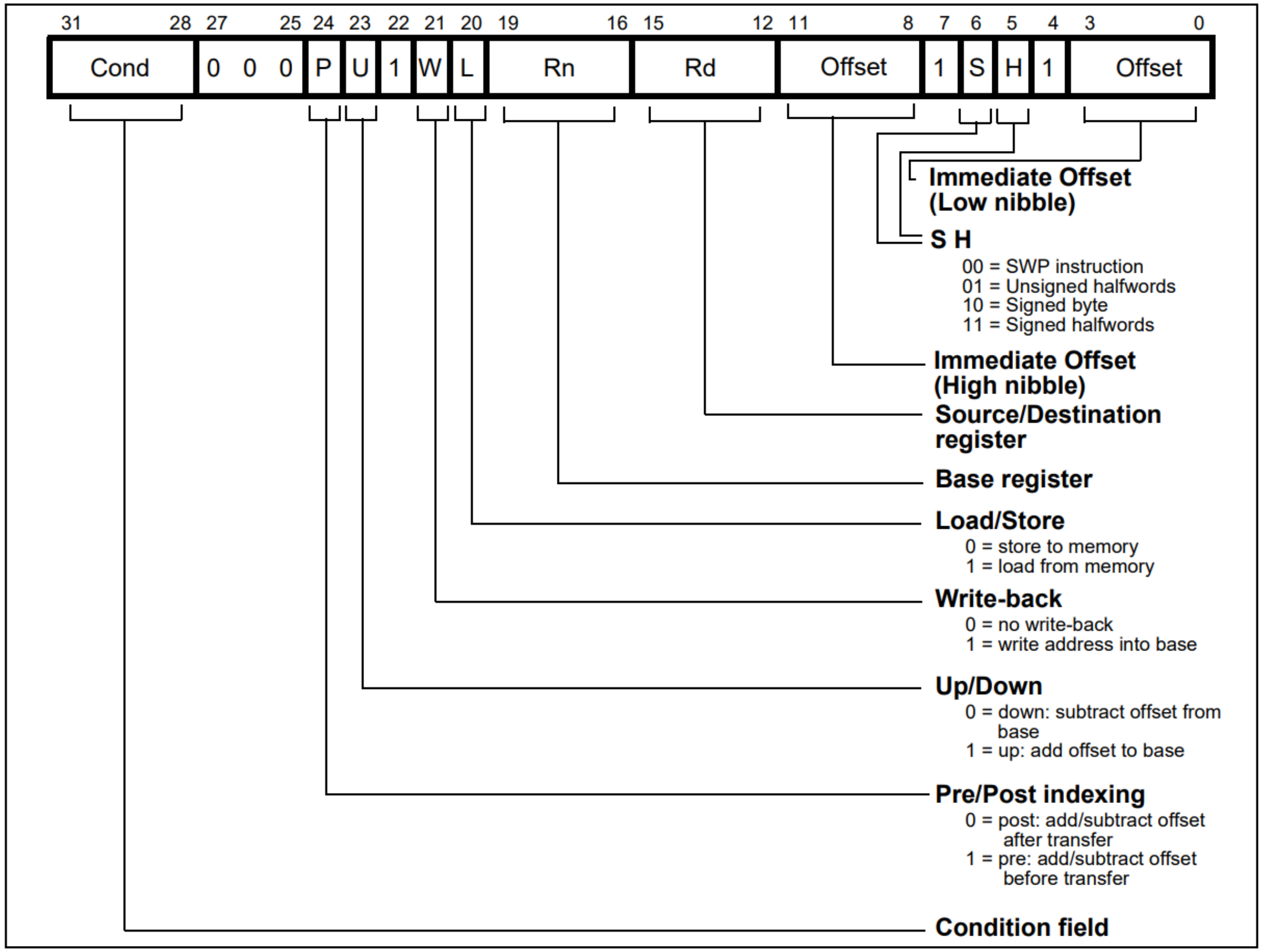
RM Instruction Set Format



ARM Condition Codes

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

ARM Instruction Set Format



COMMON OPERATIONS

- Load and store instructions: LDR, STR, etc.
- Move instructions: MOV, etc.
- Branch (jump) instructions: B, BL, BX, BLX, etc.
- Stack push and pop instructions: PUSH, POP, etc.
- Arithmetic operations: ADD, SUB, MUL, SDIV, UDIV, etc.
- No operation (null operation): NOP

MOVE OPERATIONS

MOV, MVN: “Move” and “Move NOT”; move not does a bitwise logical NOT operation before copying the value to the destination register.

- MOV R0, R1; copy the contents of R1 to R0
- MOV R0, #10; set R0 to 10

  0xFFFF FFFF − R_1

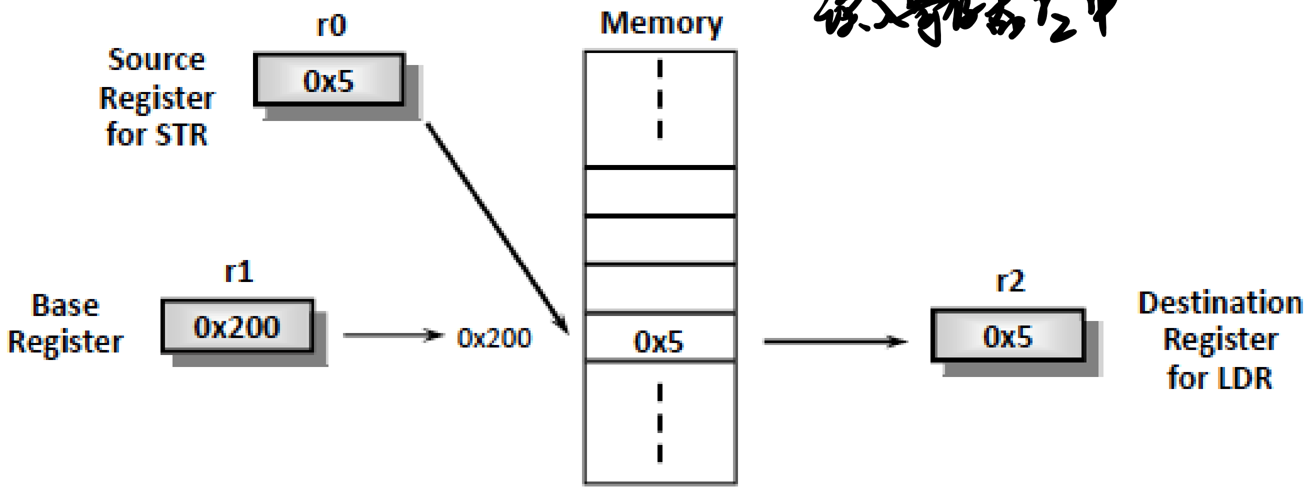
- MVN R0, R1; set R0 to bitwise NOT of contents of R1
- MVN R0, #1; set R0 to 0xFFFFFFFE (bitwise NOT of x1)

Load and Store Word or Byte: Base Register

The memory location to be accessed is held in a base register

- STR r0, [r1] ; Store contents of r0 to location pointed to 将r_0中的数据输入到存储在R_1上的地址中
- ; by contents of r1.
- LDR r2, [r1] ; Load r2 with contents of memory location
- ; pointed to by contents of r1.

将存储地址为R_1的字数据读入寄存器r_2中



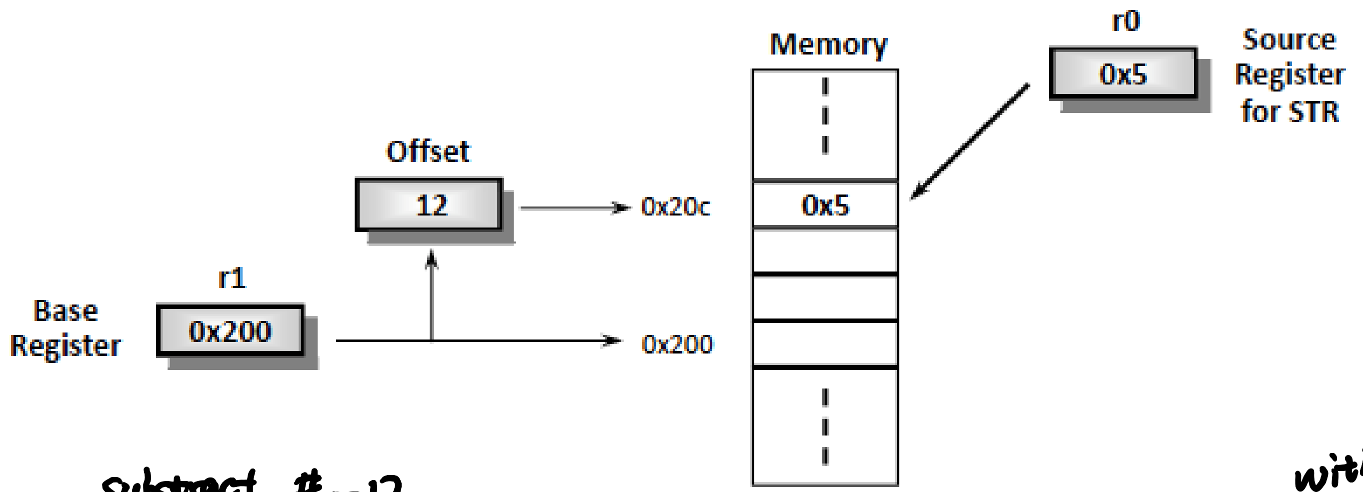
Load/Store: Offsets from the Base Register

- Accessing a location offset from the base register pointer.
- Either:
  - An unsigned 12bit immediate value (ie 0 -4095 bytes).
  - A register, optionally shifted by an immediate value
- This can be either added or subtracted from the base register:
  - Prefix the offset value or register with '+' (default) or '-'.
- This offset can be applied:
  - before the transfer is made: *Pre-indexed addressing*
    - optionally auto-incrementing the base register, by postfixing the instruction with an '!'.
  - after the transfer is made: *Post-indexed addressing* causing the base register to be auto-incremented.

!

Example: Load/Store : Pre-indexed Addressing

Example: STR r0, [r1,#12]



subsstract. # −12

with "!"

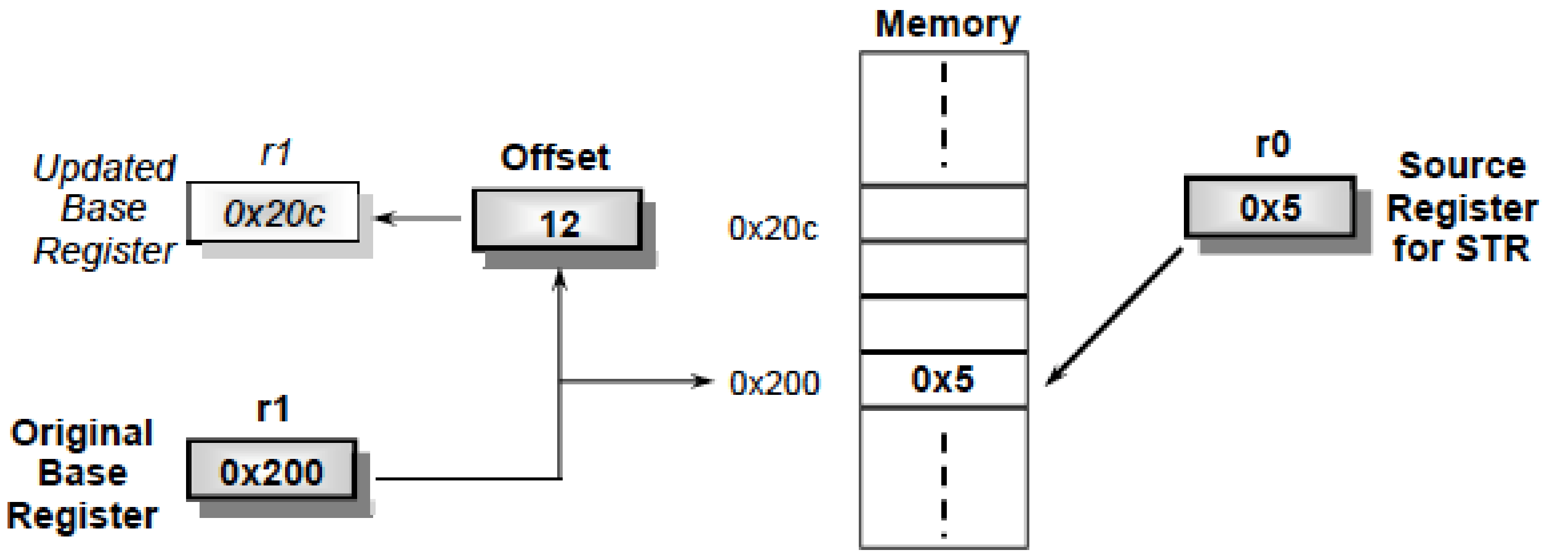
- To store to location 0x1f4 instead use: STR r0, [r1,#-12]
- To auto-increment base pointer to 0x20c use: STR r0, [r1, #12]! add #12, $r_1 = r_1 + 12$
- If r2 contains 3, access 0x20c by multiplying this by 4:

  STR r0, [r1, r2, LSL #2]

STR r0, [$r_1 + r_2 \times 4$]

Example: Load and Store: Post-indexed Addressing

Example: STR r0, [r1], #12



- To auto-increment the base register to location 0x1f4 instead use:
  - STR r0, [r1], #-12
- If r2 contains 3, auto-increment base register to 0x20c by multiplying this by 4:
  - STR r0, [r1], r2, LSL #2

write increment number outside [ ]

LOAD AND STORE INSTRUCTIONS

- LDR R1, [R0] ; load into R1 the content of the memory location whose address is in R0
- STR R1, [R0] ; store the contents of R1 into memory location whose address is in R0 R0 stores the address.
- LDR R1, [R0, #4] ; #4 specifies an offset value, load into R1 the content of the memory location whose address is given by the value R0 + 4
- LDR R1, [R0, #4]! ; #4 specifies an offset value, increment R0 by 4, load into R1 the content of the memory location whose address is given by the new contents of R0 auto-increment
- What about: STR R1, [R0] ,#4

Register names can be in upper case or lower case. Anything after ; is a comment.

The Barrel Shifter

- The ARM doesn’t have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.
- So what operations does the barrel shifter support?

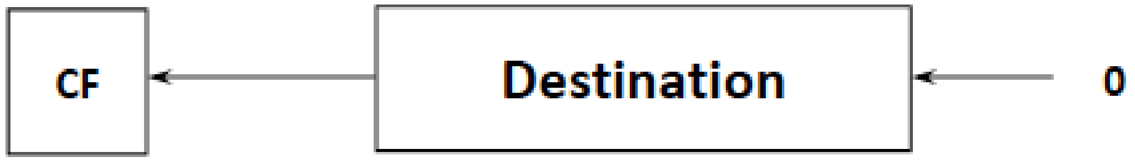
Barrel Shifter -Left Shift

- Shifts left by the specified amount (multiplies by powers of two)

e.g.

LSL #5 => multiply by 32

Logical Shift Left (LSL)



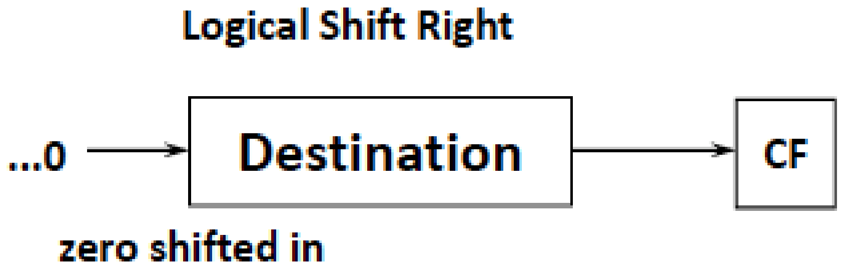
LSR, ASR

Logical Shift Right (LSR)

Shifts right by the specified amount (divides by powers of two) e.g.

不管符号
LSR, LSL

LSR #5 = divide by 32

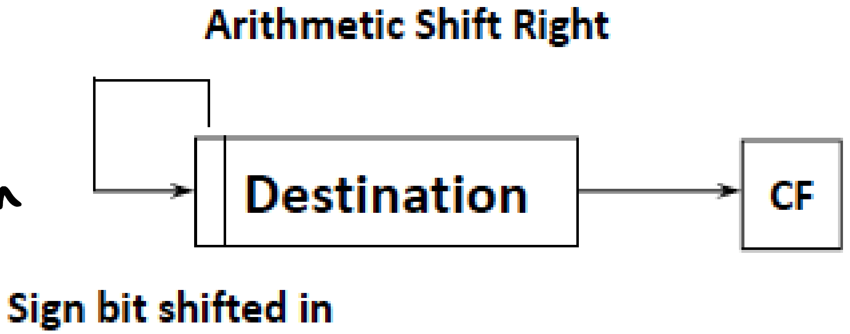


Arithmetic Shift Right (ASR)

Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g.

Consider the sign

ASR #5 = divide by 32



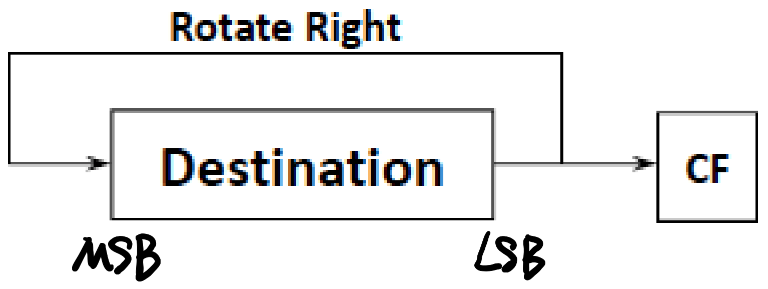
Rotations

Rotate Right (ROR)

Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

e.g. ROR #5

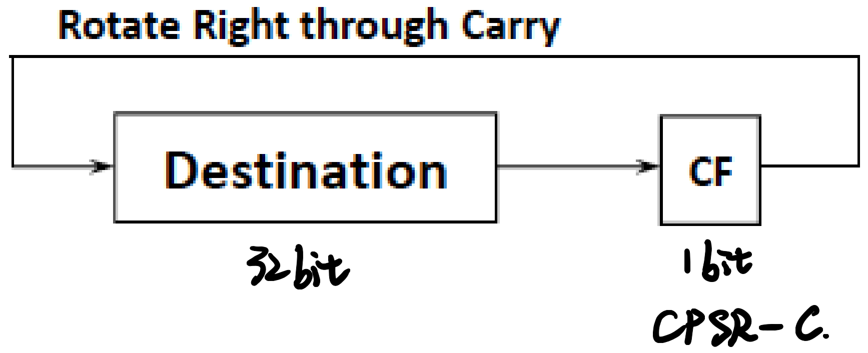
Note the last bit rotated is also used as the Carry Out.



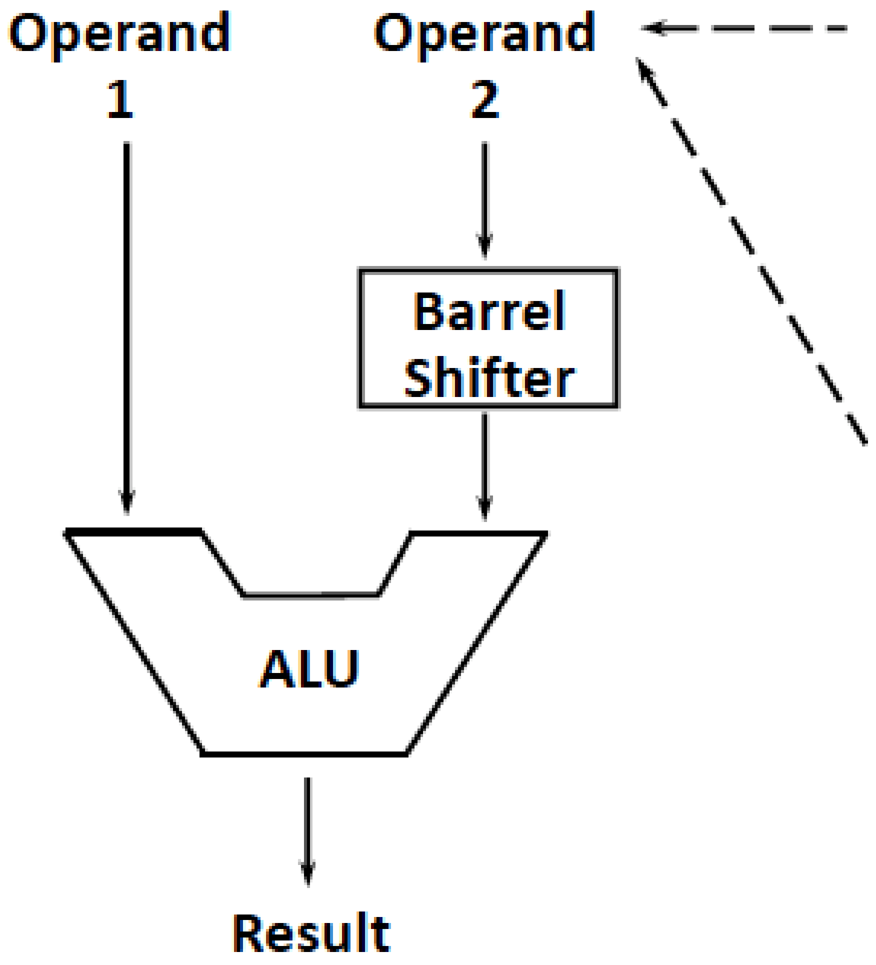
Move r0, r0, ROR#16.
32bit. swap the halftop with halfbotton

Rotate Right Extended (RRX)

This operation uses the CPSR C flag as a 33rd bit.



Using the Barrel Shifter: The Second Operand



Register, optionally with shift operation applied.

Shift value can be either be:

5 bit unsigned integer

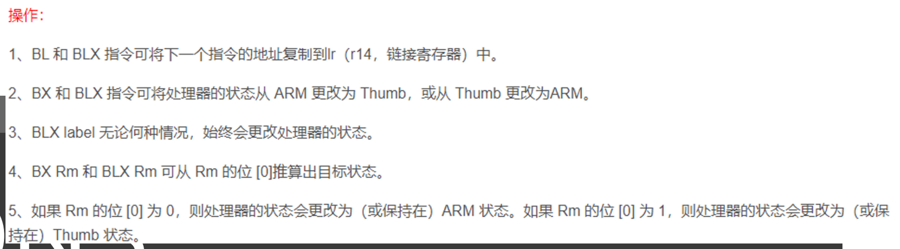
Specified in bottom byte of another register.

**Immediate value

8 bit number

Can be rotated right through an even number of positions.

Assembler will calculate rotate for you from constant.


操作:

1、BL 和 BLX 指令可将下一个指令的地址复制到lr（r14，链接寄存器）中。

2、BX 和 BLX 指令可将处理器的状态从 ARM 更改为 Thumb，或从 Thumb 更改为ARM。

3、BLX label 无论何种情况，始终会更改处理器的状态。

4、BX Rm 和 BLX Rm 可从 Rm 的位 [0]推算出目标状态。

5、如果 Rm 的位 [0] 为 0，则处理器的状态会更改为（或保持在）ARM 状态。如果 Rm 的位 [0] 为 1，则处理器的状态会更改为（或保持在）Thumb 状态。

BRANCH OPERATIONS

- B and BL are branch with “immediate” arguments (e.g. jumping to a label/function); the symbol . Is a synonym for the surrent program location

- BX and BLX are branch with “register” arguments (i.e. jump to an address stored in a register)

- BL and BLX store a bookmark to the current place in the program by writing the address of the next instruction in the link register (LR).
    - B labelA; branch to the label labelA
    - BL labelA; update LR and branch to the label labelA branch with link
    - BX LR; branch to the location whose address is in LR, e.g. return from a function call
    - BLX R0; update LR and branch to the location whose address is in R0
    - B . ; branch to the current program location (infinite loop)

PUSH AND POP OPERATIONS (STACK)

PUSH {R1} ; push the contents of R1 onto the stack

PUSH {R0,R1} ; push the contents of R0 and R1 onto the stack

PUSH {R0,R2-R4} ; push the contents of R0, R2, R3, R4 onto the stack

PUSH {R0,LR} ; push the contents of R0, LR onto the stack

POP {R0,R1} ; pop the top two 32-bit values from the stack into R0 and R1

PUSH and POP operations update the SP.

ADDITION AND SUBTRACTION

ADD , SUB : Add and subtract

ADC , SBC : “Add with carry” and “Subtract with carry”

The carry instructions also utilize the carry flag (set by previous instructions – the carry flag is stored as a bit in the application program status register)

- ADD R1, R0, R1 ; set R1 to the sum of the contents of R0 and R1
- SUB R2, R0, R1 ; set R2 to the difference of the contents of R0 and R1
- ADD R2, R0, #10 ; R2 = R0 + 10

RSB : reverse subtract (i.e., subtract the contents of the second operand from the third operand), e.g.,

RSB:

- RSB R2, R0, R1 ; set R2 to the difference of the contents of R1 and R0 $R_2 \leftarrow R_1 - R_0$

SUB R_2, R_0, R_1 = $R_2 \leftarrow R_0 - R_1$

Signed and unsigned variants: SADD16, etc.

MULTIPLICATION AND DIVISION

MUL, MLA, and MLS : “Multiply”, “Multiply with accumulate”, and “Multiply with subtract”

- MUL R2, R0, R1 ; R2 = R0*R1
- MLA R3, R0, R1, R2 ; R3 = R0*R1 + R2 MUL + ADD
- MLS R3, R0, R1, R2 ; R3 = R0*R1 – R2 MUL + SUB

Signed and unsigned variants: SMLA, etc. SDIV, UDIV : “Signed division” and “Unsigned division”

- SDIV R2, R0, R1 ; signed divide, R2 = R0/R1
- UDIV R2, R0, R1 ; unsigned divide, R2 = R0/R1

FLOATING POINT INSTRUCTIONS

Floating point instructions are available if there is a floating point unit (FPU) in the system and is enabled (the FPU is generally enabled as part of the start-up sequence on ARM Cortex-M4F).

- VADD, VSUB, etc. : floating point addition and subtraction
- VMUL, VDIV, etc. : floating point multiplication and division
- VABS : floating point absolute value

BITWISE OPERATIONS

AND, ORR (logical OR), etc.; exclusive OR is EOR

LSL, LSR : “logical shift left” and “logical shift right”

ROR : rotate right

- AND R2, R0, R1 ; R2 = R0 & R1
- OR R2, R0, R1 ; R2 = R0 | R1
- AND R2, R0, #0x10 ; R2 = R0 & 0x10
- OR R2, R0, #0x10 ; R2 = R0 | 0x10
- LSL R2, R0, #2 ; R2 = R0 << 2

CONDITIONAL EXECUTION OF INSTRUCTIONS

Many instructions support optional suffixes denoting various conditions to specify that the instruction must be executed only if the specified condition is true.

- Some examples of condition suffixes: →后缀
  - EQ for equal (i.e., Z = 1), NE for not equal (i.e., Z = 0)
  - GT for “greater than” and LT for “less than” ; these conditions are evaluated using combinations of flags Z, N, and V. For example, GT is equivalent to “ Z= 0 and N = V ”, LT is equivalent to “ N != V ”
  - GE and LE for “greater than or equal” and “less than or equal”
- Example: ADDEQ will do an addition only if the “equal” condition is currently active (i.e., a previous instruction caused the Z flag to become 1); BNE will do a branch (jump) only if the “equal” condition is not currently active.

Software Interrupt (SWI)

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	Instruction Type
Condition				1	1	1	1	SWI NUMBER																								Software Interrupt

- In effect, a SWI is a user-defined instruction.
- It causes an exception trap to the SWI hardware vector (thus causing a change to supervisor mode, plus the associated state saving), thus causing the SWI exception handler to be called.
- The handler can then examine the comment field of the instruction to decide what operation has been requested.
- By making use of the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- See Exception Handling Module for further details.

Assembler: Pseudo-ops

- AREA -> chunks of data ($data) or code ($code)
- ADR -> load address into a register
- ADR R0, BUFFER
- ALIGN -> adjust location counter to word boundary usually after a storage directive
- END -> no more to assemble

Assembler: Pseudo-ops

- Example:

```
 AREA    cacheable, CODE, ALIGN=3
rout1   ; code          ; aligned on 8-byte boundary
        ; code
        MOV     pc,lr  ; aligned only on 4-byte boundary
        ALIGN   8      ; now aligned on 8-byte boundary
rout2   ; code
```

Assembler: Pseudo-ops

- IMPORT -> name of routine to import for use in this routine
- IMPORT _printf ; C print routine
- EXPORT -> name of routine to export for use in other routines
- EXPORT add2 ; add2 routine
- EQU -> symbol replacement
- loopcnt EQU 5

Assembly Line Format

- *label*<whitespace> *instruction*<whitespace> *; comment*
- *label*: created by programmer, alphanumeric
- whitespace: space(s) or tab character(s)
- *instruction*: op-code mnemonic or pseudo-op with required fields
- *comment*: preceded by *;* ignored by assembler but useful
- to the programmer for documentation
- NOTE: All fields are optional.

REFERENCES

- Sloss, Andrew, Dominic Symes, and Chris Wright. ”ARM system developer’s guide: designing and optimizing system software.” Morgan Kaufmann, 2004.
- ARM University Program. ”ARM Processors and Architectures Comprehensive Overiew.” http://arm.com, 2012.