**DOMANDE DI CSD**

# INTERNAL FORWARDING

- ***Differenza tra internal forwarding e operand forwarding***

  Sono entrambe tecniche implementabili per risolvere i data hazard, in particolare quelli del tipo READ AFTER WRITE. La tecnica più semplice è quella dell'operand forwarding che prevede di modificare il datapath del processore modificando l'architettura interna dell'ALU in modo che l'uscita dell'ALU oltre ad andare agli stadi successivi vengono messi in dei registri buffer di appoggio. In particolare poi in ingresso all'ALU avremo un multiplexer che dirà se caricare il dato proveniente dallo stadio precedente o presente del buffer di appoggio dell'operand forwarding. In questo modo se abbiamo una read after write del tipo.
  I: R1=R2+R3
  I+1: R4= R1+R3
  Sarà compito dell'unita di controllo capire che siamo in presenza di un data hazard e selezionare opportunamente il multiplexer con l'operando R1 retroazionato. E' anche chiaro che il forwarding va portato fino a 3 istruzioni successive.
  Con Operand forwarding (con ibernation table) si intende una tecnica diversa che prevede nel caso di data hazard di scrivere i valori correnti temporanei all'interno di una copia dei registri che andremo poi a finalizzare non appena il conflitto si risolve.
- ***Internal forwarding come funziona***

  L'internal forwading prevede le seguenti 3 cose:
  - Una tabella delle istruzioni da ibernare, detta tabella di ibernazione
  - Un banco di registri uguale a quello del processore, con l'unica differenza che invece di contenere i valori conterrà i puntatori ai registri temporanei.
  - Un banco di registri temporanei che hanno come informazioni i valori temporanei, ad esempio all'interno di questo banco sono presenti diverse versioni del registro R1.

  La tabella di ibernazioni prevede sulle righe le nostre istruzioni da ibernare e per ogni istruzione sulle colonne troviamo: Operando1, operando2, risultato.
  Quando iberniamo un istruzioni andremo a scrivere in operando1 il tag che punta al registro temporaneo che presenterà la corretta versione del nostro operando. Stessa cosa per operando2 e risultato, successivamente quando andremo a scongelare e svolgere la nostra istruzioni punteremo agli operandi indicati nella riga e scriveremo il risultato dove indicato. Solo alla fine di tutto i data hazard (quando un particolare operando non ha alcuna istruzioni ibernata associata) renderemo definitivo quel registro).

- ***Internal Forwarding (con esempio)***
- ***Cosa succede se ho tanti registri operando quanti registri R vantaggi e svantaggi. (non sicura)***

  L'internal forwarding per funzionare bene prevede che il numero di registri operando sia maggiore del numero di registri R del processore, in modo tale che riesco a memorizzare più versioni dello stesso registro. E' chiaro che se il numero di operand register è pari a quello dei registri ho lo svantaggio che nel tentativo di salvare un registro temporaneo nuovo ho pià probabilità che non trovo spazio

- ***Perché l'internal forwarding è meglio rispetto ad avere il registro R2 ed una copia del registro R2old?***

# MIPS

Per i **modi di indirizzamento** abbiamo:

- M68K 6 modi principali più varianti, per un totale di 14 modi diversi
- MIPS solo tre
- AEM ne ha 3 più
  - Pc-relative: utilizzabile dalla load per caricare costanti salvate in area programma.
  - Pre-indexed: il registro base viene automaticamente incrementato prima di accedere alla memoria (++i).
  - Post-indexed: il registro base viene automaticamente incrementato dopo l'accesso alla memoria (i++).

- ***istruzioni del MIPS***

  Le istruzioni sono a lunghezza fissa 32 bit e prevedono tutte indirizzamento su registri, eccetto LOAD e STORE che sono le uniche che indirizzano la memoria. Abbiamo 3 formati di istruzioni e differiscono l'una dall'altra per il numero di operandi.
  ISTRUZIONE DI TIPO R:
  Formato: OP rs rt rd shamt funct
  6 5 5 5 5 6
  OP rappresenta il codice operativo ed è sempre presente in tutti e 3 i formati su 6 bit, poi abbiamo rs e rt operandi sorgenti su 5 bit e rd operando destinazione, shamt ha senso solo in presenza di un istruzione shift e ci dice di quanto shiftare e infine funct ci dice quale versione di quel codice operativo eseguire (es. con ADD abbiamo ADD signed, ADD unsigned ecc.)
  Formato usato principalmente per istruzione aritmentiche ma anche per movimentazione dati come la MOVE.
  Es. move rdest,rsorg (pseudo-istruzione) si traduce con add rdest,rsorg,0

  ISTRUZIONE DI TIPO I
  Formato: op rs rt imm
  6 5 5 16
  Usato per inserire un immediato all'interno del codice operativo. Siccome l'immediato è codificato su 16 bit per inserire immediati più grandi è necessario sdoppiare l'istruzione e caricare prima la metà alta dell'immediato e poi la metà bassa.
  Formato usare per codificare i salti condizionati, es BNE r1,r2,1234
  Con 1234 si indica l'offset rispetto al PC e non l'indirizzo effettivo. Essendo l'immediato su 16 bit possiamo JUMPARE in questo modo fino a 2^16 istruzioni successive al PC.
  Possiamo fare anche LOAD e STORE

ISTRUZIONE DI TIPO J
Formato: op offset
6 26
Usato per i JMP incondizionati, abbia un offset esprimibile su 26 bit ovvero una finestra massima di salto, se la destinazione esce fuori da questa finestra è necessario mettere l'indirizzo a cui vogliamo saltare in un registro e poi utilizzare il modo di indirizzamento con base e spiazzamento.
C'è inoltre la possibilità di invocare le systemcall

- ***programma in MIPS e 68k a paragone***

  In generale nel MIPS troviamo i 3 tipi di formati R I J, tutte le operazioni vengono fatte tra registri eccetto LOAD e STORE, non esiste l'indirizzamento diretto per accedere in memoria (es. nel 68k troviamo MOVE.W $E506, D0 nel MIPS sarebbe impossibile ma dobbiamo caricare il valore E5O6 in un registro e poi usare LOAD)
  Abbiamo nel MIPS dei registri appositi per memorizzare l'indirizzo di ritorno da una subroutine (si chiama con jal) e registri per passare i parametri di una rubroutine

- ***come sono gestite le eccezioni nel MIPS e quali sono?***

Il MIPS utilizza un coprocessore dedicato a gestire le interruzioni. Il MIPS divide le situazioni di interruzione in causa interna e causa esterna. Cause interne: accesso in memoria indirizzo illegale o operazioni arimentiche illegali (N.B. divisione per 0 non scatena una trap ma il risultato sarà NaN Not a number)
cause esterne: interruzioni da periferiche esterne ma anche dal processore che si occupa delle operazioni floating point.
Per la gestione, nel 68k veniva fatta in hardware con una tabella dei vettori, e in base alla causa veniva associato un identificativo (vettorizzata o autovettorizzata) che permetteva di andare nella tabella e selezionare la giusta ISR da eseguire (TUTTO FATTO IN HARDWARE). Nel MIPS è completamente diverso vengono gestite via SOFTWARE tutte le cose hardware fatte dal processore sono di settaggio dei registri, abbiamo un unico exception handler, ovvero qualunque sia la causa di eccezione viene invocato lo stesso programma SOFTWARE. La routine exception handler sarà abbastanza complessa perchè dovrà essere in grado di capire la causa dell'interruzione e gestirla opportunamente (concettualmente uno switch case), per fare la routine guarda dei registri specifici del coprocessore 0 che danno informazioni su ciò che è successo nel sistema.
Abbiamo un registro causa che codifica il tipo di eccezione che si è avuto in quel momento. quando succede qualcosa nel sistema il processore identifica la causa e la scrive codificata nel registro apposito, dopo aver salvato la causa, salva l'indirizzo di ritorno dopo che l'interruzione è stata gestita e poi salta all'exception handler.Il processore non fara un salvataggio di contesto.
Abbiamo quindi 2 registri: CAUSE E STATUS register.
in CAUSE troviamo la codifica delle interruzioni pendenti e il bit di branch delay.
in STATUS troviamo la maschera delle interruzioni, user MODE
Se esiste il concetto di priorità, questa sarà gestista dall'exception handler.

- ***Perché abbiamo diversi tipi di istruzione nel MIPS e cosa ci portiamo appresso delle varie istruzioni nella pipeline del MIPS***
- ***pseudo istruzioni-istruzioni mips***

  Il MIPS prevede l'utilizzo di pseudo-istruzioni ovvero sono disponili al programmatore dei codici operativi che in realtà non sono delle istruzioni assemblabili direttamente in codice macchina ma vengono prima tradotte in altre istruzioni che poi saranno quelle effettive.
  Un primo esempio è move r1,r2. In realtà il MIPS non prevede un istruzioni di questo tipo, ma per semplicità viene messa lo stesso a disposizione del programmatore, poi questa istruzioni verrà tradotto nell'istruzione giusta ovvero in questo caso add r1,r2,0.

# BRANCH PREDICTION

In presenza di un control hazard, abbiamo vari modi per affrontarlo:

1) APPROCCIO CONSERVATIVO, appena capiamo che un'istruzione è di salto (alla fase ID) fermiamo la pipe (disabilitiamo propagazione istruzioni) e aspettiamo che venga determinato l'indirizzo dell'istruzione a cui saltare (perdiamo tutti i vantaggi del pipeline, molto lenta)
2) BRANCH DELAY: si tratta di accorgimenti in fase di compilazione che evitano completamente alcuni control hazard, es: invertiamo l'ordine di 2 istruzioni per evitare il problema. N.B.Tutto questo a patto che modificare l'ordine di esecuzione delle istruzioni non vada a cambiare lo stato di esecuzione del processore.
3) BRANCH PREDICTION ( approccio ottimistico): si cerca di prevedere il ramo del salto e se si sbaglia si corregge. In particolare si prevede che in parallelo alla fase di fetch ci sia una tabella ( memoria full associative) che contiene gli indirizzi delle due potenziali istruzioni successive al salto e ci indica anche quale dei due prendere.

- ***su che fase del processore impatta?***

Impatta sulla fase di fetch, in particolare si prevede una modifica dell'architettura realizzando un blocco di pre-fetch che ha al suo interno la branch prediction table.
In particolare prima di fare il fetch dell'istruzione controlliamo che il suo indirizzo sia nel brench pred. table se lo è allora non passiamo al fetch dalla memoria ma al blocco di pre-fetch dando in pasto alla pipeline quello che troviamo nella table.
es. istruzione $80 è di salto e la prox istruzione può essere 84 o 112. Prima di fare il fetch vediamo che $80 è presente nella brench table con indirizzo associato 112, troviamo il bit taken alto ed allora eseguiamo direttamente la 112

- ***che tipo di memoria usa?***

Come spiegato precedentemente la brench prediction table è una memoria SRAM full associative dove facciamo una ricerca in parallelo usando come chiave l'indirizzo dell'istruzione in questione

- **branch prediction: automa a 2 fasi e automa a 4 fasi differenze**

Possiamo realizzare la branch prediction a 2 stadi o a 4 stadi:

1) 2 STATI: 1 solo bit nella table (TAKEN, NOT TAKEN), l'automa prevede di default di iniziare in taken poi successivamente se sbagliamo la previsione andiamo in NOT TAKEN, se indoviniamo la previsione restiamo in TAKEN e viceversa.
   Funziona bene ad esempio con un ciclo for con 100 iterazione dove al massimo sbagliamo 2 volte: la prima volta che controlliamo la condizione che è alla cieca e l'ultima volta quando dobbiamo uscire dal ciclo.
   Questo automa ha però problemi con un doppio ciclo for innestato(es.10x10): perchè possiamo sbagliamo previsione sia le 100 volte in cui entriamo nel ciclo interno e le 10 volte in cui cerchiamo di uscire.
2) 4 STATI: per risolvere il precedente problema si utilizzano due bit nella table e si hanno 4 stati (STRONG TAKEN, LIGHT TAKEN, STRONG NOT TAKEN, LIGHT NOT TAKEN). L'automa funziona nel seguente modo:
   Si parte ad esempio da LIGHT TAKEN, se indoviniamo andiamo in STRONG TAKEN, poi successivamente fintantoché indoviniamo restiamo in STRONG TAKEN ma quando sbagliamo andiamo in LIGHT TAKEN, ora se sbagliamo ancora andiamo in LIGHT NOT TAKEN, e se sbagliamo la terza volta consecutiva andiamo in STRONG NOT TAKEN.
   In questo modo risolviamo il problema del doppio for innestato. Perchè potremmo sbagliare solo la prima e ultima volta che accediamo al for esterno e quando dobbiamo uscire dal ciclo interno.
   N.B. Conviene sempre mettere all'interno il ciclo con numero maggiore (io mi trovo minore ma sugli appunti è scritto cosi) di iterazioni perché così sbaglieremo meno volte le uscite dal ciclo interno e quindi faremo meno errori di valutazione

# VETTORE DELLE COLLISIONI

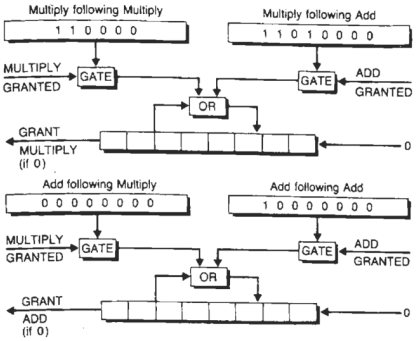
- ***vettore delle collisioni funzionamento***

Parliamo di architetture superscalari, ovvero che hanno all'interno più pipeline in modo da aumentare il throughput. c'è, quindi, la possibilità di mandare in esecuzione più di un'istruzione contemporaneamente replicando opportunamente l'hardware e verificando che non ci siano conflitti o incoerenza del processore.
Ora se le due pipe sono completamente disgiunte, esse non condividono fasi o risorse hardware e quindi non ci sono problemi. Ma se non è cosi significa che due istruzioni possono accedere contemporaneamente ad una risorsa condivisa che può essere la memoria, il bus o uno stadio dell'alu.
Siccome avere pipe disgiunte significa replicare completamente tutte le risorse, cerchiamo di usare pipe non disgiunte ma assicurandoci che non ci siano conflitti, per fare questo si usa il vettore delle collisioni. L'accesso al bus o alla memoria si può gestire tramite protocolli interni. Più complicato è l'accesso all'ALU.
In generale l'ALU è composta da più stadi ed un' istruzione attraversa un sottoinsime di questi stadi. Per essere sicuri che due istruzioni non collidono dobbiamo costruire le cosiddette maschere di collisioni che ci indicano per ogni istante di tempo l'istruzione quale stadio dell'ALU occuperà.
es. ADD OP1, OP2
    MUL OP3, OP4
Costruiamo le due maschere e poi le sovrapponiamo, se in almeno un istante di tempo notiamo che le due istruzioni occupano lo stesso stadio dell'ALU allora c'è un conflitto e mettiamo un 1 nel vettore delle collisioni, poi shiftiamo la seconda istruzioni ( come se la ritardassimo di un quanto di tempo) e ripetiamo l'operazione di sovrapposizione maschere, in questo modo quando c'è una collisione mettiamo 1,

quando non c'è 0. In questo modo se abbiamo come vettore 110110, sappiamo che la moltiplicazione possiamo eseguirla o ritardata di 3 oppure di 6 senza avere conflitti. N.B Non vale la proprietà commutativa, non si verificano gli stessi conflitti cambiando l'ordine delle operazioni, il vettore di ADD MUL è diverso da quello MUL ADD.

- ***se abbiamo 4 codici operativi quanti vettori delle collisioni possiamo avere?***
  Dobbiamo considerare tutte le possibili coppie in entrambi i versi.
  4 codici operativi: A B C D
  AA AB AC AD
  BA BB BC BD
  CA CB CC CD
  DA DB DC DD
  Sono quindi 16 vettori delle collisioni, in generale n*n (4x4=16)
- ***perchè abbiamo l'operazione di OR nel vettore delle collisioni, cosa significa che abbiamo trovato due valori pari a 1?***

La generazione di tutti i possibili vettori delle collisioni sarà compito del compilatore, sarà poi un blocco di prefetch a valutare se date 2 istruzioni posso mandarle su pipe diverse senza generare conflitti.




Abbiamo 2 possibili istruzioni, quindi 4 vettori
Supponiamo di avere un'operazione precedente di tipo MUL che è stata abilitata (multiply granted). Quello che si fa con il blocco OR è un OR bit a bit tra quello che è l'attuale vettore delle collisioni delle operazioni analizzate prima e l'attuale operazione che vogliamo fare, in questo modo avremo il reale vettore delle collisioni ed un'effettiva garanzia dell'operazione di add che stiamo richiedendo, soltanto se il risultato dell'OR è 0. Se è così abilitiamo il caricamento di una ADD e shiftiamo il vettore delle collisioni finale mettendo uno 0 sulla destra.
Riassumendo tutto il comportamento totale:
Vogliamo carica una ADD sulla seconda pipe prima di andarci vediamo cosa sta facendo la prima pipe per capire se la possiamo mandare subito in esecuzione l'add oppure se abbiamo una collisioni e dobbiamo ritardarla, dobbiamo vedere il vettore

delle collisioni che sta sotto, il quale viene abilitato in funzione dell'operazione precedente. Se l'operazione precedente era una moltiplicazione vediamo il vettore add following multipliy, se era un'addizione vediamo il vettore add following add, lo mettiamo in OR bit a bit con l'attuale vettore delle collisioni e decidiamo se abilitare o meno l'add corrispondente. Questa è l'architettura del modulo i prefetching usato dalle pipeline a partire dal Pentium 1 per gestire collisioni nel caso in cui si vogliono usare architetture superscalari. L'hw diventa più complesso.

## INTERRUZIONI

- ***interruzioni precise: perché sono utili***

***https://www.docenti.unina.it/webdocenti-be/allegati/materiale-didattico/76148***
***https://communitystudentiunina-my.sharepoint.com/:b:/g/personal/cir_marrazzo_studenti_unina_it/ES_V1XV1dmBGpVh2njoipTIB8jbnxGu7Wfqqv1jCmG5mvA?e=MrN3dF***

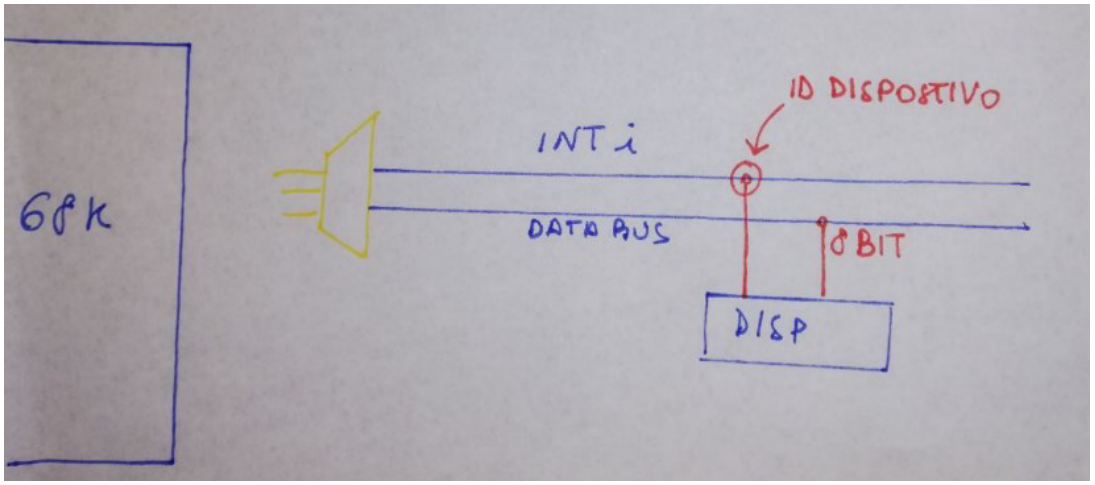
- Una interruzione è precisa se:
  - il PC è salvato in un posto noto,
  - TUTTE le istruzioni precedenti a quella puntata dal PC sono state eseguite COMPLETAMENTE,
  - NESSUNA istruzione successiva a quella puntata dal PC è stata eseguita,
  - lo stato dell'esecuzione dell'istruzione puntata dal PC è noto.
- Se una macchina ha interruzioni imprecise:
  - è difficile riprendere esattamente l'esecuzione in hardware,
  - la CPU riversa tutto lo stato interno sullo stack e lascia che sia il SO a capire cosa deve essere fatto ancora,
  - rallenta la ricezione dell'interrupt e il ripristino dell'esecuzione $\Rightarrow$ grandi latenze...

- Avere interruzioni precise è complesso:
  - la CPU deve tenere traccia dello stato interno: hardware complesso, meno spazio per cache e registri,
  - "svuotare" le pipeline prima di servire l'interrupt: aumenta la latenza, entrano bolle (meglio avere pipeline corte).
- Pentium Pro e successivi, PowerPC, AMD K6-II, UltraSPARC, Alpha hanno interrupt precisi, mentre IBM 360 ha interrupt imprecisi.

- ***vettorizzate vs auto-vettorizzate***

Il nostro obiettivo è fornire l'identificativo della causa interrompente in modo da individuare la ISR giusta da eseguire:
AUTOVETTORIZZATA: utilizziamo il livello della interruzione per identificare in maniera univoca la ISR e il dispositivo. es:
Il 68k ha tre linee di interruzione connesse ad un decoder che permette di decodificare 8 livelli di interruzioni, ora se il dispositivo alza il segnale di INT (es. INT3) in automatico verrà identificato con 3 ed il decoder fornisce 011, poi andiamo nella table e con l'offset troviamo la ISR associata.

VETTORIZZATA: Con questa tecnica il dispositivo quando genera un'interruzione, da un alto alza la linea INT e questa andra al decoder, dall'altro manderà sul bus dati un vettore di 8 bit che serve per identificare la ISR




## *SCHEDINO:*

- ***Interruzioni***

L'ARM non distingue tra i diversi tipi di interruzioni ma le tratta tutte allo stesso modo tramite 2 intermediari (Interrupt controller) GIC e NVIC.
Si fa uso di una Interrupt vector table.
Per gestire un interruzione si fanno i seguenti passi:

1) Salvataggio contesto, in hardware e indirizzo di ritorno.
2) Si va nella interrupt tablet e tramite l'identificativo fornito al PIC capisce quale ISR eseguire.
3) Al termine della ISR ripristina lo stato del processore

- ***Come farei il progetto dello schedino in ASIM***
- ***Come gestire mutua esclusione (chiesto molte volte)***

L'equivalente del TAS in C non si trova perché non esiste. ARM ce l'ha? Sì, ma dobbiamo scendere tantissimo di livello per usarla.

- ***Come funziona lo scheduling (chiesto molte volte)***
- ***Organizzazione architettura schedino***
- ***Descrizione di scrittura programma sullo schedino usando un timer per alzare e abbassare un'asta***

## 6.4 Progetto 8: LED lampeggianti con uso delle interruzioni

Vogliamo ancora che i LED lampeggino, ma questa volta sfruttando il meccanismo delle interruzioni.

L'esercizio si può fare in tanti modi e tante varianti, nel nostro caso realizzeremo un programma che fa accendere e spegnere i LED ogni mezzo secondo (ma si potrebbe anche, in maniera leggermente diversa, implementare un algoritmo che ogni mezzo secondo accenda i LED ma li tiene accesi solo per un periodo molto piccolo di tempo, magari gestendo l'attesa semplicemente con un ciclo for).

La prima differenza rispetto alla versione senza interruzioni è che bisogna abilitare le interruzioni del timer TIM2 alzando il bit UIE (*Update Interrupt Enable*) del registro DIER.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Res. | TDE | Res. | CC4DE | CC3DE | CC2DE | CC1DE | UDE | Res. | TIE | Res. | CC4IE | CC3IE | CC2IE | CC1IE | UIE |
| | rw | | rw | rw | rw | rw | rw | | rw | | rw | rw | rw | rw | rw |

Abbiamo già visto che la ISR relativa al timer è la numero 28, quindi bisogna alzare il 28° bit del NVIC_ISER [0], così che venga eseguito il 28° sottoprogramma. Dobbiamo a questo punto realizzare l'effettiva ISR, e lo facciamo mediante una funzione che ha lo stesso nome dell'interruzione: **TIM2_IRQHandler**.

| 28 | 35 | settable | TIM2 |
|---|---|---|---|

```
DCD     TIM2_IRQHandler
```

```
        PUBWEAK TIM2_IRQHandler
        SECTION .text:CODE:REORDER:NOROOT(1)
TIM2_IRQHandler
        B TIM2_IRQHandler
```

```
void TIM2_IRQHandler(){
  TIM2->SR &= ~1;
  if(flag==0){ GPIOE->ODR |= 0X0000FF00; flag=1;}
  else {GPIOE->ODR = 0; flag=0;}
}
```

Nel nostro caso, quando raggiunge mezzo secondo scatta UIF, e quindi UIE, e viene eseguita la nostra funzione handler, che non fa altro che abbassare l'UIF e accendere/spegnere i LED. Sfruttiamo un flag, che deve essere dichiarato globalmente per essere visibile dall'handler.

L'ultima cosa che rimane da fare è aggiornare l'ARR, poiché ti ricordo che nel system_stm32f30x.c è stata modificata la frequenza di clock. Quindi se prima in mezzo secondo c'erano 4 milioni di conteggi, ora ce ne sono ben 4x9=36 milioni.

➔ **Il valore di TIM2_ARR è 36.000.000**

- ***I2C funzionamento***

Il protocollo di comunicazione **I²C**, scritto anche come I2C, è un protocollo di comunicazione seriale sincrono creato da Philips che permette a molti dispositivi di scambiarsi dati fra loro utilizzando solo due fili, uno per i dati ed uno per sincronizzare la comunicazione (clock).
Nella comunicazione seriale, il master controllerà sempre il segnale SCL e scambierà i dati con lo slave tramite il segnale SDA. Una comunicazione avviene all'interno di due sequenze di segnali che il master invierà sul bus I²C, queste sono il comando di START e di STOP

START: Prima di inviare il segnale di start, lo stato del bus I²C si trova nella condizione in cui entrambi i segnali SDA ed SCL sono al livello logico alto. Per inviare il comando di start, il master pone a livello basso il segnale SDA mentre SCL è a livello logico alto, così il comando di start ( S ) viene letto sul fronte di discesa del segnale SDA.

In questo modo segnala agli altri dispositivi che sta per iniziare una comunicazione.

STOP: Prima di inviare il segnale di stop, sono stati trasferiti dei dati per cui lo stato del bus I²C si trova nella condizione in cui entrambi i segnali SDA ed SCL sono al livello logico basso. Per inviare il comando di stop, il master pone prima a livello alto il segnale SCL e poi il segnale SDA, così il comando di stop ( P ) viene letto sul fronte di salita del segnale SDA.
In questo modo segnala allo slave che la comunicazione e terminata e scollegandosi dal bus.I²C questo ritorna con entrambe le linee SDA ed SCL a livello logico alto.

- ***I2C può lavorare con una frequenza di clock elevata?***

Lo standard per la velocità di clock del bus è di 10 KHz e di 100 KHz, ma esistono anche varianti a 400 KHz e 3.4 MHz. Il bello è che queste sono i limiti superiori della velocità. Non esiste un limite inferiore quindi è possibile ad esempio impostare un clock di 1 Hz

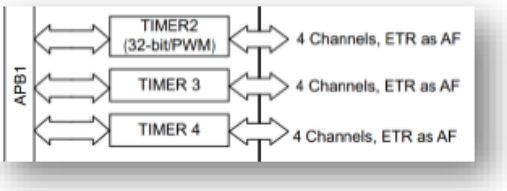
- ***se avessi collegato più sensori su un filo che protocollo avrei utilizzato (i2c)***

A questo bus possono essere collegati fino a 128 dispositivi, che potranno comunicare tra loro assumendo di volta in volta uno il ruolo di master ed uno il ruolo di slave.
Sul bus I2C possono esserci fino ad 8 dispositivi collegati, il master è quello che inizia la trasmissione attraverso il comando di START, ma come fa a stabilire qual'è lo slave con cui comunicare? Lo fa attraverso l'invio di una sequenza di 8 bit di cui i primi 7 rappresentano il device address, a cui uno ed uno solo degli altri dispositivi risponderà. Il device address di un dispositivo collegato al bus deve essere univoco,

- ***usart***
- ***timer e interruzioni***

## 5.1 Timer general-purpose

I timer general-purpose TIM2, TIM3 e TIM4 sono contatori rispettivamente a 32, 16 e 16 bit.




L'unità time-base include:

- TIMx_**CNT** (*Counter Register*)
- TIMx_**PSC** (*Prescaler Register*)
- TIMx_**ARR** (*Auto-Reload Register*)


Il **CNT** contiene il valore del conteggio, che comincia a contare quando viene abilitato.

L'**ARR** è precaricato con un certo valore, che rappresenta il numero di conteggi da effettuare prima di azzerare il contatore. Quando si modifica l'ARR, il suo contenuto può venir letto all'evento di update (alla fine del ciclo) o immediatamente, a seconda se il registro è bufferizzato o meno.

Il **PSC** serve per dividere la frequenza di clock, qualora fosse troppo alta. $T_{PSC} = (PSC + 1) \cdot T_{ck}$

- ***Se hai un dispositivo i2c, puoi mettere un altro dispositivo dello stesso modello sulla stessa linea i2c?***

Si è possibile, perché una parte dell'identificativo è assegnato dal produttore del dispositivo ma c'è un altra parte che è programmabile, e quindi su quest'ultima possiamo scrivere 2 valori diversi e i dispositivi avranno 2 identificativi diversi.

## MEMORIE E CACHE

- ***che pro ha una memoria con alto grado di associatività?***

Nel dimensionamento di una cache i parametri in gioco sono i seguenti:
D = dimensione totale della cache
L = lunghezza del blocco (es.4 word da 16 bit quindi 64bit)
N (chiamato anche S dal prof) = numero di set
K = grado di associatività (numero di linee per set)
E' chiaro che D = k*L*N. Questa equazione non è risolvibile, perchè, assegnata una certa dimensione D, avrebbe 3 incognite. Per cui questa equazione va risolta utilizzando un approccio simulativo/tecnologico.
Di solito L viene fissata da banda di memoria e dalla politica di write: se ho Wtorugh conviene L piccolo se ho Wback conviene L grande). K di base determinando il numero di comparatori è limitata da vicnoli tecnologici.
Tornando alla domanda un K grande ha vari vantaggi vediamo perchè:
Nella fase di scrittura un determinato blocco di memoria può appartenere solo ad un determinato insieme ( tramite funzione modulo si calcola) poi successivamente dobbiamo cercare una linea vuota in quel set, se non la troviamo dobbiamo effettuare un operazione di swap. Quindi avendo un k basso può capitare che anche se la cache è praticamente vuota, troviamo pieno il nostro insieme e dobbiamo fare una swap. Aumentando il grado di associatività questo problema si abbassa e il comportamento tende più ad una full associative.
Fare operazioni di swap non è una buona cosa sia per il costo temporale, ma dobbiamo anche decidere una politica di swap ( il blocco sacrificato potrebbe servirci tra poco) è chiaro anche come avere un k alto fa ridurre la percentuale di cache miss.

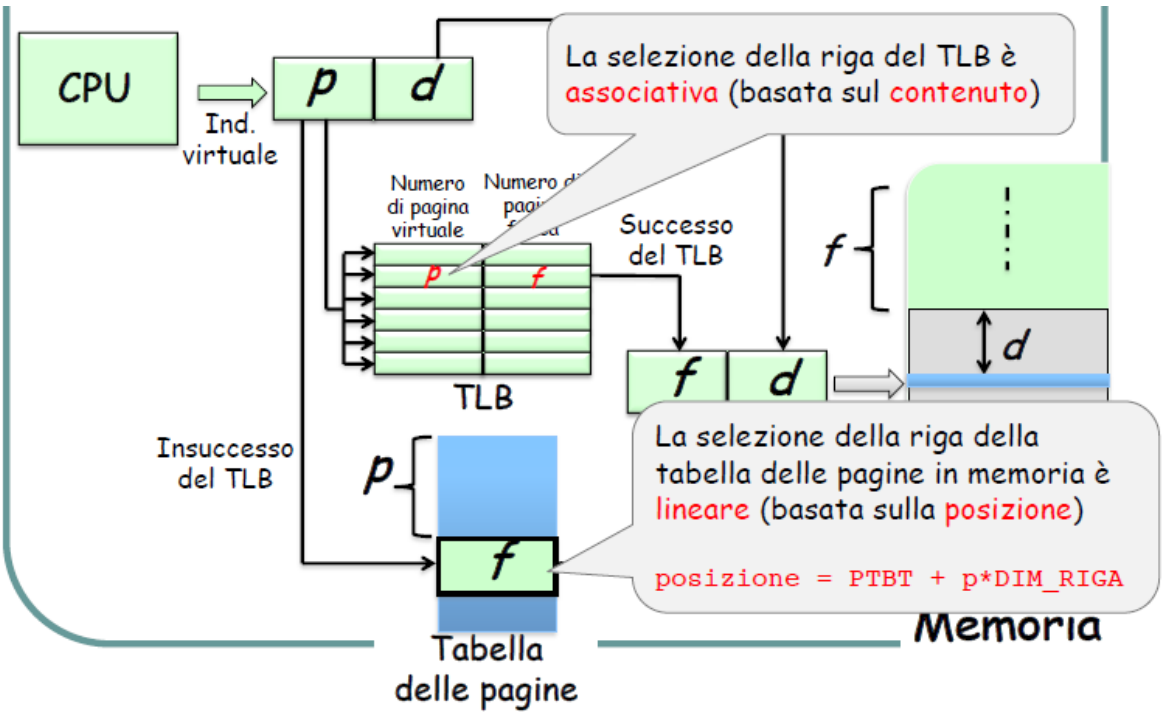
- ***disegno di memoria full-associative con grado di associatività 16***
- ***disegno di memoria set associative 2 settori e grado di associatività***

- ***Nella catena il processore lavora ad indirizzi logici, come diventano indirizzi fisici? Se lavori con la virtual cache che tipo di indirizzi hai? Se la label logica sta nella cache che succede e se non ci sta?***

Tutti i programmi ed i dati sono organizzati in unità di lunghezza predefinita: le pagine. Le pagine rappresentano l'unità minima di informazioni che viene spostata dalla memoria secondaria alla memoria principale.
C'è una distinzione tra indirizzi virtuali che sono prodotti dal programma durante la sua esecuzione e fisici che sono gli indirizzi effettivi in memoria.

Esiste un componente(hardware) della CPU chiamato MMU che si occupa di tradurre gli indirizzi virtuali in indirizzi fisici.
Nella traduzione l'indirizzo virtuale è diviso in Virtual page number e offset, il page number serve come chiave per indirizzare la pagina all'interno della tabella delle pagine (page table address + virtual page number) e poi l'offset serve per indirizzare il blocco che ci interessa all'interno della pagina. Il problema è che ho una tabella che contiene tutti gli indirizzi virtuali da considerare, inoltre se la pagina non è stata ancora caricata in memoria abbiamo un page fault (troviamo nella tabella il bit di validità non attivo) e dobbiamo andare in memoria secondaria (Hard Disk).
Ovviamente la tabella in questione è allocata in memoria quindi ogni accesso porta una perdita di efficienza (raddoppiano gli accessi in memoria), per ovviare a ciò la MMU prevede una cache full associative chiamata TLB, che contiene una porzione della tabella (si comporta come una normale cache).
Quindi ricapitolando tutto:

CASO 1: PAGE HIT in TLB
Abbiamo un indirizzo virtuale suddiviso in page number + offset -> si fa una ricerca full associative nel TLB -> se troviamo la page allora abbiamo un page HIT -> abbiamo l'indirizzo fisico
CASE 2: PAGE MISS in TLB e Page FAULTin table -> IL TUTTO SI CHIAMA PAGE FAULT
Abbiamo un indirizzo virtuale suddiviso in page number + offset -> si fa una ricerca full associative nel TLB -> non troviamo la page allora abbiamo un page miss -> andiamo in memoria nella table ma non lo troviamo, abbiamo un page fault e quindi dobbiamo andare prima nell'hard disk.
CASO 3: PAGE MISS in TLB ma Page HIT in table -
Abbiamo un indirizzo virtuale suddiviso in page number + offset -> si fa una ricerca full associative nel TLB -> non troviamo la page allora abbiamo un page miss -> andiamo in memoria nella table e lo troviamo, abbiamo un page hit e troviamo l'indirizzo virtuale
N.B. nell'indirizzo virtuale la chiave (virtual page number) è la stessa che si usa nel TLB per la ricerca associativa, nella page table come offset per trovare la pagina giusta.




- ***Non ci può mai essere un cache miss e un page fault insieme. Se il blocco è più grande della pagina , non può comunque accadere. Perché?***
- ***Differenza tra cache miss e cache hit***

Spiegata bene sopra

- ***Se tu hai un page hit e un cache miss che cosa vuol dire? Se hai un cache miss e un page miss che cosa vuol dire?***

Spiegata bene sopra

- ***dimensionamento cache k=4 s=2 e cosa cambia se s=4 vantaggi e svantaggi***

Se a parità di dimensione totale della cache passo da k=4 s =2 a K=4 s = 4.
Ho aumentato il numero di insiemi, quindi sicuramente se k è invariato avrò una lunghezza del blocco minore.
Per rispondere bene a questa domanda bisogna capire se K resta pari a 4 oppure varia.

- ***Virtual cache + cache miss***

Spiegato bene prima

- ***tabella cache miss e page fault:se e come cambierebbe nel caso usassi pagine di dimensione più piccole dei blocchi,***

- ***Simulazione delle cache e analisi delle tracce***
- ***Esempio architettura di una cache set associativa***
- ***le callback come vengono mappate in memoria dinamico o statico***
- ***Cos'è il refresh periodico nelle DRAM?***
- ***Coerenza nella memoria: automi write through e write back, perchè uno degli automi ha 2 stati e l'altro 3***

Questa domanda fa parte sicuramente del capitolo multicomputer, perchè gli automi sono presenti nella gestione della memoria li. Qui spiego solo in generale la differenza tra write through e write back

E' un problema della strategia di scrittura. Il problema nasce dalla necessità che – quando si deve scrivere il risultato di un'operazione – si vuole certamente che anche l'istruzione di scrittura sia eseguita velocemente (e quindi accedendo alla cache), ma si vuole anche che l'informazione contenuta – in qualsiuasi istante – nella cache sia consistente con quella contenuta nella RAM.

Le possibili strategie per la gestione delle scritture sono writethrough e write-back

1) Nell'approccio write-through, quando si esegue un'istruzione di scrittura l'informazione viene scritta simultaneamente nel blocco della cache e nel blocco della memoria principale. La coerenza è quindi sempre rispettata, a prezzo però di un maggior tempo richiesto da ogni operazione di scrittura.
2) Nella soluzione write-back (o copy back), invece, al momento dell'esecuzione dell'istruzione l'informazione viene scritta solo nel blocco della cache. Il blocco modificato viene scritto nel livello inferiore della gerarchia solo quando se ne decide la sostituzione. Al termine della istruzione di scrittura nella cache, quindi, la memoria RAM conterrà un valore diverso da quello presente nella cache; in questo caso si dice che la memoria e la cache sono inconsistenti (cioè non sono coerenti): il blocco della cache può essere clean (unmodified) se non ci sono state scritture, oppure dirty (modified) se sono state effettuate scritture.

## ARCHITETTURE MULTICOMPUTER

- ***prodotto tra due vettori SIMD e MIMD (spimd è un caso speciale di mimd)***
- ***speed-up e efficienza differenze e quando conviene parallelizzare, con esempio***

## BUS E PROTOCOLLI

- ***Protocollo sul bus nel caso di interruzioni vettorizzate***

# VIRTUALIZZAZIONE

## ● Hypervisor e full/para virtualization

L'Hypervisor, anche noto come VMM (Virtual Machine Manager/Monitor) è un software di sistema che gestisce in maniera efficiente più macchine virtuali che coesistono contemporaneamente. In particolare, l'Hypervisor ha il compito di intercettare e gestire le chiamate che i SO Guest effettuano verso le risorse hardware della macchina fisica Host. La VMM inoltre deve effettuare il multiplexing (allocazione e scheduling) delle risorse fisiche, in modo appropriato, ed in particolare per garantire tre criteri fondamentali:

- Security and Reliability: vogliamo fare in modo che la VM sia completamente isolata rispetto alla altre VMs e soprattutto dal SO Host. Es. se la VM crasha, non vogliamo assolutamente che ci siano ripercussioni sulle altre macchine virtuali.
- Performance: si deve garantire alla VM le risorse che gli sono state allocate durante la configurazione. Il sistema virtuale deve quindi mostrare nel peggiore dei casi una lieve diminuzione delle prestazioni.
- Equivalenza: la VM deve essere completamente equivalente al computer sottostante.

Esistono due tipi principali di Hypervisor:

1) Bare-Metal: l'Hypervisor non viene eseguito su un sistema operativo host, ma deve comunicare direttamente con l'hardware fisico del sistema. L'HV quindi deve eseguire lo scheduling e l'allocazione REALE delle risorse. (i.e. Xen)
2) Hosted: l'Hypervisor opera su un sistema operativo host già esistente, quindi in questo caso ha solo il compito di creare e configurare le macchine virtuali. (i.e. VirtualBox, Parallels, VMware Fusion)

Detto ciò, la virtualizzazione può essere effettuata in due modi differenti:

1) Full-Virtualizzation: in questo caso la VM è completamente estranea della presenza di un SO Host e quindi le operazioni sensibili vengono tradotte in delle trap di sistema, catturate dall'Hypervisor ed emulate (Trap and emulate);
2) Para-Virtualizzation: nel secondo caso invece, la VM è a conoscenza della virtualizzazione e quindi effettua delle specifiche chiamate, dette Hypercalls, per utilizzare in maniera esplicita i servizi dell'Hypervisor.

Wikipedia:

**Full virtualization** was implemented in first-generation x86 VMMs. It relies on binary translation to trap and virtualize the execution of certain sensitive, non-virtualizable instructions. With this approach, critical instructions are discovered (statically or dynamically at run-time) and replaced with traps into the VMM to be emulated in software. Binary translation can incur a large performance overhead in comparison to a virtual machine running on natively virtualized architectures such as the IBM System/370. VirtualBox, VMware Workstation (for 32-bit guests only), and Microsoft Virtual PC, are well-known commercial implementations of full virtualization.

**Paravirtualization** is a technique in which the hypervisor provides an API and the OS of the guest virtual machine calls that API, requiring OS modifications.

- **Perché è meglio usare root/non-root rispetto al caso in cui non hai queste due modalità? (non sicuro)**

Con le due modalità operative Root/Non-Root abbiamo la possibilità di implementare il meccanismo del trap and emulate in maniera completamente automatica. Infatti l'Hypervisor, mentre si trova nella modalità Root può configurare a priori il funzionamento e le operazioni sensibili per la VM può eseguire. In questo modo, la VM può tranquillamente lavorare nella modalità Non-Root e se effettua una operazione sensibile, in automatico viene generata una trap software che provoca l'uscita (VM Exit) ed il cambio di contesto alla modalità Root. L'Hypervisor analizza la trap, elabora la risposta e la trasferisce alla VM (VM Entry).
Nel caso in cui queste due modalità non siano presenti, è obbligatorio emulare le istruzioni privilegiate. Ogni qualvolta una VM esegue una istruzione privilegiata, l'Hypervisor deve emularne il funzionamento effettuando il re-mapping, ad esempio andando a riscrivere il codice del kernel. Chiaramente in questo caso abbiamo un notevole rallentamento delle performance.

- Full virtualization con supporto HW **(non sicuro)**

Un esempio è l'architettura Intel x86-64: inizialmente non era pienamente virtualizzabile, ossia non riusciva a fare in modo che istruzioni privilegiate, eseguite a livello di privilegio più basso, innescassero delle trap. Per risolvere questo problema, gli sviluppatori hanno introdotto un supporto hardware che consiste nell'introduzione di diverse componenti hardware:

- Estensione dell'ISA (Instruction Set Architecture)
- Registri dedicati che andassero ad estendere il modello di programmazione:
    1) Registri di controllo
    2) Registri di stato, con i bit IF (Interrupt Enable Flag) ed TF (Trap Flag);

L'Hardware in questo caso quindi è fondamentale per poter rilevare le trap ed emularle.

Vantaggi:
La virtualizzazione assistita dall'hardware riduce il sovraccarico di manutenzione della paravirtualizzazione poiché riduce (idealmente, elimina) le modifiche necessarie nel sistema operativo guest. È anche notevolmente più facile ottenere prestazioni migliori.

Svantaggi:
La virtualizzazione assistita da hardware richiede un supporto esplicito nella CPU host, che è disponibile non su tutti i processori x86/x86_64. Un approccio di virtualizzazione "pura" assistita da hardware, che utilizza sistemi operativi guest completamente non modificati, comporta molte trap delle macchine virtuali e quindi costi elevati della CPU, limitando la scalabilità e l'efficienza del consolidamento dei server. Questo calo di prestazioni può essere mitigato dall'uso di driver paravirtualizzati; la combinazione è stata denominata "virtualizzazione ibrida".

## *PIA*

- Chi inizializza nella pia CB2 e CA1, il fatto che si alza CB2 e CA1… li fa più precisamente il dma o il processore

## PIC

- ***Cos'è e come funziona***

Il Priority Interrupt controller è un dispositivo introduce un meccanismo flessibile e programmabile per la gestione delle priorità e il mascheramento delle interruzioni Un singolo PIC gestisce 8 livelli di priorità, è possibile mettere 2 pic in cascata per arrivare a 64. Due modalità di priorità: fully nested mode e rotate mode. Di default abbiamo uno schema a priorità fissa, ma si può attivare anche uno schema a rotazione di priorità.
Il PIC funziona così:

1) abbiamo uno o più dispositivi connessi che inviano un interruzione sulla linee alla quale sono connessi, questa interruzione arriva al PIC e viene posto a 1 il relativo bit del registro IRR (Interrupt Request Register).
2) C'è un priority resolver che decide, sulla base delle priorità associate e sul contenuto di un registro che fa da maschera di interruzione IMR, quale dispositivo servire ed invia la richiesta di interruzione alla CPU
3) La CPU ricevuta la richiesta sul pin INT, invia un ACK
4) Il PIC ricevuto l'ACK setta il bit relativo alla linea interrompente del ISR (In Service Register) a 1 e resetta il corrispondente bit dell'IRR
5) Il PIC infine trasmette l'identificativo a 8 bit relativo al dispositivo interrompente
6) Il PIC resetta il bit dell'ISR ( può essere fatto in modalità automatica o manuale)

E' possibile programmare il PIC tramite due command word generate dalla CPU, la command word di inizializzazione che serve ad inizializzare il dispositivo con una sequenza di comandi, oppure la command word di operazione che consente di modificare la configurazione.
Riassumendo: Il PIC si frappone tra tutti i dispositivi che richiedono l'interruzione e il processore: si prende quindi carico di sentire se ci sono dispositivi che stanno interrompendo e se ce n'è più di uno adotta un meccanismo di priorità prefissato, per scegliere quello più prioritario. A questo punto, il PIC attiva l'unica linea di interruzione del processore, dandogli anche il "vettore" dell'interruzione, dal quale si ottiene, grazie ad una tabella, l'indirizzo del programma (ISR, ovvero Interrupt Service Routine) che il processore stesso dovrà eseguire per soddisfare la richiesta del dispositivo più prioritario. Come si vede, il processore non dovrà più preoccuparsi di andare in giro e stabilire quale interruzione dovrà servire per primo: l'unica cosa che dovrà fare sarà quella di eseguire il programma di interruzione (ISR) che gli verrà indicato dal PIC.

- ***approfondimenti sull'utilità***

Senza il PIC tutte le richieste di interruzione dei vari dispositivi confluiscono su un unico filo INT, e teoricamente una volta interrotto, il processore deve interrogare i vari dispositivi per individuare chi è stato ad attivare l'interruzione, a meno che non si usano le autovettorizzate, la situazione diventa ancora più complessa se più dispositivi richiedono contemporaneamente un interruzione, in tal caso il processore dovrebbe prevedere uno schema di priorità. Con l'introduzione del PIC tutti i seguenti problemi vengono gestiti da un dispositivo apposito e il processore ricevere solo la linea INT e l'identificativo della ISR da eseguire sul bus dati.

- ***quando interrompe su che linea mette l'interruzione?***

Il dispositivo è direttamente connesso al PIC e quando interrompe si scrive solo 1 nel registro IRR del PIC, sarà il PIC successivamente ad inoltrarla la richiesta al processore sulla classica linea INT del processore

- ***profilo del pic sul b***us

# DMA

Il Direct Memory Access è l'unico dispositivo oltre la CPU in grado di generare degli indirizzi sul bus per accedere direttamente alla memoria o ai dispositivi mappati in memoria.
Tramite il DMA è molto utile quando è necessario trasferire grossi blocchi di dati senza generare un interruzione per ogni carattere trasferito. Per fare cio il DMA si pone al centro dell'architettura di bus e si impadronisce del bus per indirizzare la memoria. Chiaramente è necessario un protocollo di comunicazione tra DMA e processore per il controllo del bus.

- ***Il dma inizializza la periferica o la inizializza il processore? Sempre il processore? Le interruzioni vanno al dma ?***

In generale le periferiche vengono inizializzate dal processore, il DMA in generale non lo fa. Il processore deve occuparsi di inizializzare lo stesso DMA (anche prime delle altre periferiche). C'è un eccezione, in un particolare schema dove la periferica non è connessa al bus ma solo al DMA, in questo caso il processore non può inizializzare la periferica (perchè non è sul bus) e se ne occupa il DMA.
In generale in questo schema la periferica invia l'interruzione al DMA.

- ***multiprocessore con coerenza e DMA***
- ***modello di programmazione del DMA, protocollo sul bus tra DMA e processore ogni singolo passo spiegato con tutti i segnali e registri coinvolti***

Il DMA è formato da due canali simmetrici programmabili mediante i registri BADDR-CADDR-BCOUNT-CCOUNT (indirizzo base e corrente del trasferimento, valore base-corrente del conteggio in numero di byte da trasferire). Abbiamo poi registri di MODE (modalità di trasferimento, DEV-MEM, MEM-DEV, MEM-MEM), flag di richiesta e flag di mask (per mascherare il trasferimento sul canale). I registri di CNTRL, STATUS e TEMP sono in comune ai due canali.
I bit del registro MODE hanno ognuno un significato: il BIT 5 per incremento o decremento del registro indirizzo dopo il trasferimento, BIT 4 per abilitare l'auto-inizializzazione, BIT 2-3 per indicare il tipo di trasferimento (read or write), il BIT 7 indica il modo di trasferimento, 0 se Single 1 se BLOCK, quindi cambia il modo in cui viene rilasciato il bus al processore alla fine di ogni trasferimento.
Il registro CNTRL si divide in 4 bit meno significativi per lo stato del componente e gli altri 4 per i bit di controllo. I BIT 0-1 sono per il trasferimento MEM-MEM. Il BIT 2 abilita il controller, BIT 3 tipo di tempificazione, BIT 4 tipo di priorità tra i canali (fissa o round-robin).
Il registro STATUS gestisce la parte di comunicazione col processore e quindi sono più legati alla questione delle interruzioni. Ad ogni canale è associato un flag per il request (richiesta di DMA da un device attraverso DREQ) e per il mask (disabilita il canale da richieste sia hardware che software).

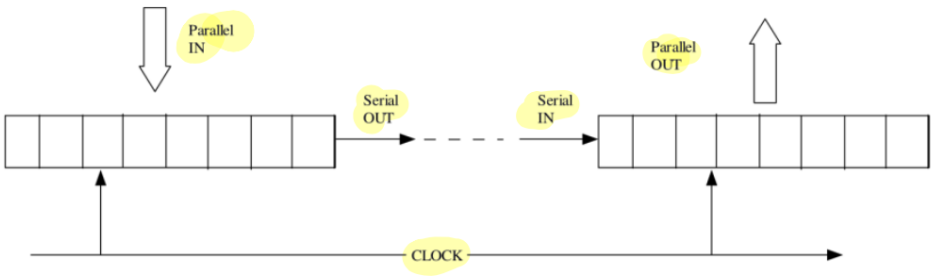
Per quanto riguarda i PROTOCOLLI, abbiamo che generalmente le fasi in cui si ha un trasferimento da DEV-MEM sono:

- 1) il DEVICE fa una richiesta al DMA con DREQ;
- 2) il DMA chiede al Bus Master (cioè la CPU) di acquisire il bus con HRQ;
- 3) Si attende HLDA;
- 4) Ricevuto il Bus Grant, si pone l'indirizzo base sull'address bus e si manda un ACK al DEVICE;
- 5) supponendo trasferimento DEV-MEM, abilitiamo IOR e MEMW;
- 6) il DEVICE mette i dati sul bus e disabilita la richiesta;
- 7) la MEM scrive i dati e manda un segnale di ready al controller che incrementa il CADDR (partendo da BADDR) e decrementa CCOUNT (partendo da BCOUNT);
- 8) il ciclo è ripetuto finchè EOP non viene alzato, indicando il termine dell'operazione.

Inoltre, il DMA può funzionare sia da SLAVE (il DMA viene visto dalla CPU come una qualunque periferica) che da MASTER (sostituisce la CPU, interfacciandosi direttamente con la periferica). [DIFFERENZA NELLA RAPPRESENTAZIONE (CI VUOLE DISEGNO) SE UN DISPOSITIVO E' COLLEGATO AL BUS, ALLORA SARA COLLEGATO ANCHE AL DMA CHE SARA' SLAVE; OPPURE DISPOSITIVO COLLEGATO SOLO AL DMA E NON AL BUS, IL DMA E' COLLEGATO AL BUS E FUNGE DA MASTER).

# USART

L'USART è un dispositivo hardware che converte flussi di bit di dati da un formato parallelo a un formato seriale o viceversa. È fondamentale per poter permettere la comunicazione tra interfacce seriali di diversi dispositivi. La comunicazione seriale, rispetto a quella parallela, permette di raggiungere distanze più elevate con un hardware molto semplice. Per la trasmissione, la USART riceve dal processore, in maniera parallela, 8 bit alla volta, i quali vengono opportunamente serializzati utilizzando uno shift register. In ricezione è presente un ulteriore SR, il quale accoglie i bit che riceve in maniera seriale; quando lo SR è pieno, invia in maniera parallela il messaggio ad un registro interno, il quale può essere letto dal processore. Semplificando:




USART supporta due tipi di comunicazione: sincrona ed asincrona; le due modalità si differenziano nel modo in cui viene effettuata la sincronizzazione durante la comunicazione.

### MODALITÀ ASINCRONA

In questa modalità la sincronizzazione avviene all'interno del messaggio. Sono presenti alcuni bit per determinare l'inizio e la fine del messaggio, rispettivamente START bit e STOP bit. Inoltre sono previsti anche bit di parità per la gestione degli errori ed infine abbiamo il messaggio vero e proprio. Chiaramente in questo modo stiamo introducendo

un elevato overhead, ma allo stesso tempo garantiamo la sincronizzazione e la correttezza della comunicazione su ogni messaggio. In questa modalità nasce ovviamente la problematica nel caso in cui trasmettitore e ricevitore lavorano a frequenze diverse; per essere sicuri che il ricevitore riesca ad individuare lo start bit, impostiamo la frequenza di ricezione pari ad un valore che sia 16/32 volte il valore della frequenza di trasmissione. Una volta ricevuto il bit di start, la frequenza viene rifasata per posizionare il campionamento al centro del bit.

**MODALITÀ SINCRONA**

In questo caso invece la sincronizzazione viene effettuata in maniera separata ed il messaggio contiene tutte informazioni significative. In maniera periodica vengono inviati un numero programmabile di segnali di sincronizzazione, detti SYNC BYTE, i quali vengono utilizzati per la correzione degli errori. A differenza della modalità precedente, non abbiamo un forte overhead sul singolo messaggio, ma se ci sono degli errori, ce ne accorgiamo molto in ritardo e soprattutto per correggerli bisogna ritrasmettere un numero elevato di messaggi, mentre nella modalità sincrona in caso di errore occorre ritrasmettere solo il singolo carattere.