

ARCHITETTURE dei SISTEMI di ELABORAZIONE

0) INTRODUZIONE al COMPUTER DESIGN

Il computer market si divide in 5 aree:

- Personal Mobile → efficienza energetica e applicazioni real-time
- Desktop → prezzo-performance
- Server → scalabilità e throughput
- Clusters/Warehouse-Scale → floating-point performance e reti interne veloci
- Embedded → special-purpose (spesso FPGA)

Per aumentare le prestazioni, si parla di **PARALLELISMO** su diversi livelli (dati, task, istruzioni, vettori/matrici [GPU], thread, richieste...). Una **COMPUTER ARCHITECTURE** comprende **INSTRUCTION SET ARCHITECTURE**, **ORGANIZATION** e **HARDWARE** (richiamo alla LEGGE DI MOORE = numero di transistor in 1 chip raddoppia ogni 18/24 mesi; il maggior costo nella produzione IC è dato da validazione e testing).

Importante è il **CONSUMO ENERGETICO**: la tensione viene mantenuta costante (altrimenti rottura dispositivo), è il dispositivo che assorbe più o meno energia a seconda del carico richiesto (si differenzia tra $P_{statica} = V I$ e $P_{dinamica} = \frac{1}{2} C V^2 f$; $E_{dinamica} = C V^2$). Importante anche l'**AFFIDABILITÀ**, specialmente in alcuni settori (es. medico, trasporti... [es. usare 2 CPU al posto di 1, per controllare che siano ancora affidabili nei calcoli]). Per misurare le prestazioni si fanno **BENCHMARK**: da qui si vede anche il **TEMPO DI ESECUZIONE** (sul singolo pezzo si fa con media pesata; se non si ha il peso, media aritmetica).

Nel computer design, importante è la **LEGGE DI AMDAHL**

$$speedup_{overall} = \frac{execution\ time_{old}}{execution\ time_{new}} = \frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{speedup_{enhanced}}}$$

Esempio:

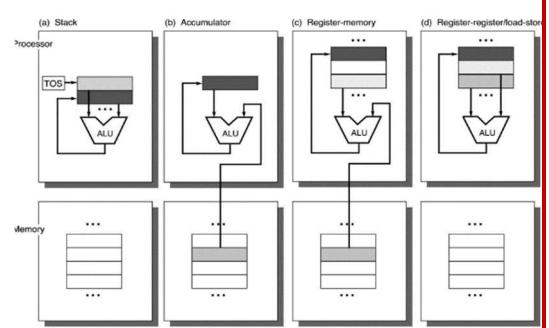
An enhancement makes one machine 10 times faster for 40% of the programs the machine runs. Which is the overall speedup?

$$\begin{aligned} fraction_{enhanced} &= 0.4 \\ speedup_{enhanced} &= 10 \end{aligned}$$

$$speedup_{overall} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56$$

1) INSTRUCTION SET ARCHITECTURE (ISA)

L'**ISA** è il livello di astrazione con cui il programmatore o il compilatore interagiscono con la CPU. Il progetto dell'ISA influenza vari aspetti del processore, come **PRESTAZIONI**, **COMPLESSITÀ** (di processore e compilatore), **DIMENSIONE DEL CODICE** e **CONSUMO ENERGETICO**. Riguardo alla **TASSONOMIA** (classificazione della CPU in base al tipo di memoria interna), si hanno basati su **stack**, **accumulator** e **registri** (es. **load-store** [detti register-register] oppure **register-memory**).



⚠ La maggior parte dei processori è **GPR** (General-Purpose Register) perché i **registri** sono più veloci/gestibili per un compilatore, rispetto alla memoria; i registri ospitano operandi.

Le caratteristiche di un set di istruzioni sono:

- **INDIRIZZAMENTO DELLA MEMORIA**
 - Posizione dell'LSB/MSB:
 - **Little Endian** → MSB ... LSB (indirizzo è quello del LSB)
 - **Big Endian** → LSB ... MSB (indirizzo è quello del MSB)
 - Allineamento degli accessi:
 - **Allineato** (permette solo accessi allineati alla memoria)
 - **Disallineato**
 - Tipi di indirizzamento:
 - **Register** (se il dato è nel registro → Add R4, R3 → $\text{Regs}[R4] = \text{Regs}[R4] + \text{Regs}[R3]$)
 - **Immediate** (per costanti numeriche → Add R4, #8 → $\text{Regs}[R4] = \text{Regs}[R4] + 8$)
 - **Displacement** (puntatori → Add R4, 100(R1) → $\text{Regs}[R4] = \text{Regs}[R4] + \text{Mem}[100+\text{Regs}[R1]]$)
 - **Deferred/Indirect** (Add R4, (R1) → $\text{Regs}[R4] = \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$)
 - **Indexed** (Add R4, (R1+R2) → $\text{Regs}[R4] = \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]+\text{Regs}[R2]]$)
 - **Direct/Absolute** (Add R4, (1001) → $\text{Regs}[R4] = \text{Regs}[R4] + \text{Mem}[1001]$)
 - **Memory Indirect** (valore puntato dal puntatore *p → Add R4, @(R3) → $\text{Regs}[R4] = \text{Regs}[R4] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$)
 - **Post Autoincrement** (Add R4, (R3) + → $\text{Regs}[R4] = \text{Regs}[R4]+\text{Mem}[\text{Regs}[R3]]$; $\text{Regs}[R3] = \text{Regs}[R3] + \text{dimensione del singolo elemento}$)
 - **Pre Autodecrement** (Add R4, - (R3) → $\text{Regs}[R3] = \text{Regs}[R3] - \text{dim}$; $\text{Regs}[R4] = \text{Regs}[R4]+\text{Mem}[\text{Regs}[R3]]$)
 - **Scaled** (Add R4, 100(R3)[R2] → $\text{Regs}[R4] = \text{Regs}[R4] + \text{Mem}[100+\text{Regs}[R3]+\text{Regs}[R2]*\text{dim}]$)

⚠ I più usati sono Register, Immediate e Displacement.

- **OPERAZIONI**
 - **Aritmetiche-Logiche** (add, and ...)
 - **Data transfer** (load, store...)
 - **Control Flow** (conditional branch, jump, procedure calls, procedure returns, trap...) [per i salti, vanno salvate anche le informazioni di ritorno e altre informazioni sul salto; l'**8086 (CISC)** salva le **informazioni da solo, se usassi RISC dovrebbe gestire il programmatore**]
 - **System** (syscall, virtual memory...)
 - **Floating point** (add, multiply...)
 - **Decimal** (add, multiply, conversions...)
 - **String** (operazioni su stringhe)
 - **Graphics** (operazioni su pixel, compressione/decompressione...)

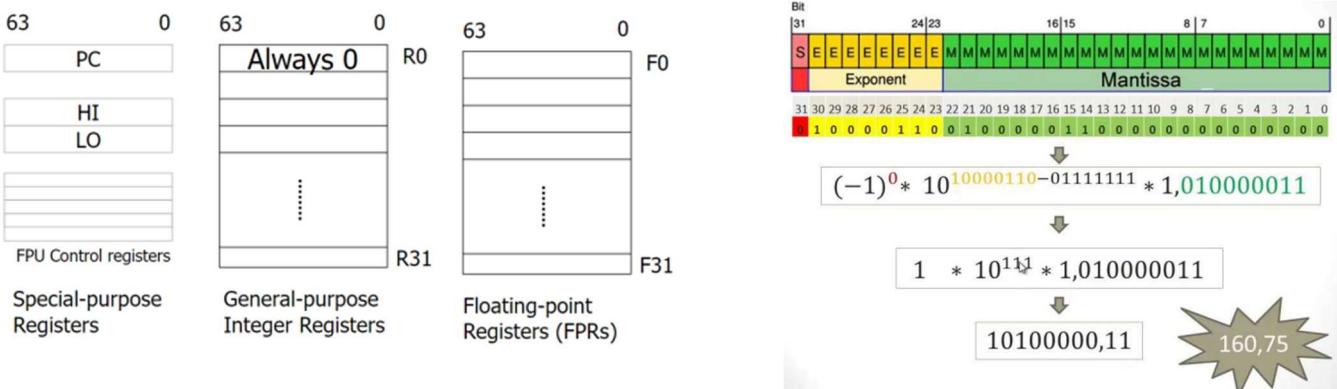
- **TIPI e DIMENSIONI DEGLI OPERANDI**
 - **char** = 1 Byte
 - **half word** = 0.5 word
 - **word** = 4 Byte (ovviamente dipende dal parallelismo, es. se a 64 bit ho word da 8 Byte)
 - **double word** = 2 word
 - **single-precision** floating-point (float) = 4 Byte
 - **double-precision** floating-point (double) = 8 Byte

- **CODIFICA DELLE ISTRUZIONI**
 - **Variabile** = numero variabile di operandi, istruzioni lunghezza variabile, meno performance, dimensione codice minima
 - **Fissa** = solo numero fisso di operandi, address specifier (per l'address mode), istruzioni lunghezza fissa, più performance, dimensione codice grande
 - **Ibrida**

⚠ Programmi assembly sono ora prodotti solo dai compilatori (quindi i designer di CPU e compilatore devono cooperare); non possiamo usare tutti i registri (alcuni sono riservati).

2) MIPS64

MIPS (Microprocessor without Interclocked Pipeline Stages) è una famiglia di RISC (utili per applicazioni embedded). Hanno un **semplice load-store instruction set** e sono progettati per la “**pipeline efficiency**” (lunghezza di istruzioni prefissata e pensati per applicazioni a basso consumo). I registri sono a **64 bit** e il registro **R0** vale **sempre 0**; ciò consente metodi di indirizzamento alternativi [sx]. Lo standard è **IEEE 754 32 bit** [dx]:

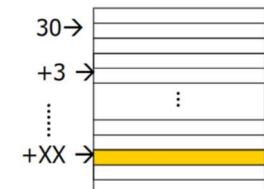


I **TIPI di dati** sono:

- **Byte** (8 bit)
- **Half Word** (16 bit)
- **Word** (32 bit)
- **Double Word** (64 bit)
- **32-bit float** (single precision)
- **64-bit double** (double precision)

I **METODI DI INDIRIZZAMENTO** sono:

- **Immediato** → viene usato un immediate field (16 bit) [es. DADDUI R1,R2,#32 = R1 ← R2 +32; DADDUI R1,R0,#32 = R1 ← 32]
- **Con Displacement** → esempio:
LD R1, 30(R2) # R1 ← R2 + 30
R2 = XX
R1 ← Mem[30+R2]
⚠ Se usassi LD R1, 0(R2) avrei “**registro indiretto**”; se usassi LD R1,64(R0) avrei “**indirizzamento assoluto**”.



Riguardo le **ISTRUZIONI**, queste sono delle single **32 bit aligned word**; includono un **opcode di 6 bit** iniziali; hanno 3 formati:

1. **Immediato:**
 - opcode (6 bit)
 - Rs (5 bit) → indirizzo sorgente
 - Rt (5 bit) → indirizzo target
 - Immediate (16 bit signed)
2. **Registro:**
 - opcode (6 bit)
 - Rs (5 bit) → indirizzo sorgente
 - Rt (5 bit) → indirizzo target
 - Rd (5 bit) → indirizzo destinazione
 - Sa (5 bit) → shift amount (per shift a sx e dx)
 - Function (6 bit) → funzione
3. **Salto:**
 - opcode (6 bit)
 - offset (26 bit) → indice spostato a sinistra di 2 bit per fornire i 28 bit di ordine inferiore dell’indirizzo destinazione del salto

Nell'**INSTRUCTION SET** le istruzioni sono raggruppate per funzionamento:

- **Load & Store** = MIPS ha un'architettura sua di load & store con cui si accede alla memoria principale:
 - **LB R1, 28(R8)** → carica 1 byte da memoria in un registro
 - **LD R1, 28(R8)** → carica 1 double word da memoria in un registro
 - **LBU R1, 28(R8)** → carica 1 byte unsigned da memoria in un registro
 - **L.S F4, 46(R5)** → carica 1 float single-precision in un registro
 - **L.D F4, 46(R5)** → carica 1 double in un registro
 - **SD R1, 28(R8)** → memorizza 1 double
 - **SW R1, 28(R8)** → memorizza 1 word
 - **SH R1, 28(R8)** → memorizza l'half word più significativa
 - **SB R1, 28(R8)** → memorizza gli 8 bit LSB

⚠ Analogi con i numeri reali; l'**estensione** dei valori avviene ripetendo il bit MSB (infatti nel floating point il MSB è il segno).
- **Operazioni ALU** = logico-aritmetiche, shift etc...; tutte le operazioni vengono eseguite con operandi salvati nei registri; vediamo alcuni esempi:
 - **DADDU R1, R2, R3** → double add unsigned
 - **DADDUI R1, R2, 74** → double add unsigned immediate
 - **LUI R1, 0X47** → load upper immediate (usata con **ORI** per numeroni)
 - **DSLL R1, R2, #3** → double shift logical [R1 ← R2 << 3]
 - **SLT R1, R2, R3** → set less than [if (R2 < R3) R1 ← 1; else R1 ← 0]
- **Branch & Jump** = salti condizionati e non:
 - **J label** → unconditional jump
 - **JAL name** → jump and link
 - **JALR R4** → jump and link register
 - **JR R4** → jump register
 - **BEQZ R4, name** → branch equal zero [if (R4 = 0) then PC ← name]
 - **BNE R3, R4, name** → branch not equal [if (R3 != R4) then PC ← name]
 - **MOVZ R1, R2, R3** → conditional move if zero [if (R3 = 0) then R1 ← R2]
 - **NOP** → “no operation” [uguale a SLL R0, R0, 0]

⚠ Un programma assembler si divide in **DATA SECTION** (variabili e costanti) e **CODE SECTION** (programma, routines e subroutines); queste 2 parti sono precedute da commenti sul titolo del programma e dalle direttive all’assemblatore. I **COMMENTI** sono preceduti da “;” e sono inline.

.Code				■ Assembler Directives
.global main				
main:	addi Jal	r1,r0,Info Input	;	Labels
	movi2fp cvti2d	f10,r1 f0,f10	;	OPcode
	addi	c2,r0,1	;	Operators
	movi2fp cvti2d	f11,r2 f2,f11	;	Comments
	Movd	f4,f2	;	
Loop:	led bfpt	f0,f4 EndL	;	
	Multd subd	f2,f2,f0 f0,f0,f4	;	
	j	Loop	;	
EndL:	Print, f2	Print,f2 r14,r0,Print	;	
	addi	r14,r0,Print	;	
	trap	5	*** end	

⚠ Noi useremo WinMIPS64 (per il codice bozza userò VSCode). Esempio di **C** vs **MIPS64**:

```
/* Sum of 10 integer values */
#include <stdio.h>

const long int values[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int main() {
    long int result;

    result = 0;
    for (int i = 0; i < 10; i++) {
        result = result + values[i];
    }
}
```

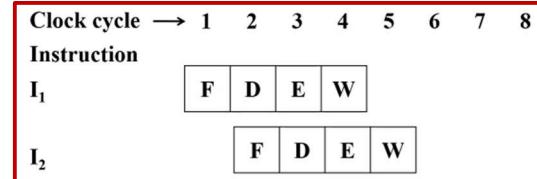
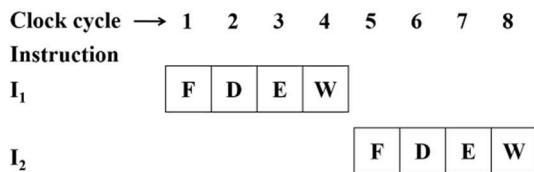
```
;-----+
; Program: 10V_sum.s
; Sum of 10 integer values
;-----+
.data
values: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ;64-bit integers
result: .space 8

.text
MAIN:  daddui R1,R0,10 ;R1 ← 10
       dadd R2,R0,R0 ;R2 ← 0   POINTER REG
       dadd R3,R0,R0 ;R3 ← 0   RESULT REG

LOOP:   ld    R4,values(R2)    ;GET A VALUE IN R4
       dadd R3,R3,R4 ;R3 ← R3 + R4
       daddi R2,R2,8 ;R2 ← R2 + 8  POINTER INCREMENT
       daddi R1,R1,-1 ;R1 ← R1 - 1 DECREMENT COUNTER
       bnez R1,LOOP
       sd    R3,result(R0) ; Result in R3
HALT          ;the end
```

3) PIPELINE

Il **PIPELINING** permette di **sovrapporre l'esecuzione di più istruzioni**, dove diverse unità (dette stadi o segmenti) eseguono parti diverse di varie istruzioni in parallelo durante gli stessi cicli di clock:



Il **THROUGHPUT** è il **numero di istruzioni che esce dalla pipeline** per unità di tempo (con la pipeline aumenta il throughput); il **CICLO MACCHINA** è la **singola iterazione del pipeline** (ovvero il tempo per eseguire 1 step, che solitamente corrisponde ad 1 ciclo di clock) ed è determinato dallo stadio più lento [**CPI** = cicli di clock per istruzione].

⚠ In una **pipeline ideale**, tutti gli stage sono perfettamente bilanciati, ovvero richiedono tutti lo stesso tempo (quindi, dato N = numero degli stage della pipeline, avremo $\text{throughput}_{\text{pipelined}} = \text{throughput}_{\text{unpipelined}} * N$).

Vediamo un ESEMPIO SENZA PIPELINE; l'esecuzione di **ogni istruzione è composta al max da 5 cicli di clock** [le **branch** ne richiedono solo 4] [ad ogni stadio associamo sotto le *PSEUDOISTRUZIONI*]:

1. (instruction) **fetch** (IF)
 - $IR \leftarrow \text{Mem}[PC]$ # IR = instruction register; PC = program counter
 - $NPC \leftarrow PC + 4$ # NPC = new program counter

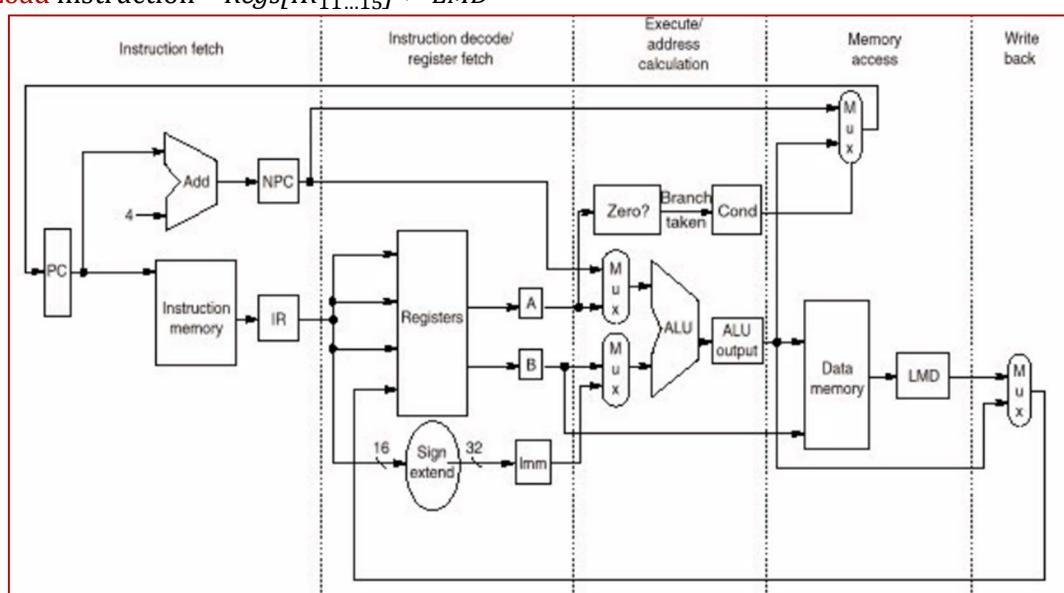
2. **decode/register fetch** (ID) [32 registri → dati e istruzioni entrambi su 32 bit, vedi sotto]
 - $A \leftarrow \text{Regs}[IR_{6..10}]$
 - $B \leftarrow \text{Regs}[IR_{11..15}]$
 - $Imm \leftarrow (([IR]_{16})^{16} \# \# IR_{16..31})$

0	5 6	10 11	15 16	31
OPcode	rs	rt	immediate	
0	5 6	10 11	15 16	20
OPcode	rs	rt	rd	sa function

3. **execution** (EX) = varia a seconda dell'istruzione:
 - ❖ Memory Reference = $ALUOutput \leftarrow A + Imm$
 - ❖ Register-Register ALU instruction = $ALUOutput \leftarrow A op B$
 - ❖ Register-Immediate ALU instruction = $ALUOutput \leftarrow A op Imm$
 - ❖ Branch = $ALUOutput \leftarrow NPC + Imm$; $Cond \leftarrow (A op 0)$

4. **memory access** (MEM) = varia a seconda dell'operazione:
 - ❖ Memory Reference = $LMD \leftarrow \text{Mem}[ALUOutput]$ oppure $\text{Mem}[ALUOutput] \leftarrow B$
 - ❖ Branch Completion Cycle = if ($Cond$) $PC \leftarrow ALUOutput$; else $PC \leftarrow NPC$

5. **write back** (WB) = varia a seconda dell'istruzione:
 - ❖ Register-Register ALU instruction = $Regs[IR_{16..20}] \leftarrow ALUOutput$
 - ❖ Register-Immediate ALU instruction = $Regs[IR_{11..15}] \leftarrow ALUOutput$
 - ❖ Load instruction = $Regs[IR_{11..15}] \leftarrow LMD$

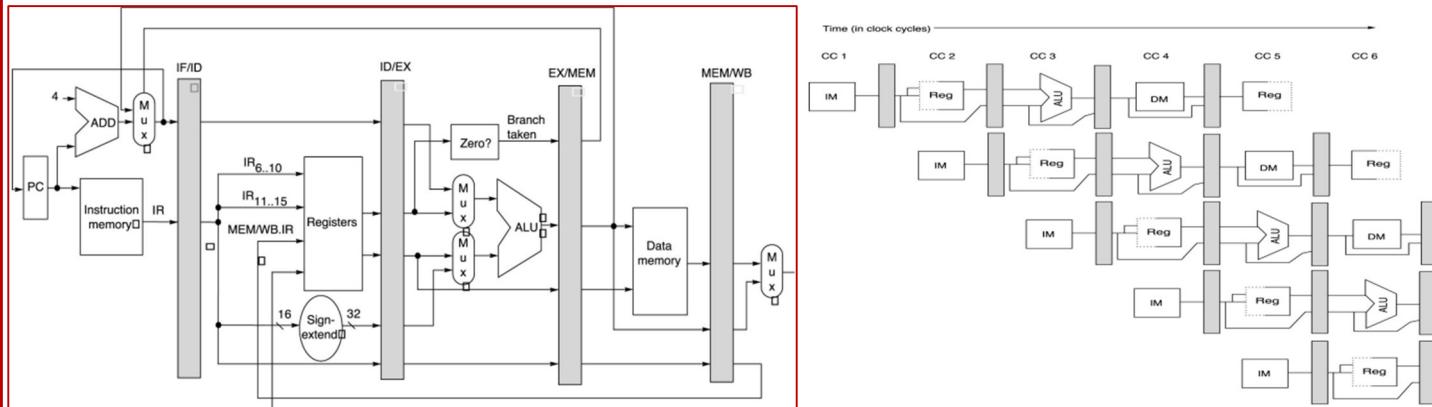


⚠ Necessario usare un **single control unit** per produrre i **segnali** necessari al **datapath**.

Potrebbero essere fatte alcune **OTTIMIZZAZIONI** per ridurre il CPI medio:

- Completare le istruzioni ALU durante il ciclo di MEM;
- Ottimizzare le risorse hardware per eliminare duplicazioni;
- Usare un'architettura single clock alternativa (1 istruzione ogni colpo di clock)

Vediamo ora un ESEMPIO PIPELINED, dove abbiamo l'**avvio di 1 nuova istruzione per ogni clock cycle**; per ciascun ciclo di clock, ogni risorsa può essere usata per 1 sola richiesta, necessitando quindi di separare le istruzioni e la memoria dati.



⚠ Usando la PIPELINE, **aumentiamo il throughput, accettando che i singoli stadi possano essere più lenti** (questo rallentamento è dovuto all'overhead per il controllo della pipeline); proprio per questo motivo, la lunghezza della pipeline è limitata dalla necessità di **bilanciare gli stadi e l'overhead per il controllo**.

⚠ Il tempo di esecuzione medio di un'istruzione è CK cycle * CPI.

I **Pipeline Hazards** ("azzardi") sono situazioni che possono far sì che un'istruzione non venga eseguita come dovrebbe; questi vengono **gestiti con** l'introduzione degli **STALLI** (operazioni in cui si richiede al processore di fermarsi per non causare problemi, in modo che le istruzioni seguenti vengano eseguite dopo che le istruzioni precedenti allo stall finiscono di essere elaborate). 3 tipi:

- **STRUCTURAL** hazards → quando un'unità della pipeline non è in grado di eseguire una certa operazione che era stata pianificata per quel ciclo; esempi:
 - unità non in grado di terminare il suo task in 1 ciclo di CK
 - la pipeline ha 1 solo register-file write port, ma non ci sono cicli in cui 2 register writers sono richiesti
 - la pipeline fa riferimento ad 1 single-port memory e ci sono cicli in cui differenti istruzioni vogliono accedere alla memoria insieme

L'unica soluzione è il **miglioramento HW** (acquistare nuove componenti); l'Average Instruction Time per pipeline con structural hazards è dato da $CPI_{ideal} + InstructionFreq * Stall_{penalty} \approx 1.33 * CK_{ideal}$

- **DATA** hazards → quando più istruzioni successive vogliono accedere/modificare dati (registri) prima che il valore nuovo (calcolato nella istruzione precedente) diventi effettivo (es. se ho ADD R1,R2,R3 [dove avrò che solo nello stadio WB viene scritto in R1 il risultato di R2+R3] seguito subito da AND R5,R1,R7, che quindi parte subito nello stadio successivo alla ADD, ma usa il registro R1 che avrà un valore errato, in quanto non è stato ancora aggiornato) [avviene sia nei registri sia in memoria il data hazard].

⚠ Se avvenisse un'**INTERRUPT** durante l'esecuzione di una porzione di codice critica, la correttezza potrebbe essere ripristinata, causando però un comportamento non deterministico.

La soluzione è usare uno **stallo** per i dati richiesti (in modo da usarli solo quando sono ultimati/disponibili) oppure usare un meccanismo di **FORWARDING** (un hardware dedicato all'interno del datapath si occupa di prendere il risultato come input per l'operazione seguente, invece di prendere il valore obsoleto nel registro; deve però anche essere in grado di non fare forwarding se l'istruzione che segue è in stallo oppure è stata eseguita un'interrupt). Il **controllo sull'attivazione del forwarding** avviene nella fase di decodifica, su un possibile data hazard relativo ad un'istruzione in fase di ID (se rilevata quindi avviene forwarding o stallo). Avviene legato a ciò "**load interlock detection**" (quando un'istruzione di load va in fase di esecuzione e un'altra

istruzione sta cercando di accedere al dato in fase di decode, si verifica se gli operandi fanno match e nel caso di data hazard si introduce uno stall per prevenire le istruzioni di fetch e decode di avanzare).

opcode di ID/EX	opcode di IF/ID	match?
Load	register-register alu	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	register-register alu	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	load, store, ALU immediate, branch	ID/EX.IR[rt] == IF/ID.IR[rs]

Data un'istruzione in stage di ID, come si introduce uno:

- **STALLO** (in fase di EX):
 - forzando tutti zeri nella pipeline ID/EX register (corrisponde ad una NOP)
 - forzando IF/ID register a contenere il valore corrente
 - fermare il PC

- **FORWARDING**:

- dal data memory output della ALU
- verso l'input della ALU, verso data memory inputs o verso zero detection unit

⚠ Per implementare **N cammini di forwarding**, vanno aggiungi **N** ingressi al **MUX** che prendono gli **N** risultati delle operazioni per metterli come input della ALU al posto dei registri con i dati ancora obsoleti.

- **CONTROL hazards** → **dovuti a salti (branch o jump) che possono cambiare il PC dopo che l'istruzione ha già eseguito il fetch** [nel MIPS, il PC è scritto con il target address (destinazione) alla fine della fase di decode]. Una soluzione è usare stalli appena l'istruzione di branch viene individuata (cioè in fase di decode) e **decidere prima se il salto avverrà**, calcolando in anticipo il NPC (questo però aumenta il costo dell'HW e prevede maggiore attenzione ai registri legati ai salti). Ci sono varie soluzioni per ridurre la degradazione delle performance dovute ai salti:

- **freezing the pipeline** = pipeline messa in stallo (o svuotata) appena il branch è rilevato e fino a che non si sa dove saltare (più semplice)
- **predict untaken** = assume che il branch non venga preso, evitando qualsiasi cambio di stato della pipeline fino a che il branch non ha "deciso". Annulla inoltre le operazioni eseguite se il branch viene preso
- **predict taken** = se si sa il target address prima del risultato del branch, il processore usa ottimizzazioni interne per fare la giusta previsione del branch
- **delayed branch** = riempire gli slot dopo l'istruzione di branch ("branch delay slots") con istruzioni che devono essere eseguite indipendentemente dall'esito del branch (più difficile, meno usata)

Parlando invece di **MULTICYCLE OPERATIONS**, ovvero come gestire le operazioni che richiedono più cicli di clock per essere eseguite. Le **FLOATING POINT OPERATIONS** sono più complicate rispetto a quelle sugli interi; per questo, per riuscire a svolgere tali operazioni in 1 solo ciclo di clock, dovremmo usare un **clock lento** e rendere **complesse le unità** (aggiungendo molta logica); entrambe queste opzioni sono infatti inaffidabili, per cui si cerca di scomporre l'operazione in più fasi, eseguite in una sorta di pipeline (tutte le attività convergono nella fase di MEM per poi concludersi nella fase di WB [write back]). **È solo la fase di execute che risulta parallelizzata.**

Definiamo:

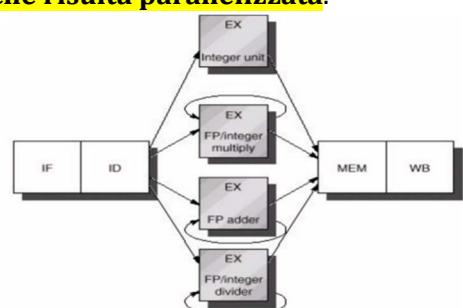
- **LATENZA** = n° cicli da eseguire per concludere l'istruzione e produrre un risultato riutilizzabile
- **INITIATION INTERVAL** ("intervallo di inizializzazione") = n° cicli che devono passare tra 2 operazioni dello stesso tipo sulla stessa unità

Proprio a causa delle multicycle operations, si ha come problema la **COMPARSA FREQUENTE DEGLI HAZARD**:

- **STRUCTURAL HAZARDS (MULTICYCLE)** [S] → a causa dell'impossibilità di usare una pipeline per l'unità di divisione, molte istruzioni potrebbero necessitarne allo stesso tempo; inoltre è possibile ottenere risultati dalle diverse unità operazionali nello stesso tempo, ma non è realmente possibile.

La soluzione è usare ulteriori "write ports" (però molto costosa) oppure **forzare uno structural hazard**:

- le istruzioni sono poste in stallo nella fase di decode
- le istruzioni sono messe in stallo prima della fase di MEM o WB (write back)



- **DATA HAZARDS (MULTICYCLE) [s]** → a causa di una più lunga latenza nelle operazioni, gli stalli per i data hazards possono fermare una pipeline per un quantitativo di tempo >; inoltre NUOVI tipi di data hazards sono possibili a causa dei tempi maggiori per raggiungere la WB (RAW hazard = Read After Write; WAW hazard = Write After Write)

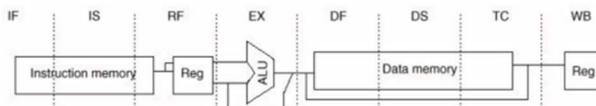
La soluzione è **controllare**, prima di entrare in fase di esecuzione, **se l'istruzione andrà a scrivere su un registro che è già in fase di esecuzione**.

⚠ Quindi **riassumendo**, se l'hazard detection è sempre fatto nella fase di ID, vanno fatti 3 controlli:

- **Structural hazards** (riguardo alla divide unit e alla write port)
 - **RAW data hazards** (verificare se un registro sorgente è presente nei registri destinazione delle istruzioni pendenti)
 - **WAW data hazards** (verifica se l'istruzione nella fase di ID ha lo stesso registro destinazione di qualche istruzione)

⚠ Non sempre ci si ferma in fase di decode, ma può succedere che si fermi anche in fase di execute.

⚠ Il MIPS R4000 è un processore a 64 bit con istruzioni simili al MIPS64; usava una pipeline a 8 stage per risolvere problemi di cache dovuti agli accessi (accesso alla MEM in più fasi), ed una frequenza di clock alta [le pipeline lunghe si chiamano **SUPERPIPELINES**]. La FP unit è composta da 3 unità funzionali (divider, multiplier e adder) ed aveva 8 stage.



ESERCIZIO (capire le situazioni rischiose, inserendo gli stalli nei momenti giusti, facendo da control unit "umana")

Data l'architettura MIPS64 con caratteristiche [sx] e il programma [dx]:

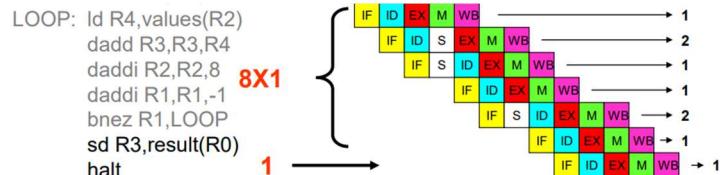
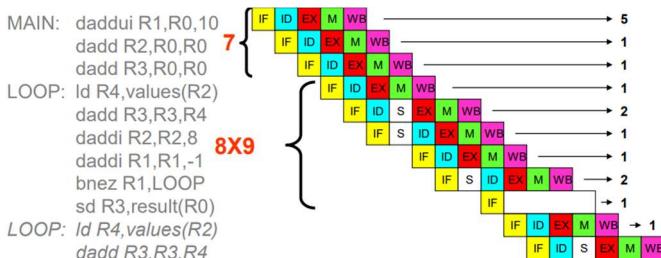
- Integer ALU: 1 clock cycle
 - Data memory: 1 clock cycle
 - FP multiplier unit: pipelined 8 stages
 - FP arithmetic unit: pipelined 2 stages
 - FP divider unit: not pipelined unit, requiring 12 clock cycles
 - branch delay slot: 1 clock cycle (disabled)
 - forwarding is enabled

```

; 10 integers addition
;-----
;.data
values:.word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ;64-bit integers
result:.space 8
.text
MAIN: daddui R1,R0,10      ;R1 <- 10
      dadd R2,R0,R0       ;R2 <- 0 POINTER REG
      dadd R3,R0,R0       ;R3 <- 0 RESULT REG
LOOP: ld R4,values(R2)      ;GET A VALUE IN R4
      dadd R3,R3,R4        ;R3 <- R3 + R4
      daddi R2,R2,8         ;R2 <- R2 + 8 POINTER INCREMENT
      daddi R1,R1,-1        ;R1 <- R1 - 1 DECREMENT COUNTER
      bnez R1,LOOP
      sd R3,result(R0)     ;Result in MEM
      halt                 ;the end

```

Ci viene chiesto di calcolare il numero di cicli:



Dunque un totale di **$7 + 8 \times 10 + 1 = 88$** .

⚠ Ricorda che **se avessimo avuto una div.d** avremmo dovuto riservare **8 stage alla sua "E"** (perché nella traccia c'è scritto FP multiplier unit con 8 stages) ad essa; **se abbiamo 2 div.d in sequenza**, essendo che non possiamo fare la pipeline sulla divisione, dobbiamo inserire degli **stalli fino a che non finisce la "E" della div.d precedente**.

⚠ Se dobbiamo completare la "tabella" (scaletta) delle fasi dei vari stadi **all'esame**, dovremmo scrivere le lettere **F** (fetch) **D** (decode) **E** (execute) **M** (memory) **W** (write back).

In un altro **ESERCIZIO** ne abbiamo 6 stage della E dell'FP multiplier unit; la "**Halt**" è come se fosse composta da F e poi 4 N (non) come fossero D E M W

- ⚠ Il doppio **D** (evidenziato in rosa) è come se fosse uno stallo S per far arrivare il valore di r2 dalla fase prima di E, ma essendo che non si "blocca", viene allungata la fase di decode (e non stallata).
 - ⚠ Gli stalli si propagano sulla **verticale sotto di essi**.
 - ⚠ Se prima ho una **LOAD**, il dato sarà disponibile solo dopo la M.
 - ⚠ Se ho operazioni che usano unità architetturali diverse, posso farle in pipeline (es. add e div).
 - ⚠ Il **controllo sul salto avviene nella decode** (D), quindi la D deve prendere il dato almeno dopo la E se il dato viene dalla istruzione prima.

In un altro **ESERCIZIO** simile al precedente, vediamo anche l'applicazione della **LEGGE DI AMDAHL**:

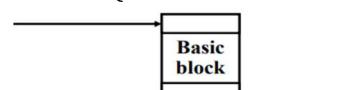
Il testo ci dice che i clock richiesti dalla add.d passano da 2 a 1 (SPEEDUP = 2X). Dalla **SIMULAZIONE** ottengo $6+31*100 = 3106$; perciò SPEEDUP = $3206/3106 = 1.032X$. Con **AMDAHL** devo fare:

$$\text{fraction}_{\text{enhanced}} = \frac{\text{n° di volte (istruzioni) dello speedup}}{\text{n° istruzioni totali}} = \frac{200}{2 + 100 * 32} = 0.0625 \rightarrow$$

$$\text{speedup}_{\text{enhanced}} = \frac{1}{(1 - \text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}}} = \frac{1}{(1 - 0.0625) + \frac{0.0625}{2}} = 1.032$$

Ora parliamo di **INSTRUCTION LEVEL PARALLELISM**, che ha 2 approcci: **DYNAMIC & STATIC** (lo statico è meno frequente, ma prevalente nel mercato embedded).

Un **BASIC BLOCK** è una sequenza di istruzioni **senza branch-in** (ad esclusione dell'ingresso) e **senza branch-out** (ad esclusione dell'uscita). In un basic block, il compilatore perciò può fare **RESCHEDULING** delle istruzioni per **ottimizzare il codice**, un codice che faccia $a = b + c$: $d = e - f$ (assumendo che la load abbia una latenza di 1 clock).



```

1 LD Rb, b      IF ID EX MEM WB
2 LD Rc, c      IF ID EX MEM WB
3 ADD Ra, Rb, Rc  IF ID st EX MEM WB
4 SD Ra, Va      IF st ID EX MEM WB
5 LD Re, e        IF ID EX MEM WB
6 LD Rf, f        IF ID EX MEM WB
7 SUB Rd, Re, Rf  IF ID st EX MEM WB
8 SD Rd, Vd      IF st ID EX MEM WB

```

1	LD Rb, b	IF	ID	EX	MEM	WB	
2	LD Rc, c	IF	ID	EX	MEM	WB	
3	LD Re, e	IF	ID	EX	MEM	WB	
4	ADD Ra, Rb, Rc	IF	ID	EX	MEM	WB	
5	LD Rf, f	IF	ID	EX	MEM	WB	
6	SD Ra, Va		IF	ID	EX	MEM	WB
7	SUB Rd, Re, Rf		IF	ID	EX	MEM	WB
8	SD Rd, Vd		IF	ID	EX	MEM	WB

⚠ Nell'esempio iniziale sono necessari 14 clock cycle; con il rescheduling, sono necessari **12** cicli.

⚠ Dato che le istruzioni potrebbero dipendere da altre, il parallelismo del basic block è **limitato** (altre tecniche).

Si passa al **LOOP LEVEL PARALLELISM** (se ogni istruzione del ciclo è indipendente, possiamo parallelizzarle):

- **LOOP UNROLLING** → duplicare il codice all'interno del ciclo, in modo da fare più istruzioni in parallelo.

Vantaggi: riduciamo il numero di controlli effettuati, aumentiamo la chance che il compilatore elimini gli stalli.

Svantaggi: aumenta la dimensione del codice

```
1 // originale
2 for (i=0;i<N;i++) {
3     body
4 }
```

```
1 // unrolled
2 for (i=0;i<N/4;i++) {
3     body
4     body
5     body
6     body
7 }
```

- **SIMD** → portare le single instruction stream a multiple data streams (SIMD), usando “vector processors”, istruzioni vettoriali che lavorano su un set di dati, GPU e unità funzionali per eseguire task simili in parallelo lavorando su dati multipli

Ci sono 3 tipi di **DIPENDENZE**:

- **DIPENDENZE DI DATI** → istruzione *i* dipende da istruzione *j*, con *i* che deve produrre un risultato che viene usato da *j*, oppure istruzione *j* dipende da istruzione *k*, con *k* che a sua volta dipende da *i*
 - **Dipendenze e hazard:** le dipendenze sono parte del programma, mentre gli hazards sono parte della pipeline (gli stalli dipendono sia dal programma sia dalla pipeline)
 - **Dipendenze di memoria** (a differenza di quelle a livello di registri, sono difficili da trovare e sono rilevate solo a runtime)
- **DIPENDENZE DI NOME** → 2 istruzioni fanno riferimento allo stesso registro o alla stessa locazione di memoria (nome), ma non c'è flusso di dati associato al nome. 2 tipi:
 - **ANTIDEPENDENCE** = istruzione *j* scrive un registro o una locazione di memoria che l'istruzione *i* deve leggere, ed *i* viene eseguita prima
 - **OUTPUT DEPENDENCE** = entrambe le istruzioni *i* e *j* scrivono nello stesso registro o nella stessa porzione di memoria

Esempio (tra riga 2 e 4 c'è antidependence, tra 1 e 4 output dependence):

```
1 Loop: L.D F0, 0(R1)
2          ADD.D F4, F0, F2
3          S.D F4, 0(R1)
4          L.D F0, -8(R1)
5          ADD.D F4, F0, F2
6          S.D F4, -8(R1)
7          L.D F0, -16(R1)
8          ...
```

Questo viene risolto con **REGISTER RENAMING** (identificare i casi in cui la dipendenza dei dati possa non esistere usando un **ulteriore registro**).

⚠ In queste dipendenze, troviamo anche RAW (Read after write) hazards, WAW hazards e WAR hazards.

- **DIPENDENZA DI CONTROLLO** → quando un'istruzione dipende da un branch. Esempio (S1 dipende da p1, S2 dipende da p2) [dx] (qui attenzione anche ai **data flow**).

```
1 if p1 {
2     S1;
3 }
4 j
5 if p2 {
6     S2;
7 }
```

Passiamo ora a **ECCEZIONI & INTERRUZIONI**: **ECCEZIONI (EXCEPTIONS)** sono eventi **interni** che modificano la normale esecuzione del programma; **INTERRUZIONI (INTERRUPTS)** sono eccezioni dovute ad eventi **esterni**. Ci sono varie cause di eccezioni: **I/O request**, **System Call**, **Tracing instruction execution**, **Breakpoint** (interrupt richiesto dal programmatore) [**Trap** (SW Interrupt)], **Arithmetic overflow/underflow**, **Page fault** (MMU fault), **Reset**, **Misaligned memory access**, **Memory-protection violation**, **HW malfunction**, **power failure** ...

Le eccezioni si **classificano** in:

- Sincrone o Asincrone
- Richieste dall'utente o Forzate (coerced)
- Mascherabili (maskable) o Non mascherabili
- Intra-istruzione o Inter-istruzioni
- Con return o Senza return

La maggior parte dei dispositivi sono “**restartable machines**” (data un’eccezione, ripartire da dove si è bloccato). Quando avviene un’eccezione, la pipeline deve fare i seguenti **passi**:

1. forzare una **trap nella pipeline nella prossima fase di F** (fetch)
2. finchè non viene presa la trap, devono essere **disabilitate tutte le W** per le istruzioni che stanno causando l’eccezione e per tutte le istruzioni nella pipeline
3. quando la procedura di gestione dell’eccezione (“**routine**”) prende il controllo, **salva il PC** (program counter) **dell’istruzione che ha causato l’eccezione**

Una volta terminata la gestione, un’istruzione speciale fa tornare la macchina all’origine dell’eccezione e ricarica il PC originale (facendo ripartire lo stream di istruzioni).

⚠️ Interrupt protocol in:

- **80x86** → arriva un’eccezione da un dispositivo esterno, la CPU rileva l’interruzione, legge un numero dal databus (“N”), salva il valore del Processor Status Word (PSW) e dell’indirizzo di ritorno nello stack, resetta l’interrupt flag (disabilitando le interruzioni esterne e la trap mode); usando N come indice, il processore legge dalla Interrupt Vector Table (IVT) l’indirizzo a cui saltare, e salta alla Interrupt Service Routine (ISR)
- **ARM** → arriva un’eccezione da un dispositivo esterno, la CPU rileva l’interruzione, fa il push del current stack frame (composto da 8 registri incluso il PC e il Processor Status Register), aggiorna i flag del processore e salta all’indirizzo dato dall’interruzione di tipo “N” (in tale posizione verrà messo il salto alla attuale ISR)

Distinguiamo:

- **Precise exceptions** = quando c’è un’istruzione che genera eccezione, **tutte le precedenti devono essere completate, mentre quelle che seguono vengono rieseguite dall’inizio** (⚠️ ripartire dopo un’eccezione è complesso se non sono gestite in modo preciso) [ha costo alto in performance ed è complicato con le multiple cycle instructions]
- **Imprecise exceptions** = opposto rispetto a prima, vediamo un esempio:

```

1 DIV.D F0, F2, F4
2 ADD.D F10, F10, F8
3 SUB.D F12, F12, F14

```

L’istruzione ADD.D e SUB.D sono completate prima di DIV.D (out-of-order completion). Se dovesse essere rilasciata una eccezione da parte di SUB.D, questa sarebbe gestita in modo impreciso.

Pipeline stage	Cause of exception
IF	Page fault on instruction fetch, Misaligned memory access, Memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch, Misaligned memory access, Memory-protection violation
WB	None

⚠️ Riprendendo il discorso di prima riguardo alle tipologie di eccezioni, vediamole nei vari stadi della pipeline.

Altro caso sono le **eccezioni contemporanee**; vediamo un esempio: immaginiamo di avere un’eccezione nella MEM di LD e nella EX di DADD; la 1^a viene processata e, se la causa viene rimossa, la 2^a viene gestita. Esistono dei casi in cui però le 2 eccezioni possono avvenire in ordine inverso; si risolve usando un flag di stato per ogni istruzione della pipeline: se avviene un’eccezione viene settato il flag di stato, e se questo è settato, l’istruzione non può fare operazioni di write (quando un’istruzione raggiunge l’ultimo stage, se il flag è settato, viene rilasciata un’eccezione).

⚠️ Quando un’istruzione è garantito che finisca, è detta “**committed**”; alcune macchine hanno istruzioni che possono cambiare stato prima di essere committed e, se una di queste istruzioni venisse abortita a causa di un’eccezione, lascerebbe la macchina in uno stato alterato, rendendo difficile implementare eccezioni precise.

⚠️ Dal laboratorio si vede che per fare lo schema della pipeline **senza forwarding**, se ho un’istruzione che usa un dato dell’istruzione prima, la E dopo deve partire un clock dopo la W prima.

4) CACHE

Le **CACHE** sono memorie più veloci di hard disk e ram (**velocità di accesso >>**); si basano sui **principi di**:

- **località temporale** → se al tempo t la CPU accede ad una cella di memoria, è probabile che dovranno riaccedere a quella cella in un tempo $t+\delta t$
- **località spaziale** → se CPU accede ad una cella di memoria con indirizzo x , è probabile che dovranno riaccedere a celle di memoria a lei vicine con indirizzo $x \pm e$

Dunque se un **blocco intero** viene caricato in cache ad un tempo t_0 , è probabile che ad un certo tempo δt il programma troverà in cache tutte le word necessarie. Il **TEMPO DI ACCESSO ALLA MEMORIA MEDIO** sarà:

$$t_{access} = hC + (1 - h)M \quad \text{con} \quad \begin{cases} h = \text{cache hit ratio} \approx 0.9 \\ C = \text{cache access time} \\ M = \text{memory access time (se il dato non è in cache)} \end{cases}$$

La cache si divide in **parte dati** (divisa in **directory array** e **data array**, dove ogni entry è una **cache line** caratterizzata da 1 **bit di validità** [indicano se blocco presente o no], 1 **tag** [parte di indirizzo (non intero) che indica il blocco di memoria presente in quel momento] e 1 **blocco dati/memoria** [può contenere più word]) e **parte controllo**.

Si differenzia **cache hit** (quando i dati richiesti alla cache sono presenti nella cache) e **cache miss**.

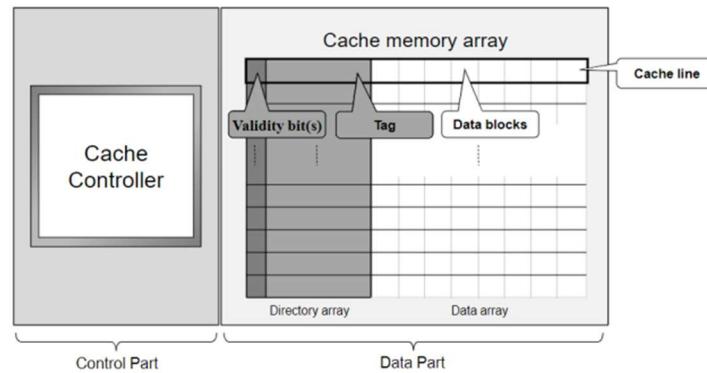
Per **trovare un blocco in cache**? Quando un dato della memoria deve essere indirizzato, il suo **CPU Address** è composto da **TAG** (indica il blocco di memoria), **INDEX** (indica la cache line) e **OFFSET** (indica la word all'interno del blocco). Per trovare il blocco in cache è sufficiente un **MUX (decoder)** che opera sull'indirizzo [es. se ho 1024 byte → tag = 19 bit, index = 10 bit, offset = 3 bit].

La cache sta **tra CPU e bus** (o tra memoria principale e bus, ma non conviene). Ogni volta che il processore fa un accesso in memoria, la cache interpreta l'indirizzo e **controlla se la word è in cache o no**:

- se **cache hit** → cache riduce il tempo di accesso alla memoria di un fattore dipendente dal rapporto tra il tempo di accesso alla cache e alla memoria
- se **cache miss** → cache risponde in 2 modi:
 - accede alla memoria, carica il blocco mancante e risponde alla richiesta (tempo di accesso >)
 - accede alla memoria e risponde alla richiesta (load through o early restart) (costo >, performance >)

Oggi le cache sono separate in **cache istruzioni** e **cache dati (architettura Harvard)**; le caratteristiche sono:

- **cache size** (dimensione [kB, pochi MB]): se dimensione >, costo >, prestazioni sistema >, ma velocità cache <
- **block size** (dimensione blocco di memoria)
- **MAPPING** → meccanismo con cui una cache line viene associata ad un blocco di memoria; il tipo di mappatura (modello di associatività):
 - **Direct**: ciascun blocco di memoria è associato staticamente ad un set k in cache, usando l'espressione $k=index \% N$ (N = numero di cache lines) [il calcolo di k può essere fatto prendendo i bit LSB dell'index]. Pro = facile implementazione HW; Contro = se programma accede spesso a 2 blocchi corrispondenti alla stessa cache line, avviene una miss a ciascun accesso in memoria
 - **Set associative**: per calcolare dove scrivere il valore serve il numero di set $s=N/W$ (N = numero di cache lines; W = numero di word lines); ciascun blocco è associato ad un set k con $k=index \% s$. Un blocco con indice $index$ può essere inserito in una qualsiasi delle word lines del set k . Permette quindi di salvare ogni elemento della memoria in 2 linee della memoria (dati più usati trovabili in cache).
 - **Fully associative**: ciascun blocco della memoria principale può essere messo in un **blocco qualsiasi** della cache. Pro = maggiore flessibilità; Contro = maggiore complessità HW.
- **REPLACING ALGORITHM** (algoritmo di rimpiazzamento) → per sostituire le cache lines, deve essere usato un algoritmo che individui quale rimuovere; vari tipi:
 - **LRU** (least recently used) = scegli la meno recentemente usata
 - **pseudolRU** = età di ciascuna cache line è salvata in albero binario, di cui ogni nodo è un history bit



- **FIFO** (first in first out) = scegli la prima; con lui si usa il “second chance”, ovvero ogni nodo ha 1 bit di utilizzo: quando viene usato viene posto a 1 (dandogli una seconda chance in quanto, essendo stato appena usato, potrebbe essere ancora utile); vengono rimossi solo i nodi con il bit a 0
- **LFU** (least frequently used) = scegli la meno usata [più complesso ma più efficace]
- **Random**

⚠ Quando avviene un’operazione di scrittura su un dato presente in cache, bisogna **aggiornare anche il dato in memoria principale**; si usano 2 soluzioni:

- ✓ **Write Back** = per ogni cache block, c’è un **dirty bit** che indica se il blocco è stato cambiato o meno da quando è stato caricato in cache; l’aggiornamento in memoria principale avviene solo quando il blocco viene sostituito dalla cache ed è settato il dirty bit [lenta, inconsistenti in sistemi multicore]
- ✓ **Write Through** = ogni volta che la CPU fa una scrittura, questo viene scritto sia in cache sia in memoria principale [meno efficiente, ma le scritture sono molto meno frequenti delle letture quindi nemmeno troppo]

La **CACHE COHERENCE** (**coerenza tra le cache**) è uno dei problemi principali in sistemi multicore con memoria condivisa (in cui ogni CPU ha la sua cache) [stesso problema se ho un DMA controller]. Per risolverlo si mette un **validity bit** per ogni cache line: inizialmente disabilitati, se rimango disabilitati vuol dire che non è stato fatto alcun accesso al blocco e quindi deve dare una miss.

⚠ Conviene usare più bit di cache per avere più livelli di cache (L1 = 1° livello [piccolo e veloce], L2 = 2° [lento e capiente], L3 = 3° [molto lento ma molto capiente]).

Esempio:

Immaginiamo di avere una memoria cache con le seguenti caratteristiche:

- 64kb di dimensione
- direct mapping
- 4byte blocks
- 32 bit di indirizzo

Determina la struttura della cache (numero di linee, dimensione del tag field).

Ogni blocco è di 4 byte, quindi se ho 2^{32} byte avrò 2^{30} blocchi. Se la cache ha 64 kbyte, avrò 2^{16} blocchi quindi la cache ha 2^{14} linee. Il tag è composto da 30 bit, ma 14 fanno riferimento alla linea e dunque solo 16 sono per il tag field.

La dimensione totale della cache è dunque:

$$2^{14} * (32 + 16) = 2^{14} * 48 = 768\text{kbit} = 96\text{kByte}$$

5) BRANCH PREDICTION

I **salti** possono impattare molto sulle prestazioni della pipeline, per questo conviene fare **predizioni sul risultato** delle branch (“**forecasting branches**”) in modo da non perdere troppe performance, con lo scopo di ridurre le chance che il controllo dipenda da **cause di stallo**. Se ne occupa la **BPU (Branch Prediction Unit)** con tecniche:

- **STATIC branch prediction** → può essere utile in combinazione con altre tecniche statiche come l'**enabling delayed branches** e il **rescheduling to avoid data hazards**. Il compilatore può:
 - predire **sempre** i branch come **taken** (presi)
 - predire **in base al branch direction** (se siamo in un ciclo) [i forward branches sono poco presi, mentre i backward branches sono spesso presi]
 - predire **in base alle informazioni di profiling** (presa una sequenza di input, eseguire il programma con quella sequenza e fare una statistica sul comportamento del branch) prese dalle prime esecuzioni
- **DYNAMIC branch prediction** → sono **implementati in HW** e usano gli indirizzi delle istruzioni di branch per attivare meccanismi differenti di predizioni. Si basano sui principi di località spaziale, località temporale e località di branch (se il conditional branch è ristretto ad un insieme di poche possibili destinazioni).

Funzionamento:

- 1) Una predizione sul branch è fatta nello stage D (decode)
- 2) L’istruzione predetta è segnata come prossima istruzione da eseguire, senza stallare la pipeline
- 3) Il branch è completato e la predizione viene verificata nello stage E (execution):

- a. Se TRUE, si va avanti senza stallo
- b. Se FALSE, la pipeline è “flushed” (viene pulita) e l’istruzione corretta viene segnata

Tecniche di implementazione:

- o **BHT (Branch History Table)** [metodo più facile]: è una piccola memoria indicizzata dalla più piccola porzione di indirizzo dell’istruzione di branch; ogni entry ha **1 o più bit che registrano se il branch è stato preso (taken) o no l’ultima volta che è stato eseguito.**

Ogni volta che viene decodificata (D) un’istruzione di branch, viene eseguito un accesso alla BHT (usando la porzione di indirizzo per l’indice). La predizione viene salvata nella tabella e il NPC (nuovo PC) viene calcolato con tale predizione; quando il risultato è noto, la tabella viene aggiornata.

L’efficacia dipende dal metodo usato, ma è soggetto ad **aliasing** (una linea della tabella potrebbe far riferimento ad un altro salto e non a quello corrente) ed al problema di accuratezza della predizione [nel MIPS questo metodo è inutile perché la valutazione del branch avviene nella fase di F]. **Esempio:** Prendiamo un loop che è stato preso 9 volte di seguito e poi al 10° deve uscire; se ipotizziamo che la entry non è condivisa con altri branch, la predizione usando una BHT di 1 bit verrà sbagliata solo la 1^ e la 10^ con un successo dell’80%.

- o **2-bit prediction schemes:** predizione migliore; per ogni branch vengono tenuti **2 bit** (la **predizione viene cambiata solo dopo 2 miss**). Un’estensione sono gli **schemi a n-bit**, dove i bit si comportano come un contatore (cosa che non avveniva nel 2-bit), ovvero +1 ad ogni branch taken, -1 ad ogni miss; quando il counter > metà del max, il branch viene predetto come preso.

Le performance dei branch vengono impattate da **accuratezza della predizione, costo del branch** (penalità per errore) e **branch frequency** (minore in floating point programs).

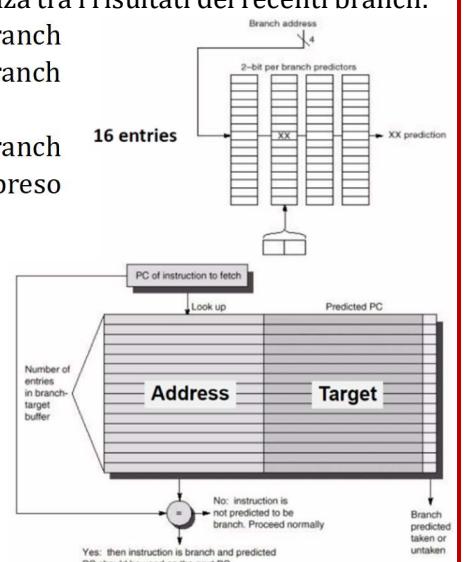
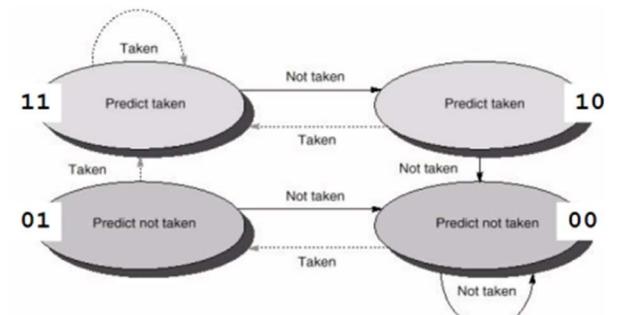
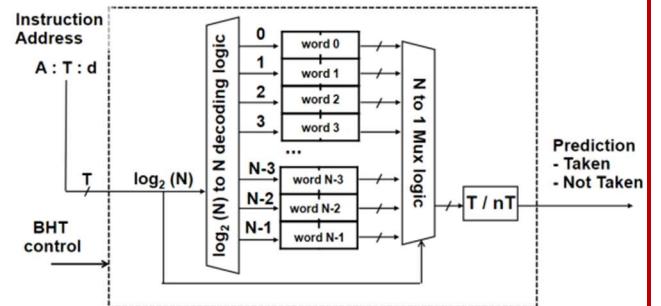
- o **2-level predictors (Correlating predictors):** basato sulla dipendenza tra i risultati dei recenti branch:

- **(m,n) predictors:** uso il comportamento degli ultimi m branch (salvato in uno shift register ad m -bit) per scegliere da 2^m branch predictors, ciascuno di n -bit
- **(1,1) predictor** [$m=1, n=1$]: 1 riporta la predizione se il branch precedente è stato preso, 1 se il branch precedente non è stato preso
- **(2,2) predictor** [shift register a 2 bit] [img dx]

- o **Branch-target buffer:** per ridurre il numero di effetti sul controllo delle dipendenze (come sapere se il branch è taken o no, e il nuovo valore del PC), si usa il branch-target buffer (o cache): ogni sua entry contiene **l’indirizzo del branch considerato e il target value da caricare nel PC**; il nuovo valore viene caricato nel PC alla fine dello stadio F (fetch), prima ancora che la branch sia decodificata (D) [buona, ma costosa]. **Es:**

Let assume the following penalty parameters

Instruction in buffer	Prediction taken	Actual branch taken	Penalty c.c.
Yes	taken	taken	0
Yes	taken	not taken	2
No		taken	2
No		not taken	0



Let also assume that

- the prediction accuracy is 90%
- the hit rate in the buffer is 90%
- taken branches are 60%.

Which is the total branch penalty?

Branch Penalty =

- Hit branches & Not Taken +
- Miss branches & Taken
- = (percent buffer hit rate × percent incorrect predictions × 2) + ((1 - percent buffer hit rate) × taken branches × 2)
- = (90% × 10% × 2) + (10% × 60% × 2)
- = $0.18 + 0.12 = 0.30$ C.C.

Vediamo degli ESERCIZI:

- BHT (Caso 1):

Considering a 2-bit saturating counter BHT of 1K entries; and assuming that the processor executes the following code fragment, determine the BHT final state and calculate the misprediction ratio for the presented case.

```
L0: ...
for(i=1;i<=100;i++){
    read_data();
    if (aa==2)
        aa = 0;
    if (bb==2)
        bb = 0;
    if (aa == bb)
        ...
    L1: ...
    DADDUI R3, R1, #2
    BNEZ R3, L1
    DADD R1, R0, R0
    BNEZ R3, L2
    DADD R2, R0, R0
    DSUB R3, R1, R2
    BNEZ R3, L3
}
L2: ...
DADDUI R3, R2, #2
BNEZ R3, L2
DADD R2, R0, R0
DSUB R3, R1, R2
BNEZ R3, L3
...
L3: ...
DADDUI R4, R4, #1
BNEZ R4, L0
```

Iteration 1				
address	Instruction	BHT	Prediction	misP. counter
0x0000	L0: ...	0	NT	
...	...	0	NT	
0x0010	DADDUI R3, R1, #2	0	NT	
0x0014	BNEZ R3, L1	3	T	2
0x0018	DADD R1, R0, R0	0	NT	
0x001C	L1: DADDUI R3, R2, #2	0	NT	
0x0020	BNEZ R3, L2	3	T	2
0x0024	DADD R2, R0, R0	0	NT	
0x0028	L2: DSUB R3, R1, R2	0	NT	
0x002C	BNEZ R3, L3	3	T	2
0x0030	...	0	NT	
0x0034	L3: ...	0	NT	
0x0038	DADDI R4, R4, #1	0	NT	
0x003C	BNEZ R4, L0	3	T	2
0x0040	...	0	NT	

Iteration 2				
address	Instruction	BHT	Prediction	misP. counter
0x0000	L0: ...	0	NT	
...	...	0	NT	
0x0010	DADDUI R3, R1, #2	0	NT	
0x0014	BNEZ R3, L1	2	NT	2
0x0018	DADD R1, R0, R0	0	NT	
0x001C	L1: DADDUI R3, R2, #2	0	NT	
0x0020	BNEZ R3, L2	2	NT	2
0x0024	DADD R2, R0, R0	0	NT	
0x0028	L2: DSUB R3, R1, R2	0	NT	
0x002C	BNEZ R3, L3	2	NT	2
0x0030	...	0	NT	
0x0034	L3: ...	0	NT	
0x0038	DADDI R4, R4, #1	0	NT	
0x003C	BNEZ R4, L0	2	NT	2
0x0040	...	0	NT	

- BHT (Caso 2):

Case 2 - Iteration 1

address	Instruction	BHT	Prediction	misP. counter
0x0000	L0: ...	0	NT	
...	...	0	NT	
0x0010	DADDUI R3, R1, #2	0	NT	
0x0014	BNEZ R3, L1	1	NT	1
0x0018	DADD R1, R0, R0	0	NT	
0x001C	L1: DADDUI R3, R2, #2	0	NT	
0x0020	BNEZ R3, L2	1	NT	1
0x0024	DADD R2, R0, R0	0	NT	
0x0028	L2: DSUB R3, R1, R2	0	NT	
0x002C	BNEZ R3, L3	1	NT	1
0x0030	...	0	NT	
0x0034	L3: ...	0	NT	
0x0038	DADDI R4, R4, #1	0	NT	
0x003C	BNEZ R4, L0	1	NT	1
0x0040	...	0	NT	

Case 2 - Iteration 100

address	Instruction	BHT	Prediction	misP. counter
0x0000	L0: ...	0	NT	
...	...	0	NT	
0x0010	DADDUI R3, R1, #2	0	NT	
0x0014	BNEZ R3, L1	3	T	2
0x0018	DADD R1, R0, R0	0	NT	
0x001C	L1: DADDUI R3, R2, #2	0	NT	
0x0020	BNEZ R3, L2	3	T	2
0x0024	DADD R2, R0, R0	0	NT	
0x0028	L2: DSUB R3, R1, R2	0	NT	
0x002C	BNEZ R3, L3	3	T	2
0x0030	...	0	NT	
0x0034	L3: ...	0	NT	
0x0038	DADDI R4, R4, #1	0	NT	
0x003C	BNEZ R4, L0	2	T	3
0x0040	...	0	NT	

Case 2 - Iteration 4

address	Instruction	BHT	Prediction	misP. counter
0x0000	L0: ...	0	NT	
...	...	0	NT	
0x0010	DADDUI R3, R1, #2	0	NT	
0x0014	BNEZ R3, L1	0	NT	2
0x0018	DADD R1, R0, R0	0	NT	
0x001C	L1: DADDUI R3, R2, #2	0	NT	
0x0020	BNEZ R3, L2	0	NT	2
0x0024	DADD R2, R0, R0	0	NT	
0x0028	L2: DSUB R3, R1, R2	0	NT	
0x002C	BNEZ R3, L3	0	NT	2
0x0030	...	0	NT	
0x0034	L3: ...	0	NT	
0x0038	DADDI R4, R4, #1	0	NT	
0x003C	BNEZ R4, L0	3	T	2
0x0040	...	0	NT	

Case 2 - Iteration 100

address	Instruction	BHT	Prediction	misP. counter
0x0000	L0: ...	0	NT	
...	...	0	NT	
0x0010	DADDUI R3, R1, #2	0	NT	
0x0014	BNEZ R3, L1	0	NT	50
0x0018	DADD R1, R0, R0	0	NT	
0x001C	L1: DADDUI R3, R2, #2	0	NT	
0x0020	BNEZ R3, L2	0	NT	50
0x0024	DADD R2, R0, R0	0	NT	
0x0028	L2: DSUB R3, R1, R2	0	NT	
0x002C	BNEZ R3, L3	0	NT	50
0x0030	...	0	NT	
0x0034	L3: ...	0	NT	
0x0038	DADDI R4, R4, #1	0	NT	
0x003C	BNEZ R4, L0	2	T	3
0x0040	...	0	NT	

Case 2 - Iteration 3

address	Instruction	BHT	Prediction	misP. counter
0x0000	L0: ...	0	NT	
...	...	0	NT	
0x0010	DADDUI R3, R1, #2	0	NT	
0x0014	BNEZ R3, L1	1	NT	2
0x0018	DADD R1, R0, R0	0	NT	
0x001C	L1: DADDUI R3, R2, #2	0	NT	
0x0020	BNEZ R3, L2	1	NT	2
0x0024	DADD R2, R0, R0	0	NT	
0x0028	L2: DSUB R3, R1, R2	0	NT	
0x002C	BNEZ R3, L3	1	NT	2
0x0030	...	0	NT	
0x0034	L3: ...	0	NT	
0x0038	DADDI R4, R4, #1	0	NT	
0x003C	BNEZ R4, L0	3	T	2
0x0040	...	0	NT	

Case 2:

- Program inputs for aa and bb alternate values different than 2 and equal to 2.

Misprediction ratio =

Number of mispredicted branches / Total branches

Misprediction ratio = 153 / 400

Misprediction ratio = 38.25%

(2,2) - Iteration 1

address	Instruction	00	01	10	11	2-b sft reg	misP. counter
0x0000	L0: ...	0	0	0	0	00	
...	...	0	0	0	0	00	
0x0010	DADDUI R3, R1, #2	0	0	0	0	00	
0x0014	BNEZ R3, L1	1	0	0	0	01	1
0x0018	DADD R1, R0, R0	0	0	0	0	00	
0x001C	L1: DADDUI R3, R2, #2	0	0	0	0	00	
0x0020	BNEZ R3, L2	0	1	0	0	11	1
0x0024	DADD R2, R0, R0	0	0	0	0	00	
0x0028	L2: DSUB R3, R1, R2	0	0	0	0	00	
0x002C	BNEZ R3, L3	0	0	0	1	11	1
0x0030	...	0	0	0	0	00	
0x0034	L3: ...	0	0	0	0	00	
0x0038	DADDI R4, R4, #1	0	0	0	0	00	
0x003C	BNEZ R4, L0	0	0	0	1	11	1
0x0040	...	0	0	0	0	00	

(2,2) - Iteration 100

address	Instruction	00	01	10	11	2-b sft reg	misP. counter
0x0000	L0: ...	0	0	0	0	00	
...	...	0	0	0	0	00	
0x0010	DADDUI R3, R1, #2	0	0	0	0	00	
0x0014	BNEZ R3, L1	1	3	0	0	10	3
0x0018	DADD R1, R0, R0	0	0	0	0	00	
0x001C	L1: DADDUI R3, R2, #2	0	0	0	0	00	
0x0020	BNEZ R3, L2	0	1	0	3	00	3
0x0024	DADD R2, R0, R0	0	0	0	0	00	
0x0028	L2: DSUB R3, R1, R2	0	0	0	0	00	
0x002C	BNEZ R3, L3	0	0	0	3	00	2
0x0030	...	0	0	0	0	00	
0x0034	L3: ...	0	0	0	0	00	
0x0038	DADDI R4, R4, #1	0	0	0	0	00	
0x003C	BNEZ R4, L0	2	0	0	3	00	5
0x0040	...	0	0	0	0	00	

- Program inputs for aa and bb alternate values different than 2 and equal to 2.

Misprediction ratio =

Number of mispredicted branches / Total branches

Misprediction ratio = 13 / 400

Misprediction ratio = 3.25%

6) DYNAMIC SCHEDULING

Lo **SCHEDULING DINAMICO** consente di identificare le dipendenze che sono **sconosciute nella compilazione**; semplifica il lavoro del compilatore e permette al processore di tollerare ritardi non predibili (permette anche di eseguire lo stesso codice su processori diversi). Vediamo un **esempio**:

```
1 DIV.D F0, F2, F4  
2 ADD.D F10, F0, F8  
3 SUB.D F12, F8, F14
```

→ la pipeline ha uno stallo dopo la DIV.D per la dipendenza tra il risultato della div che è operando della ADD.D; la SUB.D si ferma, anche se non ha dipendenze con altre operazioni (praticamente è un **rescheduling automatico** non fatto da noi)

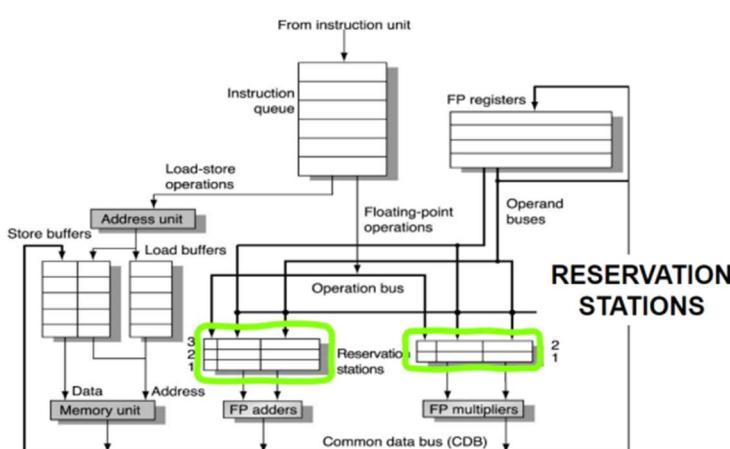
L'**esecuzione out-of-order** prevede l'esecuzione di istruzioni non ancora eseguite; questo può introdurre **WAR hazards** (Write after Read), **WAW hazards** e esecuzione **imprecisa** delle eccezioni (se l'esecuzione out-of-order è concessa, è difficile fare gestione precisa delle eccezioni → quando un'eccezione è rilasciata, un'istruzione precedente o successiva all'eccezione deve ancora essere completata).

Per consentire l'esecuzione fuori ordine, bisogna **dividere lo stage di ID (decode, D) in 2 parti**:

- **issue** = decode instruction, controlla gli hazard strutturali; legge l'istruzione da un registro o da una coda, solo dopo si attende per gli operandi (fase seguente) e si entra in fase di E (esecuzione)
- **read operands** = aspetta fino a che non ci sono data hazards, solo dopo legge gli operandi; dopo si va in E
- ⚠ Se il processore ha più unità funzionali, molte istruzioni possono essere eseguite in parallelo.

Alcune **strategie HW** potrebbero essere applicate per risolvere i problemi di scheduling dinamico, tra cui Scoreboarding e l'algoritmo di Tomasulo. Vediamo il **Tomasulo's algorithm** (architettura di Tomasulo) le principali idee sono tracciare la disponibilità degli operandi e l'introduzione del register renaming. Le **Reservation Stations** sono la novità di Tomasulo e hanno come funzioni:

- **bufferizzare gli operandi** delle istruzioni in attesa dell'esecuzione, non appena questi sono disponibili
- implementare la **logica di emissione**
- **identificare univocamente un'istruzione** nella pipeline (le istruzioni in attesa designano la stazione di prenotazione che fornirà loro un operando in ingresso)



MIPS FP unit structure using Tomasulo approach

Ciascuna reservation station relativa ad un'unità funzionale controlla quando un'istruzione può iniziare l'esecuzione in quella unità.

Ogni volta che viene impartita un'istruzione, il registro specifica che gli operandi "pendenti" sono rinominati con i nomi delle reservation stations in carica di calcolarli (ciò è il **register renaming** in grado di eliminare i WAW e WAR hazards).

I risultati sono passati direttamente alle altre unità funzionali, piuttosto che andare nei registri. Tutti i risultati dalle unità funzionali e dalla memoria sono inviati sul **CDB** (Common Data Bus); il CDB permette a tutte le unità in attesa di operandi di caricarli insieme appena disponibili.

Parliamo di **Instruction execution steps** → le istruzioni vengono **eseguite in 3 fasi**:

1. **Issue** = quando un'istruzione viene presa dalla coda FIFO:
 - a. se non sono disponibili reservation stations, avviene uno structural hazard e l'istruzione è messa in stallo fino a che 1 reservation station non diventa disponibile
 - b. se è disponibile almeno 1 reservation station, le viene mandata l'istruzione con gli operandi (se disponibili, altrimenti si attende anche la loro disponibilità)
2. **Execute** = quando un operando appare sul CDB, viene letto dalla reservation unit/station e appena tutti gli operandi dell'istruzione sono disponibili nella reservation station, l'istruzione può essere eseguita (in questo modo si eliminano i RAW hazards).

Le istruzioni di **load** e **store** sono eseguite in 2 step:

- appena il base register è disponibile, l'effective address viene calcolato e scritto nel load/store buffer

- nel caso della load è eseguita appena la memoria è disponibile, nel caso della store si attende che gli operandi che devono essere scritti siano disponibili

⚠ Per evitare di modificare il comportamento delle eccezioni, nessuna istruzione è autorizzata a iniziare l'esecuzione **fino a che tutti i branch precedenti non sono stati completati** (per questo motivo si può usare la **speculazione** per migliorare questo meccanismo).

3. **Write back** (o write result, che coincide con lo stadio di Mem [M]) = quando il risultato dell'istruzione è disponibile, viene subito scritto nel CDB (dove i registri e le unità funzionali attendono)

Ciascuna reservation station è associata ad un **identificatore ("instruction identifiers")**, il quale **identifica anche gli operandi** necessari per l'istruzione (è in questo modo che l'istruzione li riconosce); ogni reservation station ha quindi i seguenti campi:

Op	V _j	V _k	Q _j	Q _k	A	Busy
----	----------------	----------------	----------------	----------------	---	------

- **Op** = operazione da eseguire
- **V_j** = valore dell'operando sorgente "j"
- **V_k** = valore dell'operando sorgente "k"
- **Q_j** = reservation station che produrrà un operando sorgente "j"
- **Q_k** = reservation station che produrrà un operando sorgente "k"
- **A** = usato dal buffer della load/store per memorizzare prima il campo immediato e poi l'effective address
- **Busy** = status della reservation station

⚠ I **register file** ci sono ancora (sebbene i registri vengano quasi bypassati con Tomasulo); ogni elemento in un register file contiene un campo Qi, il quale contiene il n° della reservation station contenente l'istruzione il cui risultato dovrebbe essere salvato in un registro; se **Qi == null**, non è attiva nessuna istruzione che sta calcolando il risultato per quel registro [vengono eliminati stalli WAW e WAR e la hazard detection logic è distribuita, ma si ha alta complessità HW e il CDB potrebbe essere un bottleneck].

Vediamo sotto un esempio di reservation stations:

Name	Busy	Op	V _j	V _k	Q _j	Q _k	A
Load1	no						
Load2	yes	Load					45 + Regs[R3]
Add1	yes	SUB		Mem[34 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]		Load2	
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

⚠ Il loop unrolling non è più necessario (in quanto le istruzioni si possono eseguire normalmente in parallelo).

⚠ Riguardo a **load/store**, il processore deve fare dei controlli se vengono riordinate:

- ogni volta che una load è pronta per l'issue, **il store buffer è controllato** (per vedere se è segnato per store che agiscono sullo stesso indirizzo della load; in questo caso si aspetta la terminazione delle store)
- ogni volta che una store è pronta per l'issue, **si controllano sia store sia load buffer** (per vedere se sono segnati per istruzioni che agiscono sullo stesso indirizzo della store; in questo caso si aspetta la terminazione delle istruzioni)

Esempio sull'architettura di Tomasulo (partendo dalla base senza "commit" [speculazione]) clock dopo clock:

Instruction Status		j	k	Issue	Execution Complete	Write Result				
Instruction					Busy	Address				
LD	F6	34+	R2	1			Load1	Yes	34+R2	
LD	F2	45+	R3				Load2	No		
MULTD	F0	F2	F4				Load3	No		
SUBD	F8	F6	F2							
DIVD	F10	F0	F6							
ADDD	F6	F8	F2							

Reservation Stations		Time	Name	Busy	OP	S1	S2	RS for j	RS for k	
0 Add1	No									
0 Add2	No									
0 Add3	No									
0 Mult1	No									
0 Mult2	No									

Register result status		Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
1			FU			Load1					

Instruction Status		j	k	Issue	Execution Complete	Write Result				
Instruction					Busy	Address				
LD	F6	34+	R2	1			Load1	Yes	34+R2	
LD	F2	45+	R3	2			Load2	Yes	45+R3	
MULTD	F0	F2	F4				Load3	No		
SUBD	F8	F6	F2							
DIVD	F10	F0	F6							
ADDD	F6	F8	F2							

Reservation Stations		Time	Name	Busy	OP	S1	S2	RS for j	RS for k	
0 Add1	No									
0 Add2	No									
0 Add3	No									
0 Mult1	No									
0 Mult2	No									

Register result status		Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
2			FU			Load2		Load1			

Instruction Status		j	k	Issue	Execution Complete	Write Result				Busy	Address
Instruction							Load1	Yes	34+R2		
LD F6	34+	R2	1	3			Load2	Yes	45+R3		
LD F2	45+	R3	2				Load3	No			
MULTD F0	F0	F2	3								
SUBD F8	F6	F2									
DIVD F10	F0	F6									
ADDD F6	F6	F8									

Reservation Stations		Time	Name	Busy	OP	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk		
0 Add1	No										
0 Add2	No										
0 Add3	No										
0 Mult1	Yes	MULTD				R(F4)	Load2				
0 Mult2	No										

Register result status		Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1						

Instruction Status		j	k	Issue	Execution Complete	Write Result			Busy	Address
LD F6	34+	R2	1	3	4		Load1	No		
LD F2	45+	R3	2	4	5		Load2	No		
MULTD F0	F0	F2	3				Load3	No		
SUBD F8	F6	F2	4							
DIVD F10	F0	F6	5							
ADDD F6	F6	F8	F2							

Reservation Stations		Time	Name	Busy	OP	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk		
0 Add1	Yes	SUBD	M[34+R2]	M[45+R3]							
0 Add2	No										
0 Add3	No										
0 Mult1	Yes	MULTD	M[45+R3]	R(F4)							
0 Mult2	Yes	DIVD		M[34+R2]	Mult1						

Register result status		Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M[45+R3]		M[34+R2]	Add1	Mult2				

Instruction Status		j	k	Issue	Execution Complete	Write Result			Busy	Address
LD F6	34+	R2	1	3	4		Load1	No		
LD F2	45+	R3	2	4	5		Load2	Yes	45+R3	
MULTD F0	F0	F2	3				Load3	No		
SUBD F8	F6	F2	4							
DIVD F10	F0	F6	5							
ADDD F6	F6	F8	F2							

Reservation Stations		Time	Name	Busy	OP	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk		
0 Add1	Yes	SUBD	M[34+R2]					Load2			
0 Add2	No										
0 Add3	No										
0 Mult1	Yes	MULTD	M[45+R3]	R(F4)							
0 Mult2	No										

Register result status		Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M[34+R2]	Add1					

Instruction Status		j	k	Issue	Execution Complete	Write Result			Busy	Address
LD F6	34+	R2	1	3	4		Load1	No		
LD F2	45+	R3	2	4	5		Load2	No		
MULTD F0	F0	F2	3				Load3	No		
SUBD F8	F6	F2	4							
DIVD F10	F0	F6	5							
ADDD F6	F6	F8	F2							

Reservation Stations		Time	Name	Busy	OP	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk		
2 Add1	Yes	SUBD	M[34+R2]	M[45+R3]							
0 Add2	Yes	ADDD			M[45+R3]	Add1					
0 Add3	No										
10 Mult1	Yes	MULTD	M[45+R3]	R(F4)							
0 Mult2	Yes	DIVD		M[34+R2]	Mult1						

Register result status		Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	M[45+R3]		Add2	Add1	Mult2				

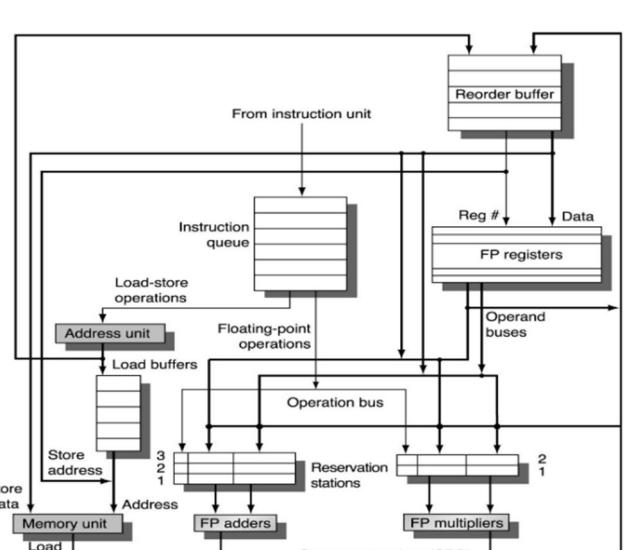
Instruction Status		j	k	Issue	Execution Complete	Write Result			Busy	Address
LD F6	34+	R2	1	3	4		Load1	No		
LD F2	45+	R3	2	4	5		Load2	No		
MULTD F0	F0	F2	3				Load3	No		
SUBD F8	F6	F2	4							
DIVD F10	F0	F6	5							
ADDD F6	F6	F8	F2							

Reservation Stations		Time	Name	Busy	OP	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk		
2 Add1	Yes	SUBD	M[34+R2]	M[45+R3]							
0 Add2	Yes	ADDD		M[45+R3]	Add1						
0 Add3	No										
10 Mult1	Yes	MULTD	M[45+R3]	R(F4)							
0 Mult2	No										

Register result status		Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
57	FU	M*F4	M[45+R3]		(M-M)+M	(M M)-M	M F M	M*F4/M			

Legato al dynamic scheduling, la **SPECULATION HW** è una tecnica per ridurre l'effetto delle dipendenze di controllo in un processore che implementa il dynamic scheduling. Se un processore supporta branch prediction con dynamic scheduling, le istruzioni di fetch e issue sono eseguite come se la predizione fosse sempre corretta; se un processore invece supporta la speculazione, le **esegue sempre**. Le operazioni sono eseguite non appena gli operandi sono disponibili.

L'architettura di Tomasulo viene **estesa per supportare la speculazione**. Lo stadio E (execute) si separa nuovamente in **calcolo dei risultati e instruction commit** (aggiornamento di file register/memoria; eseguito quando una istruzione non è più speculativa); i risultati delle istruzioni che non



hanno ancora fatto commit sono messi nel **ROB (ReOrder Buffer)**, il quale fornisce dei virtual register e integra lo store buffer dell'architettura base di Tomasulo [dunque se le istruzioni non hanno fatto commit, i dati vengono letti dal ROB; altrimenti dal register file]. Ogni entry della ROB ha 4 campi:

- **Instruction type** (branch, store o register)
- **Destination** (registro o indirizzo di memoria)
- **Value** (contiene il valore se l'istruzione è completata ma non ha ancora fatto il commit)
- **Ready** (indica se l'istruzione ha finito la sua esecuzione)

Qui l'**esecuzione delle istruzioni** avviene in **4 fasi**:

1. **Issue** = un'istruzione viene estratta dalla coda delle istruzioni se è disponibile una reservation station vuota e uno slot libero nel ROB, altrimenti viene messa in stallo. Gli operandi delle istruzioni, se presenti nel register file o nel ROB, sono inviati alla reservation station. Il n° della entry del ROB dell'istruzione viene mandata alla reservation station per indicare l'istruzione
2. **Execute** = l'istruzione è eseguita appena sono disponibili tutti gli operandi (presi dal CDB appena un'istruzione li produce) [la lunghezza di questo step varia in base al tipo di istruzione]
3. **Write back/result** = scriveremo nel CDB appena il dato è libero e verrà mandato al ROB. Tutte le reservation stations aspetteranno per il risultato prima di leggerlo, e ciascuna entry verrà poi segnata come disponibile
4. **Commit** = il ROB viene ordinato in base all'ordine originale. Appena un'istruzione raggiunge la head del ROB:
 - a. se è un branch non predetto correttamente, il buffer viene ripulito e l'esecuzione riparte con la corretta istruzione seguente
 - b. se è un branch predetto correttamente, il risultato viene scritto nel registro/memoria

In entrambi i casi, la entry del ROB viene segnata come libera; il ROB è implementato come un **circular buffer**.

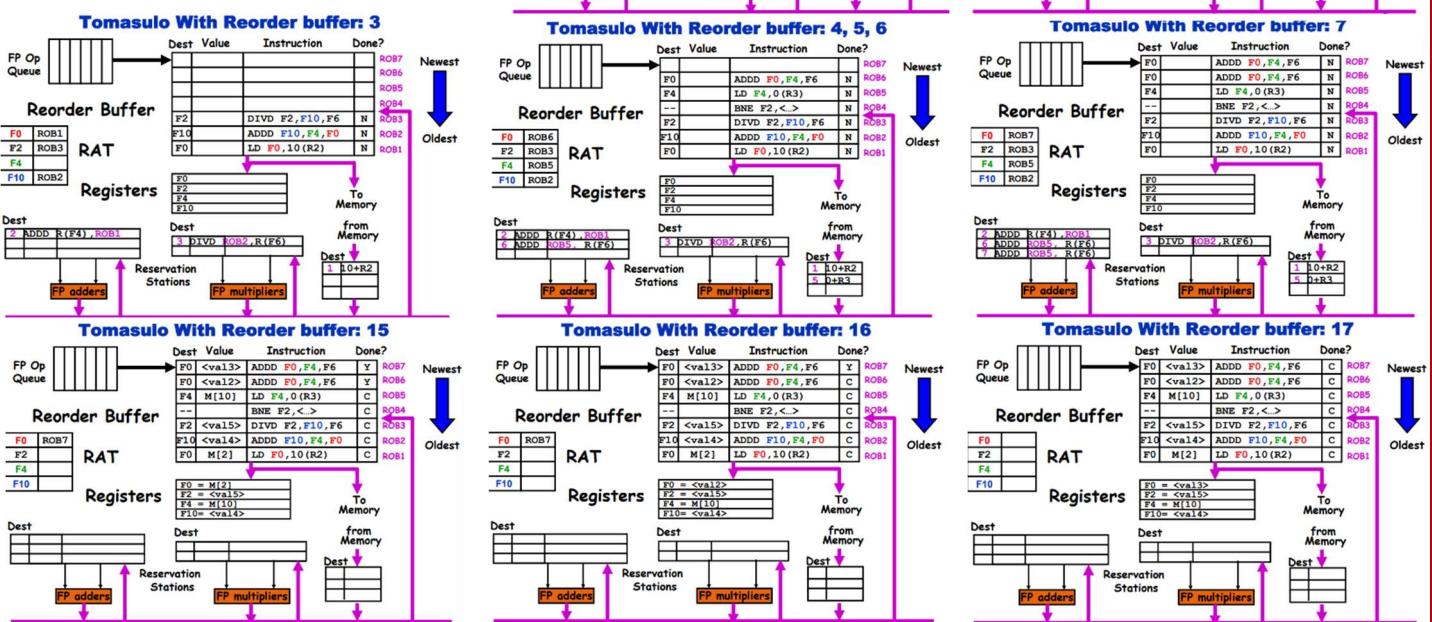
⚠ **WAR e WAW hazards** sono evitati (perché c'è dynamic renaming e l'aggiornamento della memoria avviene in ordine); i **RAW hazards** sono evitati perché avviene un enforcing del program order; inoltre una load non entra nel 2° step se qualche entry della ROB è occupata da una store che ha un destination field = al field della load.

⚠ Le **Store** scrivono in memoria solo quando è stato fatto il commit; per questo, gli operandi in input sono richiesti al momento della commit piuttosto che nello stadio di scrittura dei risultati (ciò significa che il ROB deve avere un ulteriore campo che specifica da dove gli operandi in input dovranno essere presi per ogni store).

⚠ Le eccezioni sono eseguite quando sono aggiunte al ROB (e non appena vengono rilasciate). Quando un'istruzione fa commit, la possibile eccezione viene eseguita e le istruzioni successive vengono tolte dal buffer; se l'istruzione viene eliminata dal buffer, la sua esecuzione viene ignorata (quindi supporta completamente la gestione "precise" delle eccezioni).

Esempio di Tomasulo + speculazione:

1. LD **F0**, 10 (R2)
2. ADDD **F10**, **F4**, **F0**
3. DIVD F2, **F10**, **F6**
4. BNE F2, <...>
5. LD **F4**, 0 (R3)
6. ADDD **F0**, **F4**, **F6**
7. ADDD **F0**, **F4**, **F6**



→ MULTIPLE-ISSUE PROCESSORS (VLIW):

I processori **superscalari** sono sempre più complessi con logiche complesse per gestire le dipendenze, schedulare dinamicamente, fare previsione e speculazione sui branch. Abbiamo anche i **VLIW (Very Long Instruction Word)**, ovvero processori che hanno istruzioni molto lunghe e hanno code con molte operazioni caricate in parallelo (si ha anche però una semplificazione dell'HW che controlla le dipendenze tra istruzioni in quanto non viene effettuato se le istruzioni vengono eseguite in parallelo).

Le **prestazioni** sono limitate dalla limitazione di programmi ILP, difficoltà HW, limitazioni di superscalari e vliw, e dal fatto che è difficile trovare un n° sufficiente di istruzioni indipendenti da eseguire in parallelo. Inoltre, se aumenta il n° di unità funzionali, aumenta la bandwidth del register file e della memoria, quindi aumenta la complessità HW e si riducono le performance (che si può risolvere con accessi multipli alla memoria per clock). La dimensione del codice dei VLIW è molto maggiore.

⚠ Si può implementare un multiple-issue processor con dynamic scheduling usando uno schema simile a quello di Tomasulo; per rendere l'implementazione più facile, le istruzioni non sono mai "issued" alle reservation station out-of-order. Dato che in alcuni clock si possono leggere dal CDB più istruzioni, bisogna duplicare il CDB.

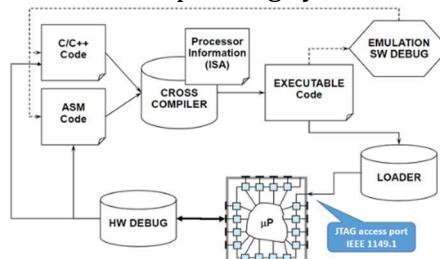
Esempio (con issue e commit di 2 istruzioni per clock; a **sx senza** e a **dx con speculazione**):

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5	6		Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3	4		Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11	12		Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8	9		Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17	18		Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14	15		Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5	6		7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3	4		8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8	9	10	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6	7		11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11			12	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9	10		14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

7) ARM

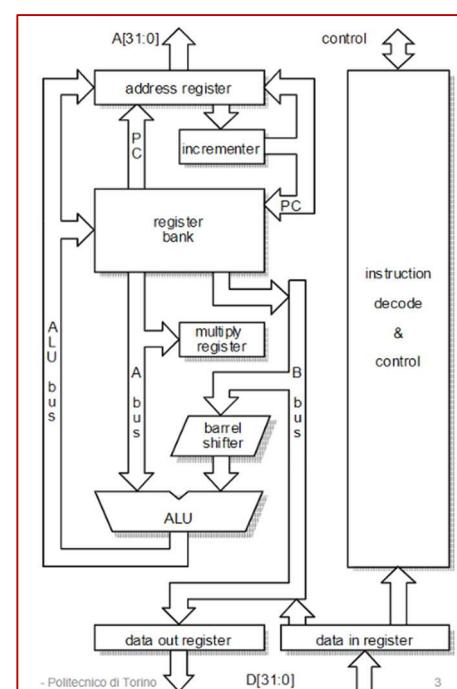
ARM è un'architettura RISC che ha preso il mercato di processori in ambito mobile e embedded. L'architettura si divide in ARM architecture embedded in SoC, ARM operating Systems e ARM compile-support-debug tools. Sotto la **toolchain** di sviluppo ARM:



L'**ARCHITETTURA GENERICA** di un processore arm è mostrata a dx; di particolare vediamo il **barrel shifter** (consente di fare gli shift in modo automatico del 2° registro di un'operazione senza usare un'istruzione apposita).

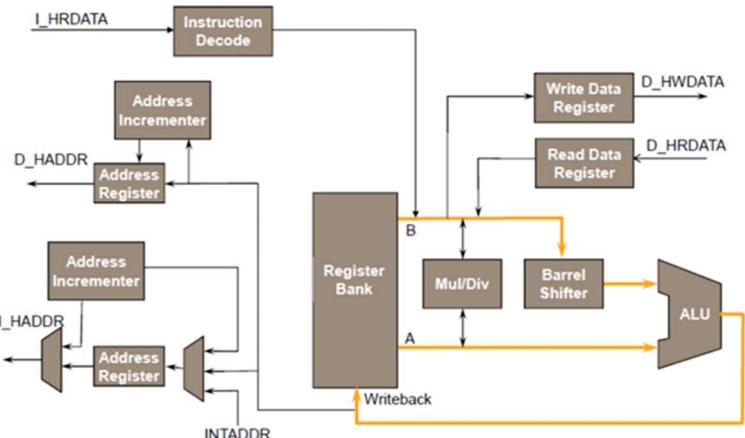
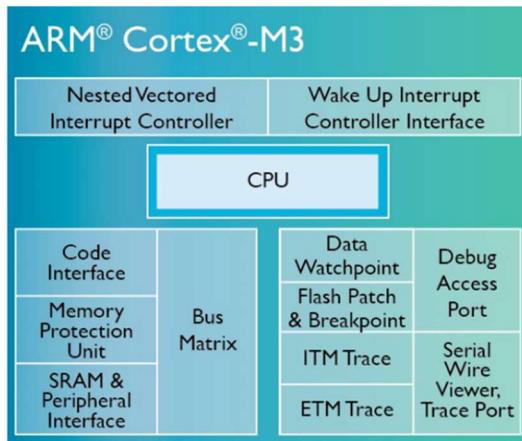
Parliamo quindi di **DATA PROCESSING** di un'architettura arm:

- Istruzioni **reg-reg** (da registro a registro):
 - 2 operandi vengono letti dai registri Rn e Rm
 - 1 operando potrebbe essere "ruotato" (0x00000A→0x000A00)
 - la ALU genera il risultato
 - il risultato viene scritto nel registro Rd

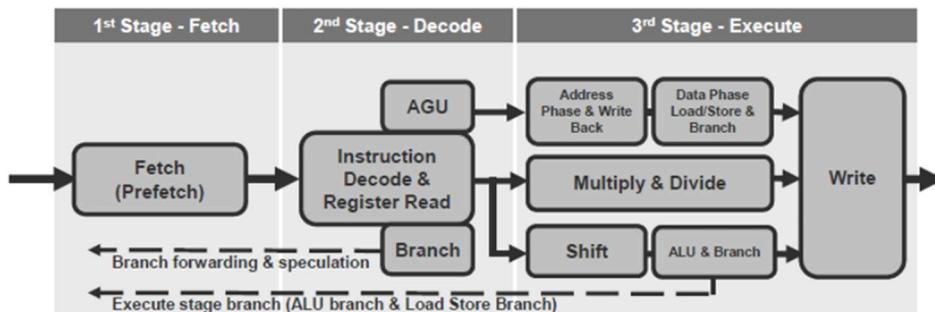


- la prossima istruzione viene recuperata dalla memoria
 - il PC viene aggiornato
- Istruzioni reg-imm (registro-immediato):
- 1 operando viene letto dal registro Rn, l'altro è immediato
 - 1 operando potrebbe essere "ruotato"
 - la ALU genera il risultato
 - il risultato viene scritto nel registro Rd
 - la prossima istruzione viene recuperata dalla memoria
 - il PC viene aggiornato
- Data transfer functions → richiedono 2 cicli nella Execute: nel 1° viene calcolato l'indirizzo usando un registro e un immediato, nel 2° avviene un accesso in memoria nel quale viene trascritto il dato
- Branch instructions → prima viene calcolato l'indirizzo target, aggiungendo un immediato (shiftato di 2 posizioni) al PC, poi la pipeline viene svuotata e riempita nuovamente
- ⚠ Nelle Branch & Link instructions, è necessario 1 clock aggiuntivo in quanto bisogna salvare l'indirizzo di ritorno in R14

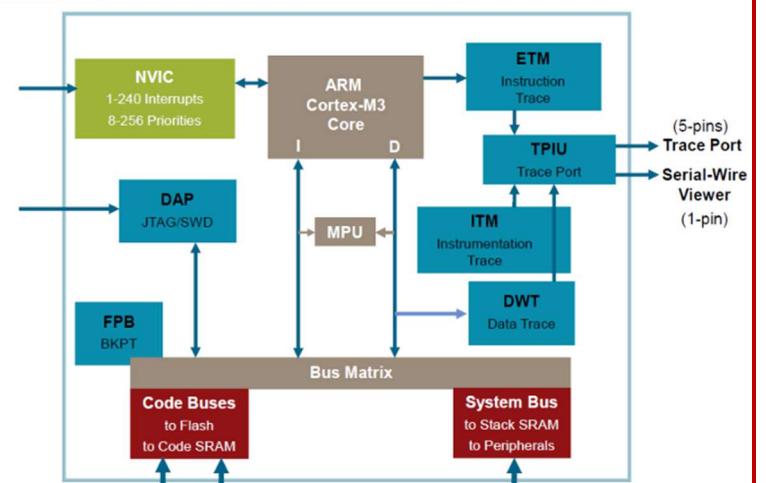
Parliamo ora del processore che useremo in questo corso, ovvero **ARM Cortex M3**, la cui architettura è:



Vediamo quindi che il Cortex M3 non è solo la CPU, ma è circondato da tutti gli elementi nella figura sopra; nella figura sopra a dx vediamo il suo **DATAPATH**. La **pipeline** del Cortex M3 ha 3 stadi:



I salti richiedono **3 cicli** per essere completati. Sono però un problema perché, nel caso peggiore (cioè **salto indiretto preso**), la **pipeline viene sempre svuotata e riempita**. Inoltre **non è supportato il delayed branch**. Con una **read dalla memoria**, perdiamo 1 ciclo di clock (si ha 1 sola porta WB [write-back] nel register file).



⚠ A dx lo schema a blocchi dell'architettura del Cortex M3 con i moduli di debug.

Vediamo il Cortex M3 **dal punto di vista del programmatore**; sono messi a disposizione:

- **18 registri a 32 bit** (**r13 = sp** [stack pointer]; **r14 = lr** [link register, usato nella branch&link]; **r15 = pc** [program counter]) [è anche presente una replica dell'sp (PSP) per facilitare la gestione degli interrupt]
- handling (gestione) efficiente delle **interruzioni**
- power management enable **idle mode** (supporta sleep now, sleep on exit e deep sleep)
- **debug efficiente** e features per il supporto allo sviluppo
- supporto del sistema (**user/supervisor**)
- completamente **programmabile in C**

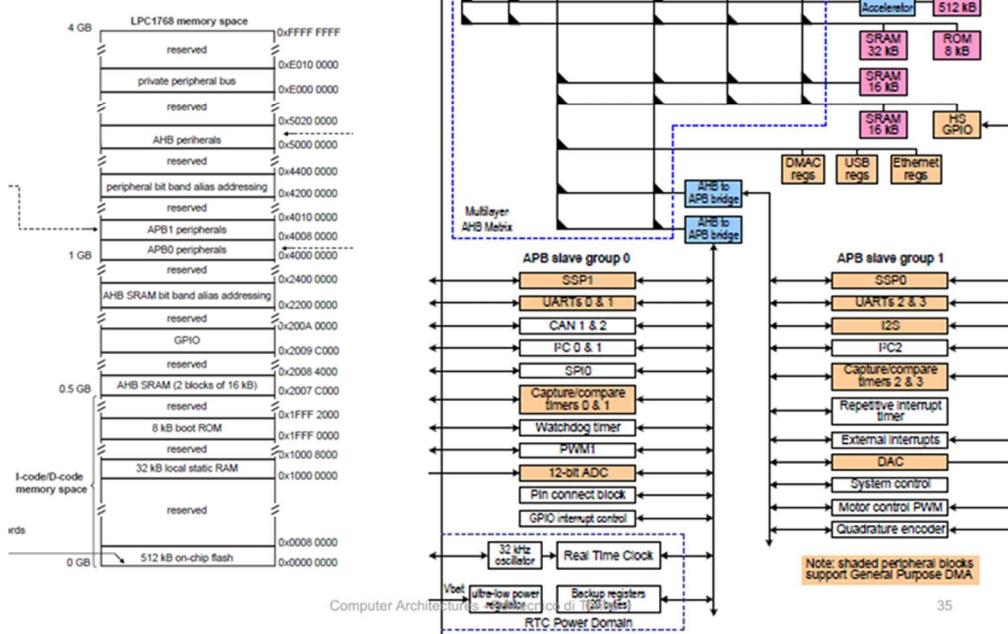
Parliamo ora del **BUS** del Cortex M3, ovvero **AMBA** bus system che prevede 3 bus:

- **AHB** (Advanced High Performance Bus)
- **ASB** (Advanced System Bus)
- **APB** (Advanced Peripheral Bus)

⚠ I sistemi arm hanno 2 clock: 1 alta frequenza (per CPU e componenti ad alta velocità) e 1 bassa frequenza (periferici)

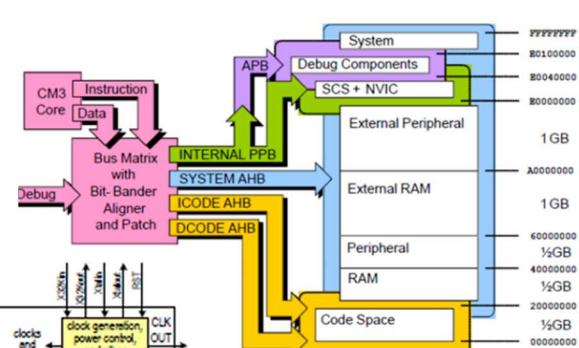
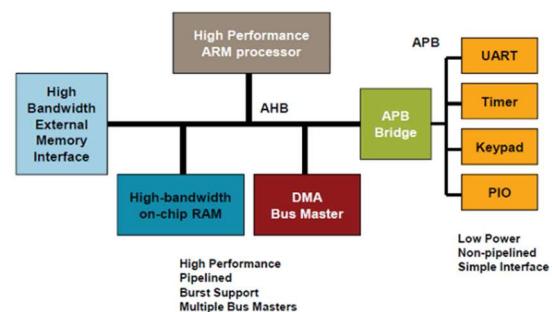
La **MEMORIA** è mappata in **4gb** (2^{32} perché ho 32 bit di indirizzi) e la bus matrix viene acceduta con i bus AHB e PPB (Private Peripheral Bus) [da cui parte l'APB] [non tutti i 4gb sono usati]

NXP LPC176x/5x block diagram and memory map



Le **ECCEZIONI** sono: Reset, NMI, Faults (hard fault, memory manage, bus fault, usage fault), SVCall, Debug Monitor, PendSV, SysTick Interrupt e External Interrupt. È presente la **IVT** (Interrupt Vector Table). Riguardo agli **INTERRUPT**, è supportata un'interruzione non mascherabile INTNMI; inoltre, insieme al processore troviamo il **NVIC** (Nested Vectored Interrupt Controller) [supporta fino a 240 interruzioni esterne].

⚠ Il Cortex M3 supporta il clock in tutte le modalità (anche da fonte esterna). È presente un WIC (Wake-up Interrupt Controller), cioè una fonte esterna di wake-up che permette al processore di spegnersi completamente.



Vediamo le **ISTRUZIONI** del Cortex M3: le istruzioni sono a **32 bit** (o **16** se Thumb) e possono essere eseguite in **modo condizionale**. C'è un'architettura di load/store (**la processazione di dati avviene solo su registri**) [l'accesso alla memoria avviene con istruzioni dotate di auto-indexing]. Il formato usato prevede **3 operandi** e combina ALU + shifter. Ci sono **18 registri a 32 bit** (supportano **byte** [8 bit], **halfword** [16 bit] e **word** [32 bit]).

Come detto sopra il **PC** è il registro **r15** [quindi posso avere scrittura diretta sul PC]; quando il processore è eseguito in "**ARM state**" (e non Thumb), tutte le istruzioni sono lunghe 32 bit e sono allineate a word: pertanto, il PC value è salvato nei bit [31:2] con i bit [1:0] pari a 0.

Il **link register** (**LR**) è il registro **r14** (usato per memorizzare l'indirizzo di ritorno di un **branch&link**, calcolato a partire dal PC) [per tornare da un branch&link, basta fare **MOV r15, r14** oppure **MOV pc, lr**].

Lo **stack pointer** (**SP**) è il registro **r13** e viene aggiornato **automaticamente** (cioè al boot viene preso dalla IVT oppure si aggiorna quando il programma esegue una istruzione "*stack-oriented*").

Altro elemento importante è il **PSR** (**Program Status Register**), che può essere acceduto da solo o come combinazione di 3 registri:

- **APSR (Application PSR)** = contiene i flag: **N** (risultato negativo dalla ALU), **Z** (risultato zero dalla ALU), **C** (Carry dell'operazione ALU), **V** (overflow dell'operazione ALU), **GE** ($>=$), **Q** (sticky/saturation)
- **EPSR (Execution PSR)** = contiene: **IT** (IF-THEN instruction status), **ICI** (Interrupt-Continuable Instruction) e **T** (Thumb; se T = 1, il processore interpreta il codice fetchato come una sequenza di Thumb operations, mentre se T = 0, normali istruzioni) [Thumb operations = operazioni a 16 bit, meno potenti e in n° minore]
- **IPSR (Interrupt PSR)** = contiene un "**exception number**" usato nella gestione delle eccezioni

Oltre alle branch, in ARM **tutte le istruzioni hanno un campo condizionale** [31:28] che determina se la CPU le eseguirà o meno; le istruzioni non eseguite richiedono 1 ciclo (richiedono di completare il ciclo per consentire il corretto fetch e decode delle prossime istruzioni). Questo permette di **togliere i branch** che stallano la pipeline. La **condizione si aggiunge come suffisso** all'istruzione che sto per scrivere e i suffissi possono essere:

Codice	Significato	LO	Unsigned lower (identical to CC)	LS	Unsigned lower or same
EQ	Equal	MI	Minus or negative result	GE	Signed greater than or equal
NE	Not equal	PL	Positive or zero result	LT	Signed less than
CS	Carry set (identical to HS), il carry vale 1	VS	Overflow	GT	Signed greater than
HS	Unsigned higher or same (identical to CS)	VC	No overflow	LE	Signed less than or equal
CC	Carry clear (identico a LO), il carry vale 0	HI	Unsigned higher	AL	Always (this is the default)

Un esempio:

- If R4-R3 == 0 then R0 = R1
- Else R0 = R2

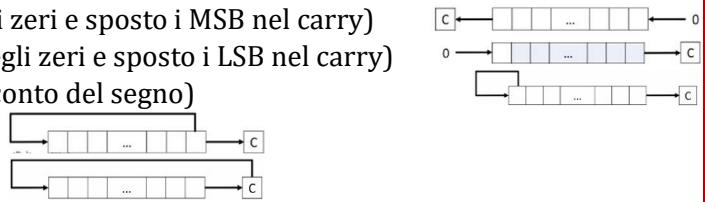
SUBS	R4, R4, R3
MOVEQ	R0, R1
MOVNE	R0, R2

⚠ Il conditional flag "**S**" setta il flag condizione == 0 sul registro destinazione dell'operazione segnata con S (è lo "start" delle operazioni seguenti che si basano sul confronto dell'istruzione con la S) [**quindi i flag vengono modificati solo se il suffisso "S" è messo all'istruzione**]

Ora vediamo le **OPERAZIONI/ISTRUZIONI** di assembly su arm:

- **CMP r0, #10 → compare** = sottrae #10 al registro r0 e aggiorna i flag [es. se ho r0 = #12, se faccio CMP r0, #12 ottengo N=0, Z=1, C=1, V=0]
- **CMN r0, #10 → compare negative** = aggiunge #10 al registro r0 e aggiorna i flag
- **TST r1, r2, LSL#4 → test** = calcola l'AND logico tra r1, r2; con LSL#4 mi chiedo se il 4° bit di r1 è settato; aggiorna tutti i flag eccetto V
- **TEQ r2, r3 → test equivalence** = calcola l'EOR logico tra r2 e r3 (uguaglianza); aggiorna tutti i flag eccetto V
- **MRS Rn, SReg →** copia un registro speciale (SReg, che può essere APSR, EPSR, IPSR o PSR) in un registro general purpose
- **MSR SReg, Rn →** copia un registro general purpose in un registro speciale

- **ADD Rd, Rn, op2** → $Rd = Rn + op2$ ($op2$ può essere come sempre un **registro shiftato**, una **costante** da 8 bit [per l'ADD e la SUB anche da 12 bit])
- **ADC Rd, Rn, op2** → $Rd = Rn + op2 + C$ (ovvero il carry) [si possono sommare valori su **64-bit**, ovvero se faccio ADDS r4,r0,r2 e dopo ADC r5,r1,r3 avrà **r5,r4 = r1,r0 + r3,r2** (le virgole indicano che ho i 32 **bit alti** nel registro prima della virgola e i 32 **bit bassi** nel registro dopo la virgola)]
- **ADDD Rd, Rn, op2** → è come la ADD, ma prende solo valori da **12-bit** e non aggiorna i flag
- **SUB Rd, Rn, op2** → $Rd = Rn - op2$
- **SBC Rd, Rn, op2** → $Rd = Rn - op2 + C - 1$ [si possono sottrarre valori su **64-bit**, ovvero se faccio SUBS r4,r0,r2 e dopo SBC r5,r1,r3 avrà **r5,r4 = r1,r0 - r3,r2** (le virgole indicano che ho i 32 **bit alti** nel registro prima della virgola e i 32 **bit bassi** nel registro dopo la virgola)]
- **SUBW Rd, Rn, op2** → è come la SUB, ma prende solo valori da **12-bit** e non aggiorna i flag
- **RSB Rd, Rn, op2** → è la **reverse subtraction**, cioè $Rd = op2 - Rn + C - 1$ (utile per shiftare l'altro operando prima della sub)
- **MUL Rd, Rn, Rm** → $Rd = Rn * Rm$ con risultato su **32 bit** (sia signed sia unsigned, no differenze)
- **UMULL Rd1, Rd2, Rn, Rm** → $Rd1, Rd2 = Rn * Rm$ con risultato **unsigned su 64 bit** (bit alti e bit bassi)
- **SMULL Rd1, Rd2, Rn, Rm** → $Rd1, Rd2 = Rn * Rm$ con risultato **signed su 64 bit** (bit alti e bit bassi)
- ⚠️ Tutti gli operandi devono essere registri nelle moltiplicazioni.**
- **MLA Rd, Rn, Rm, Ra** → $Rd = Rn * Rm + Ra$ (**multiplication with accumulation**)
- **MLS Rd, Rn, Rm, Ra** → $Rd = Rn * Rm - Ra$
- **UMLAL Rd1, Rd2, Rn, Rm** → $Rd1, Rd2 = Rn * Rm + Rd1, Rd2$ (unsigned)
- **SMLAL Rd1, Rd2, Rn, Rm** → $Rd1, Rd2 = Rn * Rm + Rd1, Rd2$ (signed)
- **UDIV Rd, Rn, Rm** → $Rd = Rn / Rm$ (unsigned)
- **SDIV Rd, Rn, Rm** → $Rd = Rn / Rm$ (signed)
- ⚠️ Se Rn non è esattamente divisibile per Rm, il risultato è arrotondato allo zero; inoltre non si può usare il suffisso "S" in UDIV e SDIV (ovvero queste non cambiano i flag)**
- **AND Rd, Rn, op2** → $Rd = Rn \text{ AND } op2$
- **BIC Rd, Rn, op2** → $Rd = Rn \text{ AND NOT } op2$
- **ORR Rd, Rn, op2** → $Rd = Rn \text{ OR } op2$
- **EOR Rd, Rn, op2** → $Rd = Rn \text{ XOR } op2$
- **ORN Rd, Rn, op2** → $Rd = Rn \text{ OR NOT } op2$
- **MVN Rd, Rn** → $Rd = \text{NOT } Rn$
- **LSL Rd, Rn, op2** → **shift left** (inserisco nel LSB degli zeri e sposto i MSB nel carry)
- **LSR Rd, Rn, op2** → **shift right** (inserisco nel MSB degli zeri e sposto i LSB nel carry)
- **ASR Rd, Rn, op2** → **arithmetic shift right** (tengono conto del segno)
- **ROR Rd, Rn, op2** → **rotate right**
- **RRX Rd, Rn** → **rotate right with extend**



Vediamo le **ISTRUZIONI DI BRANCH**:

- **B label** → unconditional branch (**jump**)
- **BX r0** → unconditional branch **indirect** (indirizzo del salto nel registro r0)
- **BL label** → unconditional branch & link (**jump and link**)
- **BLX r0** → unconditional branch & link **indirect**

⚠️ BL and BLX salvano l'indirizzo di ritorno (cioè l'indirizzo della prossima istruzione) in LR(r14) e sono usate per le subroutines.

- ⚠️ Un programma standalone non può continuare all'infinito senza l'uso dell'OS; un infinite loop è aggiunto come ultima istruzione con **stop B stop** oppure con la sequenza di istruzioni **LDR r1, =stop; stop BX r1****
- **LDR r1, =label** → crea una costante in un **literal pool** e usa un PC per prendere i dati dall'etichetta
- **ADR r1, label** → rispetto alla LDR, aggiunge/toglie un offset al PC [**⚠️ ADR non aumenta la dimensione del codice, ma non può creare tutti gli offset]**

⚠️ BX richiede che l'ultimo bit del registro sia 1 (altrimenti "usage fault exception") in quanto salta all'indirizzo cambiando l'ultimo bit del registro a 0 [quindi prima deve essere a 1]; si può usare quindi dopo una **LDR** (perché setta l'ultimo bit del registro a 1 se la label è nel code area, a 0 se è nel data area) oppure dopo una **ADR + ORR rd,rd,#1** (questo perché ADR/ADRL non cambia l'ultimo bit a 1, quindi con l'OR logico lo metto a mano a 1)

⚠ Nella B, l'opcode è 8 e l'immediato è su 24 bit; dato che gli indirizzi sono "halfword-aligned", l'immediato specifica il valore dei bit da 24 a 1 dell'indirizzo; il 25° bit è per il segno, quindi l'indirizzo relativo è di $\pm 2^{24}$ byte = $\pm 16\text{MB}$. La BX invece può saltare ad un valore su 32 bit = **4GB**

⚠ Un salto può essere fatto anche cambiando il valore di PC con [LDR rd, =label; MOV PC, rd; LDR PC, =label] ma è **sconsigliato** (MOV+LDR mettono l'ultimo bit del PC a 0)

- Tutti i **SALTI CONDITIONAL** si fanno con **B?? e BX??** (dove al posto di ?? si mettono i suffissi in tabella):

??	Flags	Meaning	??	Flags	Meaning
EQ	Z = 1	equal	NE	Z = 0	not equal
CS HS	C = 1	unsigned \geq	CC LO	C = 0	unsigned <
MI	N = 1	negative	PL	N = 0	positive or 0
VS	V = 1	overflow	VC	V = 0	no overflow
HI	C = 1 & Z = 0	unsigned >	LS	C = 0 & Z = 1	unsigned \leq
GE	N \geq V	signed \geq	LT	N \neq V	signed <
GT	Z = 0 or N = V	signed >	LE	Z = 1 or N \neq V	signed \leq

- **CBZ r1, label** → compare and branch if r1 == 0 (i registri usabili sono solo r0-r7)
- **CBNZ r1, label** → compare and branch if r1 != 0 (i registri usabili sono solo r0-r7)

⚠ Saltano però solo in avanti CBZ e CBNZ

⚠ La differenza tra usare CBZ e CMP+BEQ (e analogamente CBNZ e CMP+BNE) è che:

- o CMP setta i flag, mentre CBZ e CBNZ non li setta
- o CBZ e CBNZ possono saltare solo in avanti e non possono essere usati in un blocco IT

⚠ Un **ciclo while** si può fare così:

```
while (r0 != N) {
    ...
    //do something
}
```

```
test CMP r0, #N
BE exit
...
; do something
B test
exit
```

```
loop CBZ r0, exit
...
; do something
B loop
exit ...
```

⚠ Un **ciclo for** si può fare così:

```
for (i = 0; i < N; i++) {
    ...
    //do something
}
```

```
MOV r0, #0
loop CMP r0, #N
BHS exit
...
; do something
ADD r0, r0, #1
B loop
exit
```

```
MOV r0, N
loop ...
; do something
SUBS r0, r0, #1
BNE loop
exit
```

Per quanto riguarda **if-then** c'è l'istruzione **ITE... <condizione>** dove **IT** è il blocco di base e viene aggiunta una "**E**" per ogni else-if del blocco condizionale (max 4); la **<condizione>** è quella dell'if considerata come "vera" e quindi farà match con la prima istruzione; esempio:

MOV r0, #N	ITxyz <cond>
MOV r1, #M	instr1<cond> <operands>
CMP r0, r1	instr2<cond OR not cond> <operands>
ITE GE	instr3<cond OR not cond> <operands>
SUBGE r0, r0, r1	instr4<cond OR not cond> <operands>
SUBLT r1, r1, r0	
exit ... ; program continues	

⚠ Non si può branchare alle istruzioni di un blocco IT; solo l'ultima istruzione del blocco IT può essere una branch.

Vediamo le **DIRETTIVE**:

- **AREA** → definisce un **blocco di codice o dati** [è importante definire le sezioni di data e codice] [almeno 1 direttiva AREA è obbligatoria]; la sintassi è **AREA sectionName {, attributo, attributo...}** dove

sectionName è il nome associato a quell'area [se inizia con numero va messo in "||", ovvero "|1_DataArea|"], mentre gli **attributi** specificano delle proprietà (es. CODE, DATA, READONLY, READWRITE, ALIGN = expr)

- **RN** → si può usare per associare un nome ad un registro (**REGISTER RENAMING**) [es. registro1 RN 1, dove registro1 è il nome associato al registro e 1 indica l'indice del registro, ovvero r1]
- **EQU** → associa un **simbolo** ad una **costante numerica**
 - ⚠️ I numeri sono in decimale (2), esadecimale (0x3F) o altre basi (n_xxx con n = base, xxx = valore nella base)
- **ENTRY** → definisce un **entry point** al programma
- **DC?** → allocare costante in memoria (assegnandole un valore [**initial runtime contents**, ovvero il valore a cui sono poste queste aree di memoria all'inizio dell'esecuzione del programma]); **DCB** = definisce costante di tipo Byte, **DCW** = costante half-word, **DCWU** = costante half-word unaligned, **DCD** = costante word, **DCDU** = costante word unaligned. La sintassi è **label DC? Valore**, dove **label** = spazio di memoria a cui viene data quell'etichetta, **DC?** = direttiva usata, **valore** = valore dato in quello spazio di memoria. **Esempio:**

```
myData DCB 65, 0x73, 8_163
        DCB "embl"
```

Address	Value	Octal	Hex	ASCII
0x000000D2	65	101	41	A
0x000000D3	115	163	73	s
0x000000D4	115	163	73	s
0x000000D5	101	145	65	e
0x000000D6	109	155	6D	m
0x000000D7	98	142	62	b
0x000000D8	108	154	6C	l
0x000000D9	121	171	79	y

```
myData DCB 65, 0x73, 8_163
        DCW 0x626D, 0x796C
```

Address	Value	Octal	Hex	ASCII
0x000000D2	65	101	41	A
0x000000D3	115	163	73	s
0x000000D4	115	163	73	s
0x000000D5	0	0	0	NUL
0x000000D6	109	155	6D	m
0x000000D7	98	142	62	b
0x000000D8	108	154	6C	l
0x000000D9	121	171	79	y

```
myData DCB 65, 0x73, 8_163
        DCDU 0x796C626D
```

Address	Value	Octal	Hex	ASCII
0x000000D2	65	101	41	A
0x000000D3	115	163	73	s
0x000000D4	115	163	73	s
0x000000D5	109	155	6D	m
0x000000D6	98	142	62	b
0x000000D7	108	154	6C	l
0x000000D8	121	171	79	y

⚠️ Nelle **unaligned** non ci sono spazi NULL, mentre in quelle aligned (ovvero quelle senza U) troviamo dei NULL tra le costanti in modo che l'ultimo elemento messo crei un elemento dell'elemento che devo mettere (es. **se ho messo 3 Byte e poi devo mettere 1 Word devo mettere 3 Byte NULL in modo che l'ultimo elemento, ovvero il Byte, + 3 Byte NULL = 1 Word**)

- **ALIGN** → allinea data o codice ad un **memory boundary**, facendo un padding di zero; la sintassi è **ALIGN {expr, offset}** e la current location è allineata al next address nella forma **n*expr+offset** (ovvero si parte dalla riga con il primo multiplo di expr e ci si sposta di offset). Se expr non è specificata, la ALIGN setta la current location al next word boundary. **Esempio:**

```
myData DCB 65
        ALIGN 2
        DCB 115
```

Address	Value	Octal	Hex	ASCII
0x000000D4	65	101	41	A
0x000000D5	0	0	0	NUL
0x000000D6	115	163	73	s

```
myData DCB 65
        ALIGN 4
        DCB 115
```

Address	Value	Octal	Hex	ASCII
0x000000D4	65	101	41	A
0x000000D5	0	0	0	NUL
0x000000D6	0	0	0	NUL
0x000000D7	0	0	0	NUL
0x000000D8	115	163	73	s

```
myData DCB 65
        ALIGN 4, 3
        DCB 115
```

Address	Value	Octal	Hex	ASCII
0x000000D4	65	101	41	A
0x000000D5	0	0	0	NUL
0x000000D6	0	0	0	NUL
0x000000D7	115	163	73	s

- **SPACE** → riserva un **blocco di memoria** (messo **tutto a zero**) di una certa dimensione. La sintassi è **{label} SPACE expr** (es. long_var SPACE 8)
- **LTORG** → assegna uno **starting point di un literal pool**
- **END** → segna la **fine** del codice sorgente

⚠️ Le operazioni vengono modificate dal compilatore di ARM quando eseguiamo il codice assembly.

8) MEMORY, COSTANTS & LITERAL POOLS

Se abbiamo una cache si possono usare diverse **politiche** per accedere alle diverse "memorie" (Code, SRAM, periferiche...) [es. Code → bufferable, cacheable con write through, executable; SRAM → bufferable, cacheable con write back, executable...]. Vediamo le varie **LOAD/STORE** con struttura LDR rd, addressing_mode:

Load	Store	Size and type
LDR	STR	word (32 bits)
LDRB	STRB	byte (8 bits)
LDRH	STRH	halfword (16 bits)
LDRSB	-	signed byte
LDRSH	-	signed halfword
LDRD	STRD	two words
LDM	STM	multiple words

Vediamo le singole istruzioni nel dettaglio:

- **LDRD r1, r2, [r0]** → carica 2 word prese dall'indirizzo r0 in avanti (es. se r0 = 0x00008000 prendo i byte da 0x00008000 a 0x00008007 [8 byte = 2 word]) nei registri r1 e r2
- **STR r1, [r0]** → copia il contenuto di r1 in 4 blocchi di memoria (4 byte = 1 word) partendo dall'indirizzo r0 [vengono messi dal LSB al MSB, quindi byte più a dx di r1 messo in r0 e byte più a sx di r1 messo in r0+3]
- **STRB r1, [r0]** → copia l'LSB di r1 nell'indirizzo r0 (es. se r1 = 0x65737341, metterò all'indirizzo r0 0x41)
- **STRH r1, [r0]** → copia i 16 bit bassi di r1 (half-word) all'indirizzo r0 (es. se r1 = 0x65737341, metterò all'indirizzo r0 0x41 e all'indirizzo r0+1 0x73)
- **STRD r1, r2, [r0]** → copia il contenuto di 2 registri in 8 zone di memoria consecutive (2 word = 8 byte) [come la LDRD ma al contrario]

Parliamo di **ADDRESSING MODE**:

- **Pre-indexed** = indirizzo ottenuto sommando l'indirizzo in rn all'offset con sintassi **load/store rd, [rn, offset] {!}**; l'offset è una costante su 12-bit o un registro (che può essere shiftato a sx fino a 3 posizioni); il "!" indica se rn è aggiornato alla fine dell'istruzione (aggiungendo a rn l'offset costante), ovvero se avviene **WriteBack**

LDR r2, [r0]
LDR r3, [r0, #4]
LDR r4, [r0, #8]
LDR r5, [r0, #12]

At the end, r0 = 0x00008000

LDR r2, [r0]
LDR r3, [r0, #4]!
LDR r4, [r0, #4]!
LDR r5, [r0, #4]!

At the end, r0 = 0x0000800C

- **Post-indexed** = indirizzo ottenuto dall'indirizzo in rn con sintassi **load/store rd, [rn], offset**; rn è dopo aggiornato aggiungendo l'offset (uguale a prima) [non c'è "!" perché rn è sempre aggiornato]

```
AREA    .text!, CODE, READONLY
Reset_Handler    PROC
    EXPORT Reset_Handler [WEAK]
    MOV r2, #8 ;after some calculus
    LDR r0, =lookup
    LDR r4, [r0, r2, LSL #2]
    stop B stop      ;stop program
    lookup DCD 1, 1, 2, 6, 24, 120, 720,
    5040, 40320, 362880, 3628800
    ENDP
```

⚠ Si può usare una **look-up table** (array di costanti precalcolate); utile per valori molto usati (non devo farne sempre il calcolo runtime), ma richiede spazio in memoria. **Esempio** (programma che usa x valori contenuti in r2 e mette in r4 il risultato del fattoriale di x [con x tra 0 e 10]) **di LUT di word** →

Passiamo ora alle **COSTANTI** e ai **LITERAL POOL**. La **MOV** assegna un valore (contenuto di un registro o costante) ad un registro [non può assegnare in un registro un programma o un indirizzo dati (es. MOV r0, myData oppure MOV r1, myCode)]; può avere sintassi:

- **MOV rd, rm {, shift}** → copia rm in rd. Lo shift è opzionale e viene applicato a rm; può essere ASR #n, LSL #n, LSR #n, ROR #n, RRX (si preferisce però lo shift corrispondente piuttosto che la mov con lo shift)
- **MOV rd, #costante** → copia in rd il valore (può essere una costante su 16 bit, un valore ottenuto shiftando a sx una costante 8-bit oppure una forma tra 0x00AB00AB, 0xAB00AB00, 0xABABABAB)

⚠ **MOVW** è come **MOV** ma prende solo costanti su 16-bit

- **MVN r0, #costante** → copia il complemento di un operando in un registro (es. MVN r0, #0 mette 0xFFFFFFFF in r0) [non prende valori su 16-bit] [la usa il compilatore: per esempio se ho MOV r0, #-2, diventa MVN r0, #1 perché -2 non è nel range della MOV]
 - **MOVT r0, #costante** → copia un valore su 16-bit nei 16 bit alti di un registro (quindi se voglio mettere un valore su 32-bit in un registro uso MOVT per i bit alti e MOV per i bit bassi)

⚠ Si può usare LDR rd, =costante:

- se la costante è nel range della MOV, l'istruzione viene rimpiazzata con `MOV rd, #costante`
 - se la costante non è nel range della MOV, `LDR rd, [PC, #offset]`; si crea quindi un **literal pool** (blocco di costante all'indirizzo PC+offset)

⚠ L'offset è su 12 bit e si calcola come la differenza tra l'indirizzo del literal pool e il PC (il valore di PC si calcola come l'indirizzo dell'istruzione + 4, azzerando il 2° bit per word alignment). Esempio:

LDR r0, =0xC90147D2	0x00000118
...	...
0x47D2	0x00000144
0xC901	0x00000146

- ```
1. 0x118 = 2_000100011000
2. 0x118 + 4 = 0x11C = 2_000100011100
3. PC = 2_000100011100 = 0x11C
4. offset = 0x144 - 0x11C = 0x28 = 40
```

LDR r0, [PC, #40]

Il **literal pool** si trova di default nella direttiva END (dopo l'ultima istruzione). Dato che l'offset è su 12 bit, la distanza tra l'istruzione corrente e l'ultima deve essere < 4096 (altrimenti bisogna mettere il literal pool da qualche altra parte con la LTORG). Esempio:

```
AREA .text, CODE, READONLY
Reset_Handler PROC
 EXPORT Reset_Handler [WEAK]
 LDR r0, =0xC90147D2
stop B stop
myEmptySpace SPACE 4100
ENDP
END ;literal pool is saved here
```

```
AREA .text!, CODE, READONLY

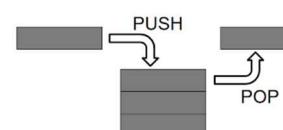
Reset_Handler PROC
 EXPORT Reset_Handler [WEAK]
 LDR r0, =0xC90147D2
 B stop
 LTORG ;literal pool is saved here
stop B stop
myEmptySpace SPACE 4100
ENDP
END
```



⚠ Si può usare **ADR** che viene rimpiazzata con **LDR rd, PC, #offset** (se l'offset è > 4095 Byte, si usa **ADRL** al posto di ADR [ADRL genera 2 operazioni e l'offset può essere fino a 1MB])

## **9) STACK & SUBROUTINES**

Lo **stack** ha un'organizzazione **LIFO**. I dati sono inseriti (scritti) ed estratti (letti) dal top della pila. Lo **stack pointer** (**r13, sp**) contiene l'indirizzo del top dello stack (opposto dell'heap). L'sp viene aggiornato dopo ogni push e può essere:

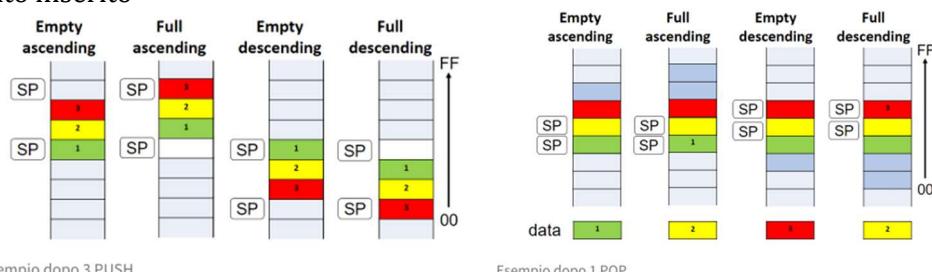


- **descendente** = indirizzo del top della pila diminuisce dopo ogni push
  - **ascendente** = indirizzo del top della pila aumenta dopo ogni push

Mentre il contenuto del top dello stack può essere:

- **empty** = sp punta all'entry dove il nuovo dato dovrà essere inserito (1° spazio libero)
  - **full** = sp punta all'ultimo elemento inserito

**⚠ Nostro è full descending.**



Le istruzioni LDM e STM sono usate per caricare e salvare più registri insieme e hanno sintassi **LDM{xx}/STM{xx}**

#### Rn {!}, regList:

- **Rn** = base register
- **xx** = metodo di indirizzamento (come e quando viene aggiornato Rn durante l'istruzione); sono 2:
  - o **IA (Increment After [default])**:
    - 1) L'indirizzo a cui eseguire l'accesso è nel base register
    - 2) Il base register è incrementato di 1 word (4 Byte)
    - 3) Se non ci sono registri nella lista si torna al punto 1)
  - o **DB (Decrement Before)**:
    - 1) Il base register è decrementato di 1 word (4 Byte)
    - 2) L'indirizzo a cui eseguire l'accesso in memoria è nel base register
    - 3) Se non ci sono registri nella lista si torna al punto 1)
- **!** = Rn viene settato al nuovo valore (senza viene impostato al valore iniziale)
- **regList** = lista di registri da caricare/salvare; possono essere consecutivi (r0-r4) o non consecutivi (r10, 1r) o mixati (es. r0-r4, r10, 1r) [sp non può apparire nella regList, mentre pc può esserci solo con LDR e in assenza di lr] [non importante l'ordine dei registri, vengono ordinati in modo crescente; r8, r1, r3-r5, r14 → r1, r3, r4, r5, r8, r14]

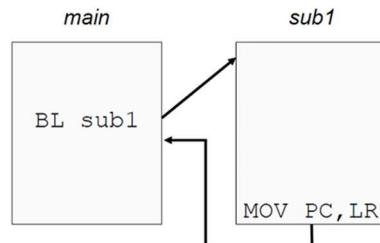
Riassunto delle varie istruzioni:

| Stack type      | PUSH                  | POP                   |
|-----------------|-----------------------|-----------------------|
| Full descending | STMDB<br>STMFD        | LDM<br>LDMIA<br>LDMFD |
| Empty ascending | STM<br>STMIA<br>STMIA | LDMDB<br>LDMEA        |

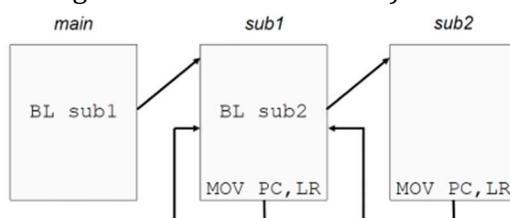
⚠ L'utilizzo di PUSH e POP facilitano l'utilizzo di un full descending stack:

- **PUSH regList** = alias per **STMDB sp!, regList**
- **POP regList** = alias per **LDMIA sp!, regList**

Una **SUBROUTINE** viene chiamata con **BL label** e **BLX Rn**, viene scritto l'indirizzo a cui tornare dalla subroutine in **1r** (una procedura “reentrant” finisce con un branch all'indirizzo salvato in **1r**) e il valore di **label** o **Rn** in **pc**. Una subroutine inizia e termina facoltativamente con le direttive **PROC/FUNCTION** e **ENDP/ENDFUNC**. Vediamo la chiamata ad una subroutine:



Ci sono problemi nel caso di **chiamate annidate**, in quanto il valore in **1r** viene sovrascritto non rendendo possibile tornare da **sub1** al **main** (per questo bisogna salvare **1r** nello stack):



Tutte le subroutines dovrebbero quindi **salvare 1r e gli altri registri usati nello stack come 1^ istruzione della subroutine e ripristinarli come ultima istruzione**:

**PUSH {regList, 1r}**

...

**POP {regList, pc}**

Si possono **passare dei parametri alle funzioni/subroutines** con 3 approcci:

- tramite **registro** (caso peggiore, sconsigliato)

- tramite **reference** (es. indirizzo salvato in un registro, "puntatore"):

```

1 MOV r0, #0x34
2 MOV r1, #0xA3
3 LDR r3, =mySpace
4 STMIA r3, {r0, r1}
5 BL sub2
6 LDR r2, [r3]
7 ; r2 contains the result
9 sub2 PROC
10 PUSH {r2, r4, r5, LR}
11 LDMIA r3, {r4, r5}
12 CMP r4, r5
13 SUBHS r2, r4, r5
14 SUBL0 r2, r5, r4
15 STR r2, [r3]
16 POP {r2, r4, r5, PC}
17 ENDP

```

- tramite **stack**:

```

1 MOV r0, #0x34
2 MOV r1, #0xA3
3 PUSH {r0, r1, r2}
4 BL sub3
5 POP {r0, r1, r2}
6 ; r2 contains the result
7 ...
8 stop B stop
9 ...
10
11 sub3 PROC
12 PUSH {r6, r4, r5, LR}
13 LDR r4, [sp, #16]
14 LDR r5, [sp, #20]
15 CMP r4, r5
16 SUBHS r6, r4, r5
17 SUBL0 r6, r5, r4
18 STR r6, [sp, #24]
19 POP {r6, r4, r5, PC}
20 ENDP

```

Un **ABI** (Application Binary Interface) è un'interfaccia che si pone tra 2 program binary modules (spesso una libreria e un programma eseguito da un utente). Le **calling convention** di un ABI determinano come i dati sono trasmessi in input o letti in output dalle computational routines. I primi 4 registri **r0-r3 (a1-a4)** sono usati come argomenti per le subroutines e il valore di ritorno di una funzione. **Una subroutine deve preservare il contenuto dei registri r4-r8, r10, r11 e sp.** Vediamo le calling conventions:

| 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<br>SB<br>TR | Platform register.<br>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.                                                 |

(\*) Lo **standard** base prevede di **passare gli argomenti nei core registers (r0-r3) e nello stack** (per le subroutine che richiedono pochi parametri sono usati solo i registri r0-r3, riducendo overhead delle chiamate). Come già detto lo stack è full-descending con il current extend dello stack nel registro sp (r13).

⚠ Si possono creare **variabili locali nello stack** nello stesso modo in cui salviamo i dati (ovvero sottraendo il numero di byte richiesti da ciascuna variabile dallo sp).

## 10) SUPERVISOR CALLS (SVC)

Distinguiamo:

- **ECCEZIONE** = evento **interno** della CPU (es. floating point, overflow, MMU fault, trap) [quindi SW]
- **INTERRUPT** = evento **esterno** di I/O (es. I/O device request e reset) [quindi HW]

Nelle architetture ARM, le istruzioni che rilasciano **interruzioni SW** sono le **SVC** (in ARM non ci sono istruzioni per la gestione di numeri reali).

La **IVT** (Interrupt Vector Table) è una tabella che permette di determinare quali procedure gestiscono ciascuna eccezione [a dx image].

Una **UDF** (Undefined instruction) è un'istruzione non definita (che potrebbe essere voluta per triggerare uno **Usage Fault**).

Una **NMI** è un non-maskable interrupt. **SySTick** viene rilasciata dal timer di sistema quando viene raggiunto lo 0 (l'OS può usarla come system tick).

⚠ Le linee degli interrupt sono **256** (16 Interrupts e 240 eccezioni generiche).

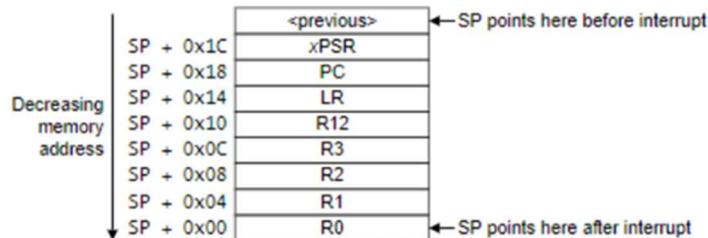
| Exception Type          | Index | Vector Address |
|-------------------------|-------|----------------|
| (Top of Stack)          | 0     | 0x00000000     |
| Reset                   | 1     | 0x00000004     |
| NMI                     | 2     | 0x00000008     |
| Hard fault              | 3     | 0x0000000C     |
| Memory management fault | 4     | 0x00000010     |
| Bus fault               | 5     | 0x00000014     |
| Usage fault             | 6     | 0x00000018     |
| SVCcall                 | 11    | 0x0000002C     |
| Debug monitor           | 12    | 0x00000030     |
| PendSV                  | 14    | 0x00000038     |
| SySTick                 | 15    | 0x0000003C     |
| Interrupts              | ≥16   | ≥0x00000040    |

Gli interrupt hanno delle **priorità**: reset = -3 (max), NMI = -2, Hard Fault = -1 e poi tutti gli altri interrupt sono configurabili.

Le eccezioni possono essere in 3 **stati** diversi:

- **inattiva** = non attiva o pendente
- **attiva** = attualmente in servizio dal processore ma non ancora completata
- **pending** = in attesa di essere servita dal processore

Quando un'eccezione viene eseguita, il processore salva le informazioni nello stack corrente (**STACKING**); la struttura di 8 parole è chiamata **stack frame**.



Gli **exception handler** di default sono dichiarati come *weak symbols* per consentire all'application writer di installare i propri handler semplicemente implementando una funzione con il nome corretto (quello che trovi di solito in uVision). Se avviene un interrupt di cui non è stato definito un handler dall'application writer viene eseguito quello di default (di solito un infinite loop).

```
SVC_Handler PROC SVC_Handler [WEAK]
EXPORT SVC_Handler
B .ENDP
```

Per generare una **SVC** si usa l'istruzione **label SVC immediate**; l'istruzione SVC (o SWI) è su 16 bit ed è composta da 0xDF + un numero (**SVC number**) usato per indicare il caller ed è un numero su 8 bit (da 0 a 255). Sul Cortex-M3, il core salva i registri degli argomenti nello stack sull'iniziale voce di eccezione; qualsiasi valore restituito deve essere restituito al chiamante mediante modifica dei valori dei registri impilati.

Il processore salva un **EXC\_RETURN value** nel lr quando l'eccezione comincia; questo meccanismo di eccezione si basa sul valore determinato quando il processore ha completato l'exception handler. I bit **[31:4]** di exc\_return value sono sempre **0xFFFFFFF**; cambia solo l'ultimo Byte (i bit [3:0] indicano il modo in cui il processore deve ritornare al chiamante):

| EXC_RETURN       | Description                                                                                                       |
|------------------|-------------------------------------------------------------------------------------------------------------------|
| 0xFFFFFFFF1      | Return to Handler mode.<br>Exception return gets state from the main stack.<br>Execution uses MSP after return.   |
| 0xFFFFFFFF9      | Return to Thread mode.<br>Exception Return get state from the main stack.<br>Execution uses MSP after return.     |
| 0xFFFFFFFFD      | Return to Thread mode.<br>Exception return gets state from the process stack.<br>Execution uses PSP after return. |
| All other values | Reserved.                                                                                                         |

Quando il processore carica nel pc un valore che fa match con il pattern di una EXC\_RETURN, riconosce che l'operazione non è una normale branch e che l'eccezione è stata completata (inizializza l'**exception return sequence**).

La **MRS cond spec\_reg, Rn** abilita l'aggiornamento di **registri special purpose** quando si ha un livello privilegiato:

- **cond** = condition code opzionale
- **Rn** = registro sorgente
- **spec\_reg** = 1 tra i registri PSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, **MSP** (master/default sp), **PSP** (alternate sp), PRIMASK, BASEPRI, BASEPRI\_MAX, FAULTMASK, CONTROL (ne parliamo sotto)

Ci sono 2 **modalità operative**:

- **thread** mode = on reset o dopo un'eccezione
- **handler** mode = quando avviene un'eccezione (**sempre privileged mode, mai PSP, ignora CONTROL[0]** ↓)

Ci sono 2 **livelli di accesso/privilegio**:

- ❖ **user/unprivileged** level = accesso limitato alle risorse
- ❖ **privileged** level = accesso a tutte le risorse

Il **CONTROL register** usa i seguenti bit:

- **CONTROL[2]** (gestisce se FPU attiva) → solo su Cortex-M4 e Cortex-M7, se = 0 FPU non attiva, altrimenti attiva
- **CONTROL[1]** (gestisce quale stack pointer è usato):
  - o handler mode: se = 0 uso MSP (non c'è mai alternate stack pointer [PSP] in handler mode)
  - o thread mode:
    - se = 0, uso MSP (master, default)
    - se = 1, uso PSP (alternate)
- **CONTROL[0]** (gestisce il livello di privilegio) → non presente in Cortex-M0, se = 0 il processore è in thread mode in privileged level, se = 1 in thread mode in user level

A tempo di **reset**, dopo le inizializzazioni, si può impostare il processore in **user level** e **thread mode**, usando **PSP**:

```
MOV r0, #3 ; settato a 3 uso PSP, altrimenti uso MSP
MSR CONTROL, r0
```

Poi però, a causa dell'ingresso di una procedura di handling, il sistema si sta muovendo in **privileged level** e in **handler mode**, usando **MSP**.

**Esempio:** **STACK segment**

```
1 Stack_Size EQU 0x00000200
2 AREA STACK, NOINIT, READWRITE, ALIGN=3
3 SPACE Stack_Size/2
4 Stack_Process SPACE Stack_Size/2
5 __initial_sp
```

**CALLER**

```
1 MOV R0, #3
2 MSR CONTROL, R0
3 LDR SP, =Stack_Process
4 SVC 0x10
```

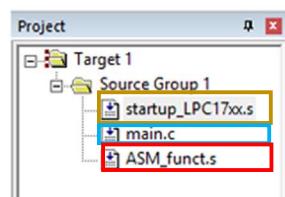
**HANDLER**

```
1 STMFD SP!, {R0-R12, LR}
2 MRS R1, PSP
3 LDR R0, [R1, #24]
4 LDR R0, [R0, #-4]
5 BIC R0, #0xFF000000
6 LSR R0, #16
7 LDMFD SP!, {R0-R12, LR}
8 BX LR
```

⚠ Prima di eseguire le istruzioni scritte nell'SVC Handler, avverrà all'inizio della routine dell'SVC handler il salvataggio dello **stack frame** usando PSP. Per vedere se uso l'indirizzo sp in MSP o PSP basta che nel simulatore guardo l'indirizzo preso in sp e lo comparo con i 2 nella sezione "banked"; poi lo stack verrà modificato con le modifiche scritte nel codice dell'SVC handler. Inoltre, per tenere sotto controllo lo stack ricorda che gli elementi vengono aggiunti dall'indirizzo in sp verso l'alto (quindi devi ingrandire la finestra della "memory" **verso l'alto** e mettere i 2 indirizzi di MSP e PSP cercando di allinearli al centro della finestra, quindi in una finestra "**msp-0x50**" e nell'altra "**psp-0x50**" [la sottrazione serve per spostare l'indirizzo di interesse al centro dello schermo visto che l'indirizzo considerato viene messo in alto dal programma]). Se faccio un salvataggio ad [sp, #24] mi sposto a dx di 24 Byte (6 word) mentre con #-4 mi sposto a sx di 1 Byte nella nostra interfaccia di uVision.

## 11) ASM+C ABI & ARM TOOLCHAIN

Il **CROSS COMPILATORE** gira su una macchina ma genera un codice che gira su un'altra macchina (es. da windows genero codice che gira su un SoC ARM). Se in un progetto di uVision ho startup.s e main.c, posso saltare dal codice dello startup al codice del main con:



```
Reset_Handler PROC
 EXPORT Reset_Handler [WEAK]
 IMPORT __main
 LDR R0, =__main
 BX R0
ENDP
```

```
extern int ASM_funct(int, int, int, int, int, int);

int main(void){

 int i=0xFFFFFFFF, j=2, k=3, l=4, m=5, n=6;
 volatile int r;

 r = ASM_funct(i, j, k, l, m, n);

 while(1);
}
```

• External ASM function invoked by a C function  
`r = ASM_funct(i, j, k, l, m, n);`

Where are parameters stored?

How to return results?

AREA asm\_functions, CODE, READONLY  
`EXPORT ASM_funct`

Parameters are in R0-R3 (a1-a4)

Stacked parameters

```
; save current SP for a faster access
; to parameters in the stack
MOV r12, sp
; save volatile registers
STMFD sp!, {r4-r8, r10-r11, lr}
; extract argument 4 and 5 into R4 and R5
LDR r4, [r12]
LDR r5, [r12, #4]
; setup a value for R0 to return
MOV r0, r5
; restore volatile registers
LDMFD sp!, {r4-r8, r10-r11, pc}
```

⚠ A differenza di come eravamo abituati in C, terminiamo il nostro main.c con un loop bloccante **while(1)**.

Lo standard usato per passare gli argomenti è quello **ABI** visto sopra (\*), ovvero:

| 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<br>SB<br>TR | Platform register.<br>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.                                                 |

⚠ Oltre a richiamare un codice ASM usando una funzione esterna `r = ASM_funct(i,j,k,l,m,n)`, posso anche scrivere codice ASM in un codice C usando `__ASM("SVC #0x10")` (**inline ASM**)

Ci sono 3 **direttive** utili:

- **EXPORT** = rende visibile una funzione fuori dal singolo file che la definisce
- **IMPORT** = rende visibile una funzione da altri files
- **extern** = permette di importare una variabile da un altro file (dov'è definita)

⚠ Ci sono delle **ottimizzazioni** che posso chiedere al compilatore riguardo ad un programma C nelle opzioni di uVision. In alcuni casi però non ci vuole ottimizzazione e in questo caso posso dichiarare davanti ad una funzione/variabile C il termine **"volatile"** (dice al compilatore che la variabile/funzione non può mai essere modificata esternamente all'implementazione).

→ Riprendendo l'esempio sopra nei riquadri vediamo il **"disassemblato"** di uVision:

```

Disassembly
4: int main(void){
5:
6: PUSH (r1-r3,lr)
7: int i=0xFFFFFFF, j=2, k=3, l=4, m=5, n=6;
8: MOV r4,#0xFFFFFFF
9: MOVS r5,#0x02
10: MOVS r6,#0x03
11: MOVS r7,#0x04
12: MOVS r8,#0x05
13: MOVS r9,#0x06
14: volatile int r=0;
15: MOV r0,#0x00
16: STR r0,[sp,#0x08]
17: r = ASM_funct(i, j, k, l, m, n);
18: MOV r3,r7
19: MOV r2,r6
20: MOV r1,r5
21: MOV r0,r4
22: STR r8,[sp,#0x08]
23: ASM_funct (0x0000224)
24: BL W
25: STR r0,[sp,#0x08]
26: while(1);
27: NOP
28: B 0x00001AC

```

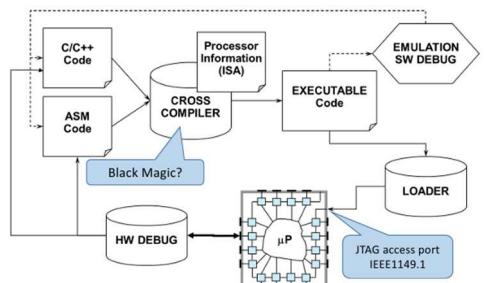
Variabili allocate in register e inizializzate

Variabile volatile allocazione in stack

Parametri i, j, k, l impostati in register r0-r3

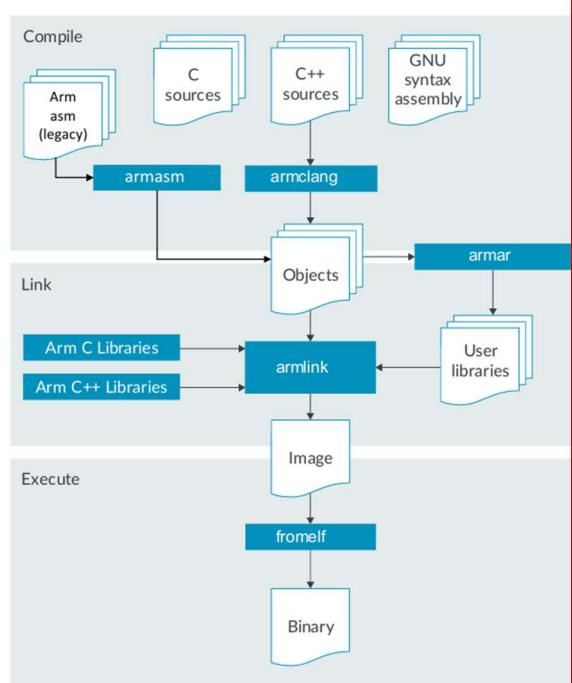
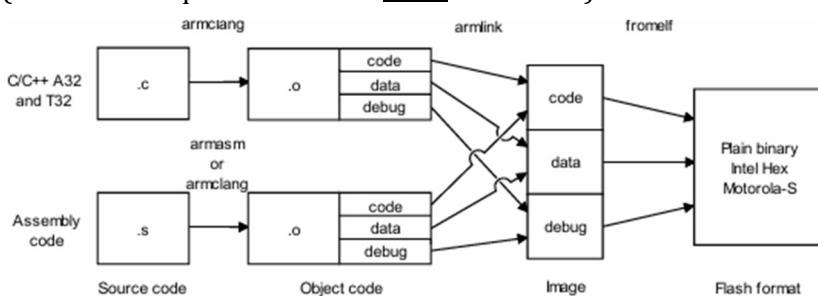
Parametri m, n impostati in stack

Valore restituito scritto all'indirizzo



Ora parliamo di **ARM Toolchain per embedded systems** (da **source code a executable**) [build + debug (sapere la struttura della toolchain aiuta)]. Una **TOOLCHAIN** è un **set di "programming tools"** usato per task di sviluppo complesse o per creare un software (nella toolchain [alto a dx] vediamo anche il cross compiler visto prima).

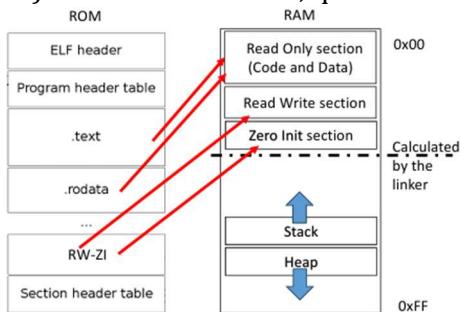
Ora guardiamo nel dettaglio la **toolchain ARM**, composta da 3 fasi (tutte incluse quando faccio la build su uVision):



- **COMPILE** → crea gli object files (.o). GCC non consente dichiarazioni “on fly” [es. `for(int i = 0; ...)`]. Usa:
  - **armclang** = compilare codice di alto livello (C/C++); assemblare codice ASM con sintassi GNU
  - **armasm** = assemblare codice ASM esistente con sintassi GNU
- Gli **optimization levels** influenzano la performance e la size degli executable (e il machine code nell’exe):
  - O0 (level 0) = no ottimizzazione, generated code corrisponde al source code
  - O1 (level 1) = ottimizzazione >, debug view miglior tradeoff tra size, performance e debug
  - O2 (level 2) = ottimizzazione >>, debug view <<
  - O3 (level 3) = ottimizzazione >>>, debug view <<<
  - Of (fast), Omax (max), Os/Oz (size)
- **LINK** → linka tutti gli object files in un singolo exe (**image ELF**) file. Ha bisogno però di **informazioni sulla memoria** per organizzare l’**image memory layout**. Questa fase risolve i **simboli del linker** (diversi dalle funzioni e dalle variabili; es. `__main`) e rimuove il “codice morto” (sezione inusata). Si possono specificare gli entry point allo startup e le informazioni di memoria nelle opzioni Linker di uVision, ma anche da uno scatter file (file custom che specifica quali sezioni di data/code prendere)
- **EXECUTE** → converte l’image ELF in altri formati per le memorie (binari, intel hex-32, motorola-s...). Si possono modificare anche qui le opzioni Output di uVision

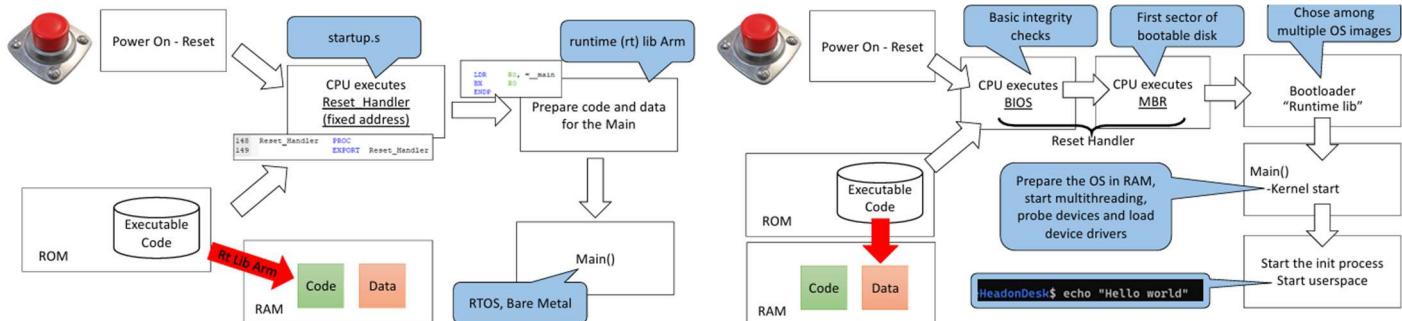
L’ARM toolchain produce come **output**:

- **Eseguibile** (executable; .exe, .elf, .axf) = data+code section; quando viene **caricato in ram** per essere eseguito:



- **Listing & dependencies files** (.d; .dep = project dependencies) = generated and used by the toolchain in the link phase. Listing files are debugging files showing how the code is translated in machine code (opzioni Listing in uVision) [lo startup object dipende solo dallo startup assembly file]
- **Map file** = output log della fase di link: include la memory map, la tabella di simboli, cross references e sizes. Può essere usato da debugging tools (opzioni Listing in uVision, sezione in basso)
- **Static call graph file** (.htm) = debugging output log della fase di link. È un control-flow graph e rappresenta la gerarchia delle chiamate tra funzioni nell’eseguibile
- **Build output log** = log dell’intero processo di build del progetto (quello che leggiamo sotto a console dopo il comando di build su uVision)

Come un SoC “starta” il programma?

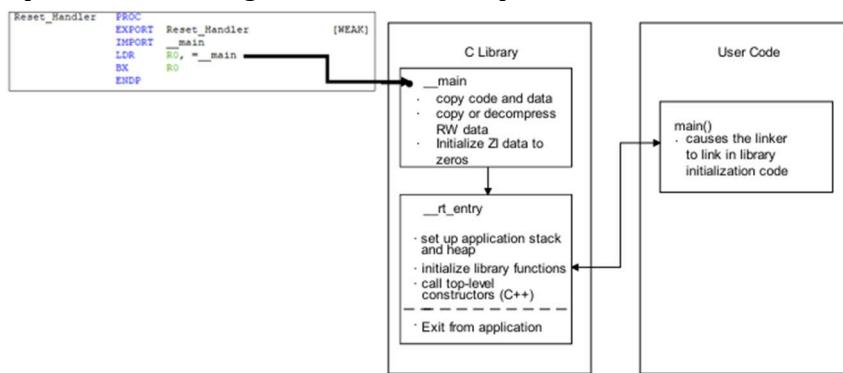


Vediamo sopra a dx il caso di OS bootstrap (ovvero lo startup dell’OS) nel caso di Linux, dove definiamo:

- BIOS (Basic I/O System) = fa integrity checks; carica ed esegue l’MBR
- MBR (Master Boot Record) = è nel 1° settore del bootable disk (< 512 byte); carica ed esegue il bootloader
- Bootloader = se ci sono più kernel images nel tuo sistema (più OS) puoi sceglierne 1 da eseguire

- Kernel = esegue il programma init (PID = 1); usa initrd (initial ram disk) come root file system fino al caricamento del vero filesystem dopo che il boot del kernel

Legato a ciò vediamo l'ARM "Magic secret sauce" (ovvero lo **startup su ARM quando mixiamo ASM e C** [se usiamo solo ASM abbiamo completo controllo sugli stack MSP e PSP, quindi li setto io nel Reset\_Handler]):



- **\_\_main** = setta la memory code e data
- **\_\_rt\_entry** = setta gli stack e l'heap (chiamando **\_\_user\_initial\_stackheap** presente nel codice startup.s), inizializza le funzioni di librerie e i static data

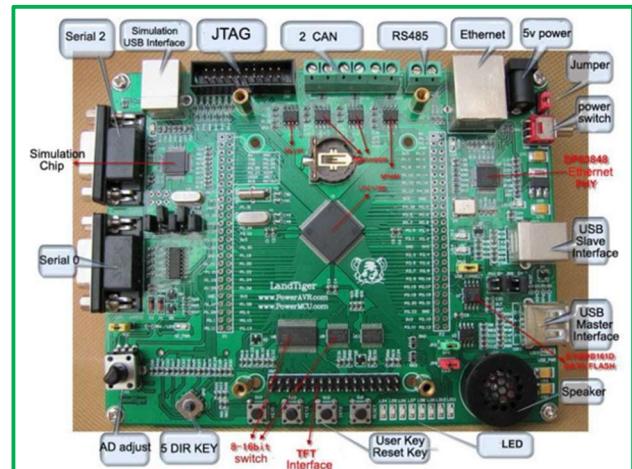
⚠ Se al posto di fare **IMPORT \_\_main** e **LDR r0, =\_\_main** facessi solo **IMPORT main** e **LDR r0, =main** avrei che, se ho delle dichiarazioni prima dell'int main di variabili globali o costanti, queste verrebbero skipgate (porta errori)

## 12) SCHEDA NXP LPC1768

Importante è lo "user manual" (fatto per l'utente per gestire la scheda; ci spiega le capacità e il **contenuto del SoC**) e poi lo "**schematic**" della scheda (con lo schema della **scheda**, come sono connessi i componenti sulla scheda).

Noi useremo il **template project**, che include:

- **startup\_LPC17xx.s**
- libreria **lib\_SoC\_board**:
  - **core\_cm3.c**
  - **system\_LPC17xx.h & system\_LPC17xx.c**
- librerie per periferici come:
  - **peripheral.c** (prototipi)
  - **lib\_peripheral.c** (base functions)
  - **IRQ\_peripheral.c** (interrupt service routines)
  - **funct\_peripheral.c** (advanced user functions)



⚠ Su Keil possiamo come compilare dal menù a tendina con opzione SW Debug o LPC... (cioè sulla scheda fisica); su Keil ora saranno di interesse anche i periferici (menù **Peripherals** in cui ci fa vedere il periferico di interesse).

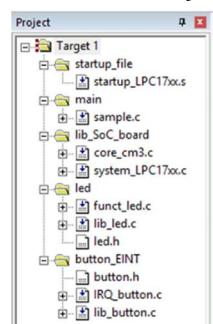
Il **Sample Project 03** (template su pulsanti e LED) prevede che (a dx template) (\* pg 37):

- premere il pulsante INT0 (porta/pin P2.10) accende LD11
- premere il pulsante KEY1 (porta/pin P2.11) accende LD10
- premere il pulsante KEY2 (porta/pin P2.12) spegne LD10 e LD11 se erano accesi

Facendo riferimento al template sopra citato, vediamo nel dettaglio la libreria **lib\_SoC\_board**:

- **system\_LPC17xx.h** → definisce **costanti** dove sono salvati (in linguaggio C) gli indirizzi di memoria **per accedere ai registri periferici** (es. #define **LPC\_APB0\_BASE** definisce le costanti per gli APB0 peripherals [fino a 32 blocchi periferici da 16kB ciascuno]). Per ogni blocco periferico, si definisce quindi un **indirizzo di partenza basato su queste costanti** (es. #define **LPC\_PINCON\_BASE** definito come **LPC\_APB0\_BASE+0x2C000**) [si vedono dalle tabelle questi indirizzi per ciascun diverso periferico]. Si può usare anche un cast per passare da struct al suo TypeDef; per esempio posso fare:

#define **LPC\_PINCON** ((**LPC\_PINCON\_TypeDef** \*) **LPC\_PINCON\_BASE**)



Dove abbiamo quindi:

- **LPC\_PINCON** = nome con cui posso accedere dal codice C (costante)
  - **(LPC\_PINCON\_TypeDef \*)** = cast al puntatore di `LPC_PINCON_TypeDef`
  - **LPC\_PINCON\_BASE** = indirizzo di ogni specifica SoC resource

Dove avremo la **struct** **LPC\_PINCON\_TypeDef** definita come →

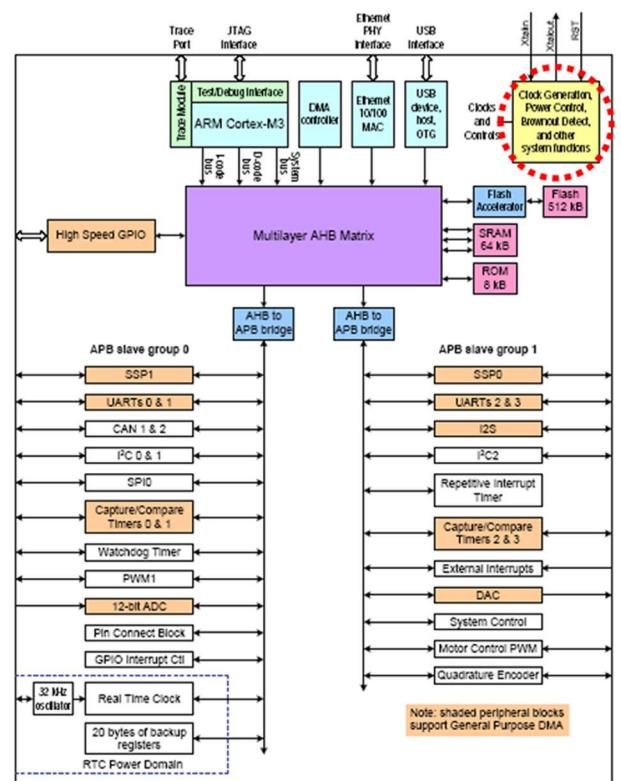
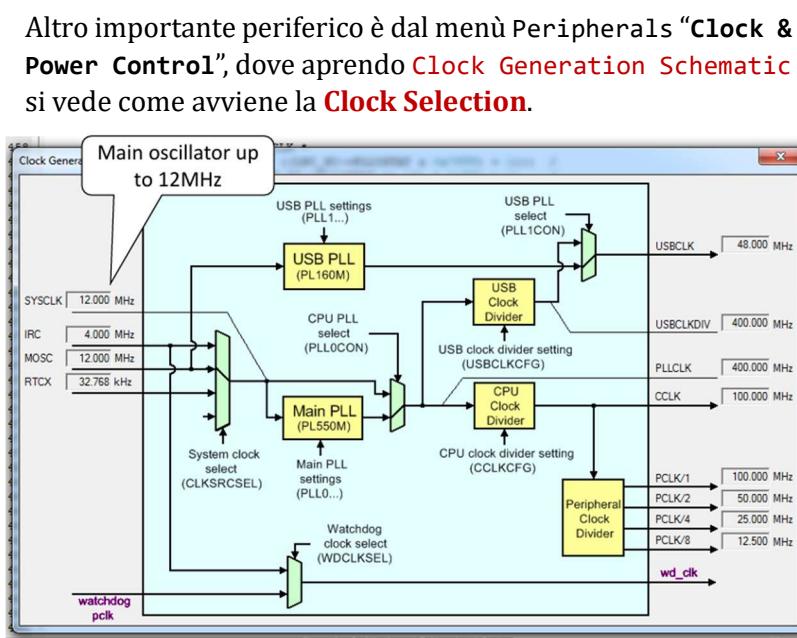
⚠ Avendo questa struct, potrò accedere ai singoli campi (elementi) della struct con la sintassi **LPC\_PINCON->PINSEL4**

- **system\_LPC17xx.c** → inizializza le frequenze di clock del SoC (la SystemInit() è chiamata dal main)
  - **core\_cm3.h** → definisce alcune costanti e funzioni a livello di CPU core (es. APSR\_Type [Program Status Register high-level usage] o NVIC [Nested Interrupt Controller peripheral block])

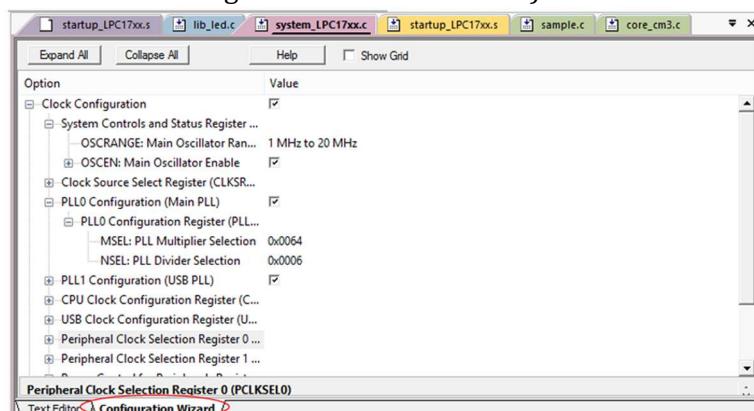
```

typedef struct
{
 _IO uint32_t PINSEL0;
 _IO uint32_t PINSEL1;
 _IO uint32_t PINSEL2;
 _IO uint32_t PINSEL3;
 _IO uint32_t PINSEL4;
 _IO uint32_t PINSEL5;
 _IO uint32_t PINSEL6;
 _IO uint32_t PINSEL7;
 _IO uint32_t PINSEL8;
 _IO uint32_t PINSEL9;
 _IO uint32_t PINSEL10;
 uint32_t RESERVED[5];
 _IO uint32_t PINMODE0;
 _IO uint32_t PINMODE1;
 _IO uint32_t PINMODE2;
 _IO uint32_t PINMODE3;
 _IO uint32_t PINMODE4;
 _IO uint32_t PINMODE5;
 _IO uint32_t PINMODE6;
 _IO uint32_t PINMODE7;
 _IO uint32_t PINMODE8;
 _IO uint32_t PINMODE9;
 _IO uint32_t PINMODE_O0;
 _IO uint32_t PINMODE_O1;
 _IO uint32_t PINMODE_O2;
 _IO uint32_t PINMODE_ODS;
 _IO uint32_t LCBRADCFG;
} LPC_PINCON_TypeDef;

```



La selezione del clock è legata al **system\_LPC17xx.c** dove troviamo le inizializzazione delle frequenze di clock (editabili nel text editor ma anche nel configuration wizard affianco):



Lo **schema dei LED** e il modo in cui questi si rilezano con la CPU sono nello “schematic” di LPC1768 (si vede che l’ordine dei LED è **inverso rispetto ai pin** [LD4 – p2.7, LD5 – p2.6, ..., LD11 – p2.0]). Una volta che si trovano i PIN della CPU cercati, vanno date al SoC anche altre informazioni come la **SoC PIN functionality** e la **PIN direction** (input/output). Legato ai PIN abbiamo nella scheda:

➤ PIN Connect Block = associa i PIN alle funzioni in base ai bit:

| PINSEL4 | Pin name | Function when 00 | Function when 01 | Function when 10 | Function when 11 | Reset value |
|---------|----------|------------------|------------------|------------------|------------------|-------------|
| 1:0     | P2.0     | GPIO Port 2.0    | PWM1.1           | TXD1             | Reserved         | 00          |
| 3:2     | P2.1     | GPIO Port 2.1    | PWM1.2           | RXD1             | Reserved         | 00          |
| 5:4     | P2.2     | GPIO Port 2.2    | PWM1.3           | CTS1             | Reserved         | 00          |
| 7:6     | P2.3     | GPIO Port 2.3    | PWM1.4           | DCD1             | Reserved         | 00          |
| 9:8     | P2.4     | GPIO Port 2.4    | PWM1.5           | DSR1             | Reserved         | 00          |
| 11:10   | P2.5     | GPIO Port 2.5    | PWM1.6           | DTR1             | Reserved         | 00          |
| 13:12   | P2.6     | GPIO Port 2.6    | PCAP1.0          | RI1              | Reserved         | 00          |
| 15:14   | P2.7     | GPIO Port 2.7    | RD2              | RTS1             | Reserved         | 00          |
| 17:16   | P2.8     | GPIO Port 2.8    | TD2              | TXD2             | ENET_MDC         | 00          |
| 19:18   | P2.9     | GPIO Port 2.9    | USB_CONNECT      | RXD2             | ENET_MDIO        | 00          |
| 21:20   | P2.10    | GPIO Port 2.10   | EINT0            | NMI              | Reserved         | 00          |
| 23:22   | P2.11    | GPIO Port 2.11   | EINT1            | Reserved         | I2STX_CLK        | 00          |
| 25:24   | P2.12    | GPIO Port 2.12   | EINT2            | Reserved         | I2STX_WS         | 00          |
| 27:26   | P2.13    | GPIO Port 2.13   | EINT3            | Reserved         | I2STX_SDA        | 00          |
| 31:28   | -        | Reserved         | Reserved         | Reserved         | Reserved         | 0           |

➤ GPIO (General Purpose I/O) [si occupa quindi anche dei LED]:

| Generic Name | Description                                                                                                                                                                                                                                                                                                                                                                                     | Access | Reset value | PORTn Register Name & Address                                                                                                  |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|-------------|--------------------------------------------------------------------------------------------------------------------------------|
| FIODIR       | Fast GPIO Port Direction control register. This register individually controls the direction of each port pin.                                                                                                                                                                                                                                                                                  | R/W    | 0           | FIO0DIR - 0x2009 C000<br>FIO1DIR - 0x2009 C020<br>FIO2DIR - 0x2009 C040<br>FIO3DIR - 0x2009 C060<br>FIO4DIR - 0x2009 C080      |
| FIOMASK      | Fast Mask register for port. Writes, sets, clears, and reads to port (done via writes to FIOPIN, FIOSET, and FIOCLR, and reads of FIOPIN) alter or return only the bits enabled by zeros in this register.                                                                                                                                                                                      | R/W    | 0           | FIO0MASK - 0x2009 C010<br>FIO1MASK - 0x2009 C030<br>FIO2MASK - 0x2009 C050<br>FIO3MASK - 0x2009 C070<br>FIO4MASK - 0x2009 C090 |
| FIOPIN       | Fast Port Pin value register using FIOMASK. The current state of digital port pins can be read from this register, regardless of pin direction or alternate function selection (as long as pins are not configured as an input to ADC). The value read is masked by ANDing with inverted FIOMASK. Writing to this register places corresponding values in all bits enabled by zeros in FIOMASK. | R/W    | 0           | FIO0PIN - 0x2009 C014<br>FIO1PIN - 0x2009 C034<br>FIO2PIN - 0x2009 C054<br>FIO3PIN - 0x2009 C074<br>FIO4PIN - 0x2009 C094      |
| FIOSET       | Fast Port Output Set register using FIOMASK. This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect. Reading this register returns the current contents of the port output register. Only bits enabled by 0 in FIOMASK can be altered.                                                                             | R/W    | 0           | FIO0SET - 0x2009 C018<br>FIO1SET - 0x2009 C038<br>FIO2SET - 0x2009 C058<br>FIO3SET - 0x2009 C078<br>FIO4SET - 0x2009 C098      |
| FIOCLR       | Fast Port Output Clear register using FIOMASK. This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect. Only bits enabled by 0 in FIOMASK can be altered.                                                                                                                                                            | WO     | 0           | FIO0CLR - 0x2009 C01C<br>FIO1CLR - 0x2009 C03C<br>FIO2CLR - 0x2009 C05C<br>FIO3CLR - 0x2009 C07C<br>FIO4CLR - 0x2009 C09C      |

⚠ Nella LED\_init troviamo sintassi come:

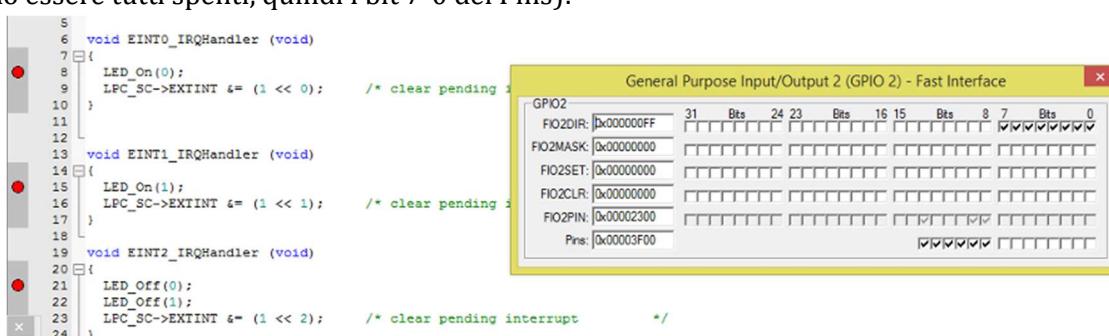
```
LPC_PINCON->PINSEL4 &= 0xFFFF0000; // setto PIN mode GPIO (00b value da P2.0 a P2.7) [setup Pinsel]
LPC_GPIO2->FIODIR |= 0x000000FF; // P2.0 ... P2.7 Output (LED su PORT2 definito come Output) [Direction]
```

Riprendendo il discorso dei **BUTTONS (PULSANTI)** (\* pg 35), questi funzionano in 2 modi: attivo se premo (0) o attivo se rilascio (1). Quando metto le porte/pin 2.10, 2.11, 2.12 e 2.13 a 01, queste si comportano come EINT0, EINT1, EINT2, EINT3 (**External Interrupt**). Vediamo **come possiamo configurare EINT**:

```
LPC_PINCON->PINSEL4 |= (1<<20); // setto EXTINT functionality (per EINT0)
LPC_GPIO2->FIODIR &= ~(1<<10); // setto la direction (per EINT0)
LPC_PINCON->PINSEL4 |= (1<<22); // EINT1
LPC_GPIO2->FIODIR &= ~(1<<11);
LPC_PINCON->PINSEL4 |= (1<<24); // EINT2
LPC_GPIO2->FIODIR &= ~(1<<12);
```

```
LPC_SC->EXTMODE = 0x7; // EXTINT pins mode: edge sensitive
NVIC_EnableIRQ(EINT2_IRQn); // NVIC selective enable of EINTs (* pg 39)
NVIC_EnableIRQ(EINT1_IRQn);
NVIC_EnableIRQ(EINT0_IRQn);
```

Ora dobbiamo capire **come possiamo triggerare un EINT** (prerequisito è avere definito DCD EINT0\_IRQHandler come WEAK nel startup.s per poter scriverne il corpo in C): bisogna mettere i breakpoint all'inizio del corpo dei vari EINT\_IRQHandler in C per vedere cosa succede nella **GPIO-Fast Interface di Keil** (dobbiamo guardare che all'inizio i BUTTONs siano tutti rilasciati, quindi i bit 12-10 della sezione Pins al fondo siano checkati, mentre i LED devono essere tutti spenti, quindi i bit 7-0 dei Pins):



Quando entriamo in **EINT0\_IRQHandler**, ovvero pressione del Bottone **INT0** (bit 10 non checkato), vediamo che avviene **LED\_On(0)** e viene “pulito” l'interrupt pending con **LPC\_SC->EXTINT &= (1<<0)**.

## 13) INTERRUPT CONTROLLER

Gli **eventi di sistema** possono essere:

- Eventi poco frequenti ma importanti
- I/O synchronization
- Periodic interrupts
- Data acquisition samples ADC

Per gestire le periferiche ci sono **2 approcci**:

- **POLLING** → il processore **controlla periodicamente lo stato del dispositivo** (CPU sempre sveglia e interroga [“poll”] per capire, quindi semplice ma inefficiente):

```
While(1){
 Check_peripheral(1);
 Check_peripheral(2);
 Check_peripheral(3);
 Check_peripheral(4);
 Check_peripheral(5);
}

void Check_peripheral(int i){
 If new_input(i)
 Goto handler(i)
 •
}
```

Per determinare se è disponibile o meno, si controlla lo status register (*best practice*) oppure verificando i data registers. Sopra vediamo l'implementazione software del polling, usando un ciclo infinito che esegue una sequenza di check per i vari periferici. Quindi:

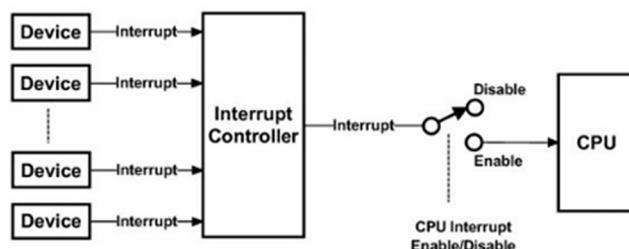
- ✓ Facile da implementare
- ✗ Tanto tempo speso nel ciclo software (inefficienza energetica)
- ✗ Alta latenza
- ✗ Difficile gestione delle richieste annidate (low performance)

- **INTERRUPT** → periferiche comunicano direttamente con la CPU mediante gli **interrupt** (interruzioni) implementati in **hardware**. Il sistema può quindi stare in uno **stato di basso consumo** (sospensione, **idle**) e il sistema **si risveglia quando un evento di sistema si verifica**. Quando viene ricevuta la richiesta, la CPU ha bisogno di riconoscere la sorgente in modo da eseguire il **corretto handler**: la gestione degli interrupt avviene con la **IVT** (Interrupt Vector Table), quindi la CPU collabora con periferici mediante l'**INTERRUPT CONTROLLER**.

Per **configurare la interrupt mode**, bisogna inizializzare al boot le strutture dati come counters, pointers e flag per abilitare gli interrupt (semafori); bisogna anche configurare l'interrupt controller abilitando le sorgenti e impostando la priorità di ogni sorgente (ciò dovrà essere fatto in ogni routine di servizio per l'interrupt).

Anche a **runtime** è necessario un ACK pulendo i flag che indicano gli interrupt attivi (può essere fatto in parti diverse della routine); è necessario mantenere il contenuto di **R4-R8, R10, R11 (standard ABI)** comunicando attraverso variabili globali condivise (+ viene creato lo **stack frame** con R0-R3, R9, R12-R15 [come sempre])

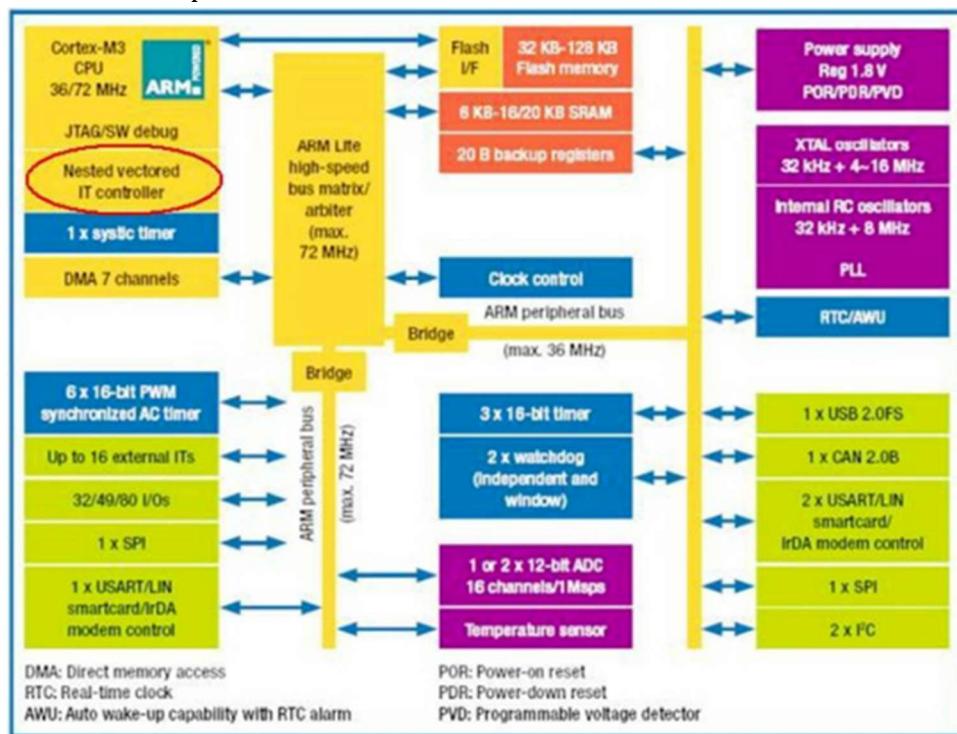
**L'INTERRUPT CONTROLLER (IC)** è un dispositivo che viene usato per **combinare diverse sorgenti di interrupt in 1 o più linee della CPU**, consentendo di gestire più livelli di priorità assegnabili agli interrupt outputs (quindi gestisce i segnali di interrupt ricevuti da più dispositivi combinandoli in 1 solo interrupt output):



⚠ Nell'immagine si vede che si può staccare IC e CPU, ma si può fare sull'8086, non in ARM sulla nostra scheda

## Sul Cortex-M3, c'è **NVIC** (Nested Vectorial Interrupt Controller):

- Lo stretto accoppiamento con la CPU consente interruzioni con bassa latenza e efficiente ritardo in arrivo degli interrupt
- Gestisce 35 possibili interrupt esterni



Per **abilitare un external interrupt** di un device nella NVIC, usiamo (\* pg 37):

```
static __INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
{
 NVIC->ISER[((uint32_t)(IRQn) >> 5)] = (1 << ((uint32_t)(IRQn) & 0x1F)); /* enable interrupt */
}
```

⚠ È importante mettere i **breakpoint** all'inizio del corpo degli enableIRQ perché altrimenti non capisco!

Per settare la **priorità** dell'interrupt (ricorda che **priorità >** ha numero di priorità **<**):

```
\param [in] IRQn Number of the interrupt for set priority
\param [in] priority Priority to set
*/
static __INLINE void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
{
 if(IRQn < 0) {
 SCB->SHP[((uint32_t)(IRQn) & 0xF)-4] = ((priority << (8 - __NVIC_PRIO_BITS)) & 0xff); /* set Priority for Cortex-M System
 } else {
 NVIC->IP[((uint32_t)(IRQn))] = ((priority << (8 - __NVIC_PRIO_BITS)) & 0xff); } /* set Priority for device specific
```

⚠ Se chiamo una routine di interrupt con priorità inferiore, mentre una con priorità superiore è in corso, quella inferiore viene segnata come **pending** (P); viceversa, quella con priorità superiore prende il controllo (attiva) e solo dopo che questa viene gestita si ritorna a quella con priorità inferiore (**nested interrupt**)

Come visto in precedenza uso la costante per riferirmi alla struct con tutti gli elementi della NVIC, ovvero **#define NVIC ((NVIC\_Type \*) NVIC\_BASE)**, dove **NVIC\_Type** è:

```
typedef struct
{
 __IO uint32_t ISER[8]; /*!< Offset: 0x000 (R/W) Interrupt Set Enable Register
 uint32_t RESERVED0[24];
 __IO uint32_t ICER[8]; /*!< Offset: 0x080 (R/W) Interrupt Clear Enable Register
 uint32_t RSERVED1[24];
 __IO uint32_t ISPR[8]; /*!< Offset: 0x100 (R/W) Interrupt Set Pending Register
 uint32_t RESERVED2[24];
 __IO uint32_t ICPR[8]; /*!< Offset: 0x180 (R/W) Interrupt Clear Pending Register
 uint32_t RESERVED3[24];
 __IO uint32_t IABR[8]; /*!< Offset: 0x200 (R/W) Interrupt Active bit Register
 uint32_t RESERVED4[56];
 __IO uint8_t IP[240]; /*!< Offset: 0x300 (R/W) Interrupt Priority Register (8Bit wide)
 uint32_t RESERVED5[64];
 __IO uint32_t STIR; /*!< Offset: 0xE00 (/W) Software Trigger Interrupt Register
} NVIC_Type;
```

## 14) CLOCKING & POWER CONTROL FUNCTIONS

**PCON** (Power Control Register) è il registro di controllo che consente di entrare nelle varie modalità di funzionamento energetico (seleziona power-down o deep power-down):

- **Power-down mode** = come deep sleep, ma anche la flash viene spenta (> risparmio energetico, ma risveglio più lento); viene settato il bit **PDFLAG** di PCON
- **Deep power-down mode** = alimentazione completamente spenta per il chip eccetto per il real-time clock, il reset pin, il WIC, il registro di backup e RTC; viene settato il bit **DPDFLAG** di PCON

I bit **PM1** e **PM0** di PCON permettono di selezionare una modalità di risparmio energetico e garantiscono retrocompatibilità con dispositivi che non supportano sleep e power-down.

**PCONP** (Power Control Register for Peripherals) è il registro relativo all'alimentazione periferica (utile se voglio usare il timer 2 e 3, vedremo dopo); consente di spegnere singolarmente le periferiche non necessarie per il funzionamento dell'applicazione (> risparmio energetico) [alcune non si possono spegnere (es. watchdog timer, SCB e il pin connect block) [si spegne la sorgente del clock per determinare quali periferiche spegnere]]

⚠ Il **SCR** (System Control Register) appartiene ai “**core peripherals**” e controlla entrata/uscita da low power state; consiglio è non usare deep sleep quando faccio debug.

L'ingresso in modalità di **low power state** inizia con **WFI** (wait for interrupt) o **WFE** (wait for exception). Il Cortex M3 supporta 2 modalità di risparmio energetico:

- **SLEEP MODE** → se attivata, il **clock del core viene fermato** e l'**SMFLAG** bit di PCON viene **settato**. L'esecuzione delle istruzioni viene fermata **fino a che non avviene un reset o un interrupt** (il wake-up avviene quando qualsiasi istruzione occorre). Le periferiche (non core) però continuano a funzionare e potrebbero generare interrupt che potrebbero risvegliare l'esecuzione. Elimina il dynamic power consumption
- **DEEP SLEEP MODE** → il **clock del core viene fermato interrompendo l'oscillatore principale** (PLL spenti) e il **DSFLAG** bit di PCON viene **settato**; l'IRC continua e può guidare il watchdog timer (16 nella IVT) per svegliare la CPU (anche l'oscillatore RTC a 32 kHz non è fermato e potrebbe usare un interrupt per svegliare la CPU). La **flash è in stand-by** pronta per un risveglio rapido; gli stati di cpu, registri e SRAM sono prelevati e i livelli logici dei PIN sono statici. Può essere **interrotto da reset o interrupt** (slegato dal clock)

## 15) TIMER

Un sistema può richiedere di aspettare per un periodo di tempo o fare operazioni **periodiche**: ciò richiede **periferiche** chiamate **TIMER** (consente al programmatore di sincronizzare il sistema basandosi sul conteggio). Quando un conteggio raggiunge la fine, il sistema di solito reagisce con un interrupt handler o con l'entrata in risparmio energetico.

Il timer ha un **clock signal dedicato** attraverso cui incrementa il suo contatore; ha registri che possono essere programmati con un n° di clock cycle da contare.



Un timer segue diverse modalità di funzionamento:

- **Decreasing** count → si interrompe quando il counter = 0
- **Increasing** count → si interrompe quando il counter raggiunge un valore max

$$time[s] = count * Clock\_Period[s]$$

$$count = time[s] / Clock\_Period[s]$$

$$count = time[s] * frequency[1/s]$$

Se il tempo di attesa è troppo alto, può non rientrare all'interno del timer del registro; si risolve con soluzione:

- hw = cascade of counter oppure prescaler
- sw = handler count of hw events

Nel nostro **LPC1768** abbiamo **più timer**:

- **STANDARD TIMERS** = programmati dall'utente per implementare intervalli regolari e ritardi; conta i cicli del **peripherical clock** (PCLK) o di un **clock esterno**. Può generare interruzioni o eseguire azioni per valori specifici del timer, usando **4 match registers**. Prevede 4 capture inputs per fare il trap del "timer value" quando avviene una transizione del segnale, generando opzionalmente un'interruzione [timer 0 e 1 sono di default]
  - o **MATCH REGISTERS** = 4 registri a 32 bit che permettono:
    - operazioni continue con interrupt opzionale su match
    - fermare il timer su match (con eventuale interrupt)
    - resettare il timer su match (con eventuale interrupt)
    - generazione di un unique interrupt → l'ISR deve capire quale dei 4 match registers ha generato l'interrupt (leggendo il registro IR)
  - o **CAPTURE REGISTERS** = una transizione può essere registrata da un pin configurato per caricare uno dei capture registers (con il valore del timer counter e eventualmente genera un interrupt)
  - o **EXTERNAL MATCH OUTPUT** = quando un match register (MR3:0) è uguale ad un timer counter, questo output può essere "toggled", andare basso, alto o non fare niente
  - o **MAIN REGISTERS**:
    - **IR** (Interrupt Register) = identifica quale degli 8 interrupts si sta verificando (4 bit per i match interrupt e 4 bit per i capture interrupt)
    - **TCR** (Timer Control Register) = controlla le funzioni del timer counter (può anche resettarlo); il bit 0 è il Counter Enable, mentre il bit 1 è il Counter Reset
    - **TC** (Timer Counter) = incrementato ogni PR+1 cycles del PCLK
    - **MCR** (Match Control Register) = controlla se avviene un interrupt e se TC viene resettato quando avviene un match
    - **MR0** (Match Register 0) = resetta TC, stoppa PC e TC e fa interrupt se TC matcha MR0
- **OPERATING SYSTEM (OS) TIMERS** = usati dall'OS per gestire i processi
- **EXTRA TIMERS** = danno all'OS funzionalità specifiche (es. ripetitive interrupt timer e PWM)

Riguardo al codice, le funzioni dei timer sono in:

- **lib\_timer.c**
  - o `init_timer(timer#, timerInterval)`
  - o `enable_timer(timer#)`
  - o `disable_timer(timer#)`
  - o `reset_timer(timer#)`
- **IRQ\_timer.c**
  - o `TIMER0_IRQHandler()`
  - o `TIMER1_IRQHandler()`

⚠ Per accendere il timer 2 bisogna: aprire system\_LPC1768.c (nella cartella lib\_SoC\_board), andare dalle tab in basso nel configuration wizard e mettere a 1 la voce PCTIM2; per abilitarlo, si usa `enable_timer(2)`

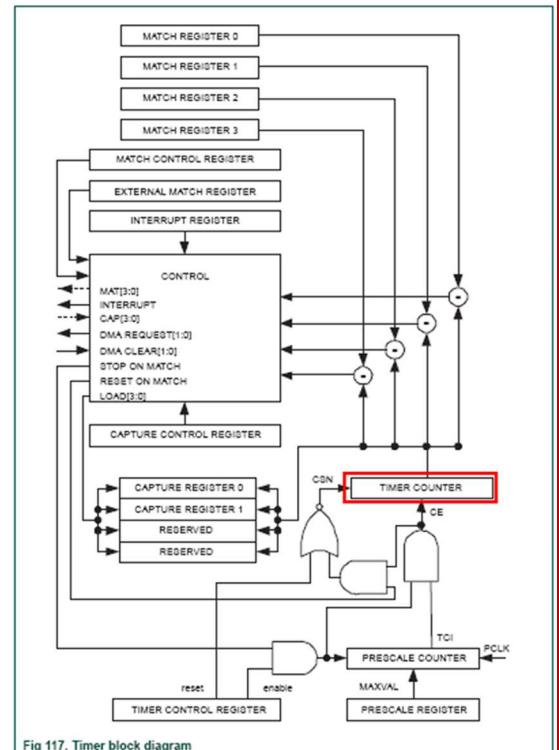
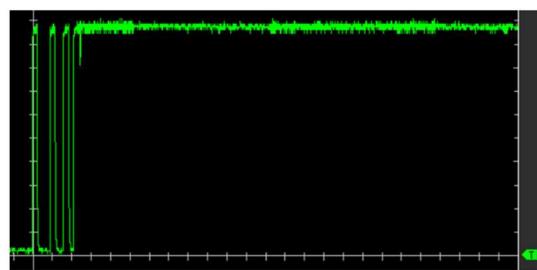


Fig 117. Timer block diagram

## 16) SWITCHING BOUNCING

Quando un pulsante viene premuto o quando viene invertito un toggle switch, 2 parti metalliche si toccano e il contatto sembra immediato, ma non è così in quanto negli switch ci sono parti che si muovono. Quindi, quando un pulsante viene premuto, prima avviene il contatto con l'altra parte metallica in qualche microsecondo, poi c'è un ulteriore contatto leggermente più lungo e così via fino a che gli switch non sono completamente chiusi:

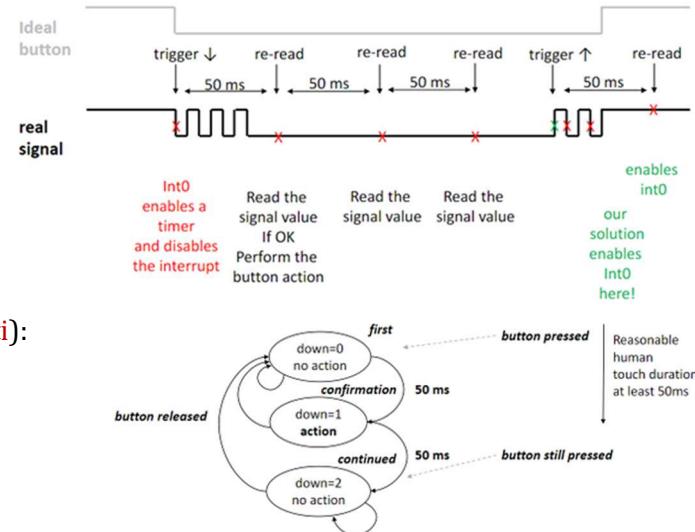
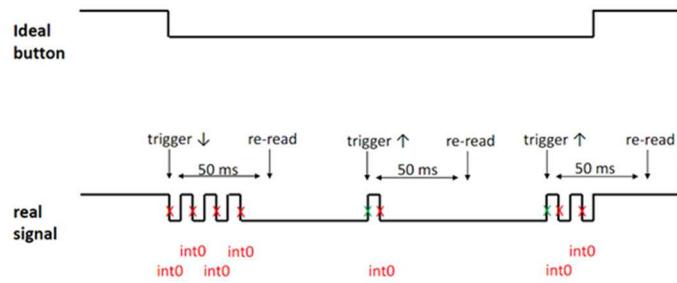
dunque lo switch sta rimbalzando tra “in contatto” e “non in contatto” (**BOUNCING**). Il problema è che il SoC lavora più velocemente del bouncing e quindi l’HW pensa che il pulsante venga premuto più volte.



Una soluzione HW è usare un condensatore, ma costa ed è poco efficiente. La soluzione SW è **rileggere il valore del pin dopo 50ms dal 1° bounce**. Per i pulsanti si preferisce usare gli interrupt (più efficienza energetica, in quanto si può entrare in power down mode), se non disponibile si usa il polling (un timer può essere usato per svegliare il sistema a tempi regolari).

⚠ L’implementazione blocking delay non è consigliata, specialmente usare for/while/do-while con blocchi vuoti

Se il pin di interrupt è settato, potrebbe non essere direttamente leggibile; perciò, **per poter leggere il valore del pulsante** è necessario disabilitare gli interrupt e accettare di leggere l’input value. **Quello che succede VS quello che vogliamo ottenere:**



Ciò si può schematizzare nella FSM (macchina a stati finiti):

Il **RIT** (Repetitive Interrupt Timer) è un timer per generare interrupt a intervalli, senza usare lo standard timer.

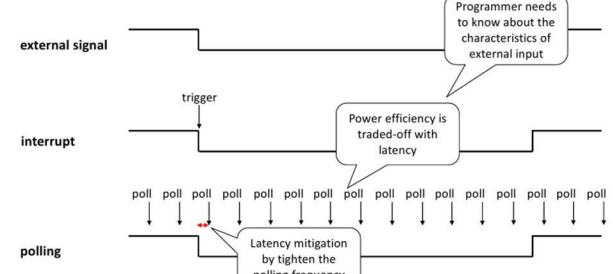
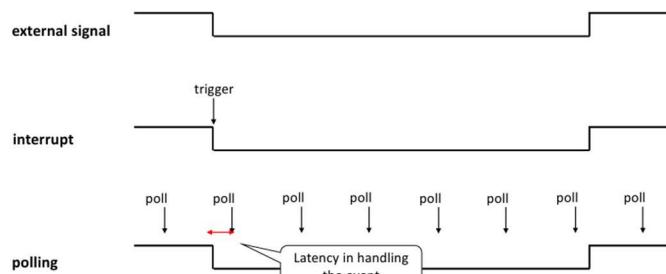
⚠ Il RIT di default è spento

⚠ Posso anche fare la `init_RIT()` nel main.c, ma devo fare la `enable_RIT()` nell’handler del pulsante con cui lo voglio abilitare

```
// RIT/IRQ_RIT.c
uint32_t init_RIT (uint32_t RITInterval){
 LPC_SC->PCLKSEL1 &= ~(3<<26);
 LPC_SC->PCLKSEL1 |= (1<<26); //RIT Clock = CCLK
 ...
 lib_RIT.c
 RIT_cnt = 50ms * 100MHz
 RIT_cnt = 5.000.000 = 0x4C4B40
 ...
}
```

## 17) POLLING SWITCHES - JOYSTICK

Abbiamo già detto come usare interrupt sia meglio di usare il polling (più timing efficient e power efficient), ma **non tutti gli eventi generano interrupt** (es. dispositivi esterni connessi a pin **non configurabili come sorgenti di external interrupt**, nel nostro caso il **JOYSTICK**). Sotto vediamo il compromesso per un buon polling:



L'approccio software base di polling (ovvero `while(1) poll(register);`) **fa schifo** (power inefficient), ma si preferisce usare soluzioni “**timer-based**” (triggerare un'interruzione a intervalli regolari, il sistema dorme mentre il timer conta).

Il joystick sulla nostra scheda LandTiger LPC1768 ha un **5-way digital joystick** (5 keys: su, giù, destra, sinistra, pressione [select]; si può premere + direzioni insieme). Gli input pins sono **hw debounced**.



Il joystick è connesso a pin non configurabili come external interrupt (già detto sopra); quindi l'unico modo per leggerne il valore è **settare i pin come GPIO in input direction e fare poll del GPIO register value**.

Possiamo usare il RIT (Repetitive Interrupt Timer, visto nei timer) per implementare il polling: **ogni 50 ms il RIT triggerà un interrupt** (una sorta di **polling con cadenza di 50 ms**). Nel RIT Handler, il valore della porta GPIO1 è letto e se il valore del bit 25 è 0 (input pin attivo a livello basso = una key è premuta), allora:

- se è la **1^ pressione**, un'azione viene fatta
- se è una **pressione ripetuta**, non si fa niente (no bouncing, altrimenti avremmo una serie di pressioni e rilasci, mentre l'effettiva pressione è 1)

⚠ A differenza del caso del solo bouncing/debouncing, nel joystick si ha sia la **init** che l'**enable** del RIT nel main.c (sample.c) perché siamo in polling

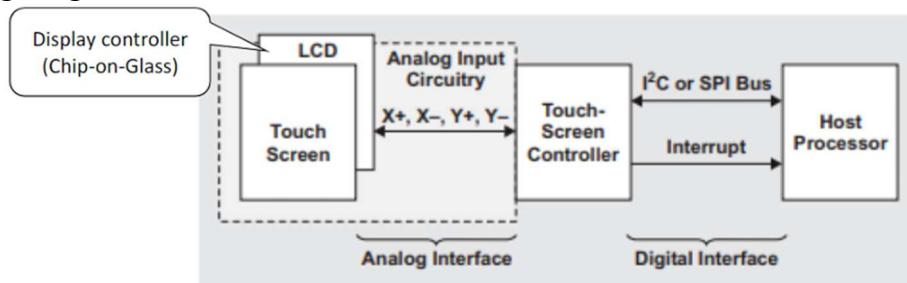
```

26 void RIT_IRQHandler (void)
27 {
28 static int select=0;
29
30 if((LPC_GPIO1->FIOPIN & (1<<25)) == 0) {
31 /* Joystick Select pressed */
32 select++;
33 switch(select){
34 case 0:
35 /* your action here */
36 break;
37 default:
38 break;
39 }
40 }
41 else{
42 select=0;
43 }
}

```

## 18) TOUCH DISPLAY

Esistono vari tipi di display, noi vedremo un touch display; le 2 componenti (il display e il touch sensor) sono indipendenti e vengono gestite da **2 interfacce differenti**:



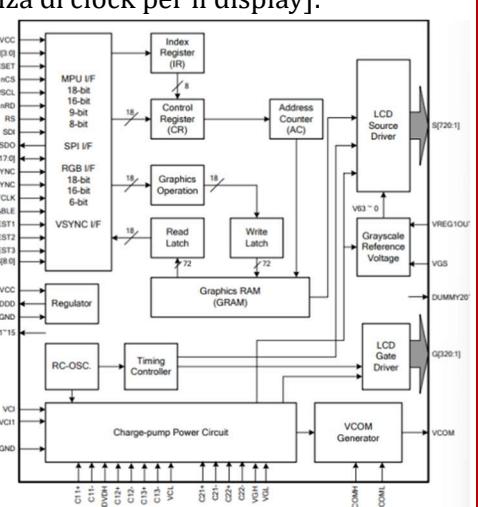
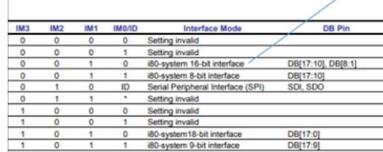
Le tipologie sono: LCD (liquid crystal display), TFT LCD (thin film transistor LCD), IPS LCD (in-plane switching LCD) e LED-backlit LCD (LCD con retroilluminazione a LED). Ci sono diverse tecnologie del touchscreen: resistivo, capacitivo, infrarossi, surface acoustic wave, near field imaging e light pens. **Noi useremo capacitivo TFT LCD.**

Vediamo i 2 controller sulla nostra scheda:

- **DISPLAY CONTROLLER (ILI9325)** [non oltrepassare i 32kHz di frequenza di clock per il display]:

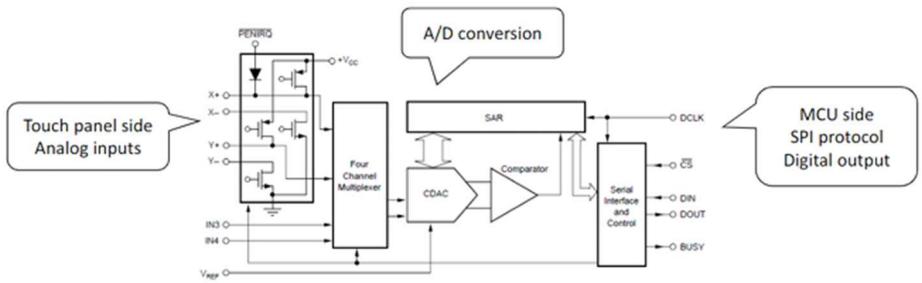
- single chip
- 320x240 resolution (RGB)
- 720-channel source driver e 320-channel gate driver
- 172800 Bytes graphic RAM
- high-speed RAM burst write function
- system interfaces (noi useremo i80 system interface)

Block Diagram  
MPU side

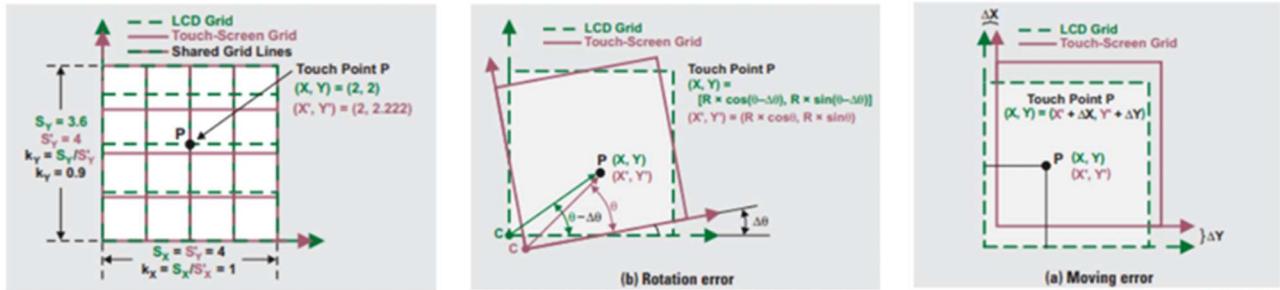


## • TOUCH SCREEN CONTROLLER (ADS7843):

- 4-wire touch screen interface
- ratiometric conversion
- single supply (2.7V - 5V)
- fino a 125kHz conversion rate
- serial interface
- programmable 8 or 12 bit resolution
- 2 input analogici ausiliari
- full power-down control



⚠ Il display e il touch non sono perfettamente allineati e perciò alla 1<sup>a</sup> accensione deve essere calibrato:



Quando il dito viene tracciato dal display, questo appare come un cerchio, ma in realtà il touch panel potrebbe salvare le coordinate come un ellisse: ciò potrebbe essere causato da alcune trasformazioni come traslazione, rotazione e scalatura.

La calibrazione consiste in:

$$\begin{cases} x_{NEW} = f_1(x_{OLD}, y_{OLD}) + c_1 \\ y_{NEW} = f_2(x_{OLD}, y_{OLD}) + c_2 \end{cases} \rightarrow \text{Dato che schermo è sistema lineare: } \begin{cases} x_{NEW} = A x_{OLD} + B y_{OLD} + C \\ y_{NEW} = D x_{OLD} + E y_{OLD} + F \end{cases}$$

Dobbiamo quindi capire i valori di A, B, C, D, E e F (6 equazioni); per farlo usiamo il **3-POINTS CALIBRATION METHOD**; si generano 3 punti indipendenti tra loro (x,y) toccando lo schermo in 3 punti, ottenendo equazioni:

$$x_{1d} = x_1 A + y_1 B + C$$

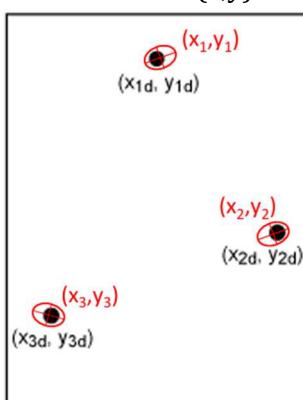
$$x_{2d} = x_2 A + y_2 B + C$$

$$x_{3d} = x_3 A + y_3 B + C$$

$$y_{1d} = x_1 D + y_1 E + F$$

$$y_{2d} = x_2 D + y_2 E + F$$

$$y_{3d} = x_3 D + y_3 E + F$$



⚠ Troviamo la funzione `setCalibrationMatrix` e `getDisplayPoint` nel nostro progetto per settare queste trasformazioni

## 19) CAN BUS

Il **CAN Bus** (Controller Area Network Bus) è uno standard di comunicazione per microcontrollori e dispositivi che usa un protocollo basato su **messaggi**. Ha costo basso, architettura centralizzata robusta e efficienza nella trasmissione dati. Dal punto di vista fisico ha uno **standard seriale multi-master** per collegare ECU (o nodi) tramite **doppino intrecciato** (impedenza di  $120\Omega$ ); la comunicazione avviene tramite transceiver che traduce bidirezionalmente i segnali logici in segnali elettrici per il bus a **2 fili** (**segnali differenziali con stati dominanti 0 e recessivi 1**):

- Trasmissione logica 0 (stato dominante) → MOSFET chiusi con CANH (3.5V) e CANL (1.5V); in Master-Slave CANH e CANL sono determinate dai trasmettitori o dai resistori di terminazione
- Trasmissione logica 1 (stato recessivo) → MOSFET aperti con CANH e CANL che fluttuano verso  $V_{DD}/2$

Tutti i nodi possono trasmettere e ricevere insieme (full-duplex) [i receivers restano recessivi mentre ascoltano]. Le modalità d'uso sono **Master-Slave** (1 nodo trasmette agli altri) e **Multi-Master** (più nodi trasmettono, ma vanno gestiti eventuali conflitti [necessita di **arbitraggio**, ne parliamo dopo con l'ID nel campo Arbitraggio]).

Il bus CAN segue logica **Wired-AND**: se tutti i nodi trasmettono 1, il bus è recessivo[1]; se anche 1 solo trasmette 0, il bus diventa dominante[0].

I periodi di attività/inattività sono regolati dal bitrate/baudrate (bit/sec). Tutti i nodi devono operare alla **stesso bitrate** (rumore e tolleranza possono generare discrepanze, quindi per avere riallineamento è richiesta risincronizzazione ad ogni transizione dominante [**sincronizzazione di start** = transizione da recessivo a dominante] e per questo i bit sono divisi in quanti/segmenti temporali (regolabili in base a bitrate e condizioni di rete).

▲ La rete CAN può essere ad alta velocità o a bassa velocità (ma tollerante ai guasti)

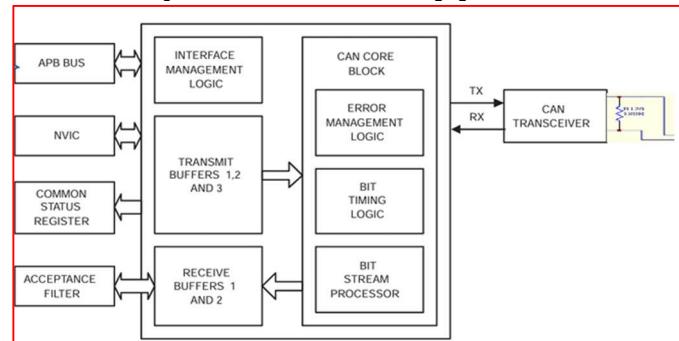
Parlando di frame (trame) CAN ci sono **2 tipi di messaggio**: **Base** frame (11 bit di identificatore) e **Extended** frame (29 bit). Ci sono **4 tipi di frame**: **Data** frame (trasmissione dei dati), **Remote** frame (richiesta di dati; non contiene il campo DATA), **Error** frame (segnalazione errori), **Overload** frame (introduce ritardi tra frame).

▲ Dato che Data frame ha bit RTR = dominante[0], vince su Remote frame con RTR = recessivo[1]

Il frame è composto da diverse parti: **Start of frame**, **Arbitration** (c'è l'identifier che indica la priorità del messaggio [il nodo con ID < (ovvero con valore binario <) ha priorità >]; così si gestisce il Multi-Master), **Control**, **Data**, **CRC** (verifica dei dati; deve essere coerente con DLC [lunghezza dei dati]), **ACK** (TX invia recessivo[1]; RX conferma messaggio inviando dominante[0] se corretto, mentre se CRC non corrisponde invia recessivo[1]; se ACK errato, viene ritrasmesso) e **End of frame**.

**La nostra scheda LandTiger LPC1768 include un controller CAN integrato:**

- **APB Bus** = fornisce accesso ai registri del controller CAN
- **Interface Management Logic** = interpreta comandi della CPU e fornisce indirizzamento interno dei registri e gestione degli interrupt
- **BSP** (Bit Stream Processor) = controlla flusso dati tra buffer e bus; rileva errori, arbitraggio e sincronizzazione (spiegata sopra nel dettaglio, sia start sia risincronizzazione)
- **TXB** (Triple Transmit Buffer) = memorizza intero messaggio da trasmettere; la CPU scrive nel buffer, che viene letto dal BSP per la trasmissione. Permette la gestione di dati fino ad un massimo di 8 Byte per frame
- **RXB** (Double Receive Buffer) = memorizza messaggi ricevuti; permette alla CPU di elaborare un messaggio mentre un altro viene ricevuto
- **CAN Controller** = modulo completo con TXB e RXB; non ha **filtro di accettazione** (messo in blocco a parte)



Nelle slide c'è l'elenco dei **registri del CAN** (vedi lì se serve). Qui cito Mode register, Command register, Global Status register, Interrupt Enable register, Interrupt & Capture register e Configuration register.

Il **filtro di accettazione** invece controlla quali messaggi ricevuti devono essere elaborati dal controller (basato su una **tavella di ricerca** [RAM dedicata per gestire gli identificatori CAM] configurabile via software; se non trova corrispondenze il messaggio viene scartato). Può funzionare in **accettazione completa** (filtro globale) o **filtraggio selettivo** (su base di intervalli o identificatori specifici) (filtro ristretto).

I canali CAN1 e CAN2 di una scheda sono collegati internamente in **loopback** per testing; 2 schede invece sono collegate in modo **incrociato** (Scheda1[CAN1]→Scheda2[CAN2]; Scheda1[CAN2]→Scheda2[CAN1]) consentendo comunicazione bidirezionale.

Dato che all'avvio del sistema i valori della tabella di ricerca sono casuali (RAM), bisogna usare **CAN\_setup** e **CAN\_start** (c'è una panoramica di **CAN.h** nelle slide). La gestione delle interruzioni (**CAN\_IRQ**) durante la ricezione avviene così: lettura del valore del messaggio ricevuto, conversione in coordinate e conteggio delle trasmissioni completate.