The ARM processor

- Born in Acorn on 1983, after the success achieved by the BBC Micro released on 1982.
- Acorn is a really smaller company than most of the USA competitors, therefore it initially develops a suitable (special purpose, i. e. early concept of RISC) low cost processor named ARM1 (Acorn Risc Machine 1) for internal use only.
- On 1987, the first ARM Archimedes platform equipped with 8 MHz ARM2 was marketed.
- VLSI Technology Inc., Acorn partner in ARM design and development, biases part of the market towards the use of a such a kind of processors.
- On 1989, ARM3 is proposed that is a powered version of ARM2 with 4 Kbit cache and 25 MHz working frequency.
- On 1990, ARM Ltd is born consisting of Acorn VLSI and Apple.
- At now ARM is a community to which the main microprocessors design brand decide to belong and the ARM acronym evolved in the more general Advanced Risc Machine.

ARM instruction set

- ARM versions.
- ARM assembly language.
- ARM programming model.
- ARM memory organization.
- ARM data operations.
- ARM flow of control.

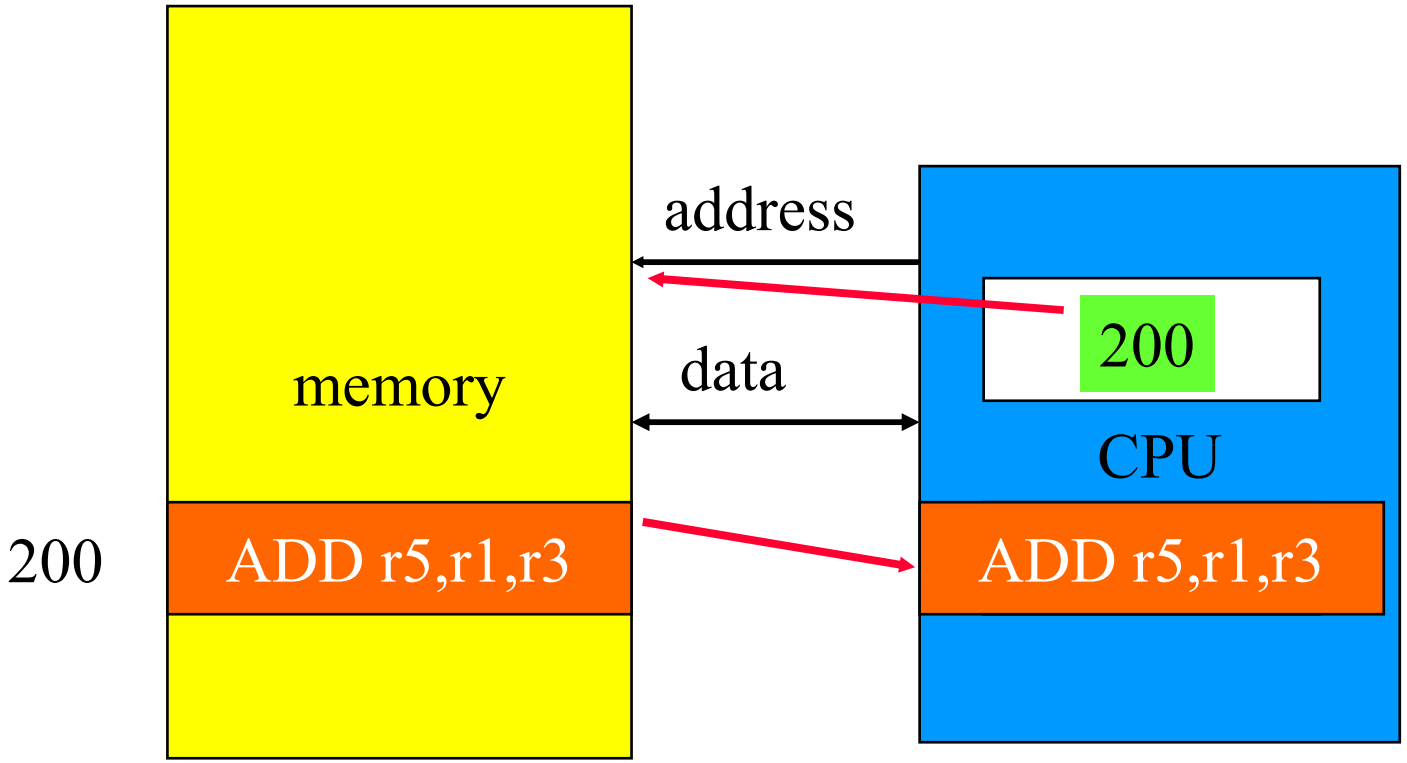
ARM versions

- ARM architecture has been extended over several versions.
- We will concentrate on ARM7 (von Neumann, while ARM9 Harvard)
- A lot of licenses
- Performance, low cost and power (cellular phones)
- Interesting extensions (Jazelle a technology to execute Java Bytecode on a ARM processor. Basically it is a Multi Tasking JVM)

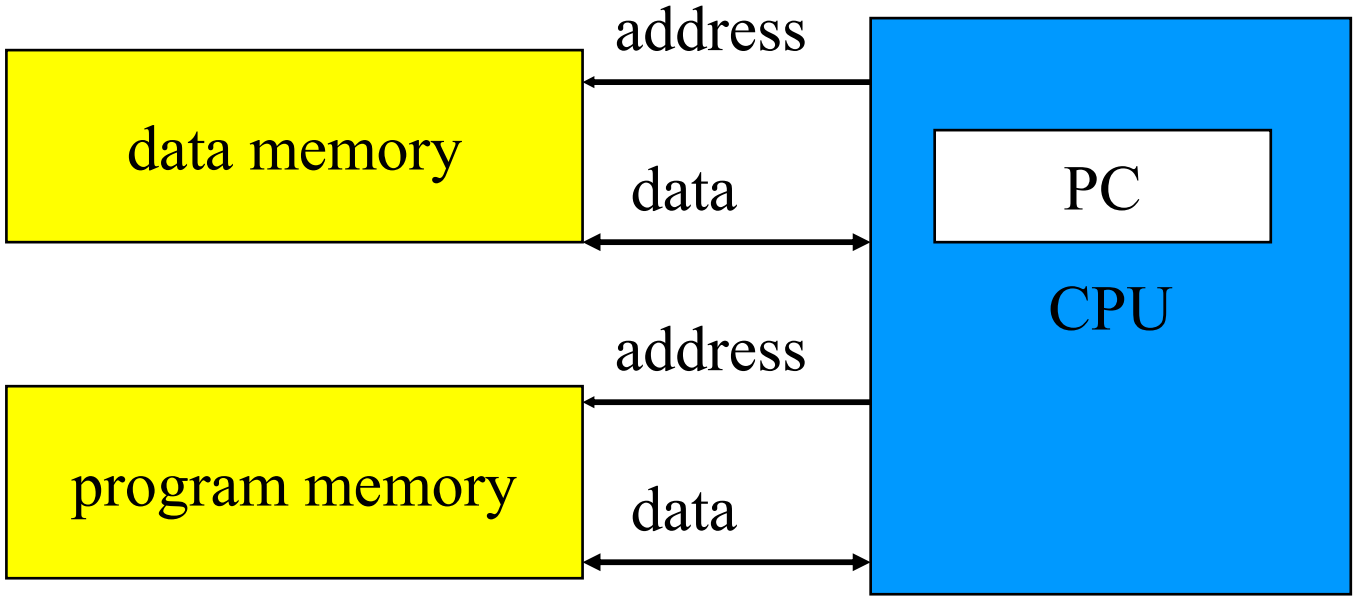
von Neumann architecture

- Memory holds data, instructions.
- Central processing unit (CPU) fetches instructions from memory.
  - Separate CPU and memory distinguishes programmable computer.
- CPU registers help out: program counter (PC), instruction register (IR), general-purpose registers, etc.

CPU + memory



Harvard architecture



von Neumann vs. Harvard

- Harvard can't use self-modifying code.
- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth.

Self modifying code

sllv $s2, variable shift left logical of a number of positions $s2

```
sllv:       li      $t0, mask          /* mask is FFFFF83F
            li      $s1, shifter
            lw      $s0, [$s1]0
            and     $s0, $s0, $t0
            andi    $s2, $s2, 0x1f
            sll     $s2, $s2, 6
            or      $s0, $s0, $s2
            sw      $s0, [$s1]0
shifter:    sll     $s0, $s1, 0
```

ARM general aspects

- Risc style aspects
  - Instruction fixed length
  - Load-store instruction to access memory
  - Arithmetic and logic on registers
- Cisc style aspects
  - Auto-inc/dec and PC relative addressing
  - Flag for branching and conditional execution
  - Multi registers load/store with single instruction
- Unusual aspects
  - Conditional execution

ARM assembly language

- Fairly standard assembly language:

```
        LDR r0,[r8] ; a comment
label   ADD r4,r0,r1
```

- Load store architecture (no direct ops in mem)
- 37 registers, 31 general purpose, 6 status
- 7 different programming modes (*user, supervisor, abort, undefined, interrupt, fast interrupt, system*)
- Data types: 8 (byte), 16 (half word), 32 bit (word)
- Three stages pipeline (ARM 7)

ARM assembly language

Processor mode			Descrizione	Codifica M[4:0]
1	User	(usr)	Modo d'esecuzione dei programmi comuni.	0b10000
2	FIQ	(fiq)	Gestione di *interrupt* veloce.	0b10001
3	IRQ	(irq)	Gestione di *interrupt* generico.	0b10010
4	Supervisor	(svc)	Modo protetto per l'esecuzione di codice del sistema operativo.	0b10011
5	Abort	(abt)	Errore nell'accesso di memoria (anche per implementare memoria virtuale o protezione della memoria).	0b10111
6	Undefined	(und)	Istruzione illegale.	0b11011
7	System	(sys)	Modo provilegiato d'esecuzione di un *task* del sistema operativo.	0b11111

- The mode can be changed through *sw privileged instructions* or through *exceptions*
- *User* mode => user programs,
- Other modes (privileged) to serve exceptions or for accessing to protected and/or shared resources
- Only sw interrupts allow to pass from user mode to other ones

ARM assembly language

- **Non privileged modes:**
  - **User** (**USR**): user program mode
- **Privileged modes:**
  - External interrupt management
    - **IRQ** (**IRQ**): normal interrupts
    - **FIQ** (**FIQ**): fast interrupt management
  - Internal interrupt management: trap
    - **Abort** (**ABT**): memory management (forbidden area accesses)
    - **Undefined** (**UDEF**): coprocessor emulation – not defined instructions
  - Internal interrupt management: system call
- **Supervisor** (**SVC**): "protected mode" to shared resources
- **System** (**SYS**): resources are used but without access limitations

ARM programming model

- Registers visible to the programmer

16 general purpose registers

r0	r8
r1	r9
r2	r10
r3	r11
r4	r12
r5	R13 (SP)
r6	R14 (LR)
r7	r15 (PC)

ARM programming model

R12 special register used by the linker also as temporary

R4 - R8 , R10 and R11 for local variables

R0 – R3 for parameters passing

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

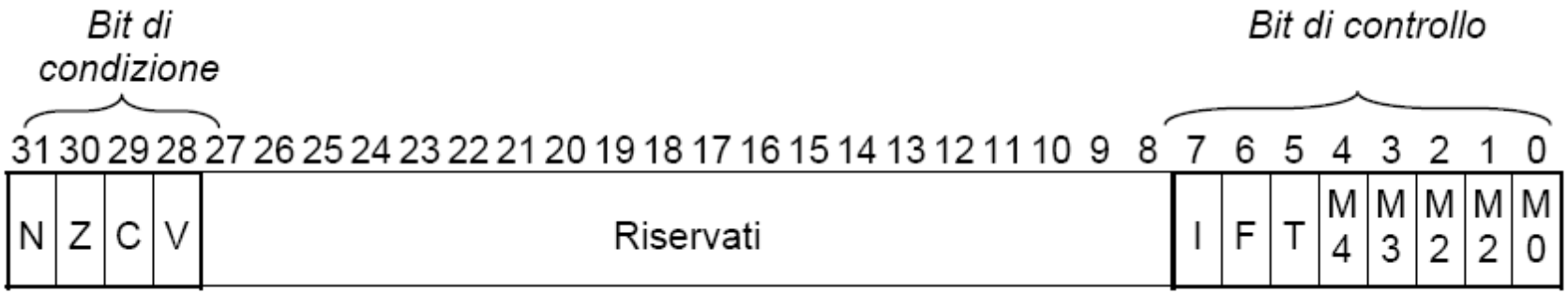
ARM programming model

Modo					
User (usr)/ system (sys)	Supervisor (svc)	Abort (abt)	Undefined (und)	Interrupt (IRQ)	Fast inturrupt (FIQ)
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABT	R14_UNDEF	R14_IRQ	R14_FIQ
PC=R15	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

- R0-R7 always correspond to same physical memory locations
- R8-R14 correspond to different physical locations in every mode
- System uses the same reg_set than User

ARM programming model

Current program status register



F=1 Fast Interrupt Disable (FIQ)

I=1 Generic Interrupt Disable (IRQ)

V=1 Overflow (Signed)

Z=1 Zero

N=0 Negative result

C=1 Carry (Unsigned Overflow)

T=1 shift to Thumb Instruction Set (Reduced 16 bit Instruction Set)

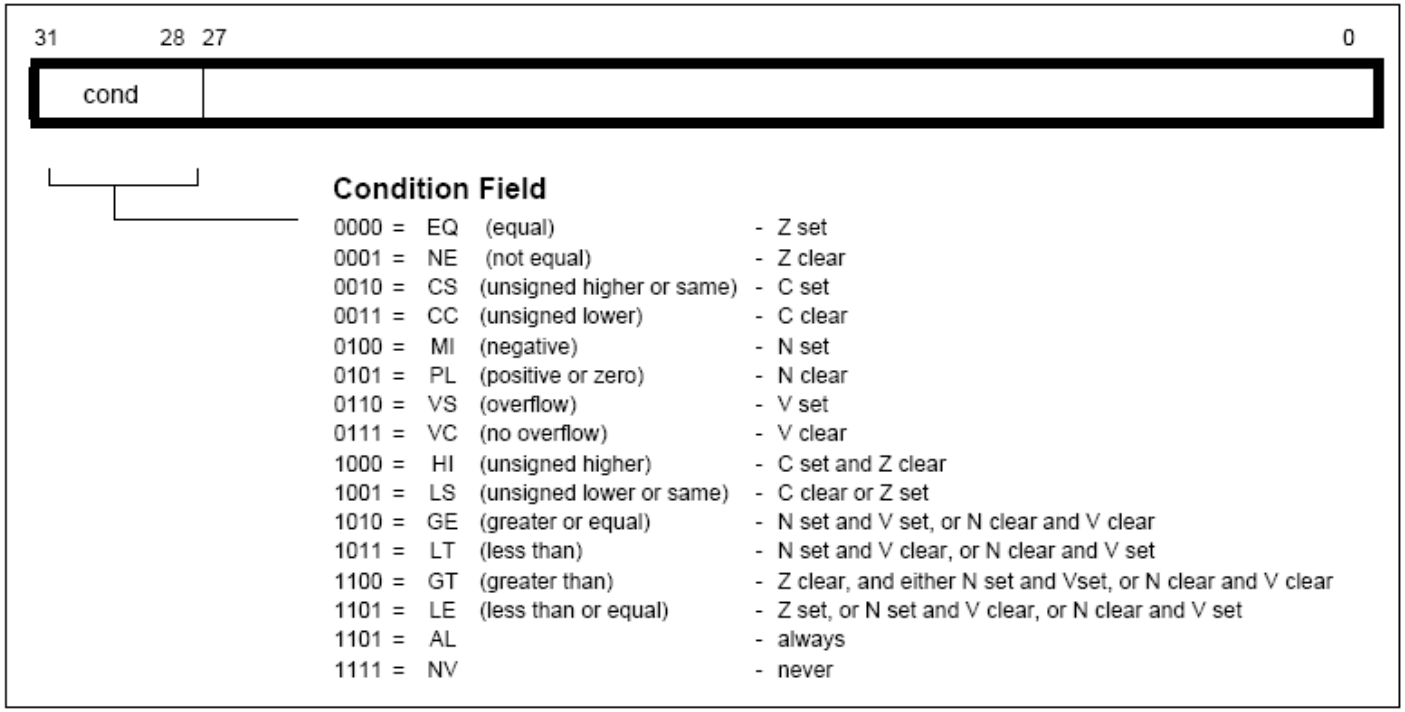
ARM status bits

- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect).
- To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".
- For example to add two numbers and set the condition flags:
- • ADDS r0,r1,r2 ; r0 = r1 + r2 ... and set flags

- Every arithmetic, logical, or shifting operation sets CPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow).
- Examples:
  - -1 + 1 = 0: NZCV = 0110.
  - 2^{31}-1+1 = -2^{31}: NZCV = 1001.

ARM status bits

- Flag bits are mapped on the most significant instruction bits to allow their conditional use

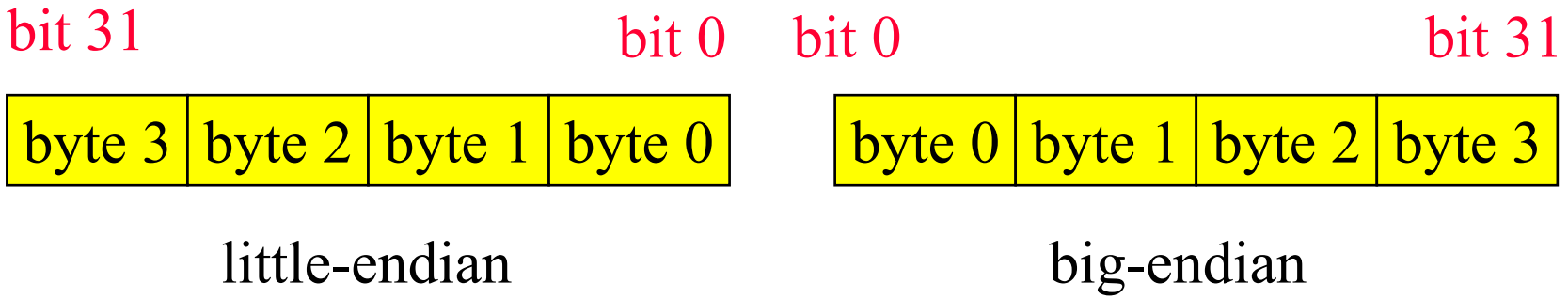


ARM data types

- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses are 32 bits long.
- Address refers to byte.
  - Address 4 starts at byte 4.
- Can be configured at power-up as either little- or bit-endian mode.

Endianness

- Relationship between bit and byte/word ordering defines endianness:



CONFIGURABLE!

ARM data instructions

- Basic format:

```
ADD r0,r1,r2
```

  - Computes r1+r2, stores in r0.

- Immediate operand:

```
ADD r0,r1,#2
```

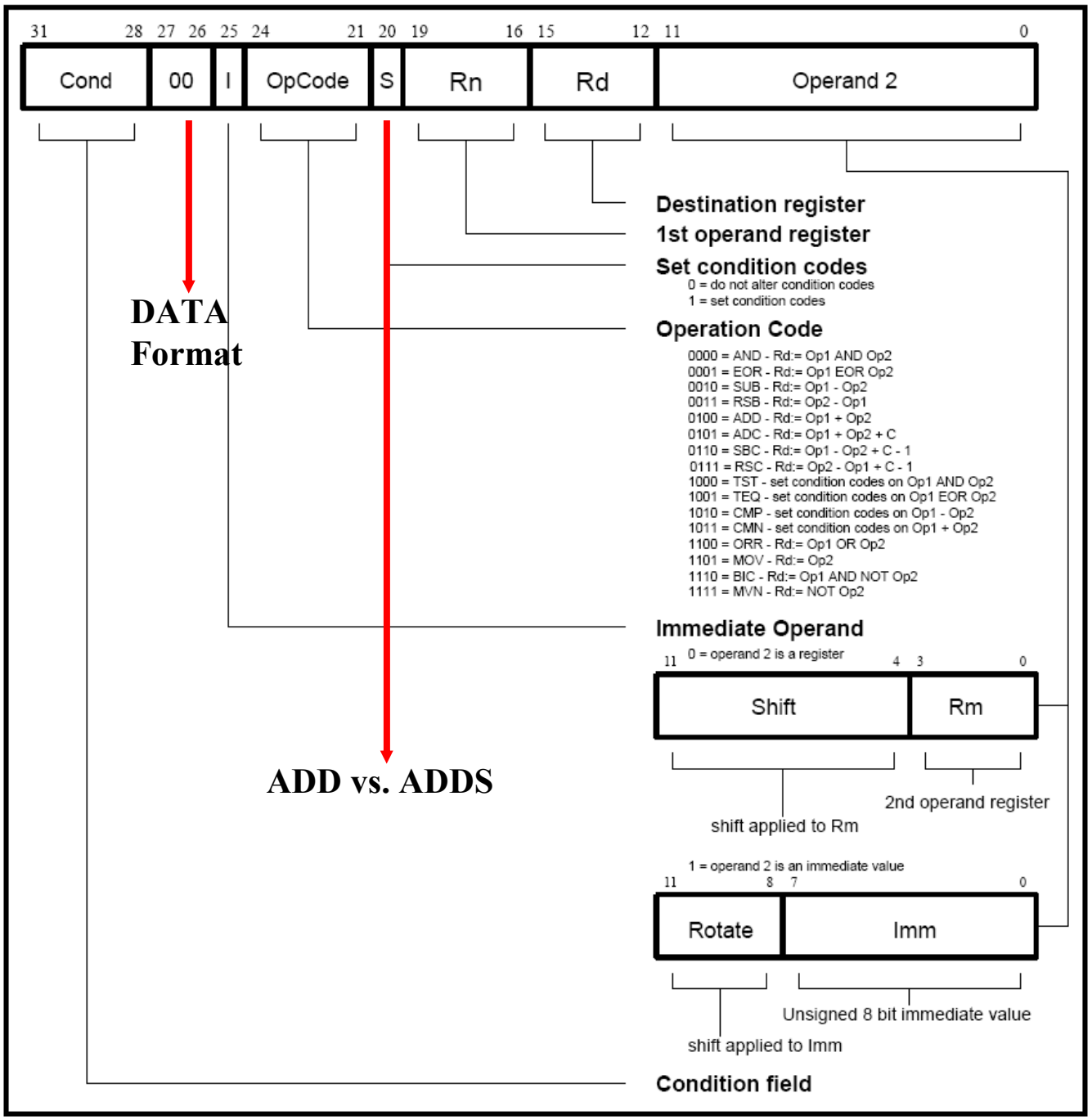
  - Computes r1+2, stores in r0.

ARM data instructions

- ADD, ADC : add (w. carry)
- SUB, SBC : subtract (w. carry)
- RSB, RSC : reverse subtract (w. carry)
- MUL, MLA : multiply (and accumulate)
- AND, ORR, EOR (ex-or)
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ASL, ASR : arithmetic shift left/right
- ROR : rotate right
- RRX : rotate right extended with C

Data operation varieties

- Logical shift:
  - fills with zeroes.
- Arithmetic shift:
  - fills with ones (if needed).
- RRX performs 33-bit rotate, including C bit from CPSR above sign bit.



Data processing instructions format: ***destination and first operand are registers, the second operand a register or a constant***

S

ARM comparison instructions

- CMP : compare (x-y)
- CMN : negated compare (x+y)
- TST : bit-wise AND
- TEQ : bit-wise XOR
- These instructions set only the NZCV bits of CPSR (no modification of registers).

ARM move instructions

- MOV, MVN : move (negated)

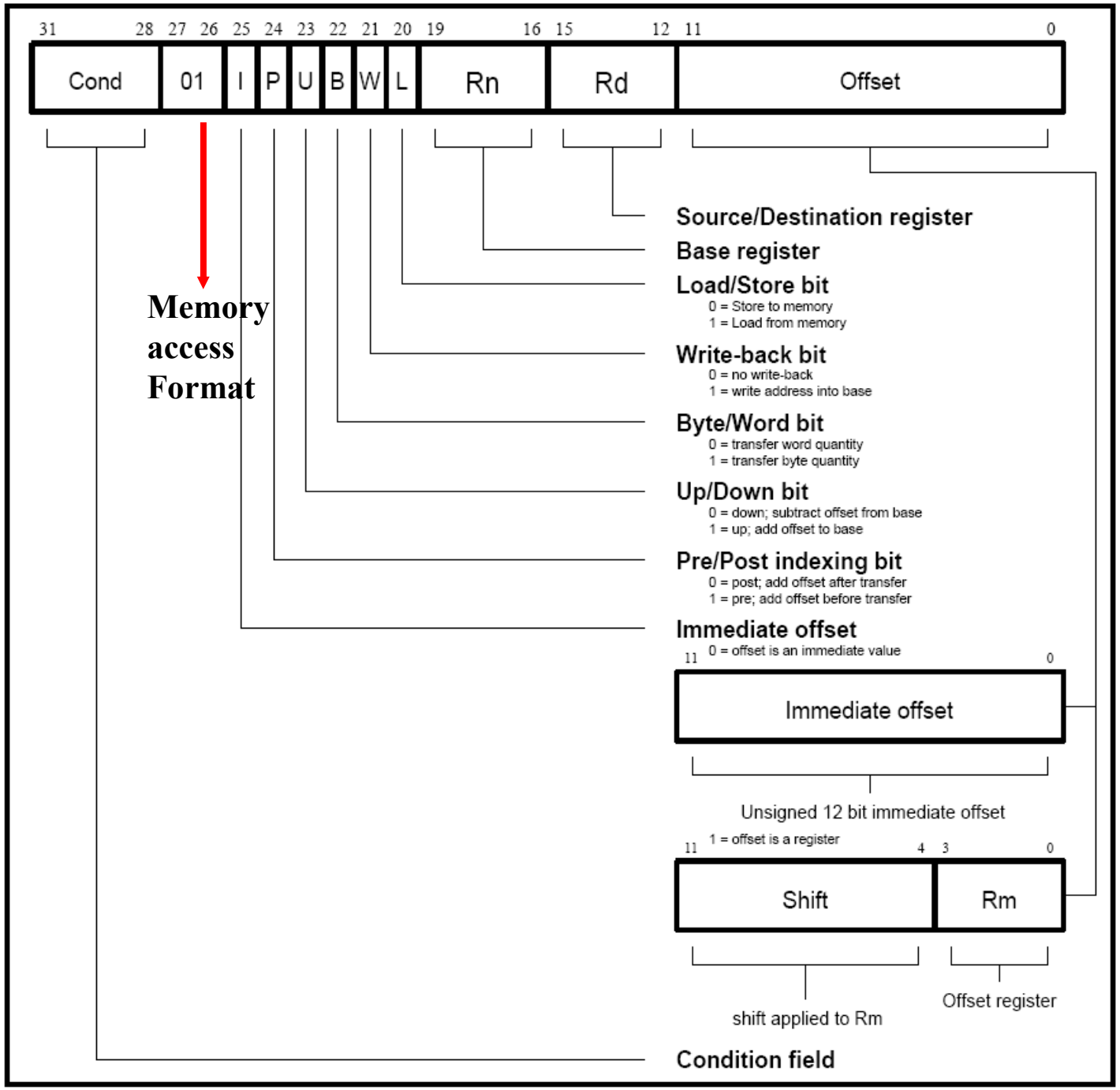
```
MOV r0, r1 ; sets r0 to r1

MVN r0, r1 ; sets r0 to r1 negated
```

ARM load-store instructions

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)

- Addressing modes:
  - register indirect : `LDR r0,[r1]`
  - with second register (offset): `LDR r0,[r1,-r2]`
  - with constant : `LDR r0,[r1,#4]`



ARM load/store instructions format

Additional addressing modes

- Base-plus-offset addressing:

```
LDR r0,[r1,#16]
```

  - Loads from location r1+16

- Auto-indexing increments base register:

```
LDR r0,[r1,#16]!
! Implies that r1 is updated
```

- Post-indexing fetches, then does offset:

```
LDR r0,[r1],#16
```

  - Loads r0 from r1, then adds 16 to r1.

ARM ADR pseudo-op

- Cannot refer to an address directly in an instruction.
- Generate an address value by performing arithmetic on PC.
- To simplify, ADR pseudo-op generates instruction required to calculate address:

```
ADR r1,FOO      (r1 with addr=FOO)
```

Example: C assignments

- C:

```
x = (a + b) - c;
```

- Assembler:

```
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a
ADR r4,b          ; get address for b, reusing r4
LDR r1,[r4]       ; get value of b
ADD r3,r0,r1      ; compute a+b
ADR r4,c          ; get address for c
LDR r2,[r4]       ; get value of c
```

C assignment, cont'd.

```
SUB r3,r3,r2     ; complete computation of x
ADR r4,x         ; get address for x
STR r3,[r4]      ; store value of x
```

Example: C assignment

- C:

```
y = a*(b+c);
```

- Assembler:

```
ADR r4,b      ;get address for b
LDR r0,[r4]   ;get value of b
ADR r4,c      ;get address for c
LDR r1,[r4]   ;get value of c
ADD r2,r0,r1  ;compute partial result
ADR r4,a      ;get address for a
LDR r0,[r4]   ;get value of a
```

C assignment, cont'd.

```
MUL r2,r2,r0  ;compute final value for y
ADR r4,y      ;get address for y
STR r2,[r4]   ;store y
```

Example: C assignment

- C:

```
z = (a << 2)|(b & 15);
```

- Assembler:

```
ADR r4,a          ;get address for a
LDR r0,[r4]       ;get value of a
MOV r0,r0,LSL 2 ;perform shift
ADR r4,b          ;get address for b
LDR r1,[r4]       ;get value of b
AND r1,r1,#15     ;perform AND
ORR r1,r0,r1      ;perform OR
ADR r4,z          ;get address for z
STR r1,[r4]       ;store value for z
```

ARM flow of control

- All operations can be performed conditionally, testing CPSR:
  - `EQ, NE, Carry Set, Carry Clear, MInus, PL (non-neg), VS/VC (ov/no ov), HIgher, unsigned LowerSame, GE, LT, GT, LE`
- Branch operation:

  `B #100`
  - Can be performed conditionally.

ARM flow of control

- Nessuna condizione ⇒AL

Estensione mnemonica	Significato	Flag di condizione	Opcode [31:28]
EQ	Uguali	Z=1	0000
NE	Non uguali	Z=0	0001
CS/HS	Carry Attivato / Senza segno maggiore o uguale	C=1	0010
CC/LO	Carry Disattivato / Senza segno minore	C=0	0011
MI	Negativo	N=1	0100
PL	Positivo o Zero	N=0	0101
VS	Overflow	V=1	0110
VC	Non Overflow	V=0	0111
HI	Senza segno maggiore	C=1 e Z=0	1000
LS	Senza segno minore o uguale	C=0 o Z=1	1001
GE	Con segno maggiore o uguale	N=V	1010
LT	Con segno minore	N!=V	1011
GT	Con segno maggiore	Z=0 e N=V	1100
LE	Con segno minore o uguale	Z=1 o N!=V	1101
AL	Sempre	-	1110
NV	Mai	-	1111

The hw does not check the field Cond

Reserved

Example: if statement

- C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

- Assembler:

```
; compute and test condition
  ADR r4,a          ;get address for a
  LDR r0,[r4]       ;get value of a
  ADR r4,b          ;get address for b
  LDR r1,[r4]       ;get value for b
  CMP r0,r1         ;compare a < b
  BLE fblock        ;if a <= b, branch to false block
```

If statement, cont'd.

```
; true block
  MOV r0,#5          ;generate value for x
  ADR r4,x           ;get address for x
  STR r0,[r4]        ;store x
  ADR r4,c           ;get address for c
  LDR r0,[r4]        ;get value of c
  ADR r4,d           ;get address for d
  LDR r1,[r4]        ;get value of d
  ADD r0,r0,r1       ;compute y
  ADR r4,y           ;get address for y
  STR r0,[r4]        ;store y
  B after            ;branch around false block
```

If statement, cont'd.

```
; false block
fblock ADR r4,c    ;get address for c
  LDR r0,[r4]      ;get value of c
  ADR r4,d         ;get address for d
  LDR r1,[r4]      ;get value for d
  SUB r0,r0,r1     ;compute a-b
  ADR r4,x         ;get address for x
  STR r0,[r4]      ;store value of x
after ...
```

Example: Conditional instruction implementation

```
; true block
  MOVLT r0,#5      ;generate value for x
  ADRLT r4,x       ;get address for x
  STRLT r0,[r4]    ;store x
  ADRLT r4,c       ;get address for c
  LDRLT r0,[r4]    ;get value of c
  ADRLT r4,d       ;get address for d
  LDRLT r1,[r4]    ;get value of d
  ADDLT r0,r0,r1   ;compute y
  ADRLT r4,y       ;get address for y
  STRLT r0,[r4]    ;store y
```

Conditional instruction implementation, cont'd.

```
; false block
  ADRGE r4,c       ;get address for c
  LDRGE r0,[r4]    ;get value of c
  ADRGE r4,d       ;get address for d
  LDRGE r1,[r4]    ;get value for d
  SUBGE r0,r0,r1   ;compute a-b
  ADRGE r4,x       ;get address for x
  STRGE r0,[r4]    ;store value of x
```

Example: switch statement

⌘C:

```
switch (test) { case 0: … break; case 1: … }
```

⌘Ass:

```
        ADR r2,test   ;get address for test
        LDR r0,[r2]   ;load value for test
        ADR r1,switchtab ;load addr. for switch table
        LDR r15,[r1,r0,LSL #2] ;index switch table

switchtab  DCD case0     ;the location of the table
                         contains relative routine address
           DCD case1     ;
           ...
case 0     code for case 0
case 1     code for case 1
           ...
```

NB. {*label*} DCD *expression* allocates one or more words of memory (4 byte boundaries) with *expression*

Example: FIR filter

- C:

```
for (i=0, f=0; i<N; i++)
  f = f + c[i]*x[i];
```

- Assembler

```
; loop initiation code
  MOV r0,#0         ;use r0 for I
  MOV r8,#0         ;use separate index for arrays
  ADR r2,N          ;get address for N
  LDR r1,[r2]       ;get value of N
  MOV r2,#0         ;use r2 for f
```

FIR filter, cont'.d

```
        ADR r3,c            ;load r3 with base of c
        ADR r5,x            ;load r5 with base of x
;loop body
loop    LDR r4,[r3,r8]      ;get c[i]
        LDR r6,[r5,r8]      ;get x[i]
        MUL r4,r4,r6        ;compute c[i]*x[i]
        ADD r2,r2,r4        ;add into running sum
        ADD r8,r8,#4        ;add 1 word offs to array index
        ADD r0,r0,#1        ;add 1 to i
        CMP r0,r1           ;exit?
        BLT loop            ;if i < N, continue
```

ARM subroutine linkage

- Branch and link instruction:

```
BL foo
```

  - Copies current PC to r14.

- To return from subroutine:

```
MOV r15,r14
```

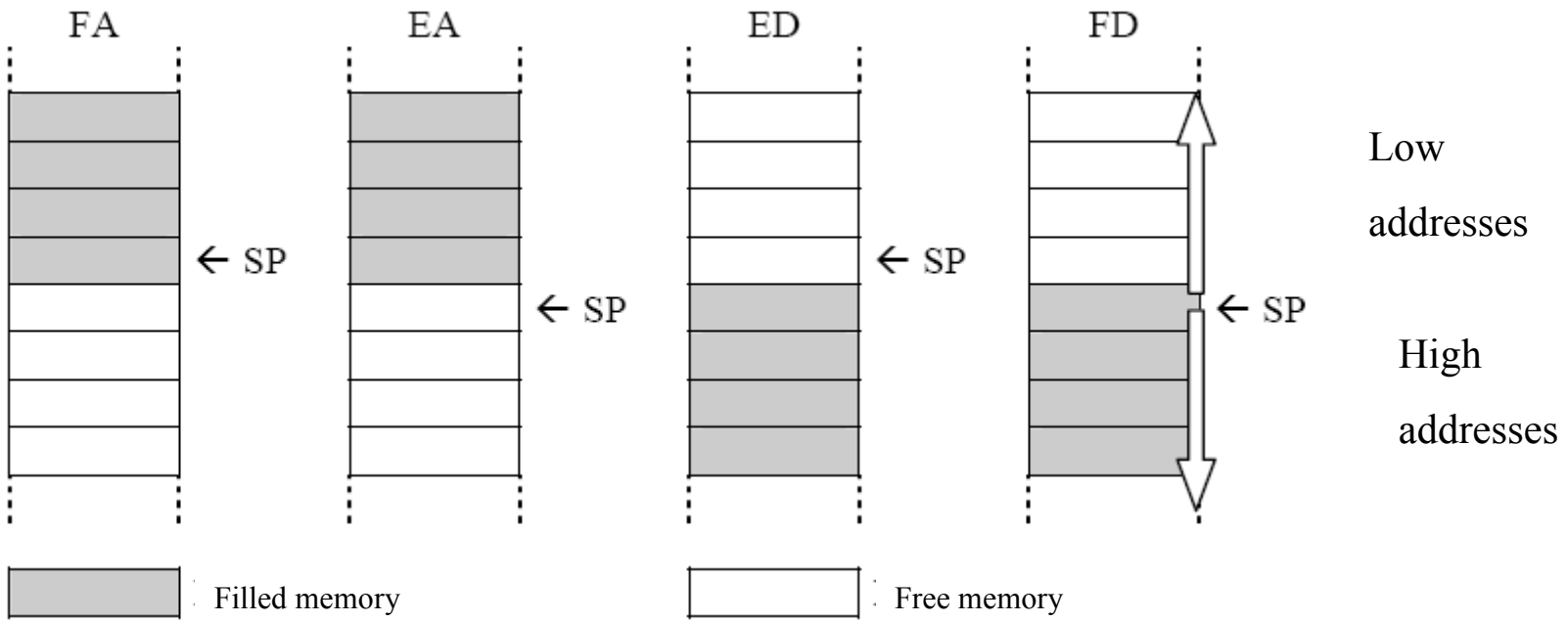
Nested subroutine calls

- Nesting/recursion requires coding convention:

```
f1      LDR r0,[r13]        ;load arg into r0 from stack
                            ;call f2()
        STR r14,[r13]!      ;store f1's return address
        STR r0,[r13]!       ;store arg to f2 on stack
        BL f2               ;branch and link to f2
                            ;……
                            ;return from f1()
        SUB r13,#4          ;pop f2's arg off stack
        LDR r15,[r13]!      ;restore register and return
```

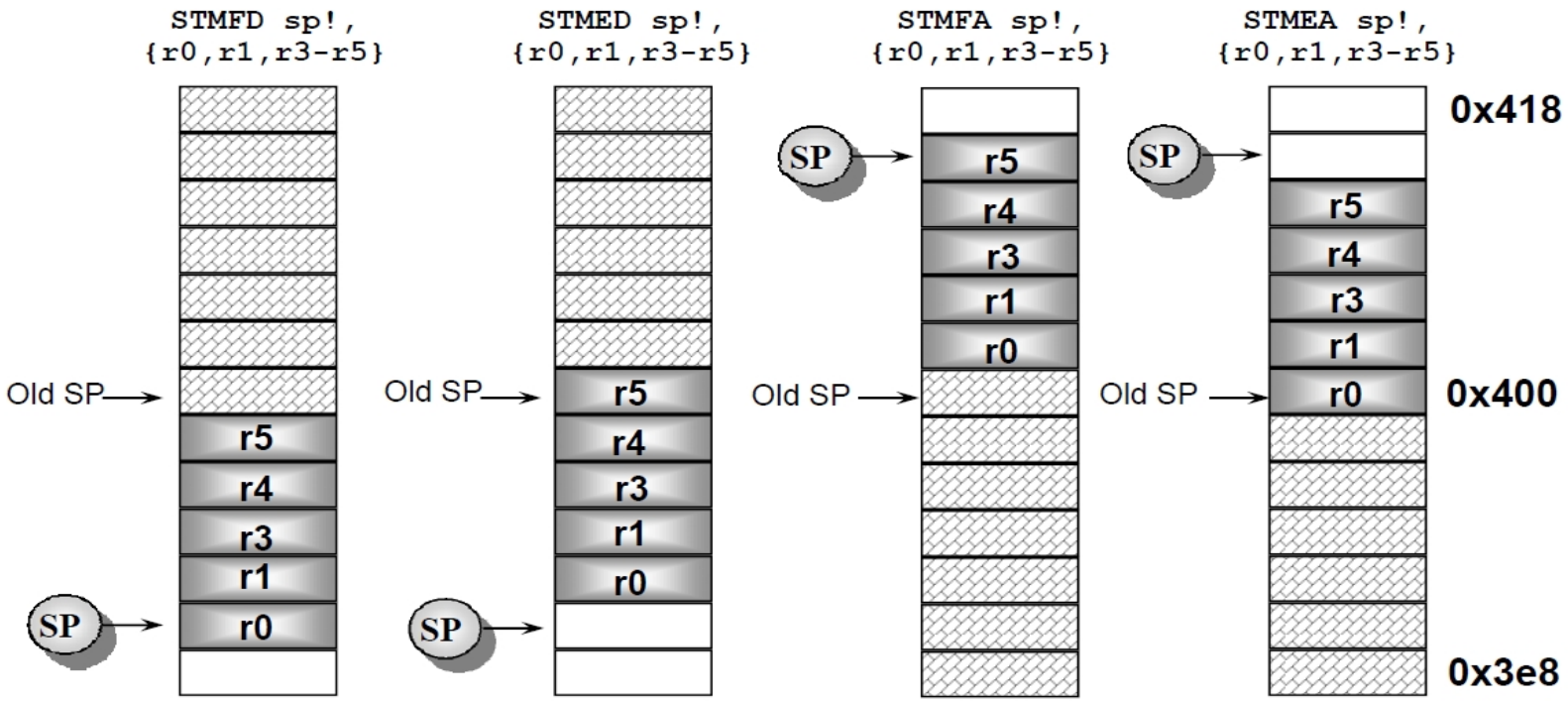
Stack types

- 4 types of stacks: full/empty, ascending/descending



Stack types

Stack Examples



Stack management instructions

- Load/store instructions with pre/post increment/decrement depending on the type of stacks acting on multiple registers

Istruzione	Modo d'indirizzamento	Tipo di stack
LDM (Load)	IA (Increment After)	FD (Full Descending)
STM (Store)	IA (Increment After)	EA (Empty Ascending)
LDM (Load)	IB (Increment Before)	ED (Empty Descending)
STM (Store)	IB (Increment Before)	FA (Full Ascending)
LDM (Load)	DA (Decrement After)	FA (Full Ascending)
STM (Store)	DA (Decrement After)	ED (Empty Descending)
LDM (Load)	DB (Decrement Before)	EA (Empty Ascending)
STM (Store)	DB (Decrement Before)	FD (Full Descending)

Stack management instructions

- Load/store instructions with pre/post increment/decrement depending on the type of stacks

Tipo stack	Commento	Suffisso	Istruzione	Modalità
Full descending (il più comune)	Lo SP si decrementa in inserimento e punta all'ultimo inserito	FD	LDM	IA
			STM	DB
Full ascending	Lo SP si incrementa in inserimento e punta all'ultimo inserito	FA	LDM	DA
			STM	IB
Empty descending	Lo SP si decrementa in inserimento e punta al primo libero	ED	LDM	IB
			STM	DA
Empty ascending	Lo SP si incrementa in inserimento e punta al primo libero	EA	LDM	DB
			STM	IA

```
STMIA R13, {R0-R7}      ; W[R13] <- R0; W[R13+4] <- R1; ... W[R13+7*4] <- R7

LDMIA R13!, {R1, R4-R5} ; R1<-W[R13]; R4<-W[R13+4]; R5<-W[R13+8]; R13<-R13+12
```

Programming I/O

- Two types of instructions can support I/O:
  - special-purpose I/O instructions;
  - memory-mapped load/store instructions.
- Intel x86 provides `in`, `out` instructions. Most other CPUs use memory-mapped I/O.
- I/O instructions do not preclude memory-mapped I/O.

ARM memory-mapped I/O

- Define location for device:

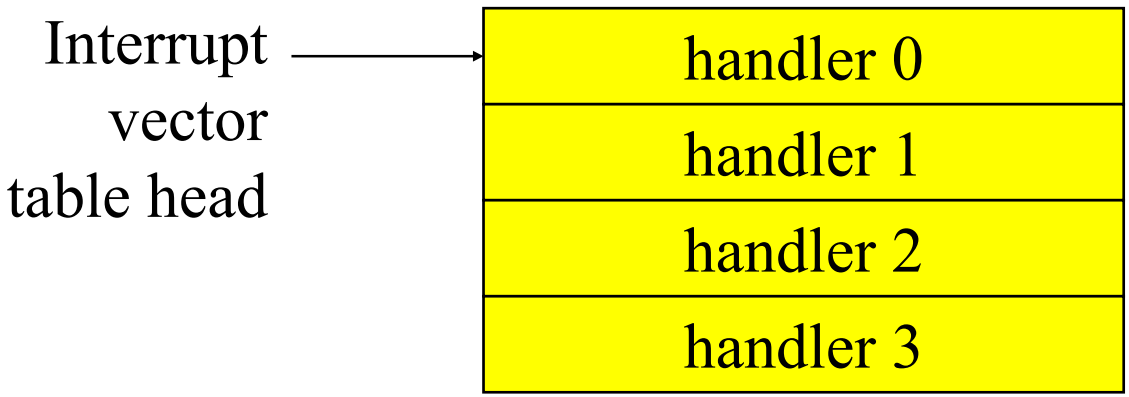
```
DEV1 EQU 0x1000
```

- Read/write code:

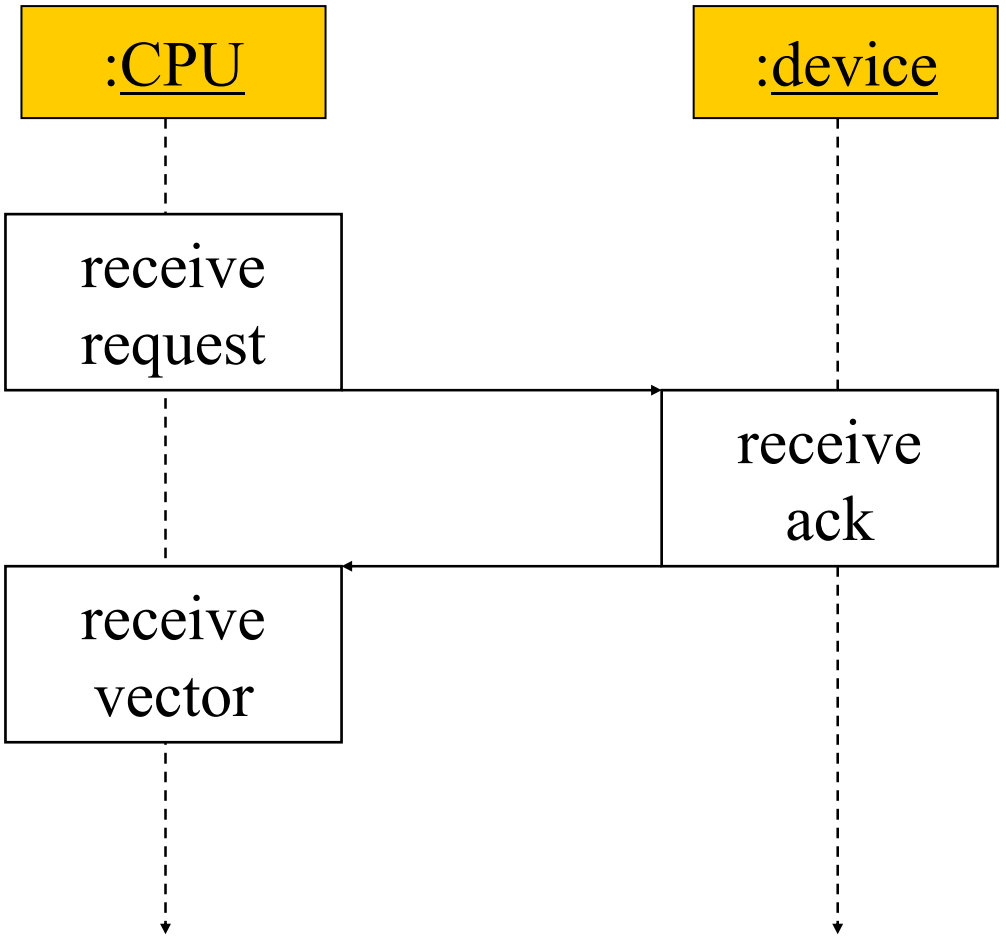
```
LDR r1,#DEV1 ;set up device address
LDR r0,[r1] ;read DEV1
LDR r0,#8 ;set up value to write
STR r0,[r1] ;write value to device
```

Interrupt vectors

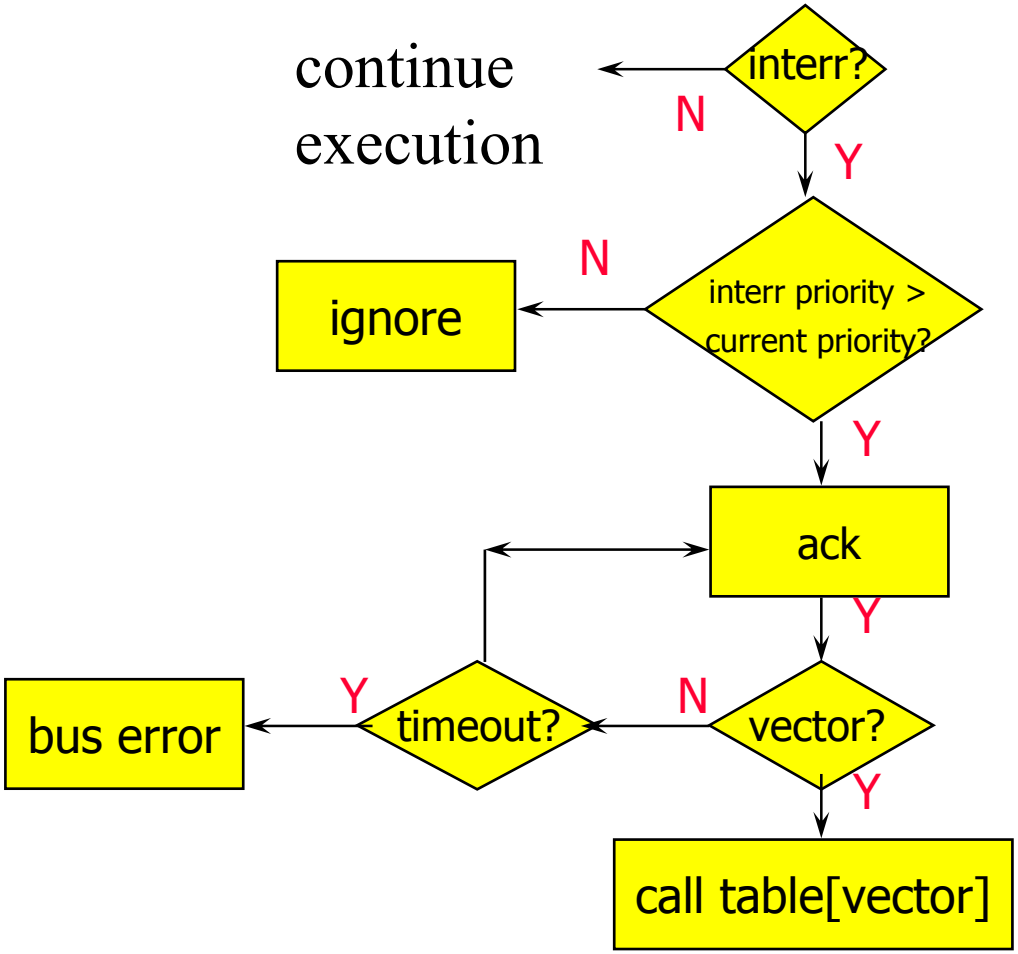
- Allow different devices to be handled by different code.
- Interrupt vector table:



Interrupt vector acquisition



Generic interrupt mechanism



Assume priority selection is handled before this point.

Interrupt sequence

- CPU acknowledges request.
- Device sends vector.
- CPU calls handler.
- Software processes request.
- CPU restores state to foreground program.

Sources of interrupt overhead

- Handler execution time.
- Interrupt mechanism overhead.
- Register save/restore.
- Pipeline-related penalties.
- Cache-related penalties.

ARM interrupts

- ARM7 supports two types of interrupts:
  - Fast interrupt requests (FIQs).
  - Interrupt requests (IRQs).
  - FIQs priority > IRQs priority
- Interrupt table starts at location 0.
- Entries contain calls to appropriate handlers.

ARM interrupt procedure

- CPU actions:
  - Save PC.
  - Copy CPSR to SPSR_mode (*saved program status register*).
  - Force some bits in CPSR to record interrupt.
  - Force PC to vector (handler).
- Handler responsibilities:
  - Restore proper PC (data process. instr. with PC destin. reg.)
  - Restore CPSR from SPSR_mode (MOVS)
  - Clear interrupt, disable flags.

ARM interrupt procedure

R14_<except_mode> <= PC + ... *PC in Link Register mode_dependent*

SPSR_<except_mode> <= CPSR *CPSR in SPSR mode_dependent*

CPSR[4:0] = exception identification *new processor mode*

if _<exception_mode> == (Reset or FIQ) then CPSR[6]=1
if Reset/FIQ disab FIQ

else CPSR[7] = 1 *disable IRQ*

PC = <exception vector> *jump to exception routine*

ARM interrupt procedure

Reset	R14_svc = unexpected
	SPSR_svc = unexpected
	CPSR[4:0] = 0b10011 //Supervisor Mode
	CPSR[5] = 0 // ARM state
	CPSR[6] = 1 // Disable FIQ
	CPSR[7] = 1 // Disable IRQ
	PC = 0x00000000

Undefined Instructions	R14_und = PC+4
	SPSR_und = CPSR
	CPSR[4:0] = 0b11011 //Undefined Mode
	CPSR[5] = 0 // ARM state
	CPSR[6] unchanged
	CPSR[7] = 1 // Disable IRQ
	PC = 0x0000004

ARM interrupt procedure

Software Interrupt	R14_svc = PC + 4
	SPSR_svc = CPSR
	CPSR[4:0] = 0b10011 //Supervisor Mode
	CPSR[5] = 0 // ARM state
	CPSR[6] unchanged
	CPSR[7] = 1 // Disable IRQ
	PC = 0x00000008

Prefetch Abort	R14_abt = PC+4
	SPSR_abt = CPSR
	CPSR[4:0] = 0b10111 //Abort Mode
	CPSR[5] = 0
	CPSR[6] unchanged
	CPSR[7] = 1 // Disable IRQ
	PC = 0x000000C

ARM interrupt procedure

Data Abort	R14_abt = PC + 8
	SPSR_abt = CPSR
	CPSR[4:0] = 0b10111 //Abort Mode
	CPSR[5] = 0 // ARM state
	CPSR[6] unchanged
	CPSR[7] = 1 // Disable IRQ
	PC = 0x00000010

Interrupt Request	R14_abt = PC+4
	SPSR_abt = CPSR
	CPSR[4:0] = 0b10010 //Abort Mode
	CPSR[5] = 0
	CPSR[6] unchanged
	CPSR[7] = 1 // Disable IRQ
	PC = 0x0000018

ARM interrupt procedure

Fast Interrupt Request	R14_abt = PC + 4
	SPSR_abt = CPSR
	CPSR[4:0] = 0b10010 //IRQ Mode
	CPSR[5] = 0 // ARM state
	CPSR[6] = 1 //Disable FIQ
	CPSR[7] = 1 // Disable IRQ
	PC = 0x0000001C

ARM interrupt procedure

Returning From an Exception Handler

- Returning from an exception handler
  - Depend on whether the exception handler uses the stack operations or not
- Generally, to return execution to the *original execution place*
  - Restore the CPSR from spsr_*mode*
  - Restore the program counter using the *return address* stored in lr_*mode*

ARM interrupt procedure

- If not require the *destination mode registers* to be restored from the stack
  - Above two operations can be carried out by *a data processing instruction* with
    - The *S* flag (bit 20) set
      - Update the CPSR flags when executing the data processing instruction
      - SUBS, MOVS
    - The program counter as the destination register
  - Example: MOVS pc, lr //pc = lr

ARM interrupt procedure

- If an exception handler entry code uses the stack to store registers
  - Must be preserved while handling the exception
- To return from such an exception handler, the stored register must be restored from the stack
  - Return by a *load multiple instruction* with ^ qualifier
  - For example: LDMFD sp!, {r0-r12,pc}^

ARM interrupt procedure

- Note, do not need to return from the reset handler
  - The reset handler executes your *main* code directly
- The actual location when an exception is taken depends on the exception type
  - The return address may not necessarily be the next instruction pointed to by the *pc*

ARM interrupt procedure

Returning from SWI and Undefined Instruction Handlers

- SWI and undefined instruction exceptions are generated by the instruction itself
  - lr_mode = pc + 4 //next instruction
- Restoring the program counter
  - If not using stack: MOVS pc, lr //pc = lr
  - If using stack to store the return address

```
STMFD  sp!, {reglist, lr}        //when entering the handler
…
LDMFD sp!, {reglist, pc}^   //when leaving the handler
```

ARM interrupt procedure

Returning from FIQ and IRQ

- FIQ and IRQ are generated only after the execution of an instruction
  - The program counter has been updated

  - *lr_mode* = PC + 4
    - Point to one instruction beyond the end of the instruction in which the exception occurred

ARM interrupt procedure

Returning from FIQ and IRQ (Cont.)

- Restoring the program counter
  - If not using stack: SUBS pc, lr, #4 //pc = lr-4
  - If using stack to store the return address

```
SUB  lr, lr, #4                    //when entering the handler
STMFD  sp!, {reglist, lr}
…
LDMFD sp!, {reglist, pc}^  //when leaving the handler
```

ARM interrupt procedure

Return from Prefetch Abort

- If the processor supports MMU (Memory Management Unit)
  - The exception handler loads the unmapped instruction into physical memory
  - Then, uses the MMU to map the virtual memory location into the physical one.
- After that, the handler must return to *retry the instruction that caused the exception.*
- However, the *lr_ABT* points to the instruction at the address *following* the one that caused the abort exception

ARM interrupt procedure

Return from Prefetch Abort (Cont.)

- So the address to be restored is at $lr\_ABT - 4$
- Thus, with simple return

  SUBS pc,lr,#4
- In contrast, with complex return

```
SUB lr,lr,#4                    ;handler entry code
STMFD sp!,{reglist,lr}
;...
LDMFD sp!,{reglist,pc}^  ; handler exit code
```

ARM interrupt procedure

Return from Data Abort

- lr_ABT points *two instructions beyond* the instruction that caused the abort
  - Since when a load or store instruction tries to access memory, the program counter has been updated.
  - Thus, the instruction caused the data abort exception is at $lr\_ABT - 8$
- So the address to be restored is at $lr\_ABT - 8$

ARM interrupt procedure

Return from Data Abort (Cont.)

- So the address to be restored is at $lr\_ABT - 8$
- Thus, with simple return

```
SUBS pc,lr,#8
```

- In contrast, with complex return

```
SUB lr,lr,#8                    ;handler entry code
STMFD sp!,{reglist,lr}
;...
LDMFD sp!,{reglist,pc}^  ; handler exit code
```

ARM interrupt procedure

Summary

	Return Instruction	Previous State		Notes
		ARM R14_x	THUMB R14_x	
BL	MOV PC, R14	PC + 4	PC + 2	1
SWI	MOVS PC, R14_svc	PC + 4	PC + 2	1
UDEF	MOVS PC, R14_und	PC + 4	PC + 2	1
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC + 4	2
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC + 4	2
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4	1
DABT	SUBS PC, R14_abt, #8	PC + 8	PC + 8	3
RESET	NA	–	–	4

- **NOTES**
  1. PC is the address of the BL/SWI/Undefined Instruction fetch which had the prefetch abort.
  2. PC is the address of the instruction which did not get executed since the FIQ or IRQ took priority.
  3. PC is the address of the Load or Store instruction which generated the data abort.
  4. The value saved in R14_svc upon reset is unpredictable.

ARM interrupt procedure

Address	Exception	Mode in Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software Interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

ARM interrupt procedure

Vector address	Exception type	Exception mode	Priority (1=high, 6=low)
0x0	Reset	Supervisor (SVC)	1
0x4	Undefined Instruction	Undef	6
0x8	Software Interrupt (SWI)	Supervisor (SVC)	6
0xC	Prefetch Abort	Abort	5
0x10	Data Abort	Abort	2
0x14	*Reserved*	*Not applicable*	*Not applicable*
0x18	Interrupt (IRQ)	Interrupt (IRQ)	4
0x1C	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	3

ARM interrupt latency

- Worst-case latency to respond to interrupt is 27 cycles:
  - Two cycles to synchronize external request.
  - Up to 20 cycles to complete current instruction.
  - Three cycles for data abort.
  - Two cycles to enter interrupt handling state.
- Best-case latency is 4 cycles

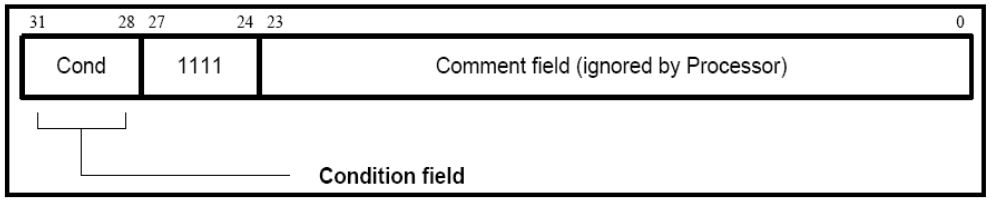
Supervisor mode

- May want to provide protective barriers between programs.
  - i. e. avoid memory interference.
- Need supervisor mode to manage the various programs.
- Not all CPUs have a supervisor mode.

ARM supervisor mode

- Use SWI instruction to enter supervisor mode, similar to call a subroutine:

```
SWI CODE_1
```



- Sets PC to 0x08.
- 24 bit argument to SWI (CODE_1) is passed to supervisor mode code to request special services (as an alternative registers r0-r3 are used).
- Saves CPSR in SPSR_SVC.
- Return, by forcing r14_SVC to PC and SPSR_SVC in CPSR

```
SWINE 0                 ;conditionally call supervisor
                        ;with 0 in comment field
```

The above examples assume that suitable supervisor code exists, for instance:

```
0x08 B Supervisor       ;SWI entry point

EntryTable              ;addresses of supervisor routines
               DCD ZeroRtn
               DCD ReadCRtn
               DCD WriteIRtn
               ...
      Zero     EQU      0
      ReadC    EQU      256
      WriteI   EQU      512

Supervisor

;SWI has routine required in bits 8-23 and data (if any) in bi
;0-7.
;Assumes R13_svc points to a suitable stack

STMFD R13,{R0-R2,R14}; save work registers and return address
LDR   R0,[R14,#-4]    ;get SWI instruction
BIC   R0,R0,#0xFF000000;
                      ;clear top 8 bits
MOV   R1,R0,LSR#8     ;get routine offset
ADR   R2,EntryTable   ;get start address of entry table
LDR   R15,[R2,R1,LSL#2];
                      ;branch to appropriate routine

WriteIRtn             ;enter with character in R0 bits 0-7
          .  .  .  .  .  .
LDMFD R13,{R0-R2,R15}^;
                      ;restore workspace and return
                      ; restoring processor mode and flags
```

Example of code for SWI management (SWI Top level Handler)

^ activates the S bit in the instruction decoding and SPSR is copied in CPSR

nts

SWI Routine written in C

- If the routines to handle each SWI in written in C
- The top-level handler uses a BL (branch and link) instruction to jump to the appropriate C function
  - BL C_SWI_Handler ;call C routine to handle
- Then, we must invoke the C routine that handles respective SWI
  - *But, how to pass the SWI number, which is now stored in r0, to the C function?*

ARM Procedure Call Convention

- Use registers *r0-r3* to pass parameter values into routines
  - Correspond to the *first* to *fourth* arguments in the C routines
- Remaining parameters are allocated to the stack in order
- A function can return
  - A one-word integer value in r0
  - A two to four-word integer value in r0-r1, r0-r2 or r0-r3.

SWI Routine in C

- Thus, the C handler is like the following

```
void  C_SWI_handler (unsigned number)
{
      switch (number)
      {
            case 0 :          /* SWI number 0 code */
                  break;
            case 1 :          /* SWI number 1 code */
                  break;
            :
            :
            default :         /* Unknown SWI - report error */
      }
}
```

SWI Routine in C (Cont.)

- However, how to pass more parameters ?
  - Make use of the *stack* (*supervisor stack*)
- The top-level SWI handler can pass the *stack pointer value (i.e. r13)* to the SWI C routine as the, for example, second parameter, i.e., *r1*
  - *sp* is pointing to the supervisor stack,

```
MOV r1, sp
BL  C_SWI_Handler
```

Exception

- Exception: internally detected error (N/0).
- Exceptions are synchronous with instructions (CPU checks if divisor is 0) but unpredictable.
- Build exception mechanism on top of interrupt mechanism.
- Exceptions are usually prioritized and vectorized.

Trap

- Trap (software interrupt): an exception explicitly generated by an instruction (undefined instruction).
  - Call supervisor mode.
- ARM uses SWI instruction for traps.

Co-processor

- Co-processor: added function unit that is called by instruction.
  - Floating-point units are often structured as co-processors.
- ARM allows up to 16 designer-selected co-processors (units).
  - Floating-point co-processor (80 bits) uses units 1 and 2 but appears as one.
- Instructions
  - CDP Coprocessor Data Processing
  - LDC Load coprocessor
  - MCR Move to Co-processor from ARM register
  - MRC Move to ARM register from Co-processor
  - STC Store coprocessor