Architettura ARM

Calcolatori Elettronici

UNIFE Ingegneria LT a.a. 2017/2018

ARM

Simulatore ARM

- http://armsim.cs.uvic.ca/Downloads/Installer.msi
- Richiede .NET Framework 3.5 https://www.microsoft.com/it-it/download/details.aspx?id=21

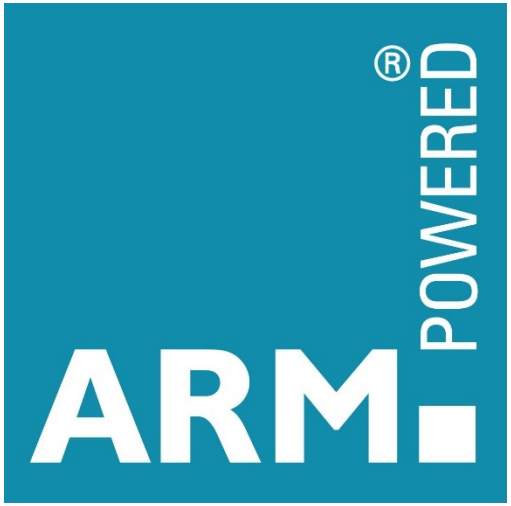
ARM

Il Consorzio

- 1990: fondata da Acorn computers, Apple Computer e VLSI technology
- Oggi detiene il 75% del mercato di processori a 32 bit per embedded

ARM

- ARM: Advanced RISC Machine
- Non produce ma licenzia i suoi design
- Fornisce il supporto necessario per sviluppare un sistema completo
- Il risultato è una serie di processori multivendor compatibili sia come linguaggio che come toolchain



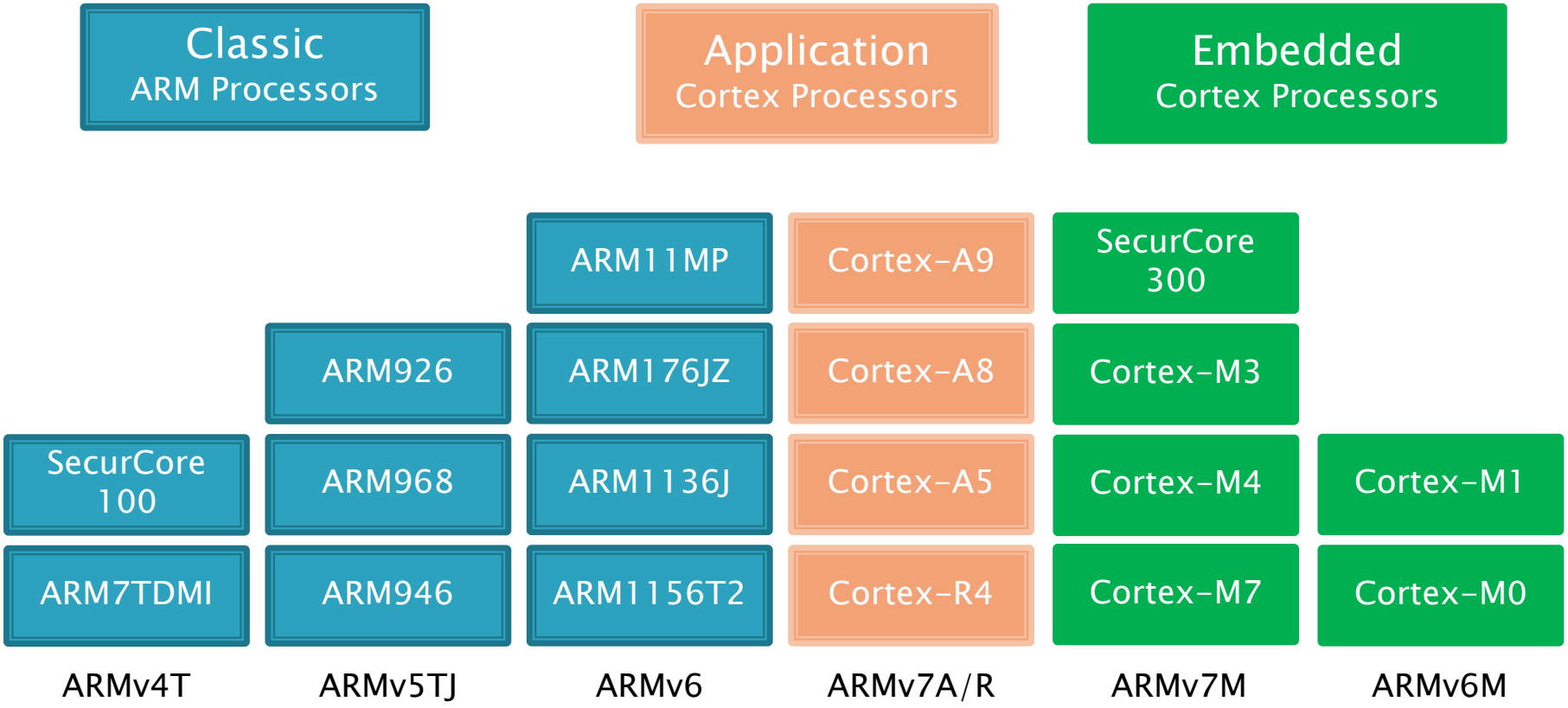
ARM

CISC vs RISC

CISC	RISC
Enfasi sull'hardware	Enfasi sul software
Istruzioni di diverse dimensioni e formati	Istruzioni della stessa dimensione con pochi formati
Meno registri	Usa più registri
Più modalità indirizzamento	Meno modalità indirizzamento
Uso esteso della microprogrammazione	Complessità nel compilatore
Le istruzioni richiedono un numero variabile di cicli di clock	Le istruzioni richiedono un solo ciclo di clock
Il pipelining è difficoltoso	Il pipelining è semplice

ARM

Le architetture



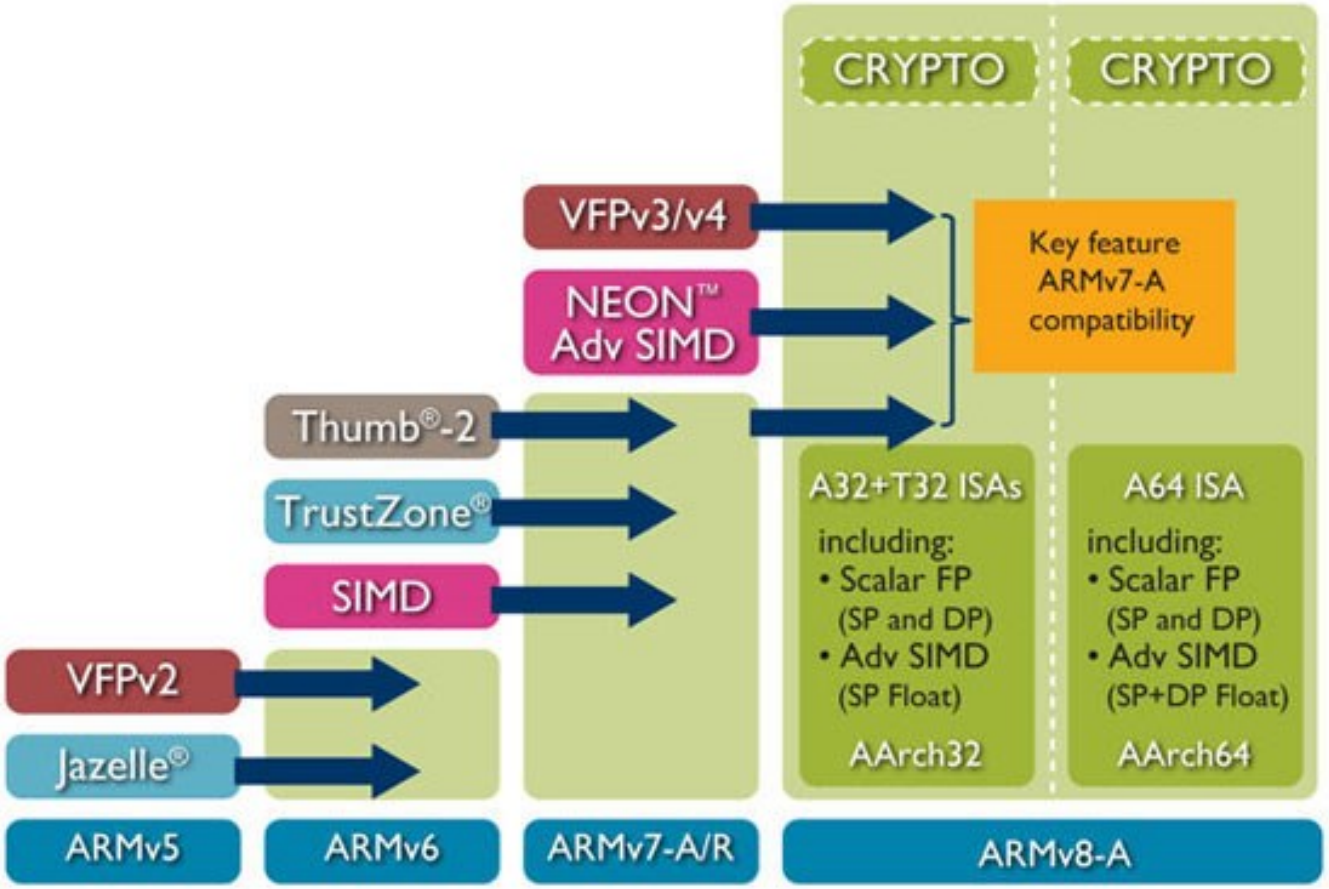
ARM

Le architetture

ARM CORE	FEATURE		
ARM v1 (obsoleto)	Istruzioni a 26 bit, nessun moltiplicatore né coprocessore		
ARM v2 (obsoleto)	Risultato a 32 bit, aggiunto coprocessore		
ARM v3 (obsoleto)	Istruzioni a 32 bit		
ARM v4 ARM v4T	Aggiunte istruzioni con segno, aggiunta Thumb mode		
ARM v5TEJ	Aggiunto supporto per algoritmi DSP e Jazelle		
ARM v6	Supporto per SIMD, aggiunto Thumb2. Aumentato supporto per virtualizzazione con TrustZone.		
ARM v6M	Creato per alte performance a basso costo.		
ARM v7	Ottimizzato THUMB2, migliorate operazioni floating point.		
	CORTEX-A	CORTEX-R	CORTEX-M
	MMU e NEON (opzionale)	Profilo real-time. MPU.	Pensato per processare velocemente interrupt. Ideale per applicazioni cost-sensitive

ARM

Le architetture



ARM

Nomenclatura prodotti

- ARMxyzTDMIEJFS:
  - X: serie;
  - Y: MMU;
  - Z: cache;
  - **T: Thumb;**
  - **D: Debugger;**
  - **M: Moltiplicatore;**
  - **I: ICE Macrocel integrato;**
  - E: istruzioni migliorate;
  - J: accelerazione Java (Jazelle);
  - F: Vector Floating-point;
  - S: versione sintetizzabile

ARM

Thumb

- E' un instruction set a 16 bit
  - Ottimizzato per la densità di codice dal C
  - Migliori performance per memorie piccole
  - Sottoinsieme di funzionalità dell'instruction set ARM
- Per la maggior parte delle istruzioni
  - L'esecuzione condizionale non è usata
  - Registro sorgente e destinazione sono identici
  - Solo i registri r0-r7 sono usati
  - Le costanti sono di dimensione limitata
  - Il barrel shifter inline non è usato

ARM

Le architetture

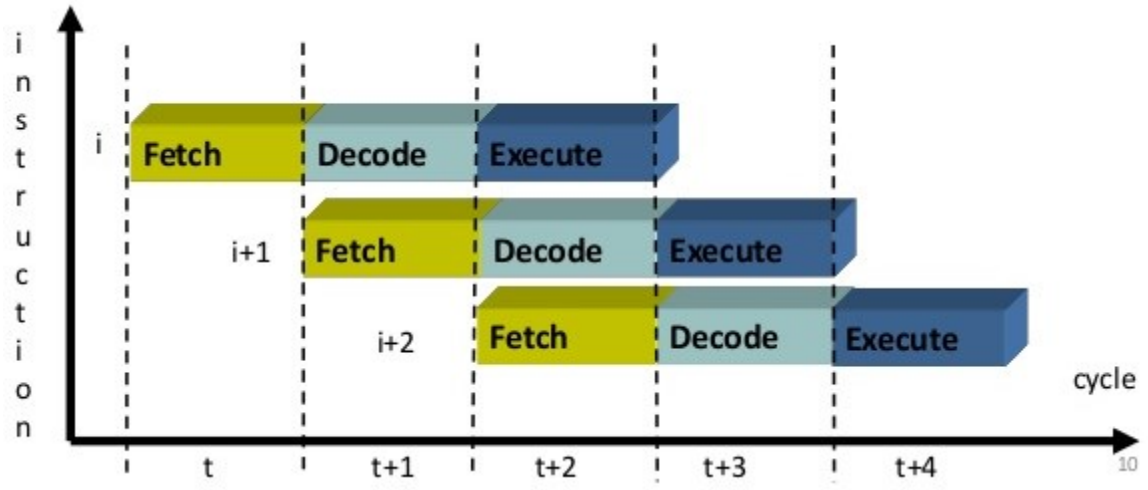
- I processori ARM sono largamente usati in embedded;
- Sono noti per il loro basso consumo e per il processing di fascia alta;
- ARM7TDMI è il loro core di maggior successo:
  - 1 miliardo di dispositivi spediti ogni 4 mesi;
  - Più di 90 al secondo;
  - Oltre 500 licenze;

Pipelining

ARM

Pipelining

- Inizialmente l'organizzazione era a 3 stadi di pipeline (fino a ARM7):
  - Fetch
  - Decode
  - Execute



ARM

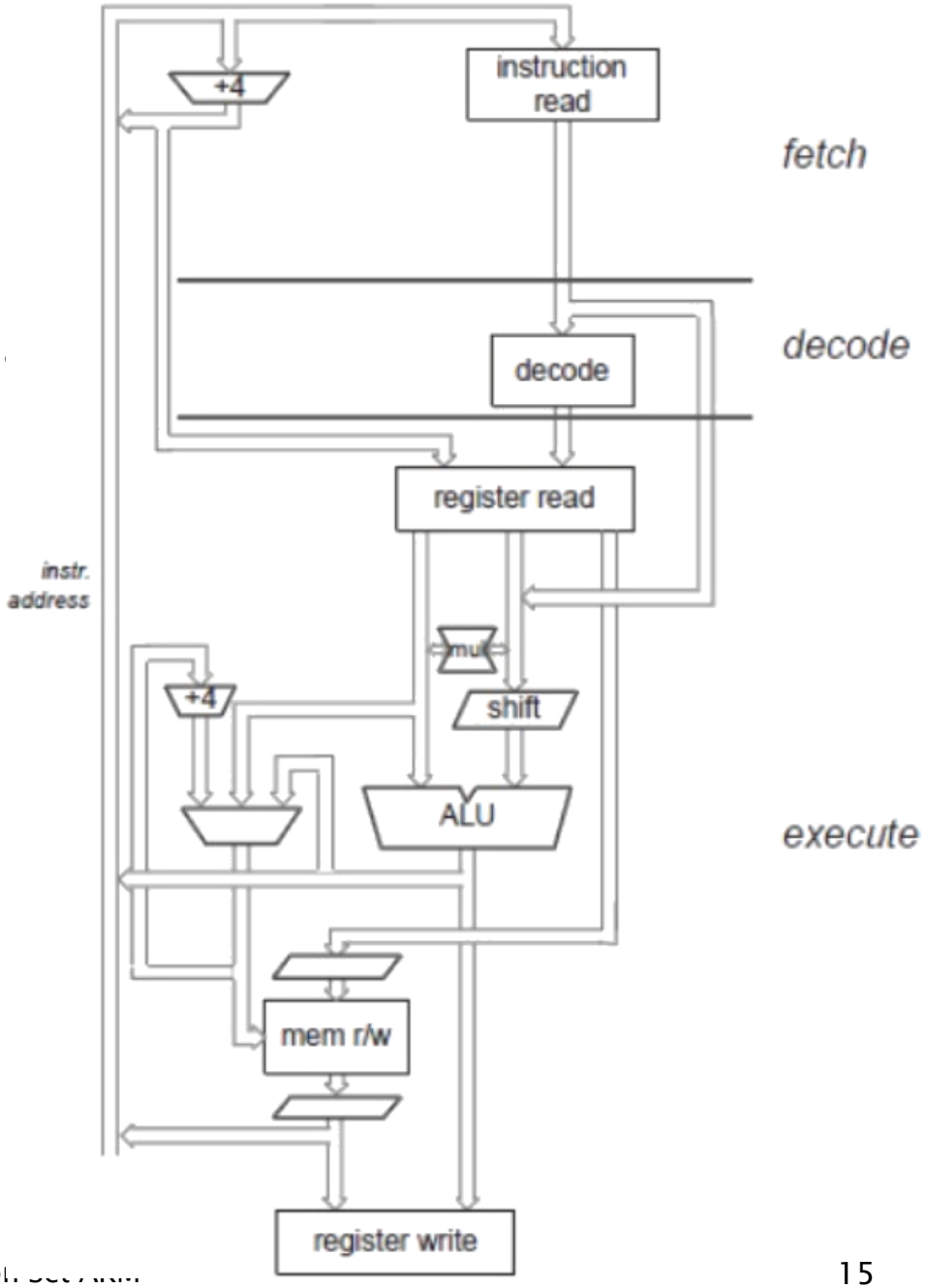
Pipelining

- L'organizzazione della pipeline a 3 stadi prevede:
  - Il banco di registri
  - Il barrel shifter
  - ALU
  - L'address register e l'incrementatore
  - I registri dati
  - Il decoder di istruzioni e la logica di controllo associata

ARM

Pipelining

- Fetch: l'istruzione è prelevata dalla memori e messa nella pipeline
- Decode: l'istruzione è decodificata e il percorso del segnale preparato per il ciclo successivo
- Execute: i banchi di registro sono letti, gli operandi shiftati, il risultato ALU generato scritto nel registro destinazione



ARM

Pipelining

- In ogni momento 3 diverse istruzioni possono occupare ognuno dei tre stadi. L'hardware di ogni stadio deve poter operare in maniera indipendente
- Quando il processore esegue istruzioni di processamento dati la latenza è pari a 3 cicli e il throughtput è 1 istruzione a ciclo
- Lato negativo: ogni istruzione di trasferimento dati causa uno stallo della pipeline. Essendo la memoria unica per dati e istruzioni non è possibile leggere l'istruzione successiva mentre si leggono dati.

ARM

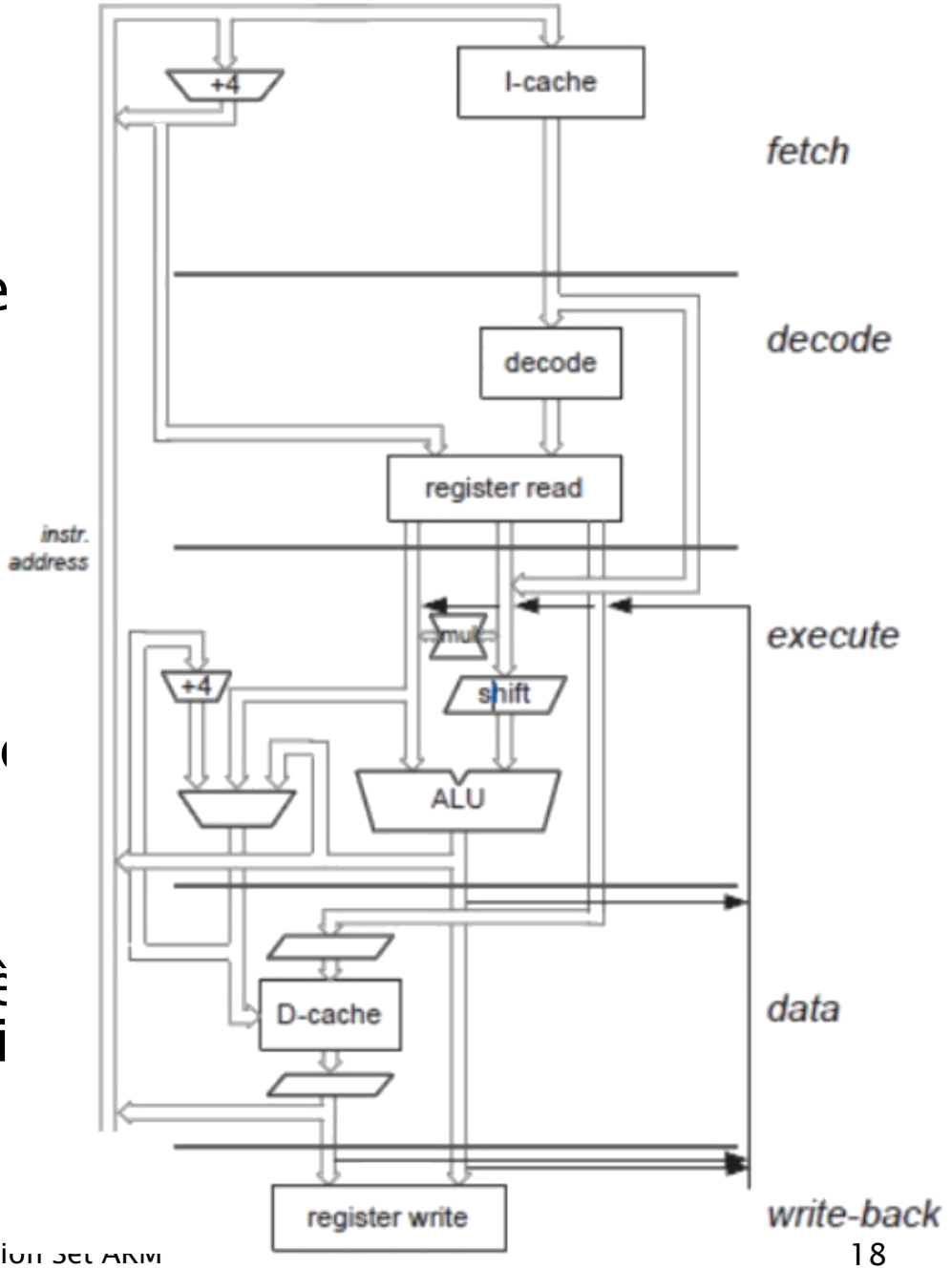
Pipelining

- Dall'architettura ARM9TDMI la pipeline è diventata a 5 stadi.
- $T_{prog} = \frac{N_{inst} * CPI}{f_{clk}}$
  - T_{prog}: tempo per eseguire un determinato programma
  - N_{inst}: numero di istruzioni ARM eseguite nel programma
  - CPI: numero medio di cicli di clock per istruzione
  - F_{clk}: frequenza di clock

ARM

Pipelining

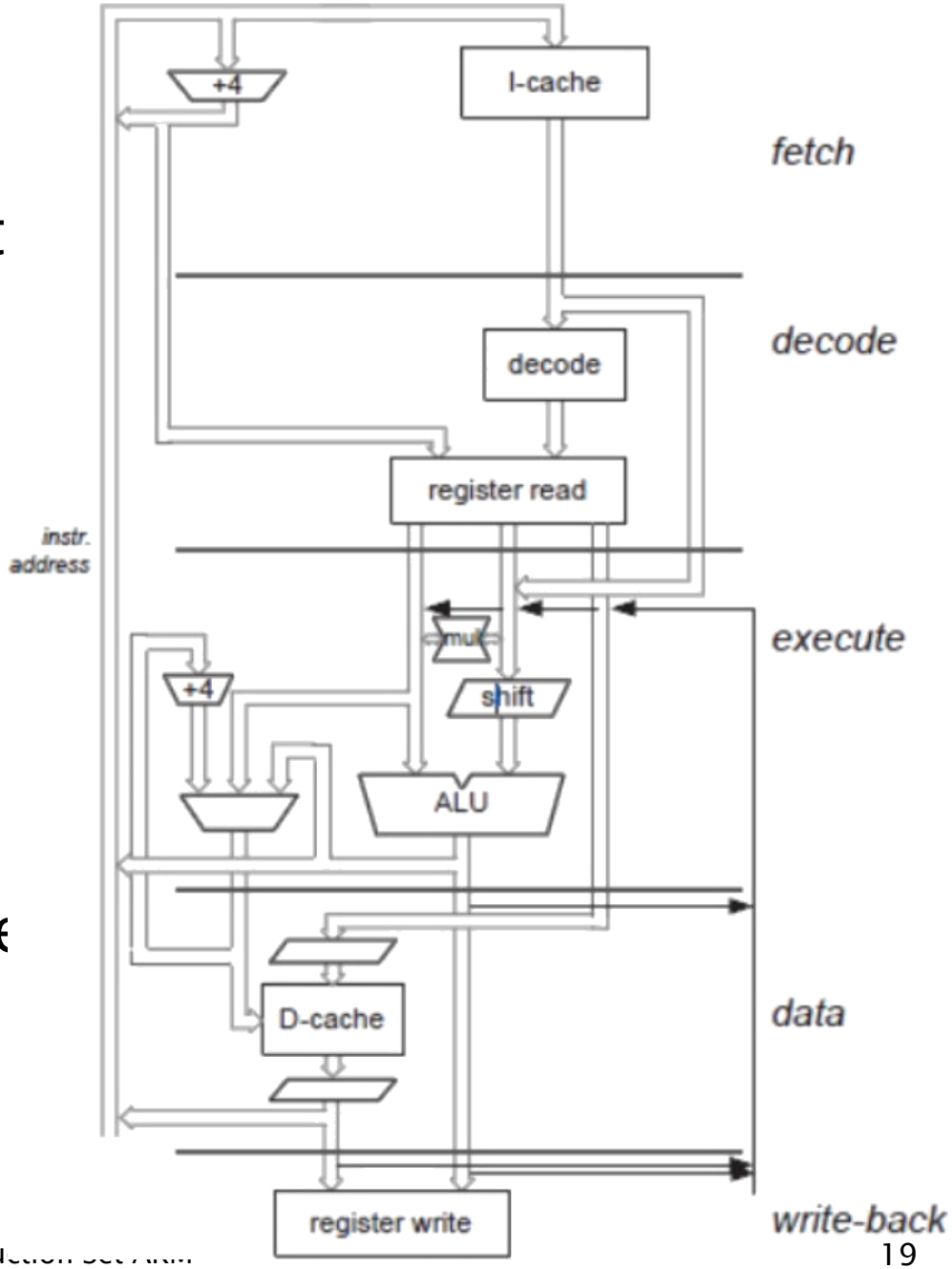
- Fetch: l'istruzione è prelevata dalla memoria e messa nella pipeline
- Decode: l'istruzione è decodificata e i registri operandi letti. Ci sono 3 porte di lettura per gli operandi in modo che la maggior parte delle istruzioni possano leggere gli operandi in un ciclo
- Execute: un operando è shiftato e il risultato ALU generato. Se l'istruzione è load o store, l'indirizzo di memoria è computato nell'ALU



ARM

Pipelining

- Buffer/Data: se richiest viene effettuato l'accesso alla memoria dati. In caso contrario i risultato della ALU è semplicemente bufferizzato per un ciclo.
- Write back: il risultato generato dall'istruzione è scritto nel registro, incluso qualsiasi dato caricato dalla memoria



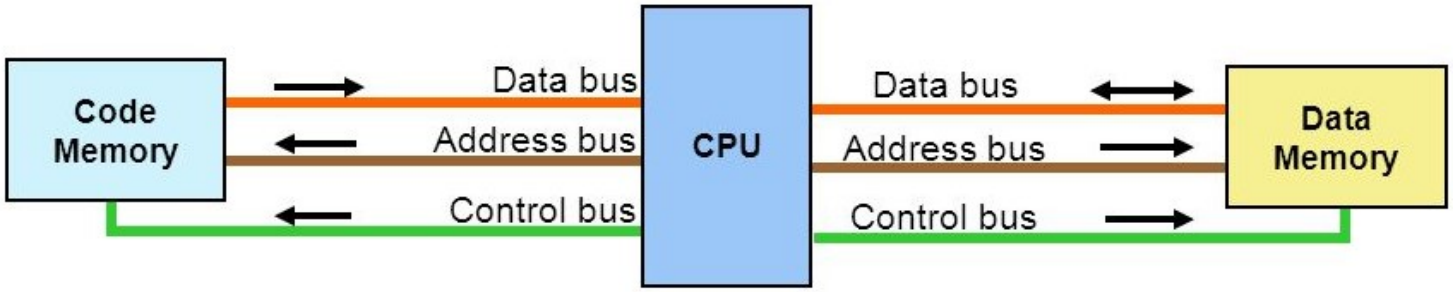
ARM

Pipelining

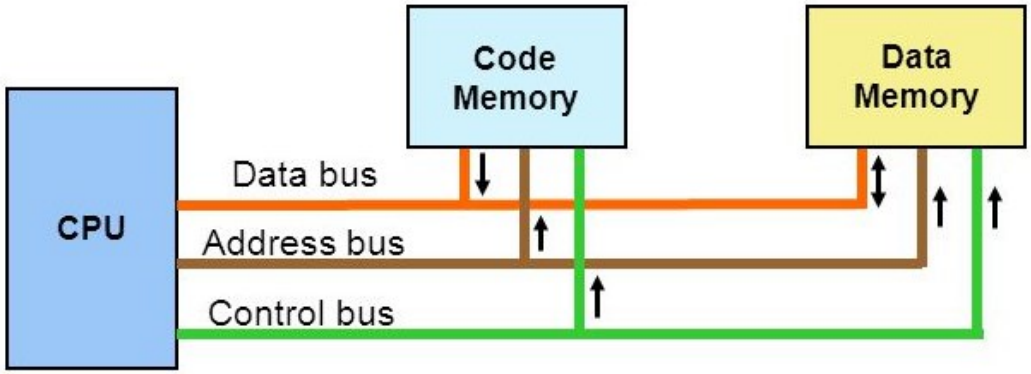
- Differenze tra pipeline a 5 stadi e pipeline a 3:
  - Spostato lo step di lettura del registro dall'execute al decode;
  - Diviso in 3 l'execute stage: ALU, memory access, write back;
  - Risultato: pipeline meglio bilanciata, con latenze minimizzate tra stage, che possono girare a clock maggiori

ARM

Hardvard vs Von Neumann



Harvard architecture



Von Neumann architecture

ARM

Pipelining

- Esistono situazioni, chiamate hazards, che impediscono l'esecuzione durante il suo ciclo designato della prossima istruzione nello stream.
- Gli hazard riducono le performance rispetto allo speed-up ideale guadagnato con il pipelining.
- Ci sono tre classi di hazard:
  - Hazard strutturali
  - Hazard di dati
  - Hazard di controllo

ARM

Pipelining

- Quando una macchina è basata su pipeline l'esecuzione sovrapposta di istruzioni richiede il pipelining di unità funzionali e la duplicazione di risorse per permettere tutte le combinazioni possibili di istruzioni nella pipeline
- Se qualche combinazione di istruzioni non può essere eseguita a causa di un conflitto di risorse la macchina ha un hazard strutturale

ARM

Pipelining

- Esempio: una macchina ha un'unica pipeline di memoria condivisa per dati e istruzioni. Come risultato, quando un'istruzione contiene un load della memoria dati, andrà in conflitto con il fetch della successiva istruzione (accesso a memoria istruzioni).

	CLOCK CYCLE NUMBER							
INSTR	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Load	IF	ID	EX	**MEM**	WB			
Instr1		IF	ID	EX	MEM	WB		
Instr2			IF	ID	EX	MEM	WB	
Instr3				**IF**	ID	EX	MEM	WB

ARM

Pipelining

- Soluzione1: inserire un ciclo di stall quando viene fatto l'accesso alla memoria. Ovviamente questo fa perdere efficienza.

	CLOCK CYCLE NUMBER								
INSTR	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Load	IF	ID	EX	MEM	WB				
Instr1		IF	ID	EX	MEM	WB			
Instr2			IF	ID	EX	MEM	WB		
Instr3				stall	IF	ID	EX	MEM	WB

ARM

Pipelining

- Soluzione2: cambiare architettura. Se ho memoria dati e memoria istruzioni separate il problema non si pone.
- ARM è passata da un'architettura von-Neumann ad una Harvard con gli ARM9:
  - Ha implementato una pipeline a 5 stadi con memoria dati e memoria istruzioni separate
  - Non soffre di questo tipo di hazard

ARM

Pipelining

- Un hazard di dati si ha quando un'istruzione dipende dal risultato di una precedente
- Esempio:

CLOCK CYCLE NUMBER							
	1	2	3	4	5	6	7
ADD R1,R2,R3	IF	ID	EX_{add}	MEM_{add}	WB		
SUB R4,R5,R1		IF	ID	EX_{sub}	MEM	WB	
AND R6,R1,R7			IF	ID	EX_{and}	MEM	WB

ARM

Pipelining

- Soluzione1: inserire uno stall -> perdita di efficienza
- Soluzione2: inserire percorsi che permettano il forwarding del dato, cioè prevedere la possibilità di passare il risultato direttamente all'unità funzionale che ne ha bisogno.

ARM

Pipelining

- L'ultimo tipo di hazard è legato alle branch e altri tipi di istruzioni che cambiano il PC

	CLOCK CYCLE NUMBER								
INSTR	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Branch	IF	ID	EX	MEM	WB				
Branch+1		IF	stall	stall	IF	ID	EX	MEM	WB
Branch+2						IF	ID	EX	MEM
Branch+3							IF	ID	EX

ARM: struttura del core

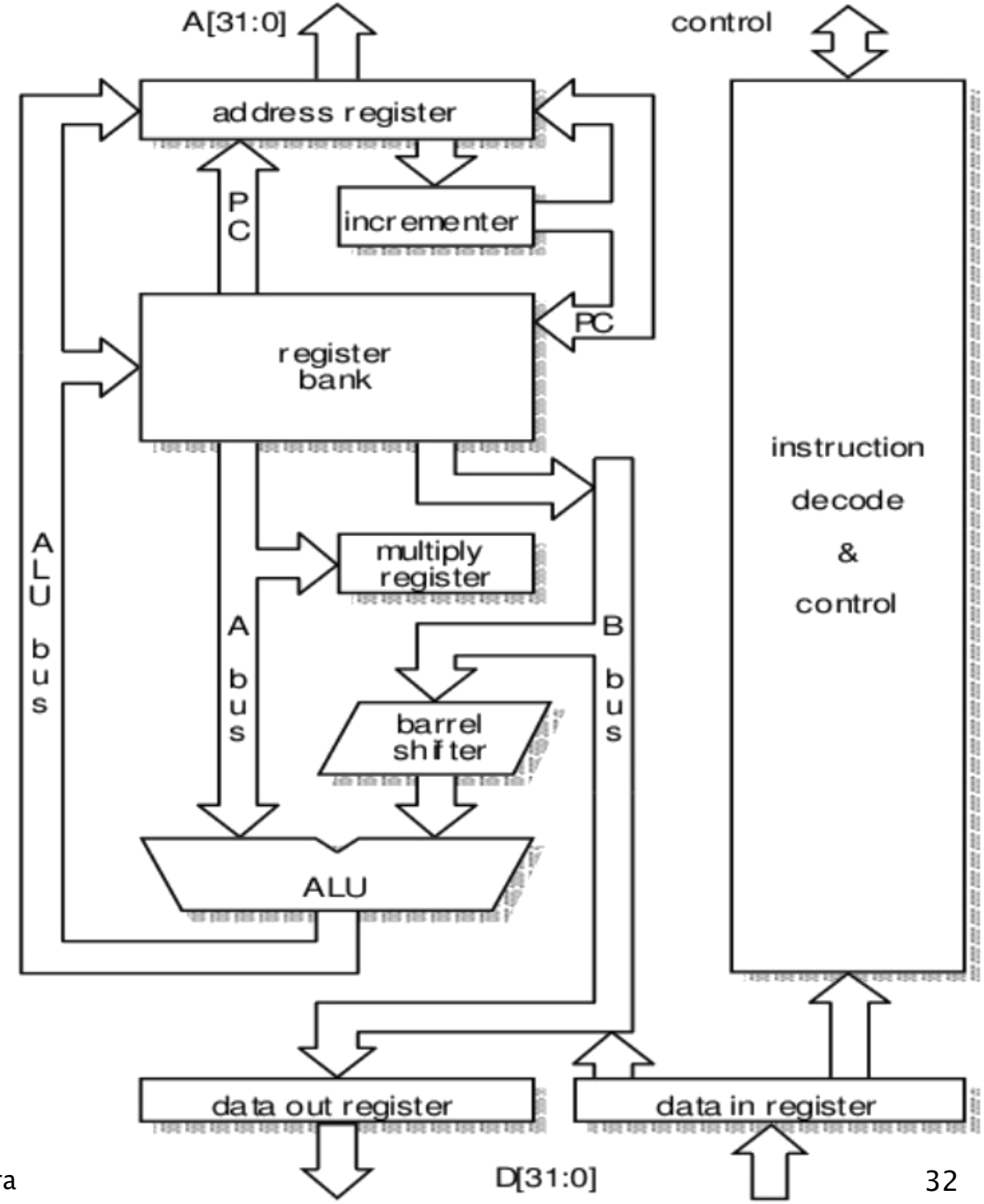
ARM

Architettura load/store

- **L'instruction set processa solo i valori nei registri**
- L'unica operazione che si applica alla memoria sono quelle di load (valori memoria copiati nei registri) e di store (valori dei registri copiati nella memoria)
- ARM non supporta le operazioni memory-to-memory
- Tutte le istruzioni sono racchiuse in tre categorie:
  - Istruzioni di processamento dati
  - Istruzioni di trasferimento dati
  - Istruzioni di controllo di flusso

ARM

Architettura



ARM

Stati instruction set

- Instruction set a 32 bit: ARM
  - Architettura load/store in cui ogni istruzione è condizionale;
- Instruction set a 16 bit: Thumb
  - Solo le istruzioni di branch sono condizionali e solo metà dei registri sono usati;
- Instruction set a 8 bit: Jazelle
  - Permette esecuzione di Java byte code in hardware

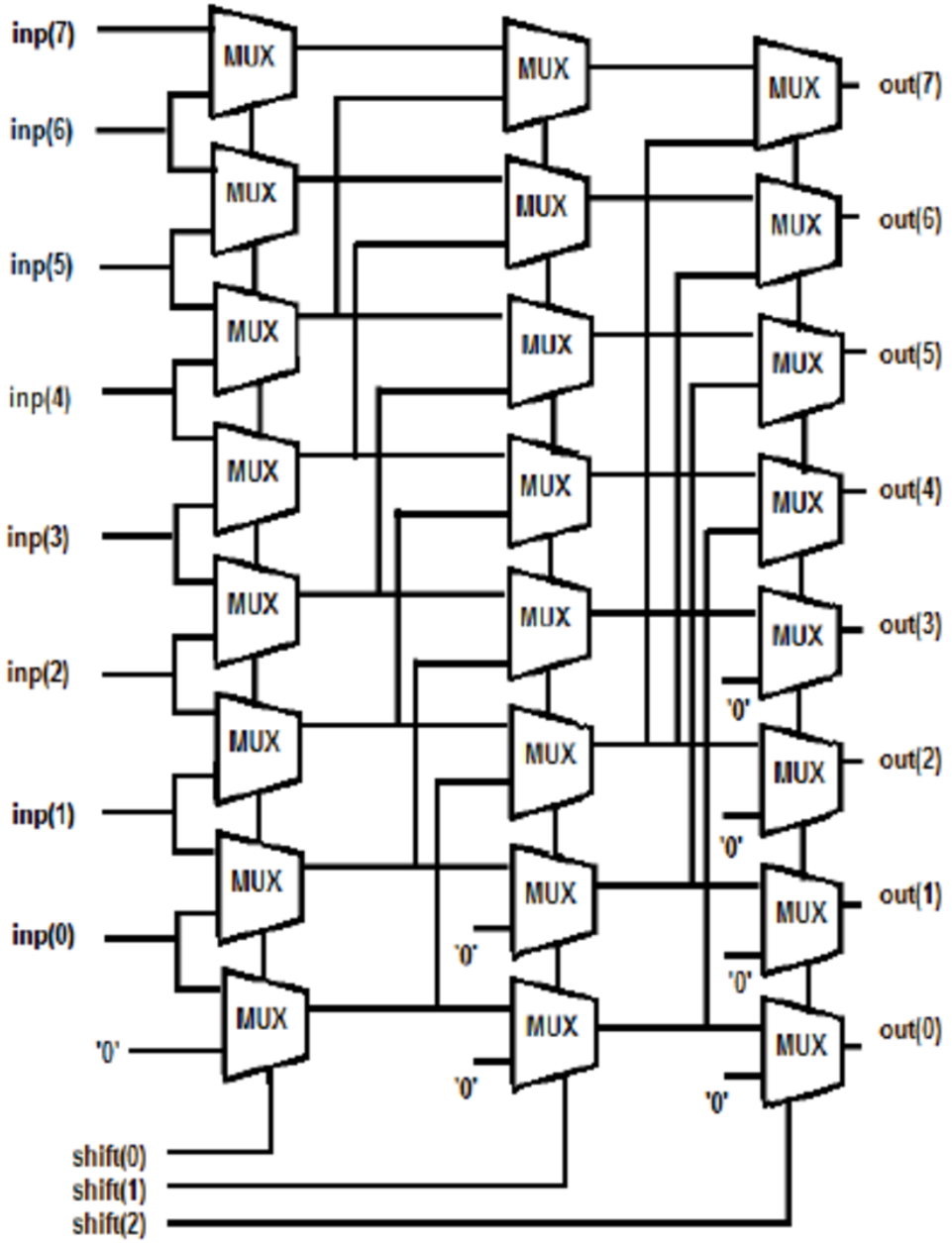
ARM

Barrel shifter

- Un barrel shifter è un circuito digitale che può shiftare una word di un numero specificato di bit in un solo ciclo di clock
- Può essere implementato come sequenza di multiplexer, con l'output di un mux connesso come ingresso del successivo in base alla distanza dello shift
- Di solito è realizzato come cascade di mux 2x1 in parallelo.
- Ci sono 2 opzioni da scegliere:
  - Tipo di shift (logico a destra, logico a sinistra, aritmetico a destra, rotazione a destra)
  - Distanza dello shift

ARM

Barrel shifter



ARM

Complemento a 2

- Per la rappresentazione dei numeri interi con segno si sfrutta il complemento a 2
- Il bit più significativo rappresenta il segno (1=numero negativo, 0=numero positivo)
- Da un numero binario positivo si ottiene il negativo seguendo due step:
  - Negare tutti i bit
  - Aggiungere 1
- Da binario negativo a positivo stessa cosa:
  - Negare tutti i bit
  - Aggiungere 1

ARM

Complemento a 2

- Esempio:
  - 5(dec) = 0000 0101
  - Step 1: negazione bit a bit. Ottengo: 1111 1010
  - Step 2: aggiungo 1. Ottengo: 1111 1011
- Volendo trovare il positivo a partire dal negativo:
  - -5(dec) = 1111 1011
  - Step 1: negazione bit a bit. Ottengo: 0000 0100
  - Step 2: aggiungo 1. Ottengo: 0000 0101

ARM

Complemento a 2

- La rappresentazione è un'astrazione. Siamo noi a decidere se considerare il binario come intero con segno o intero senza segno
- Esempio
  - Il numero dell'esempio precedente (1111 1011) può valere sia -5 che 251.
- Come **INTERPRETARE** il numero binario che abbiamo sotto mano dipende da noi

ARM

Complemento a 2

- Le operazioni aritmetiche sono indipendenti da come noi decidiamo di INTERPRETARE il numero.
- Esempio:
  - -6 = 1111 1010 = 250
  - 5 = 0000 0101 = 5
  - Sommando i due numeri binari ottengo:
  - 1111 1111
  - -1 = 1111 1111 = 255

ARM

Barrel shifter

LSL : Logical Left Shift

Multiplication by a power of 2

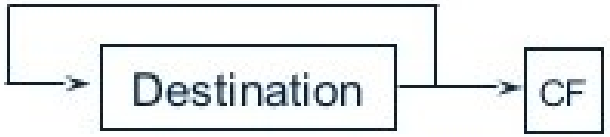
ASR: Arithmetic Right Shift

Division by a power of 2, preserving the sign bit

LSR : Logical Shift Right

Division by a power of 2

ROR: Rotate Right



Bit rotate with wrap around from LSB to MSB

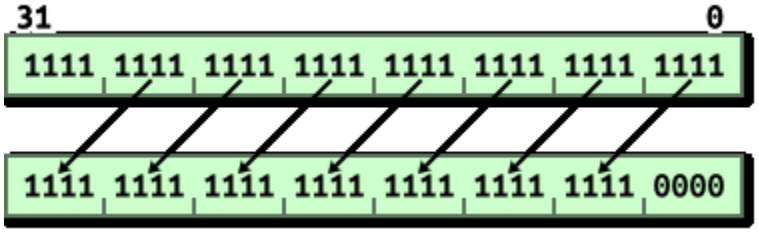
ARM

Barrel shifter

LSL : Logical Left Shift

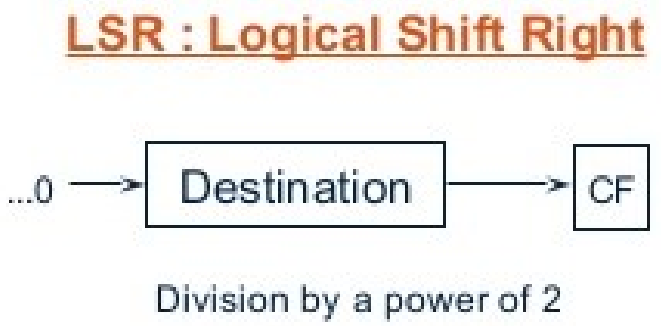
Multiplication by a power of 2

LSL di 4 bit

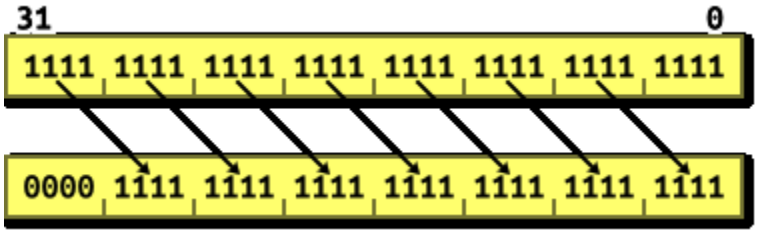


ARM

Barrel shifter



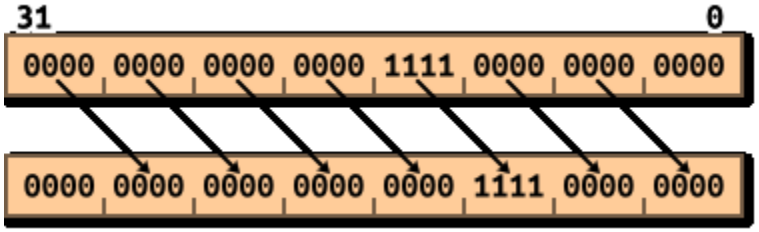
LSL di 4 bit



ARM

Barrel shifter

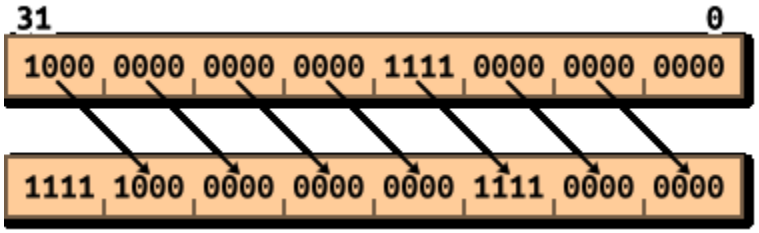
ASR di 4 bit,
valore positivo



ARM

Barrel shifter

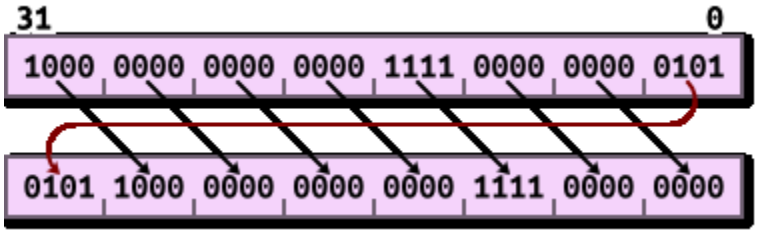
ASR di 4 bit,
valore negativo



ARM

Barrel shifter

ROR di 4 bit



ARM

Barrel shifter

- **In ARM mode non sono considerate come operazioni!!**
- **Si possono applicare all'operando 2 di altre operazioni**
- Sintassi:
  - `op #expr`
  - `op Rm`
- Op: può essere ASR, LSL, LSR, ROR
- #expr: distanza dello shift da fare su Rm
- Rm: registro che contiene la distanza dello shift
- Esempi:
  - `MOV r4, r6, LSL #4`
  - `MOV r4, r6, LSL r3`

ARM

Instruction set ARM

- Architettura load/store
- Esecuzione condizionale di ogni istruzione
- Inclusione di istruzioni load/store per registri multipli
- Abilità di effettuare un'operazione di shift e una di ALU in un'unica istruzione che viene eseguita in un unico ciclo

ARM

Interrupt

- Un interrupt è un segnale asincrono che indica il bisogno di attenzione di una periferica
- Oppure un evento sincrono che consente l'interruzione di un processo sotto determinate condizioni
- Può essere:
  - Hardware: generati da dispositivi esterni alla CPU (periferiche)
  - Software: istruzioni assembly, assimilate a chiamate di sottoprogrammi

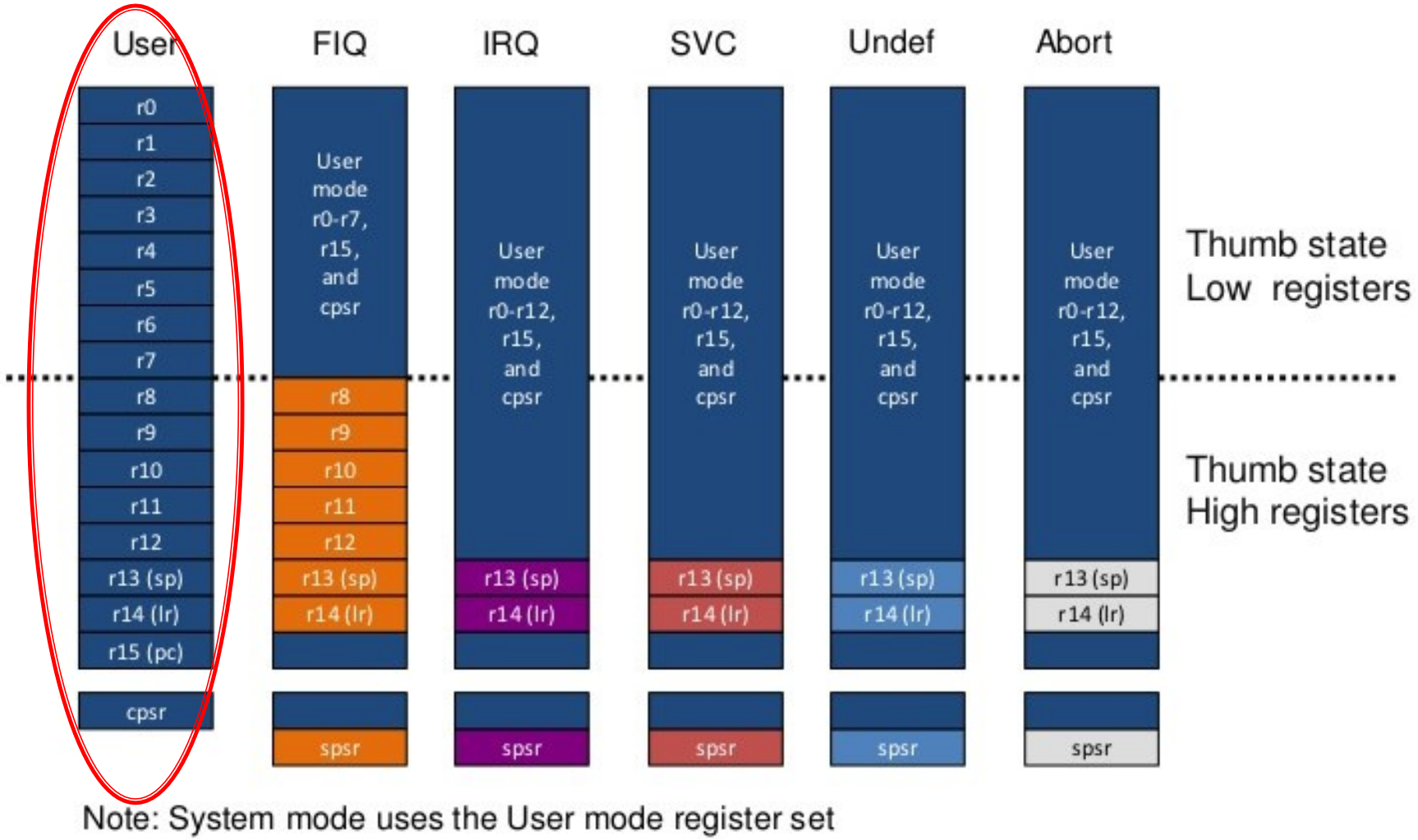
ARM

Modalità processore

- Ne esistono 7:
  - User: modalità NON privilegiata sotto cui girano la maggior parte dei task;
  - FIQ: ci si entra quando viene sollevato un interrupt ad alta priorità;
  - IRQ: ci si entra quando viene sollevato un interrupt a bassa priorità;
  - Supervisor: ci si entra al reset e quando un'istruzione di software interrupt (SWI) viene eseguita;
  - Undef: usato per gestire violazioni all'accesso di memoria;
  - System: modalità privilegiata che usa gli stessi registri dell'user mode

ARM

Organizzazione registri



Note: System mode uses the User mode register set

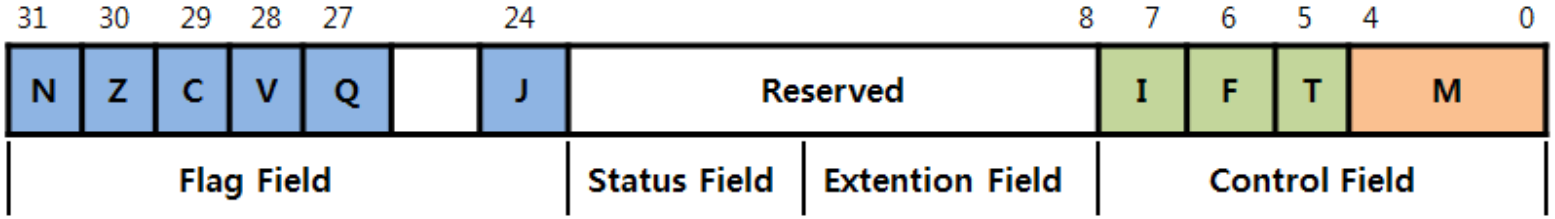
ARM

Organizzazione registri

- ARM ha 37 registri, tutti a 32 bit:
  - 1 dedicato al program counter;
  - 1 dedicato al current program status;
  - 5 dedicati al saved program status;
  - 30 general purpose;
- La modalità processore decide quale dei banchi di registri sia accessibile. Ogni modalità può accedere a:
  - Un set dedicato di r0-r12 registri (general purpose);
  - Un r13 (stack pointer) e un r14 (link register);
  - Un r15 (program counter);
  - Un CPSR (current program status register);
- Le modalità priviligiate (tranne System) hanno anche accesso a SPSR (saved program status register);

ARM

Organizzazione registri



Flag Field	
N	Negative result from ALU
Z	Zero result from ALU
C	ALU operation caused Carry
V	ALU operation oVerflowed
Q	ALU operation saturated
J	Java Byte Code Execution

Control bits	
I	1: disables IRQ
F	1: disables FIQ
T	1: Thumb, 0: ARM

Mode bits M[4:0}	
0b10000	User
0b11111	System
0b10001	FIQ
0b10010	IRQ
0b10011	SVC(Supervisor)
0b10111	Abort
0b11011	Undefined

ARM

Program counter (r15)

- Quando il processore esegue in ARM state:
  - Tutte le istruzioni sono larghe 32 bit;
  - Tutte le istruzioni devono essere allineate a word;
  - Quindi il PC è immagazzinato nei bit [31:2] con i bit [1:0] indefiniti;
- Quando il processore esegue in Thumb state:
  - Tutte le istruzioni sono larghe 16 bit;
  - Le istruzioni devono essere allineate a half-word;
  - Il valore del PC è immagazzinato nei bit [31:1] con il bit [0] indefinito;
- Quando il processore esegue in Jazelle state:
  - Tutte le istruzioni sono larghe 8 bit;
  - Il processore effettua un accesso a word e legge 4 istruzioni in una volta;

ARM

Eccezioni

- Le eccezioni previste in ARM sono

Reset	Il pin di reset va alto
Istruzione indefinita	Il processore non riconosce l'istruzione in esecuzione
Software interrupt (SWI)	Istruzione di SWI chiamata dal programma
Interruzione del prefetch	Il processore ha cercato di eseguire un'istruzione non fetchata perché l'indirizzo è illegale
Interruzione dati	Avviene quando un'istruzione di trasferimento dati cerca di caricare o immagazzinare un dato ad un indirizzo illegale
IRQ	Pin di IRQ esterno che va alto
FIQ	Pin di FIQ esterno che va alto

ARM

Eccezioni

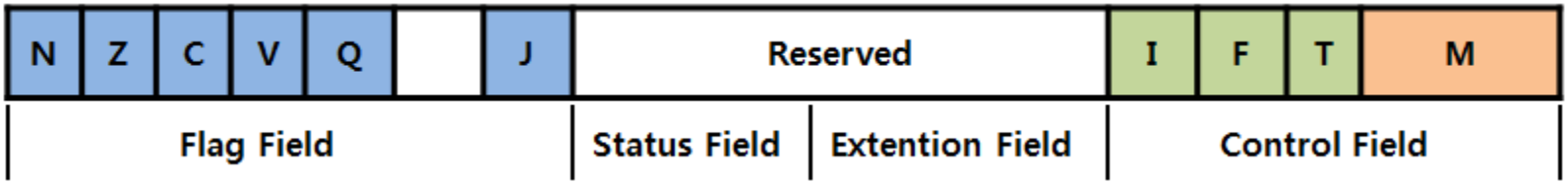
Exception	Offset from vector base	Mode on entry	A bit on entry	F bit on entry	I bit on entry
Reset	0x00	Supervisor	Disabled	Disabled	Disabled
Undefined instruction	0x04	Undefined	Unchanged	Unchanged	Disabled
Software interrupt	0x08	Supervisor	Unchanged	Unchanged	Disabled
Prefetch Abort	0x0C	Abort	Disabled	Unchanged	Disabled
Data Abort	0x10	Abort	Disabled	Unchanged	Disabled
Reserved	0x14	Reserved	-	-	-
IRQ	0x18	IRQ	Disabled	Unchanged	Disabled
FIQ	0x1C	FIQ	Disabled	Disabled	Disabled

ARM

Gestione eccezioni

- Al verificarsi di un'eccezione l'ARM:
  - Copia CPSR in SPSR;
  - Setta correttamente CSPR (cambia in stato ARM, cambia in exception mode, disabilita gli interrupt);
  - Immagazzina l'indirizzo di ritorno in LR;

Address	Interrupt Type
0x20	reset
0x24	undefined instruction
0x28	SWI (software interrupt)
0x2C	prefetch abort
0x30	data abort
0x34	(reserved for future use)
0x38	IRQ
0x3C	FIQ



ARM: modalità indirizzamento

ARM

Istruzioni spostamento

- Le operazioni di spostamento sono:
  - `MOV operand2;`
  - `MVN operand2;`
- Da notare che non si fa uso di operand 1. La sintassi è:
  - `<Operation>(<cond>){S}Rd,Operand2;`
- Esempi:
  - `MOV r0,r1;`
  - `MOVS r2,#10;`
  - `MVNEQ r1,#0;`

ARM

Istruzioni aritmetiche: ADD

- L'operazione di addizione è la seguente:
  - `ADD operand1+operand2;`
- La sintassi è:
  - `<Operation>(<cond>){S}Rd,Rn,Operand2;`
- Esempi:
  - `ADD  r0,r1,r2;`

ARM

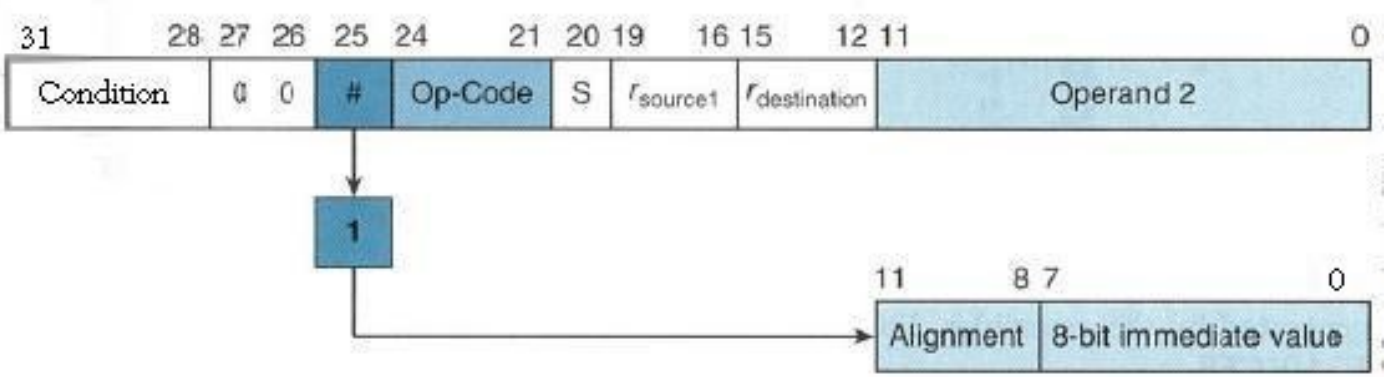
Trasferimento dati

- Il semplice load/store ha diverse opzioni, come le seguenti:
  - LDR/STR -> usato per caricare/immagazzinare words (32 bit)
- La sintassi è:
  - `<LDR|STR>(<cond>)Rd,<address>`
- Si può operare anche su blocchi inferiori alla word:
  - LDRB/STRB -> usato con byte (8bit)
  - LDRH/STRH -> half word (16 bit)

ARM

Modalità indirizzamento: immediato

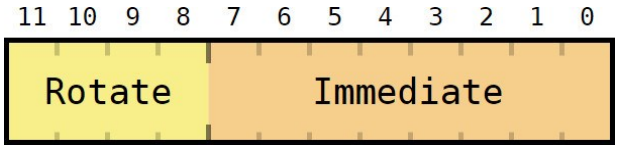
- Esempi:
  - `ADD R1, R2, #18` `(R1 = R2 + 18)`
  - `MOV R1, #30` `(R1 = 30)`
  - `MOV R1, #0xFF` `(R1 = 0xFF)`
  - `MOV R2, #0xFF000000` `(R2 = 0xFF000000)`

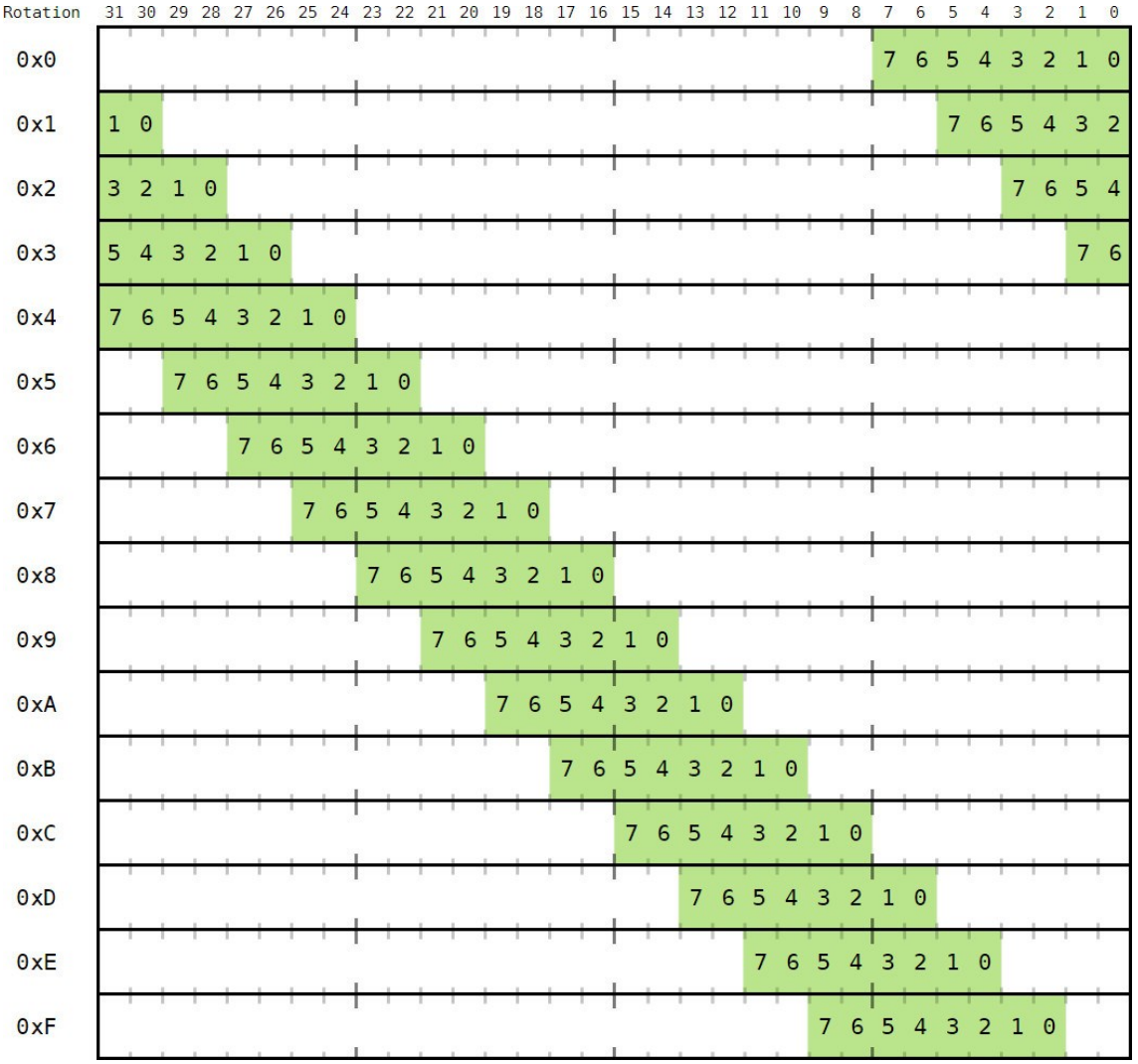


ARM

Modalità indirizzamento: immediato

- Con uno spazio a 12 bit si ottengono numeri a 32 bit
- Ovviamente alcuni valori non sono codificabili





ARM

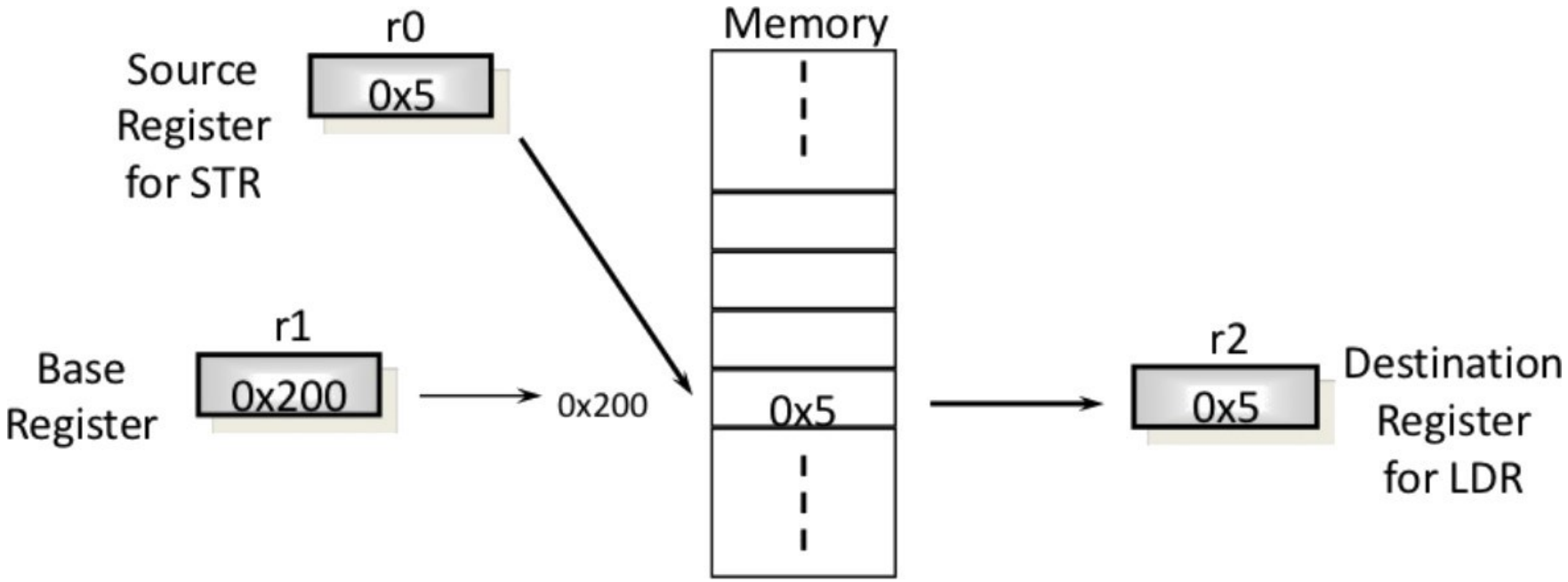
Modalità indirizzamento: registro indiretto

- Chiamato anche indirizzamento indicizzato o indirizzamento base
- Il registro contiene l'indirizzo di memoria in cui trovare l'operando
- Stesso principio dei puntatori in C
- Utilizzabile solo con operazioni di load/store
- Esempi:
  - `LDR R2, [R0]`
  - `STR R2, [R3]`

ARM

Trasferimento dati

- `STR r0,[r1]` Immagazzina il contenuto di r0 all'indirizzo indicato da r1
- `LDR r2,[r1]` Carica in r2 il contenuto dell'indirizzo indicato da r1



ARM

Modalità indirizzamento: registro indiretto con offset

- ARM permette anche di accedere a locazioni puntate come offset dall'indirizzo base
- L'offset può essere:
  - Immediate value, unsigned a 12 bit
  - Registro
- L'offset può essere aggiunto o sottratto
- Si può applicare:
  - Prima di fare il trasferimento
  - Dopo aver fatto il trasferimento (auto-incremento del base register)

ARM

Modalità indirizzamento: offset semplice

- L'offset più semplice è rappresentato da un numero immediato (o registro):
  - `STR r0,[r1,#12]` scrive all'indirizzo R1+12
  - `STR r0,[r1,#-12]` scrive all'indirizzo R1-12
  - `STR r0,[r1,r2]` scrive all'indirizzo R1+R2

ARM

Modalità indirizzamento: offset con auto-indicizzazione

- L'offset è un numero immediato (o registro)
- L'offset viene applicato all'operazione di load/store corrente
- Dopo aver effettuato l'operazione di load/store viene incrementato il registro base
  - `STR r0,[r1,#4]!` scrive all'indirizzo indicato in R1, sommandogli 4. Dopo la scrittura incrementa R1 di 4
  - È un preincremento che si accumula

ARM

Modalità indirizzamento: offset con auto-indicizzazione

- L'offset è un numero immediato (o registro)
- L'offset **NON** viene applicato all'operazione di load/store corrente
  - `STR r0,[r1],#4` — scrive all'indirizzo indicato da R1. Dopo la scrittura incrementa R1 di 4
  - È un POST incremento

ARM

Modalità indirizzamento: relativo al Program Counter

- R15 è il program counter
- Si può usare come indirizzo base per accedere agli operandi
  - `LDR r0,[R15,#4]`
  - `LDR r0,[pc,#4]`

ARM

Modalità indirizzamento: sommario delle modalità indicizzate

Assembly	Indirizzo	Valore finale in R1
LDR R0, [R1, #d]	R1+d	R1
LDR R0, [R1, #d] !	R1+d	R1+d
LDR R0, [R1], #d	R1	R1+d

ARM

Modalità indirizzamento: assoluto

- Sfrutta le label
- Permette di recuperare l'indirizzo di una variabile immagazzinata in memoria
- L'accesso richiede due passaggi:
  - Mettere in un registro l'indirizzo della label
  - Caricare/immagazzinare da/verso l'indirizzo
- Esempio:

```
LDR R0, =OPERAND1     ;recupero l'indirizzo di
LDR R1, [R0]          ;operand 1, dopodichè
                      ;carico in r1 il suo
.data                 ;valore
operand1: .word 1
```

ARM

Modalità indirizzamento: assoluto

- Si nota l'uso di .data:
  - E' una pseudo istruzione, cioè non si tratta di un'istruzione assembler ma di una direttiva per il compilatore.
  - .data stabilisce l'inizio dello spazio per i dati. Il compilatore non interpreta più ciò che segue come istruzioni.
- Esistono diversi standard di assembler
- Quello usato in queste lezioni è il GNU Assembler.
- Non è detto che in altri standard la direttiva .data sia accettata, potrebbe essere necessario usare una direttiva diversa per la stessa operazione
- Stesso discorso vale per .word

ARM

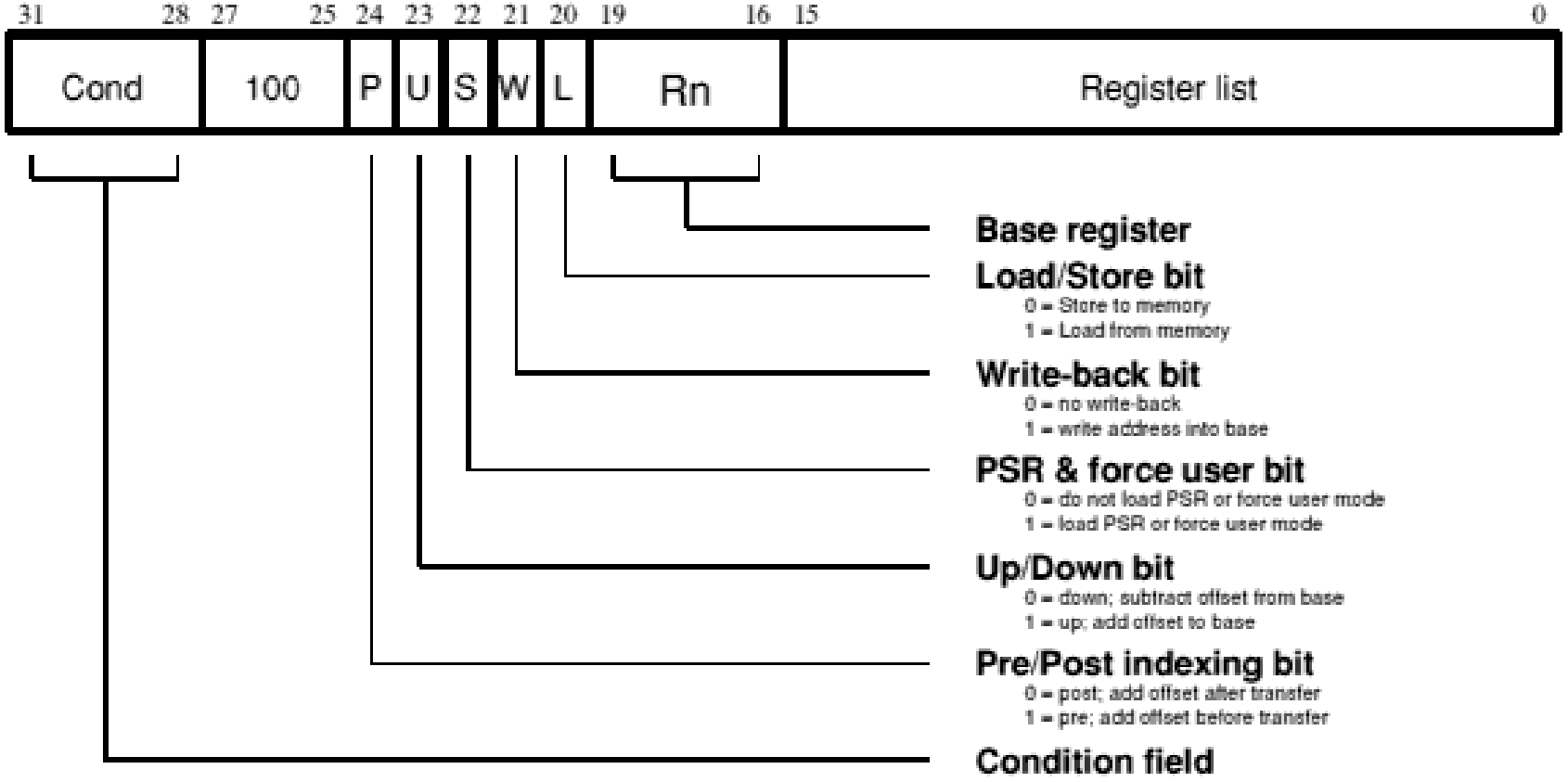
Modalità indirizzamento: sommario

Nome	Nome alternativo	Esempio
Da registro a registro	Registro diretto	MOV R0, R1
Assoluto	Diretto	LDR R0, MEM
Letterale	Immediato	MOV R0, #15 ADD R1, R2, #12
Indicizzato, base	Registro indiretto	LDR R0, [R1]
Pre-indicizzato, base con spostamento	Registro indiretto con offset	LDR R0, [R1, #4]
Pre-indicizzato, autoindicizzazione	Registro indiretto pre-incremento	LDR R0, [R1, #4]!
Post-indicizzato, autoindicizzazione	Registro indiretto post-incremento	LDR R0, [R1], #4
Doppio registro indiretto	Registro indiretto Registro indicizzato	LDR R0, [R1, R2]
Doppio registro indiretto con scalatura	Registro indiretto indicizzato con scalatura	LDR R0, [R1, R2, LSL#2]
Relativo al program counter		LDR R0, [PC, #offset]

ARM: trasferimento a blocchi

ARM

Trasferimento di blocchi



ARM

Trasferimento di blocchi

- Si possono trasferire da 1 a 16 registri da o verso la memoria con un'unica operazione.
- I registri trasferiti possono essere:
  - Un qualsiasi sottoinsieme del banco di registri corrente (default);
  - Un qualsiasi sottoinsieme del banco di registri della user mode (posporre '^' all'istruzione).

ARM

Trasferimento di blocchi

- Sintassi:
  - `op{addr_mode}{cond} Rn{!}, reglist{^}`
- op:
  - `LDM` (load multiple registers)
  - `STM` (store multiple registers)
- addr_mode:
  - `IA` (incrementa indirizzo dopo ogni trasferimento)(default)
  - `IB` (incrementa indirizzo prima di ogni trasferimento)
  - `DA` (decrementa indirizzo dopo ogni trasferimento)
  - `DB` (decrementa indirizzo prima di ogni trasferimento)
- cond: codice condizionale (opzionale)
- Rn: registro che contiene l'indirizzo base del trasferimento (no r15)
- «!»: se aggiunto il punto esclamativo, al termine del trasferimento viene scritto in Rn l'indirizzo finale
- Reglist: lista del registri da caricare/immagazzinare (va messa tra parentesi)
- «^»: suffisso opzionale, non usabile in user mode o system mode che fa utilizzare i registri della user mode per il trasferimento.

ARM

Trasferimento di blocchi

ISTRUZIONE	INDIRIZZO INIZIALE	INDIRIZZO FINALE	Valore finale R0
STMIA R0, {R1, R5-R9}	R0	R0 + numero byte scritti	R0
STMIB R0, {R1, R5-R9}	R0 + 4	R0 + 4 + numero byte scritti	R0
STMDA R0, {R1, R5-R9}	R0	R0 – numero byte scritti	R0
STMDB R0, {R1, R5-R9}	R0 – 4	R0 – 4 – numero byte scritti	R0
STMIA R0!, {R1, R5-R9}	R0	R0 + numero byte scritti	Indirizzo finale + 4
STMIB R0!, {R1, R5-R9}	R0 + 4	R0 + 4 + numero byte scritti	Indirizzo finale
STMDA R0!, {R1, R5-R9}	R0	R0 – numero byte scritti	Indirizzo finale – 4
STMDB R0!, {R1, R5-R9}	R0 – 4	R0 – 4 – numero byte scritti	Indirizzo finale

ARM

Trasferimento di blocchi

- Esempi (tra registri):
  - `LDMIA  r8,  {r0,r2,r9}  ;`
  - `STMDB  r1!, {r3-r6,r11,r12}  ;`
  - `LDMIA  r2,  {r5-r9}  ;`
  - `STMIB  r4!, {r7-r11}  ;`
- Esempio di spostamento tra aree di memoria:

```
LDR    r0,     =src     ;
LDR    r1,     =dst     ;
LDMIA  r0!,    {r4-r11}  ;
STMIA  r1!,    {r4-r11}  ;
```

ARM

Trasferimento di blocchi

- Il base register determina da dove deve partire per l'accesso alla memoria;
- Il base register si può aggiornare al termine del trasferimento posponendo '!'
- Istruzioni utili per:
  - Salvare e ripristinare il contesto
  - Spostare larghi blocchi di dato da/verso la memoria

ARM

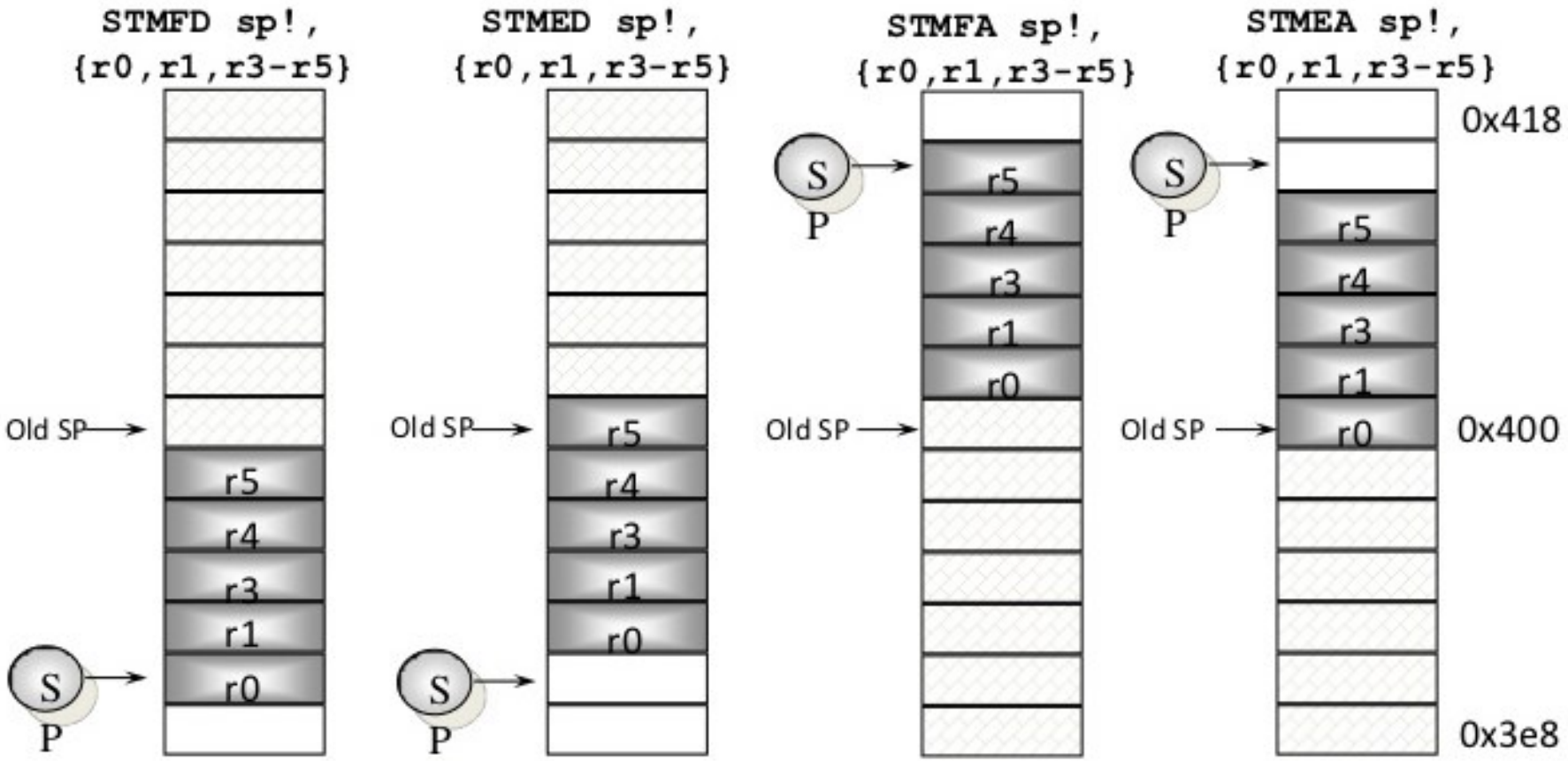
Trasferimento di blocchi

- L'address mode può essere scritta anche con suffissi stack-oriented:

Full descending	STMFD (STMDB)	LDMFD (LDMIA)
Full ascending	STMFA (STMIB)	LDMFA (LDMDA)
Empty descending	STMED (STMDA)	LDMED (LDMIB)
Empty ascending	STMEA (STMIA)	LDMEA (LDMDB)

ARM

Trasferimento di blocchi



ARM

Trasferimento di blocchi

- Esempio per creare temporaneamente spazio nei registri:

```
STMFD sp!, {r0-r12, lr}
……
……
LDMFD sp!, {r0-r12, pc}
```

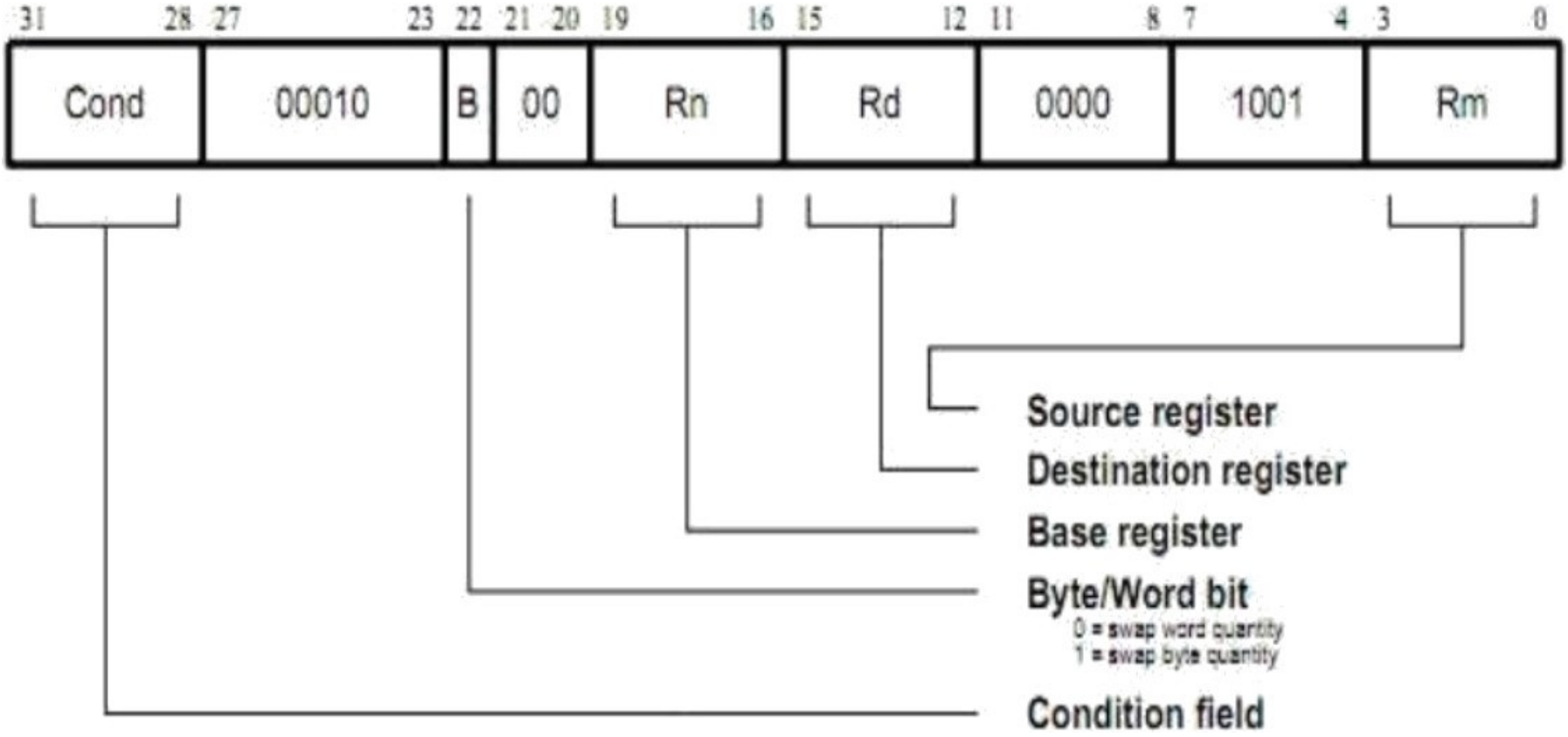
ARM

Trasferimento di blocchi

- Il base register determina da dove deve partire l'accesso alla memoria;
- Il base register si può aggiornare al termine del trasferimento posponendo '!'
- Istruzioni utili per:
  - Salvare e ripristinare il contesto
  - Spostare larghi blocchi di dato da/verso la memoria

ARM

Swap


31 28 27 23 22 21 20 19 16 15 12 11 8 7 4 3 0
Cond
00010
B
00
Rn
Rd
0000
1001
Rm
Source register
Destination register
Base register
Byte/Word bit
0 = swap word quantity
1 = swap byte quantity
Condition field

ARM

Swap

- Usata per scambiare dati tra un registro e la memoria.
- Istruzione atomica (non può essere interrotta)
- L'indirizzo per lo swap è determinato dal contenuto del base register (Rn)
- Il processore prima di legge il contenuto dell'indirizzo di swap. Poi scrive il contenuto del registro sorgente nello swap address. Infine scrive il vecchio contenuto della memoria nel registro di destinazione (Rd)
- Lo stesso registro può essere specificato sia come sorgente che come destinazione

ARM

Swap

- Sintassi:
  - `SWP{B}{cond} Rt, Rt2, [Rn]`
- Cond: codice condizione;
- B: suffisso opzionale, se presente si swappa un byte e non una word
- Rt: registro destinazione (no PC)
- Rt2: registro sorgente (no PC)
- Rn: registro che contiene l'indirizzo di memoria, deve essere diverso sia da Rt che da Rt2.
- Esempi:
  - `SWP r1, r2, [r0] ;` (Swap r2 con locazione [r0], valore in [r0] messo in R1)

ARM: instruction set

ARM

ARM instruction set

- Tutte le istruzioni sono a 32 bit;
- La maggior parte viene eseguita in un ciclo singolo;
- Ogni istruzione può essere eseguita condizionalmente;
- Architettura load/store:
  - Le istruzioni di processamento dati possono agire solo sui registri;
  - Istruzioni specifiche per l'accesso alla memoria con modalità di indirizzamento ad auto-indicizzazione.

ARM

Esecuzione condizionale

- La maggior parte degli istruction set permette l'esecuzione condizionale delle branch solo posponendo l'appropriato campo di condizione;
- In ARM ogni istruzione contiene un campo di condizione che determina se eseguire o meno l'istruzione.
- Incrementa il numero di istruzioni;
- Le istruzioni non eseguite consumano un ciclo.
- Rimuove il bisogno di molte branch (3 cicli per riempire la pipeline):
  - Codice denso, senza branch;
  - La perdita di tempo di non eseguire alcune istruzioni condizionali è minore di quella per gestire una chiamata a branch o sottoroutine.

ARM

Instruction set

- Le istruzioni si dividono nelle seguenti categorie:
  - Data processing
  - Branch
  - Load-store
  - Software interrupt
  - Program status register

ARM

Condition field

0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

ARM

Istruzioni condizionali

- Per l'esecuzione condizionale si pospone la condizione appropriata:
  - `ADD r0,r1,r2;` equivale a `ADDAL r0,r1,r2;`
  - `ADDEQ r0,r1,r2;` esegue l'addizione solo se è settato il flag di zero nel CPSR.
- Di default le operazioni di data processing non influiscono sui flag di condizione. Se si desidera che lo facciano va aggiunta una «S»:
  - `ADDS r0,r1,r2;`

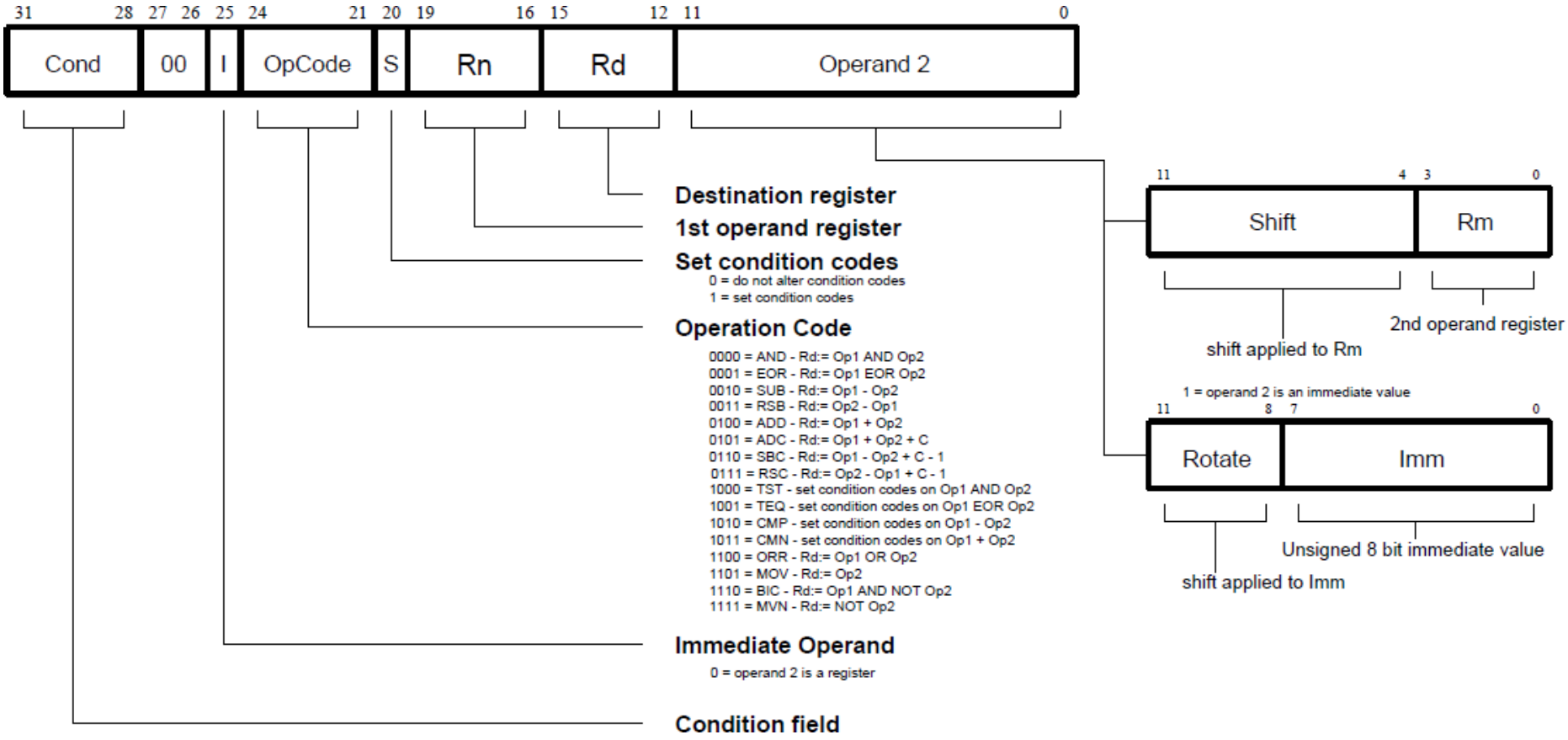
ARM

Istruzioni processamento dati

- La famiglia più popolosa di istruzioni
- Contiene:
  - Operazioni aritmetiche;
  - Comparazioni (nessun risultato, set dei flag);
  - Operazioni logiche;
  - Spostamento dati tra registri;
- **ARCHITETTURA LOAD/STORE: QUESTE ISTRUZIONI FUNZIONANO SOLO SUI REGISTRI, NON SULLA MEMORIA.**
- Ogni istruzione agisce su uno o due operandi:
  - Il primo è sempre un registro;
  - Il secondo è inviato alla ALU via barrel shifter;

ARM

Istruzioni processamento dati



ARM

Istruzioni processamento dati

MOV	Move a 32-bit value	MOV Rd,n	Rd = n
MVN	Move negated (logical NOT) 32-bit value	MVN Rd,n	Rd = ~n
ADD	Add two 32-bit values	ADD Rd,Rn,n	Rd = Rn+n
ADC	Add two 32-bit values and carry	ADC Rd,Rn,n	Rd = Rn+n+C
SUB	Subtract two 32-bit values	SUB Rd,Rn,n	Rd = Rn-n
SBC	Subtract with carry of two 32-bit values	SBC Rd,Rn,n	Rd = Rn-n+C-1
RSB	Reverse subtract of two 32-bit values	RSB Rd,Rn,n	Rd = n-Rn
RSC	Reverse subtract with carry of two 32-bit values	RSC Rd,Rn,n	Rd = n-Rn+C-1
AND	Bitwise AND of two 32-bit values	AND Rd,Rn,n	Rd = Rn AND n
ORR	Bitwise OR of two 32-bit values	ORR Rd,Rn,n	Rd = Rn OR n
EOR	Exclusive OR of two 32-bit values	EOR Rd,Rn,n	Rd = Rn XOR n
BIC	Bit clear. Every '1' in second operand clears corresponding bit of first operand	BIC Rd,Rn,n	Rd = Rn AND (NOT n)
CMP	Compare	CMP Rd,n	Rd-n & change flags only
CMN	Compare Negative	CMN Rd,n	Rd+n & change flags only
TST	Test for a bit in a 32-bit value	TST Rd,n	Rd AND n, change flags
TEQ	Test for equality	TEQ Rd,n	Rd XOR n, change flags

MUL	Multiply two 32-bit values	MUL Rd,Rm,Rs	Rd = Rm*Rs
MLA	Multiple and accumulate	MLA Rd,Rm,Rs,Rn	Rd = (Rm*Rs)+Rn

ARM

Istruzioni spostamento

- Le operazioni di spostamento sono:
  - `MOV operand1, operand2;`
  - `MVN operand1, operand2;`
- Da notare che non si fa uso di operand 1. La sintassi è:
  - `<Operation>(<cond>){S}Rd,Operand2;`
- Esempi:
  - `MOV r0,r1;`
  - `MOVS r2,#10;`
  - `MVNEQ r1,#0;`

ARM

Istruzioni aritmetiche

- Le operazioni aritmetiche sono:
  - `ADD operand1+operand2;`
  - `ADC operand1+operand2+carry;`
  - `SUB operand1-operand2;`
  - `SBC operand1-operand2+carry-1;`
  - `RSB operand2-operand1;`
  - `RSC operand2-operand1+carry-1;`
- La sintassi è:
  - `<Operation>(<cond>){S}Rd,Rn,Operand2;`
- Esempi:
  - `ADD r0,r1,r2;`
  - `SUBGT r3,r3,#1;` (eseguita solo se i flag dicono GT = greater then)
  - `RSBLES r4,r5,#5;` (eseguita solo se i flag dicono LE = less or equal e Setta i flag)
  - `SUB r4,r5,r7,LSR r2` ;(shift logico a destra di r7 del numero di bit indicati in r2, sottrazione del risultato da r5, risultato messo in r4)

ARM

Istruzioni logiche

- Le operazioni logiche sono:
  - `AND operand1 AND operand2;`
  - `EOR operand1 EOR operand2;`
  - `ORR operand1 OR operand2;`
  - `BIC operand1 AND NOT operand2;`
- La sintassi è:
  - `<Operation>(<cond>){S}Rd,Rn,Operand2;`
- Esempi:
  - `AND    r0,r1,r2;`
  - `BICEQ  r2,r3,#7;`
  - `EORS   r1,r3,r0;`

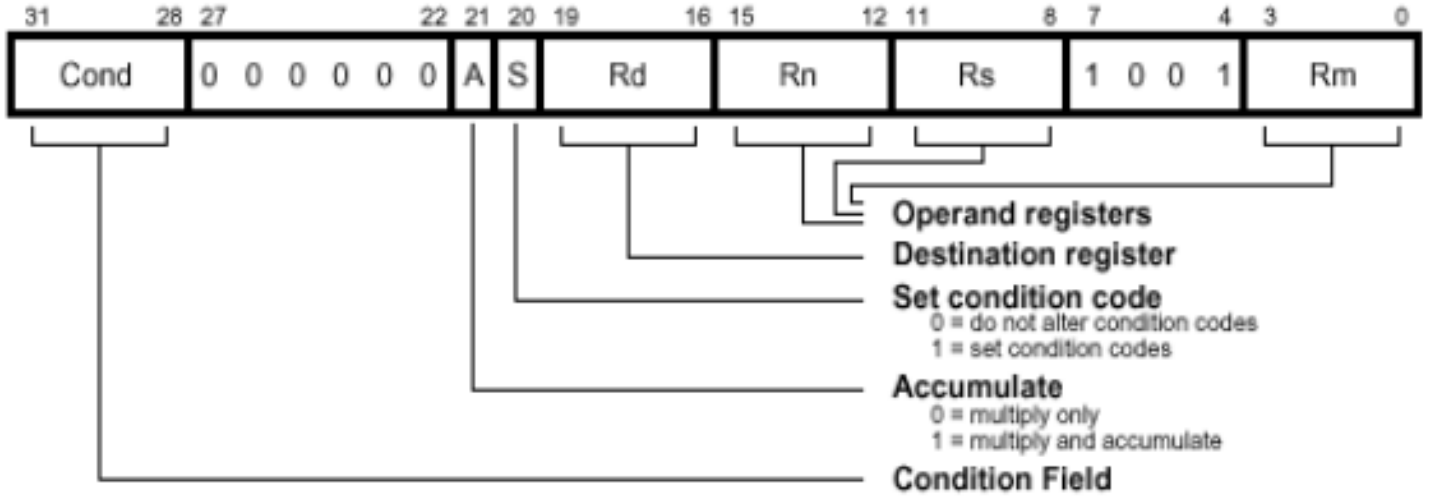
ARM

Istruzioni di moltiplicazione

- Le operazioni di moltiplicazione sono due:
  - Moltiplica:
    - `MUL(<cond>){S}Rd,Rm,Rs;` (Rd=Rm*Rs)
  - Moltiplica e accumula:
    - `MLA(<cond>){S}Rd,Rm,Rs,Rn;` (Rd=(Rm*Rs)+Rn)
- Restrizioni:
  - Rd e Rm non possono essere lo stesso registro (ma Rd e Rs possono, quindi basta girare gli operandi);
  - Non si può usare il PC.
- Gli operandi possono essere considerati con o senza segno, sta all'utente interpretarli correttamente.

ARM

Istruzioni di moltiplicazione

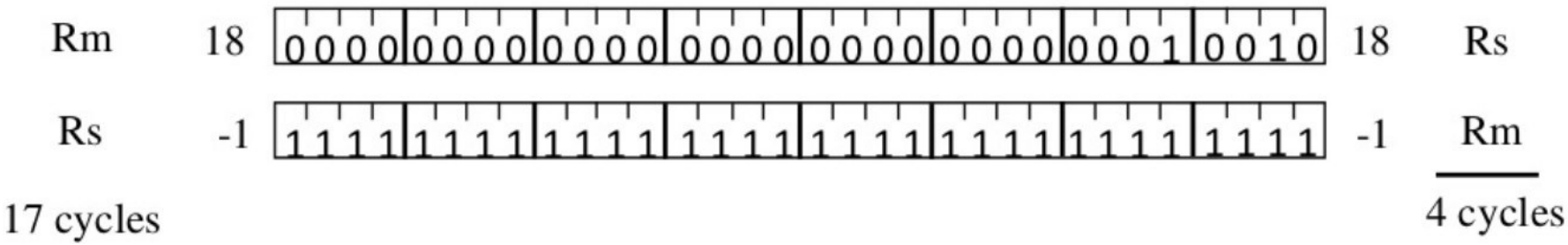


- L'istruzione di moltiplicazione è Rd:=Rm*Rs.
- Rn è ignorato e va settato a 0 per compatibilità con futuri aggiornamenti dell'instruction set.

ARM

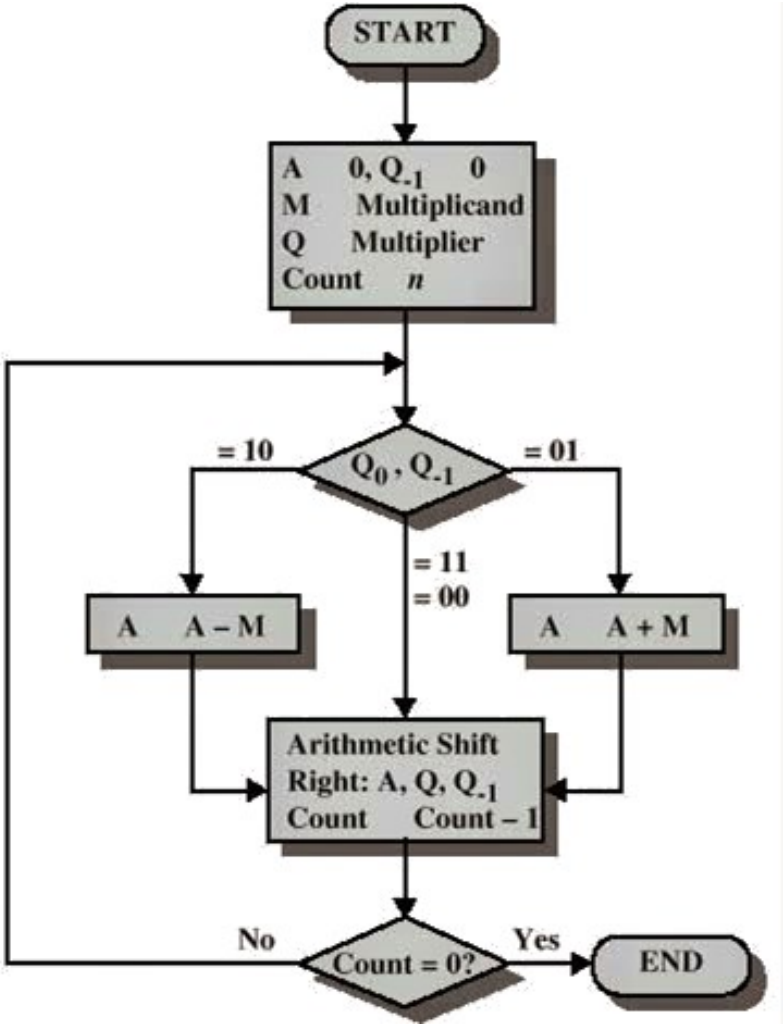
Implementazione della moltiplicazione

- L'ARM usa l'algoritmo di Booth per fare moltiplicazione tra interi.
- Sui micro non della serie M opera su 2 bit di Rs alla volta:
  - 1 ciclo per ogni coppia di bits;
  - Quando non ci sono più 1 rimasti in Rs la moltiplicazione termina anticipatamente.
  - Il compilatore non usa il principio della terminazione anticipata per ordinare gli operandi.
- Esempio: Moltiplicazione di 18 * (-1)



ARM

Implementazione della moltiplicazione (algoritmo di Booth)



ARM

Moltiplicazione estesa (MULL, MLAL)

- La serie M degli ARM ha un hardware per la moltiplicazione estesa che fornisce tre miglioramenti:
  - Si usa un algoritmo di Booth ad 8 bit (moltiplicazione più veloce, massimo 5 cicli);
  - Migliorato metodo terminazione veloce (completa la moltiplicazione anche quando i bit rimanenti sono tutti a 1);
  - Si può ottenere un risultato a 64 bit da due operandi a 32.

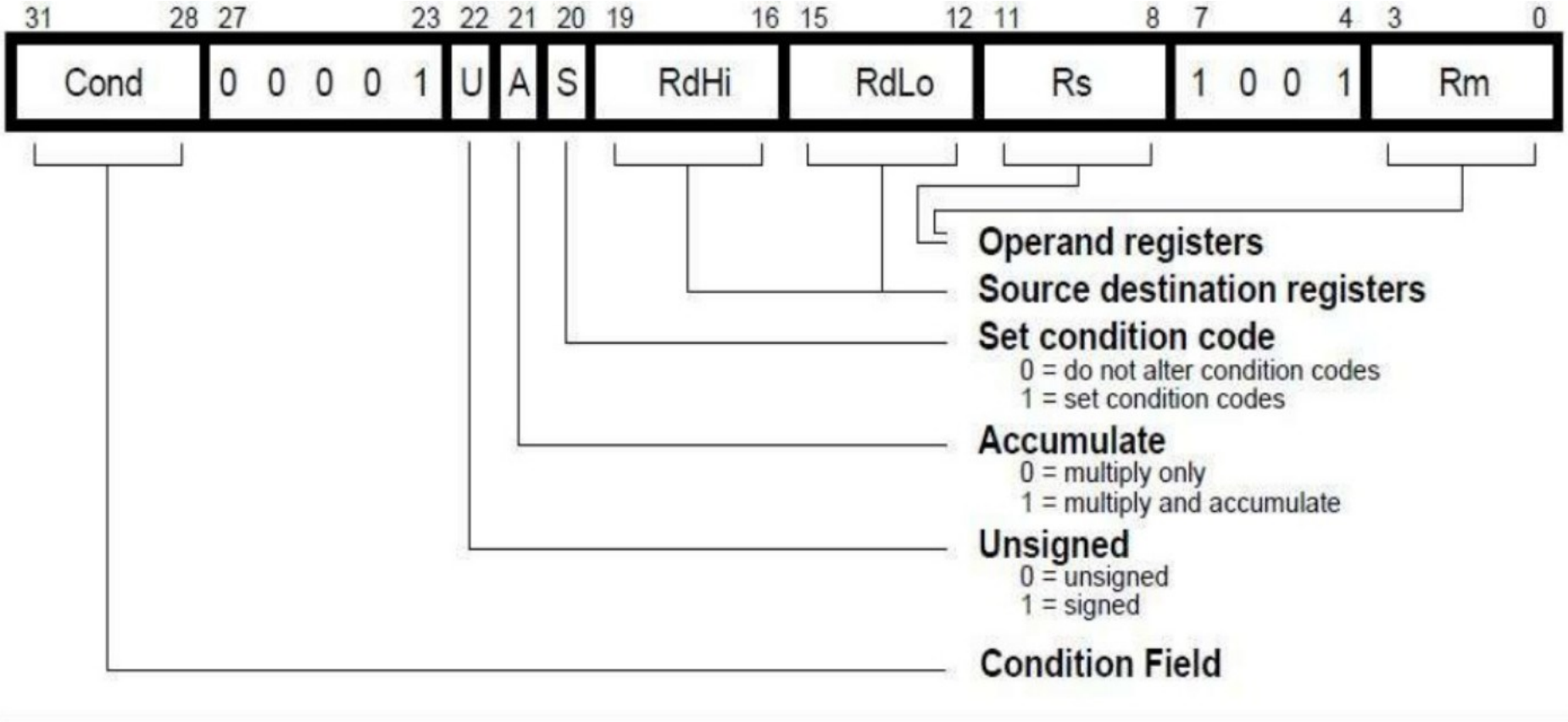
ARM

Moltiplicazione estesa

- Le istruzioni sono:
  - `MULL` che restituisce RdHi,RdLo:=Rm*Rs;
  - `MLAL` che restituisce RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo;
- Contano tutti i 64 bit del risultato, perciò occorre specificare se gli operandi sono con o senza segno;
- La sintassi è:
  - `UMULL(<cond>){S}RdLo,RdHi,Rm,Rs;`
  - `UMLAL(<cond>){S}RdLo,RdHi,Rm,Rs;`
  - `SMULL(<cond>){S}RdLo,RdHi,Rm,Rs;`
  - `SMLAL(<cond>){S}RdLo,RdHi,Rm,Rs;`

ARM

Moltiplicazione estesa



ARM

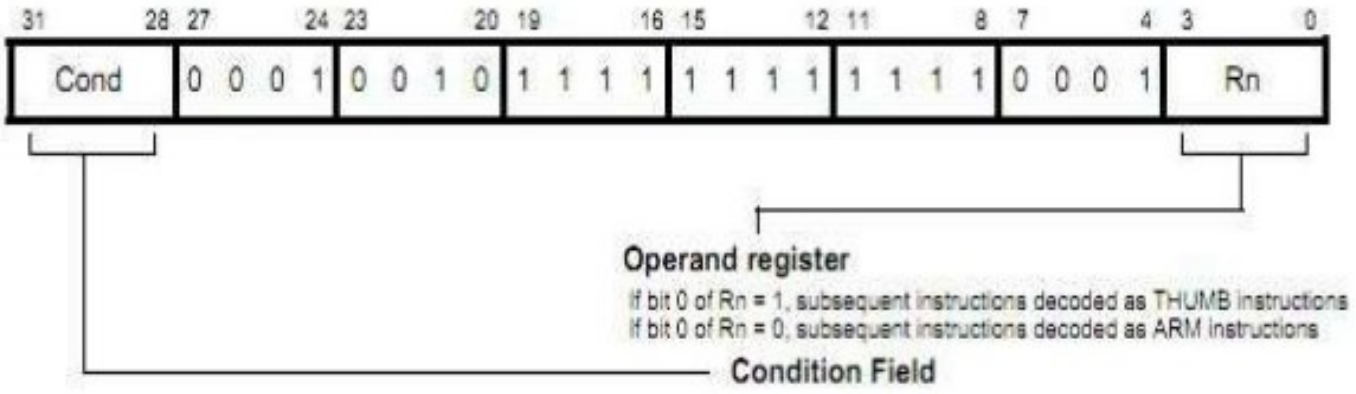
Trasferimento dati

- ARM è un'architettura load/store, perciò:
  - Carica dati dalla memoria verso i registri;
  - Immagazzina dati dai registri verso la memoria;

- Ha tre tipi di istruzioni load/store:
  - LDR/STR;
  - LDM/STM;
  - SWP;
- Già ampiamente trattate nelle lezioni precedenti

ARM

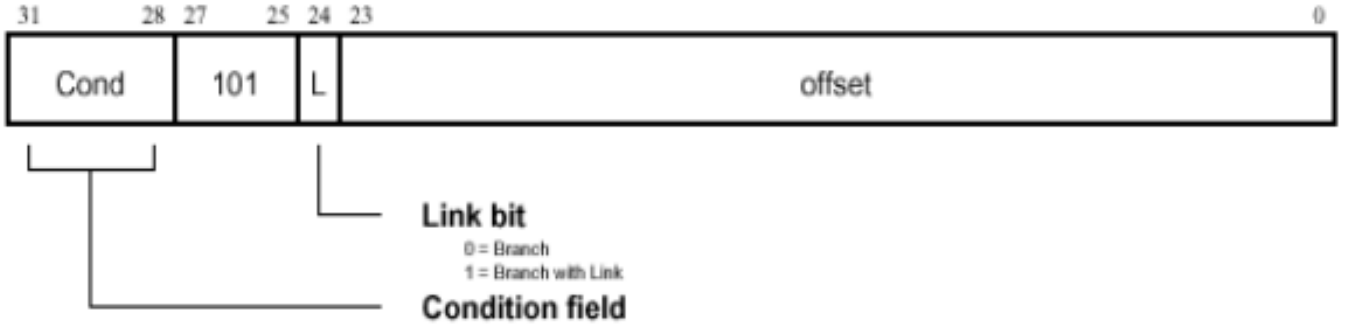
Branch (B), Branch and Exchange (BX), Branch with Link (BL) o il mix delle tre.



- Branch and Exhange: usato per passare da Thumb state ad ARM state e viceversa
- Sintassi:
  - `BX{cond} Rn`

ARM

Branch, Branch and Exchange, Branch with Link



- Branch: usato per spostarsi su un altro pezzo di codice
- Branch and Link: stessa cosa di Branch ma viene salvato l'indirizzo dell'istruzione successiva in r14 per poter tornare indietro
- Sintassi:
  - `B{L}{cond} label`

ARM

Branch, Branch and Exchange, Branch with Link

- Vista la lunghezza dell'offset (24 bit) , considerando che viene shiftato a sinistra di due bits, e preso come intero con segno, ci si può spostare di +/- 32 Mbytes
- La branch con Link:
  - Scrive il vecchio PC in r14;
  - Il CPSR non viene salvato;

ARM

Comparazioni

- Sintassi:
  - `<Operation>{<cond>}Rn,Operand2`
- Operation:
  - `CMP` operand1 – operand2
  - `CMN` operand1 + operand2
  - `TST` operand1 AND operand2
  - `TEQ` operand1 EOR operand2
- Il risultato non viene mai scritto, va solo a settare i condition flags del CPSR.
- Esempi:
  - `CMP  r0, r1`
  - `TSTEQ  r2, #5`

Divisione

- Non è presente l'operazione di divisione,
- Viene eseguita mediante un ciclo di sottrazioni ripetute, tenendo conto del numero di sottrazioni effettuate nel ciclo

```
subtract: ; label
 SUBS    r1,r1,r2 ; r1 dividendo, r2 divisore
 ADD     r0,r0,#1; quoziente
BHI     subtract ; cicla fino a quando il
risultato è diverso da zero
Provare a fare esempi con resto
Provare a fare esempi con numeri con segno
```