4/2/2014

ESERCIZIO 2 (pt 23, PROGETTO): DMAC, Task e Cache

Un sistema **S** con bus di memoria da 64 bit e bus di I/O da 8 bit dispone di **3 Porte Parallele** di ingresso da **8 bit** P_IN1, P_IN2, P_IN3 gestite in DMA, **una porta seriale di uscita** S_OUT0 gestita anch'essa in DMA, un Pentium **P**, un **PIC** e memoria costituita da **1 MB** di EPROM (mappata agli indirizzi fisici alti) e **1 MB** di RAM (mappata agli indirizzi bassi).
Il Pentium dispone di una cache dati a 2 vie da 8KB complessivi e linee da 32 byte, gestita con stato MESI e con politica di scrittura *Write-Around* in caso di miss.
Il sistema **S** funziona con *memoria virtuale disabilitata* e l'applicazione dispone di un unico **segmento dati SD mappato all'indirizzo fisico** 1 0000 H**.** Nel segmento sono memorizzati a indirizzi adiacenti **4 vettori di 2KB** l'uno, denominati rispettivamente A1, A3, A4 e RIS (A1 parte dall'indirizzo 1 0000 H).
S riceve in DMA dalle 3 porte parallele P_INi i 3 vettori Ai, quindi esegue la seguente operazione vettoriale:
RIS = (A1 OR A2) AND A3
Completata l'elaborazione, il vettore RIS viene trasmesso in DMA sulla porta di uscita S_OUT0.
I dati arrivano sulle 3 porte di ingresso con tre flussi concorrenti e *con una cadenza approssimativa di 2 KB/s per canale.* La porta seriale S_OUT0 è una UART funzionante a 9600b/s. **Un unico task T0 gestisce le tre porte parallele** ed esegue l'elaborazione una volta completata la ricezione su tutti e tre i canali che, come già detto, avviene con flussi di dati concorrenti. Ad avvenuta elaborazione, viene fatto partire il task T1 che si incarica di trasmettere sulla seriale il vettore RIS appena elaborato e, al termine della trasmissione, ciclicamente, fa ripartire la ricezione dalle tre porte di ingresso.

Si risponda ai seguenti quesiti, motivando le risposte:

1. Si disegni lo schema a blocchi di S, si disegni la mappa della memoria che mostra il segmento dati SD con i suoi componenti, si definisca il data segment **SD** e se ne costruisca il descrittore. (punti 2)

2. Su integrino nel sistema le porte di ingresso e uscita, il DMAC, il PIC e **la memoria** e si indichi quali canali di DMA si utilizzano e quanti e quali segnali di *interrupt request* vengono generati. Si dica come vengono programmati il DMAC e i registri esterni del DMAC e **si disegni il contenuto della IDT.** (punti 5)

3. Si disegnino **con cura** le flow chart dei task **T0, T1 e Idle (indicando per bene le operazioni di controllo sui dispositivi mappati in I/O)** e si indichi la sequenza dei *task running*, ci si fermi al termine della prima trasmissione di RIS alla seriale. Come si fa a essere sicuri che non si perdano interrupt e, al tempo stesso, che l'arrivo di un interrupt non faccia partire il task T0 quando questo è BUSY? Si faccia l'ipotesi che il PIC sia programmato in *positive edge triggered mode.* (punti 5)

4. Impiegando le informazioni precedenti, si indichi approssimativamente dopo quanto tempo termina la trasmissione del primo messaggio a S_OUT0. Durante questo periodo cosa fa principalmente la CPU? (punti -2...1)

5. Per *10 punti* si analizzi la dinamica della **cache** dati e si risponda in modo il più possibile **conciso e tabellare** ai seguenti quesiti:

   (a) Quali sono gli indici di set e i tag associati ai vettori A1, A2, A3 e RIS?

   (b) Qual è lo stato della cache al termine della prima ricezione dei 3 vettori A1, A2, A3?

   (c) Si scriva la sequenza di operazioni elementari con cui si esegue l'elaborazione (si scriva la prima iterazione usando ad esempio l'assembler del DLX), quindi se ne calcoli la miss rate. Si riesce a ottenere una miss rate complessiva inferiore al 2.5%? Qual è la più bassa miss rate ottenibile e qual è la sequenza di istruzioni con cui la si ottiene? E in questo caso quanti sono gli accessi?

   (d) Nella soluzione prescelta al punto precedente, vale il principio di località temporale? E quello di località spaziale? Quanto incidono (qualitativamente= questi principi sulla miss rate?

   (e) Qual è lo stato della cache negli istanti di fine elaborazione e fine trasmissione di RIS alla seriale?

   (f) Avrei ottenuto miss rate diverse con cache di 16KB? E con cache di 4KB? (Si considerino solo cache con due vie e linee di 32 byte)

0.1 Esercizio 1 - Punti 2

Lo schema a blocchi è quello in figura 1, mentre il segmento dati SD è mappato in memoria secondo la tabella 1.

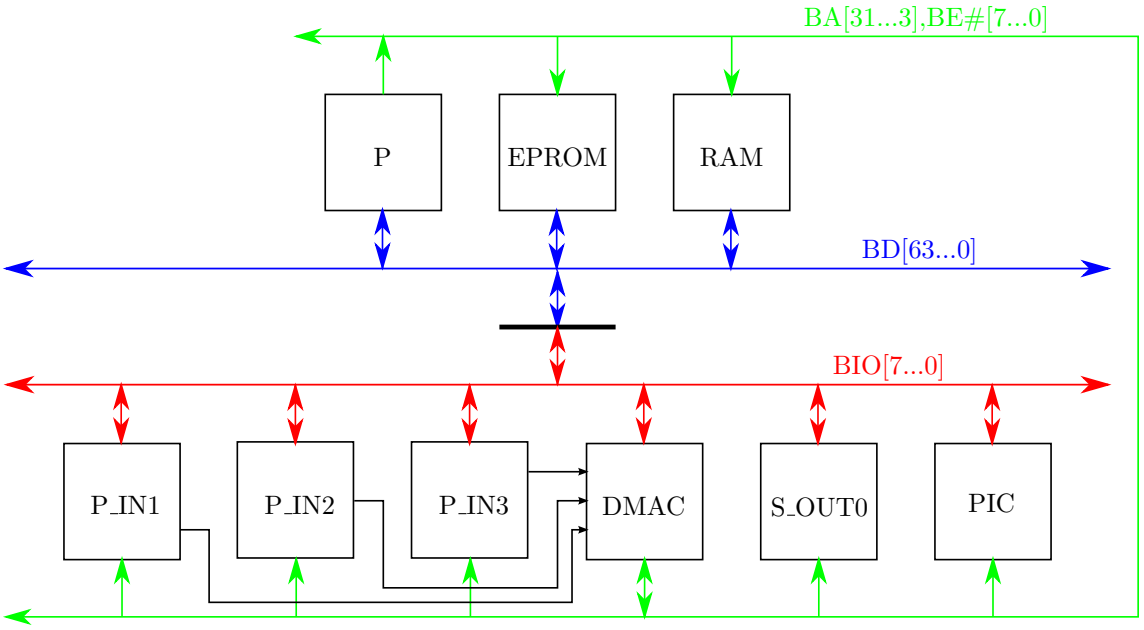


Figura 1: Schema a blocchi di **S**.

Tabella 1: Mappa della memoria del segmento SD.

	Banco 7	Banco 6	Banco 5	Banco 4	Banco 3	Banco 2	Banco 1	Banco 0	
...	...	...	...	...	...	...	...	...	...
1 1FFF H	RIS(2K-1)	RIS(2K-2)	...	...	...	...	...	RIS(2K-8)	1 1FF8 H
...	...	...	...	...	...	...	...	...	...
1 1807 H	RIS(7)	RIS(6)	...	...	...	...	...	RIS(0)	1 1800 H
1 17FF H	A3(2K-1)	A3(2K-2)	...	...	...	...	...	A3(2K-8)	1 17F8 H
...	...	...	...	...	...	...	...	...	...
1 1007 H	A3(7)	A3(6)	...	...	...	...	...	A3(0)	1 1000 H
1 0FFF H	A2(2K-1)	A2(2K-2)	...	...	...	...	...	A2(2K-8)	1 0FF8 H
...	...	...	...	...	...	...	...	...	...
1 0807 H	A2(7)	A2(6)	...	...	...	...	...	A2(0)	1 0800 H
1 07FF H	A1(2K-1)	A1(2K-2)	...	...	...	...	...	A1(2K-8)	1 07F8 H
...	...	...	...	...	...	...	...	...	...
1 0007 H	A1(7)	A1(6)	...	...	...	...	...	A1(0)	1 0000 H
...	...	...	...	...	...	...	...	...	...

In codice assembly il segmento è così definito:[1]

```
SD segment at 10000H
    A1  db   1024dup
    A2  db   1024dup
    A3  db   1024dup
    RIS db   1024dup
SD ends
```

[1] È importante utilizzare `db` anziché `w` per una correttezza *semantica*, dato che il problema è impostato con il byte come unità di misura

Il segmento parte all'indirizzo 1 0000 H ed è composto da 4 vettori di 2KB l'uno, per un totale 8KB. L'offset del segmento è quindi:

$$[8\text{KB}]_{\text{H}} - 1\,\text{H} = 20000\,\text{H} - 1\,\text{H} = 1\text{FFF}\,\text{H}$$

Il descrittore, perciò, è quello riportato in tabella 2.

Tabella 2: Descrittore di SD.

	G								E	W		
00 H	0	B	0	AVL	0 H	1	DPL	0	0	1	A	01 H
0000 H						1FFF H						

0.2 Esercizio 2 - Punti 5

Nel sistema abbiamo tre porte d'ingresso parallele e una porta d'uscita seriale. Per porta UART (Universal Asynchronous Receiver-Transmitter) non si intende altro che un dispositivo che converte un flusso di bit da una porta parallela in un formato seriale asincrono (o viceversa) con il quale si interfaccia al sistema. Per i nostri scopi, perciò, è da considerare *come una qualunque porta seriale*. Difatti, la porta 8250 è una UART.
Fissato ciò, passiamo a interfacciare i dispositivi con il sistema. È importante ricordare che *tutti i dispositivi sono gestiti in DMA*. La nostra, fortuna, in questo caso, è che i dispositivi sono "solo " quattro (tre porte parallele e una seriale), perciò un unico DMAC a quattro canali sarà sufficiente per gestirli tutti. In caso contrario, avremmo dovuto utilizzarne almeno tre: uno in *cascade mode* al quale ne sarebbero stati collegati almeno altri due in *demand mode* o *single mode*. **Ma non è questo il nostro caso**.

0.2.1 Indirizzamento

Preoccupiamoci, prima ancora di affrontare l'interfacciamento al sistema, di indirizzare ogni componente di esso. La tabella 3 organizza tutti gli indirizzi, la loro decodifica semplificata e i nomi dei segnali di *chip select*.

0.2.2 Encoder

Come prologo al nostro interfacciamento, è utile preporre l'encoder in figura 2 nella pagina seguente che traduce i segnali BE[7...0]# in BA[2..0] per indirizzare i dispositivi di I/O.

Tabella 3: Tabella degli indirizzi di tutti i dispositivi

Dispositivo	M/IO	Indirizzo iniziale	Indirizzo finale	Decodifica semplificata	Segnali di comando	Nome del CS
RAM	M	0000 0000 H	000F FFFF H	$\overline{\text{BA31}}$	MEMRDC#, MEMWRC#	CS_RAM
EPROM	M	FFF0 0000 H	FFFF FFFF H	BA31	MEMRDC#	CS_EPROM
DMAC	IO	0100 H	010F H	$\overline{\text{BA12}}\cdot\overline{\text{BA11}}\cdot\overline{\text{BA10}}\cdot\text{BA9}$	IORDC#, IOWRC#	CS_DMAC
PIC	IO	0200 H	0201 H	$\overline{\text{BA12}}\cdot\overline{\text{BA11}}\cdot\text{BA10}\cdot\overline{\text{BA9}}$	IORDC#, IOWRC#, INTA#	CS_PIC
P_IN0	IO	0300 H	0303 H	$\overline{\text{BA12}}\cdot\overline{\text{BA11}}\cdot\text{BA10}\cdot\text{BA9}$	IORDC#	CS_P_IN0
P_IN1	IO	0400 H	0403 H	$\overline{\text{BA12}}\cdot\text{BA11}\cdot\overline{\text{BA10}}\cdot\overline{\text{BA9}}$	IORDC#	CS_P_IN1
P_IN2	IO	0500 H	0503 H	$\overline{\text{BA12}}\cdot\text{BA11}\cdot\overline{\text{BA10}}\cdot\text{BA9}$	IORDC#	CS_P_IN2
S_OUT0	IO	0600 H	0607 H	$\overline{\text{BA12}}\cdot\text{BA11}\cdot\cdot\text{BA10}\cdot\overline{\text{BA9}}$	IOWRC#	CS_S_OUT0
HL0	IO	0700 H	0700 H	$\overline{\text{BA12}}\cdot\text{BA11}\cdot\cdot\text{BA10}\cdot\text{BA9}$	IOWRC#	CS_HL0
HH0	IO	0800 H	0800 H	$\text{BA12}\cdot\overline{\text{BA11}}\cdot\overline{\text{BA10}}\cdot\overline{\text{BA9}}$	IOWRC#	CS_HH0
HL1	IO	0900 H	0900 H	$\text{BA12}\cdot\overline{\text{BA11}}\cdot\overline{\text{BA10}}\cdot\text{BA9}$	IOWRC#	CS_HL1
HH1	IO	0A00 H	0A00 H	$\text{BA12}\cdot\overline{\text{BA11}}\cdot\text{BA10}\cdot\overline{\text{BA9}}$	IOWRC#	CS_HH1
HL2	IO	0B00 H	0B00 H	$\text{BA12}\cdot\overline{\text{BA11}}\cdot\text{BA10}\cdot\text{BA9}$	IOWRC#	CS_HL2
HH2	IO	0C00 H	0C00 H	$\text{BA12}\cdot\text{BA11}\cdot\overline{\text{BA10}}\cdot\overline{\text{BA9}}$	IOWRC#	CS_HH2
HL3	IO	0D00 H	0D00 H	$\text{BA12}\cdot\text{BA11}\cdot\overline{\text{BA10}}\cdot\text{BA9}$	IOWRC#	CS_HL3
HH3	IO	0E00 H	0E00 H	$\text{BA12}\cdot\text{BA11}\cdot\text{BA10}\cdot\overline{\text{BA9}}$	IOWRC#	CS_HH3

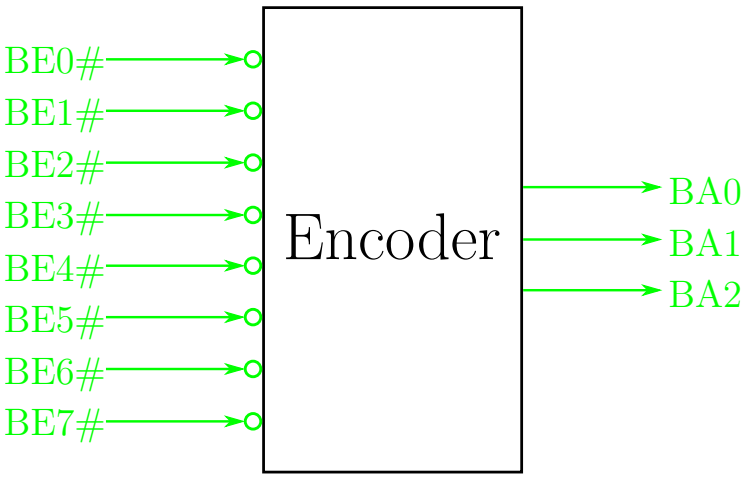


Figura 2: Schema a blocchi dell'*encoder*.

0.2.3 Porte parallele P_INi

Tutte le porte parallele si interfacciano allo stesso modo. Notare, in figura 3, che i segnali CS_P_IN e BA0 non sono direttamente interfacciati con la porta, ma in *multiplexing* con DACKi il *chip select* e con la massa i BA0. Tutti i multiplexer sono controllati dal segnale HOLDA inviato dalla CPU. Questo per far sì che il dispositivo sia attivo solo quando il DMAC ha ottenuto il controllo del bus. Non serve interfacciare BA1 con il pin A1 perché la porta gestisce un solo bit alla volta e quindi non serve indirizzarne due. Se avessimo avuto due byte in ingresso, avremmo lasciato in questo modo il pin A0 e interfacciato direttamente il pin A1 con BA1. Infine, il segnale SRQ è inviato al DREQi del DMAC. Se le porte fossero state gestite ad interrupt, i segnali sarebbero dovuti essere inviati ai piedini IRi del PIC. Ma, ancora una volta, **non è questo il nostro caso**.

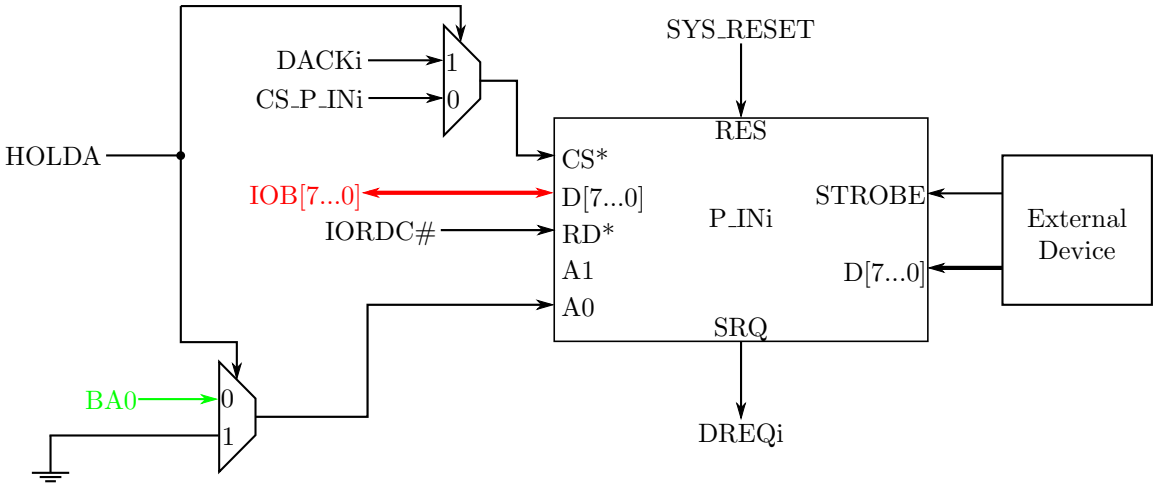


Figura 3: Schema a blocchi delle porte parallele P_INi.

0.2.4 Porta Seriale S_OUT0

L'interfacciamento della porta seriale è analogo a quello delle porte parallele, con le ovvie differenze nei segnali di comando, SRQ e *chip select*. In figura 4 nella pagina seguente, lo schema circuitale.

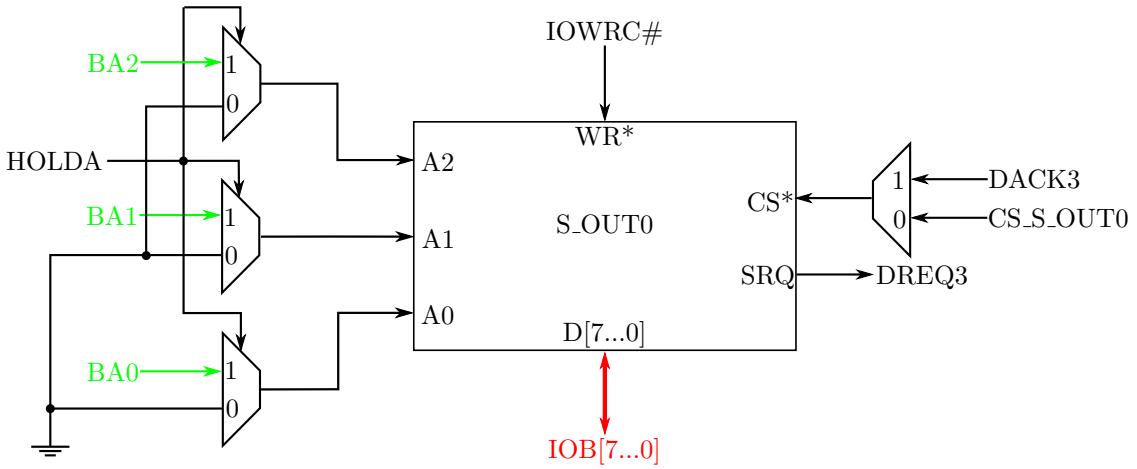


Figura 4: Schema a blocchi della porta seriale S_OUT0.

0.2.5 RAM

La RAM è formata da 8 banchi di $\frac{1\text{MB}}{8} = \frac{2^{20}\text{B}}{2^3} = 2^{17}\text{B} = 128\text{KB}$ ciascuno. Un i-esimo banco è interfacciato con il sistema come in figura 5.

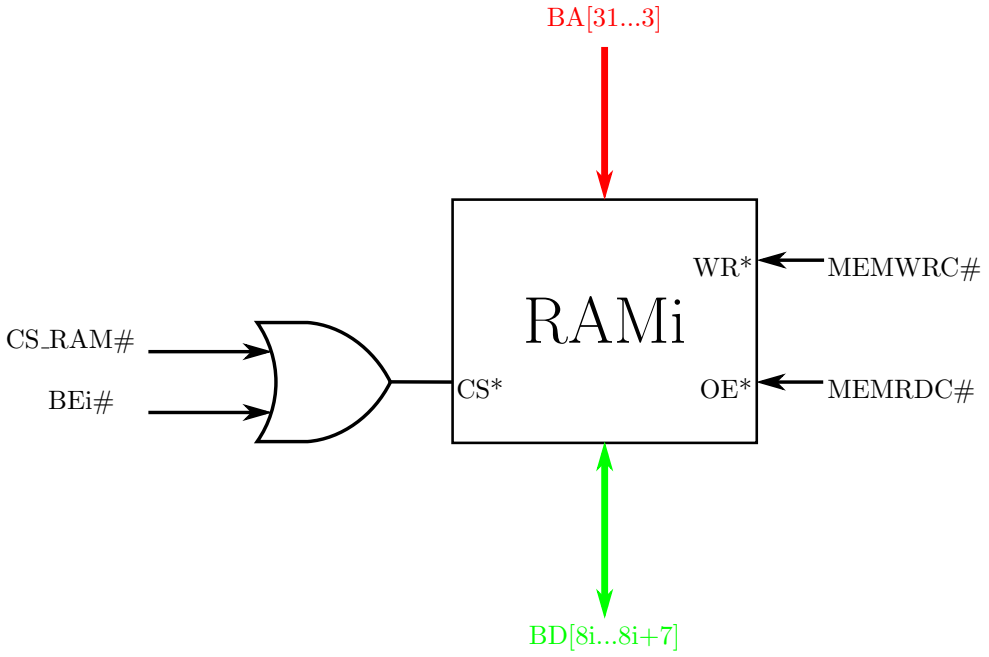


Figura 5: Schema a blocchi dell'i-esimo banco di RAM.

0.2.6 EPROM

La EPROM è organizzata esattamente come la RAM, anch'essa in banchi da 128KB. L'unica differenza, come si evince dalla figura 6 nella pagina seguente, è solo nel fatto che, per sua natura, la EPROM è una memoria *read only* e perciò non è presente alcun pin WR*.

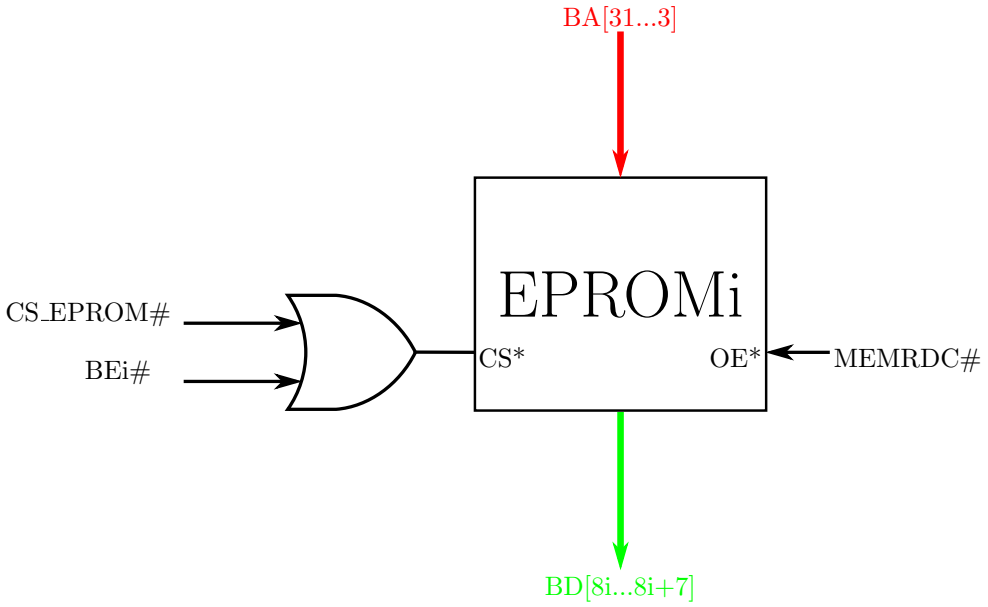


Figura 6: Schema a blocchi dell'i-esimo banco di EPROM.

0.2.7 DMAC

Il DMAC è il componente più laborioso da interfacciare. Per semplicità, dividiamo il problema in due parti. Una prima parte riguarda effettivamente il DMAC e i componenti ad esso più direttamente connessi (figura 7 nella pagina successiva). La seconda parte riguarda i *latches a 8 bit* che interfacciano l'IOB con il BA (figura 8 nella pagina seguente). È necessaria una coppia per ogni canale del DMAC perché questo è stato progettato per sistemi con bus di indirizzi a 8 bit, mentre il nostro sistema prevede un bus indirizzi a ben 32 bit. Per sopperire a questa discrepanza, si utilizzano dei *latches* opportunamente programmati dalla CPU.

Programmazione del DMAC

Il DMAC utilizza quattro canali, ognuno associato ad un dispostivo di I/O e ad una coppia di *latches*. Ogni dispositivo si occupa del trasferimento di un vettore di 2KB, conseguentemente si avranno 2K trasferimenti per ogni canale DMA, avendo il bus di I/O un parallelismo di 1 byte. Per ogni canale del DMAC dobbiamo programmare:

- BAR (*Base Address Register*): indica l'indirizzo iniziale da trasferire;
- BCR (*Base Counter Register*): indica l'offset di indirizzo dopo tutti i trasferimenti;
- LATCH_HH e LATCH_HL: sono programmati per indirizzare i 16 MSB di indirizzo;
- MR[7...0] (*Mode Register*): registro a 8 bit identifica la modalità di funzionamento del canale.

In particolare, i bit del mode register, assumono i seguenti significati:

MR[7,6] *MODE*: 00: *demand mode*; 01: *single mode*; 02: *block mode*; 03: *cascade mode.*

MR[5] Vale 0 se si vuole lavorare a indirizzi crescenti, 1 altrimenti.

MR[4] Vale 1 se è abilitato l'*autoinit*, 0 altrimenti.

MR[3,2] 00: *verify*; 01: *write*; 10: *read* 11: illegale; XX se in cascade mode.

MR[1,0] *CHANNEL SELECT*: indica quale canale si sta utilizzando

Vengono riportati in tabella 4 nella pagina successiva i valori della programmazione.

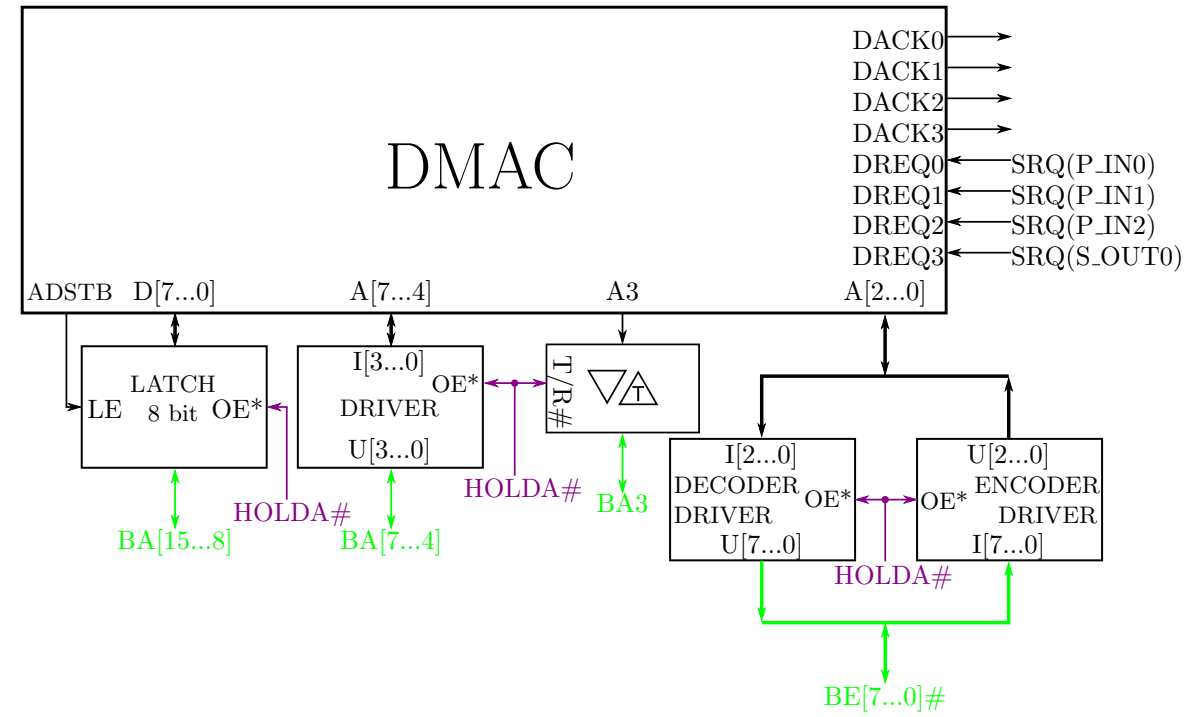


Figura 7: Schema a blocchi del DMAC.

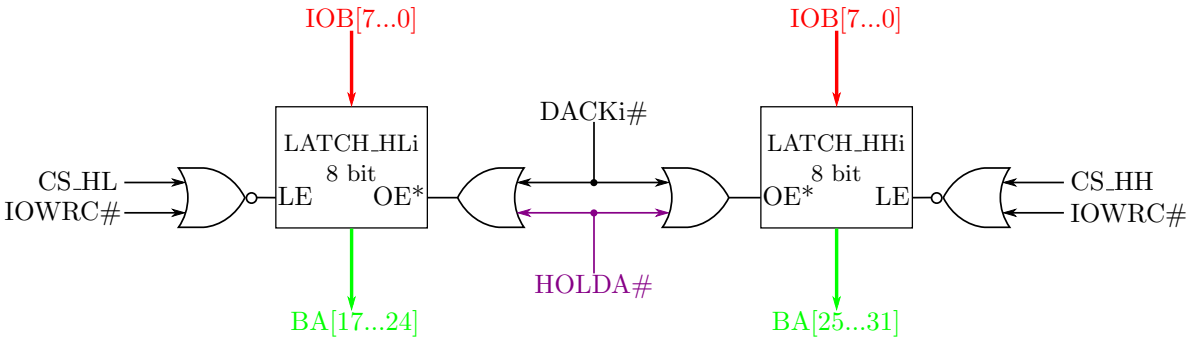


Figura 8: Schema a blocchi dei *latches* HH e HL.

Tabella 4: I valori con cui programmare il DMAC

Channel	BAR	BCR	LATCH_HL	LATCH_HH	MR[7...0]
0	0000 H	07FF H	01 H	00 H	01000100
1	0800 H	07FF H	01 H	00 H	01000101
2	1000 H	07FF H	01 H	00 H	01000110
3	1800 H	07FF H	01 H	00 H	01001011

0.2.8 PIC

Il PIC richiede un interfacciamento leggermente complesso. Ogni entrata per gli *interrupt* IRi è attivata dall'OR dei segnali EOP# e DACKi#. Questo interfacciamento ci permette di non perdere gli interrupt e non far partire un task inopportuno nel caso arrivino contemporaneamente più richieste. I dettagli saranno esaminati in dettaglio nel prossimo punto dell'esame. Per ora, si faccia riferimento allo schema a blocchi in figura 9.

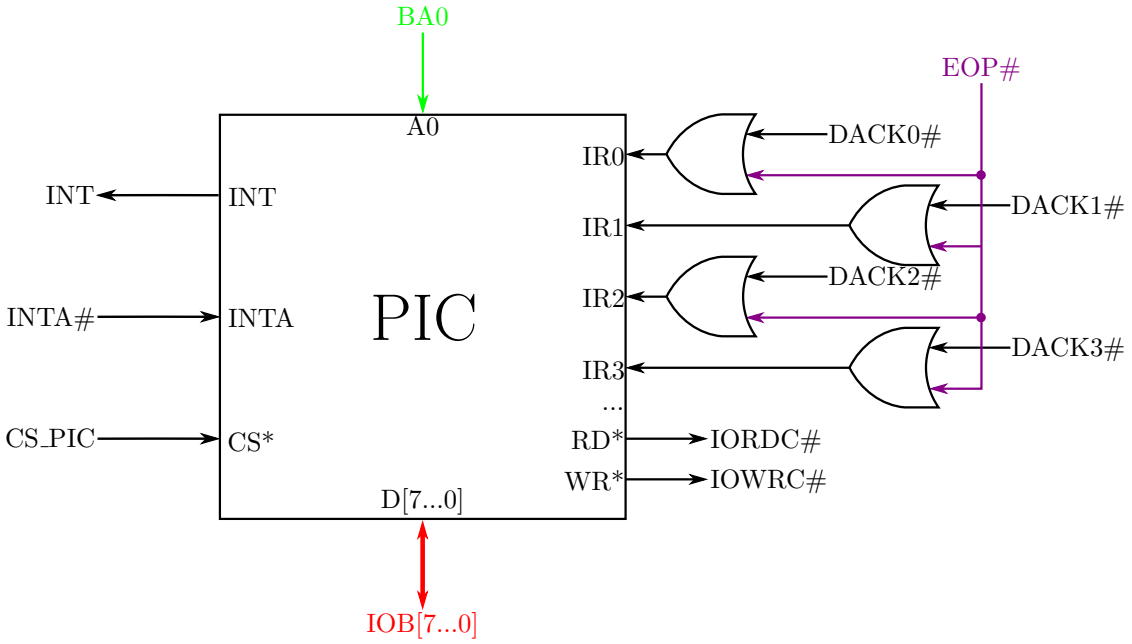


Figura 9: Schema a blocchi del PIC.

IDT

La IDT contiene *due* task gate per gestire i due tipi di interrupt possibili:

- Interrupt proveniente dal DMAC in conseguenza dell'avvenuto trasferimento *dalle porte parallele alla memoria* e inviato su IR0, IR1 e IR2 (**INT_TYPE**=30 H);
- Interrupt proveniente dal DMAC in conseguenza dell'avvenuto trasferimento *dalla memoria alla porta seriale* e inviato su IR3 (**INT_TYPE**=31 H).

In tabella 5 il descrittore del *task gate* e di seguito il codice assembly del descrittore della IDT:

Tabella 5: Descrittore del task gate.

	85 H							
Reserved	1	DPL	0	0	1	0	1	Reserved
Activating task TSS descriptor's selector	Reserved							

```
    IDT_START:
        DESCRITTORI     gate    030H dup(?)

30H T0_GATE task_gate   <0,2,0,85H,0>
31H T1_GATE task_gate   <0,3,0,85H,0>

    IDT_END
```

0.3 Esercizio 3 - Punti 5

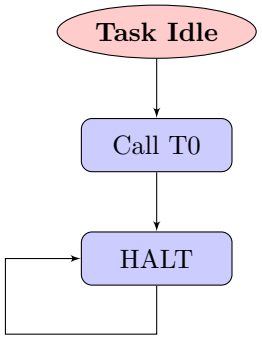
Le *flow chart* sono di facile realizzazione, purché si tengano bene a mente i seguenti punti:

- Ogni canale del DMAC possiede le proprie coppie di registri BAR, CAR, BCR e CCR, perciò non è necessario riprogrammarli ogniqualvolta venga terminato un singolo trasferimento.
- Operando in *single mode*, il DMAC rilascia il bus ogniqualvolta venga terminato un trasferimento elementare da un qualsiasi canale; questo genera sicuramente *overhead*, ma data la mole dei dati da trasferire, riduciamo l'altrimenti alto rischio di *starvation* della CPU.
- Il DMAC genera il segnale EOP solo dopo il trasferimento dell'ultimo byte su un canale.
- Dopo la generazione di EOP, il canale corrispondente *si maschera automaticamente*.

0.3.1 Flow chart dei task Idle, T0 e T1

In figura 10 è rappresentata la *flow chart* del task **Idle** che non merita alcun particolare commento: esso si preoccupa solamente di mantenere in attesa e a basso consumo energetico la CPU.

Figura 10: Flow chart del task **Idle**



In figura 11 nella pagina successiva è rappresentata la *flow chart* del task di ricezione **T0**. Dopo aver opportunamente programmato il DMAC e smascherato i canali e gli interrupt necessari, inizializza i trasferimenti e si pone in attesa di un *interrupt* dovuto ad un EOP generato da un *qualsiasi canale DMAC tra CH0, CH1 e CH2*. Ricevuto uno qualsiasi di questi, incrementa una variabile interna e attende il prossimo. Quando la variabile interna raggiunge il valore 3, cioè quando sono stati ricevuti tutti e tre i vettori, maschera gli *interrupt*, esegue le operazioni e trasferisce il controllo al task T1.

La *flow chart* del task **T1** è rappresentata in figura 12 a pagina 12. Il comportamento è analogo al task T0, ad eccezione del fatto che si deve preoccupare solo del canale DMAC CH3 e delle richieste di *interrupt* da IR3. Non deve operare su alcuna variabile interna, poiché il trasferimento è su una sola porta, né fare alcun altro tipo di operazione. Di conseguenza, al termine della trasmissione, restituisce il controllo al task T0.

La sequenza dei *task running*, infine, è la seguente:

$$T_I \to T_0 \to \underbrace{T_I \to T_0}_{\times 3} \to T_1 \to T_I \to T_1 \to T_0 \to \dots$$

0.3.2 Gestione degli interrupt

Gli interrupt non possono essere persi grazie all'interfacciamento che abbiamo dotato e alla capacità del PIC di funzionare in *positive edge triggered mode*, ovvero di riconoscere un'*interrupt* quando su un piedino IRi **non mascherato** vi è un fronte di salita, anziché riconoscerlo quando è presente un 1 logico. Inoltre siamo sicuri che mentre il task T0 è BUSY non giungano altri interrupt se settiamo a 0 il bit *Interrupt-enable Flag* nel task gate. *(La risposta fin qui è sufficiente come risposta all'esame, ciò che segue non serve ma è specificato per chiarezza espositiva.*

Figura 11: Flow chart del task **T0**

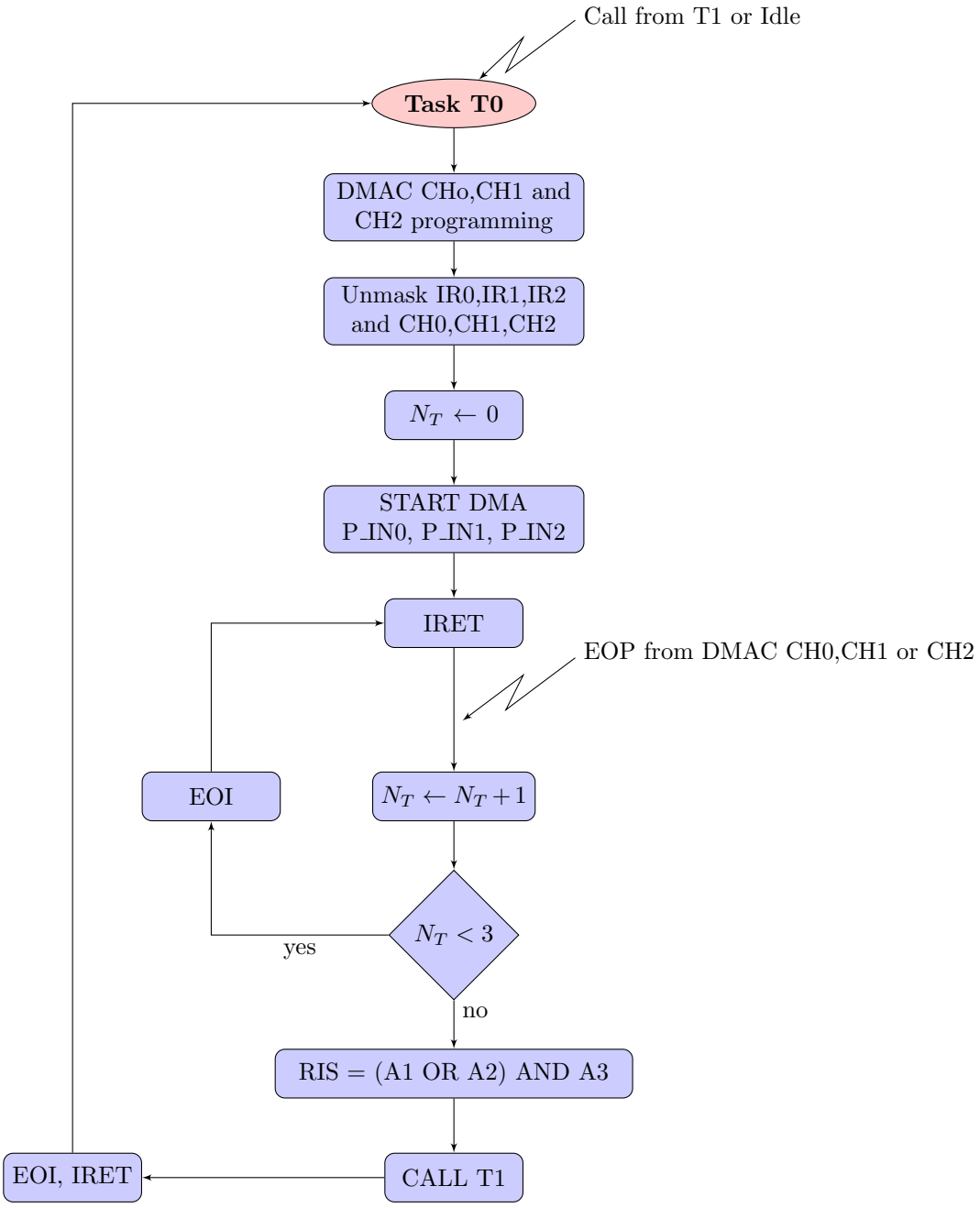
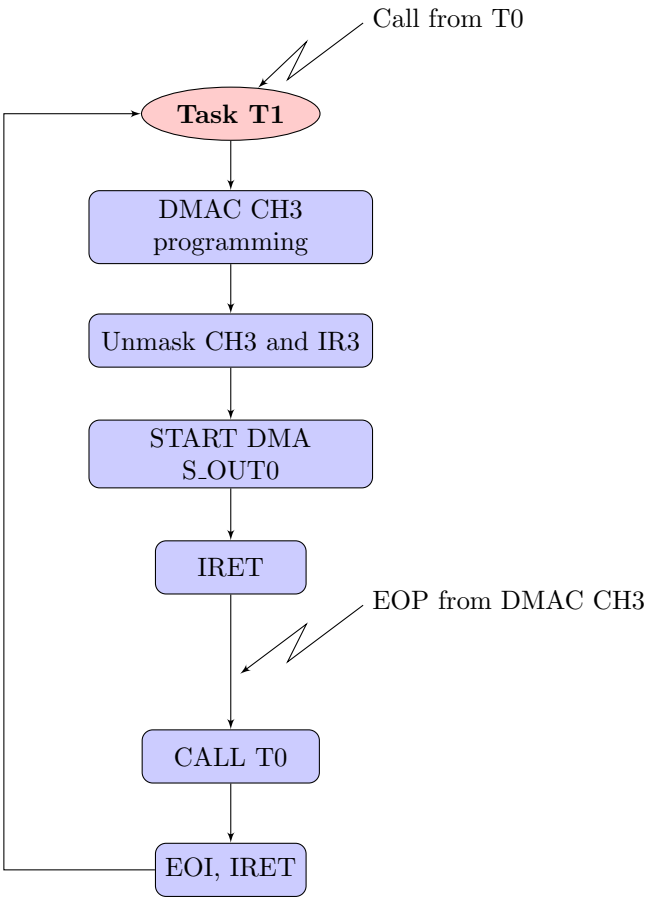


Figura 12: Flow chart del task **T1**



Utilizzando come input per IRi l'OR tra EOP# e DACKi#, questo è sempre garantito, poiché per avere un fronte positivo bisogna avere prima entrambi i segnali attivi bassi, dopodiché se ne deve disattivare uno. Nel nostro caso, ovviamente, EOP# si pone attivo basso solo se il corrispondente DACKi# è già attivo basso, portandosi poi al livello logico alto e facendo avvenire il fronte di salita sul piedino IRi del PIC. Nel grafico in figura 13 è esemplificato il caso in cui due canali terminino la loro transizione. Anche se DACK1# è attivo basso, nessun interrupt viene inviato al PIC finché non avviene il trasferimento dell'ultimo byte e viene generato il segnale EOP#. Quando in t_1 il segnale EOP# torna alto, avviene un fronte positivo sul piedino IR1 del PIC e viene servito il corrispondente interrupt. Stesso discorso può farsi per il canale CH0, con il segnale DACK0# e EOP#.

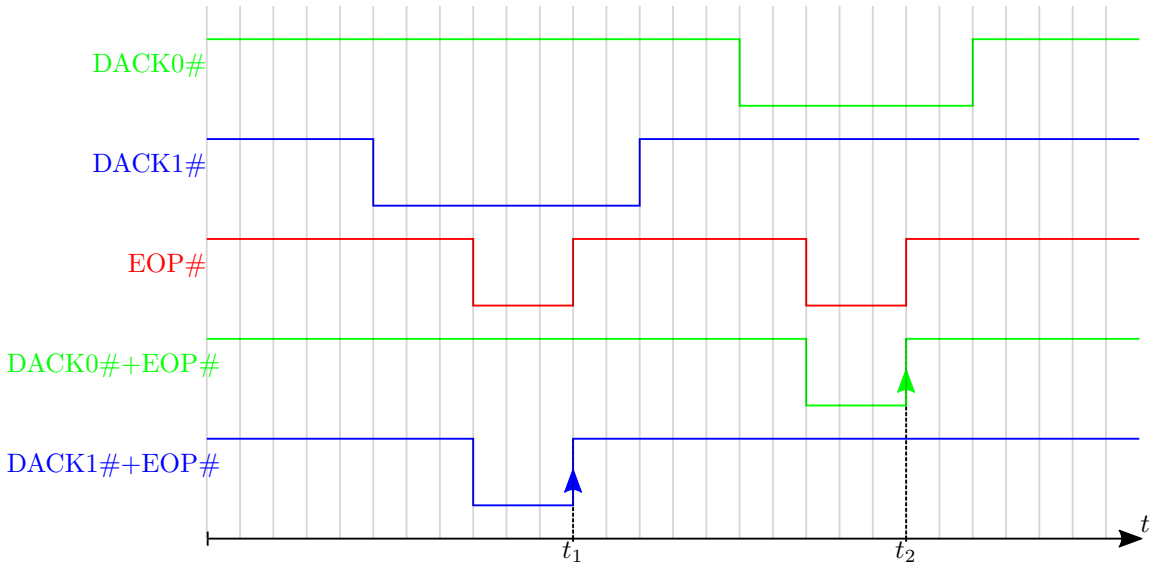


Figura 13: Segnali in ingresso al PIC.

0.4 Esercizio 4 - Punti -2...1

La ricezione avviene in maniera *concorrente*, perciò possiamo assumerla come parallela ai fini dei nostri calcoli poiché la velocità di trasferimento di un byte da una porta alla memoria è di svariati ordini di grandezza inferiore alla frequenza di arrivo dei dati nella porta. Siano $v_P = 2\text{KB/s}$ la velocità della porta parallela e $v_S = 9600\text{b/s}$ la velocità della porta seriale, il tempo totale T per la ricezione di tre vettori tramite porta parallela e la trasmissione di un quarto vettore sulla porta seriale, tutti della stessa dimensione $d = 2\text{KB}$ è:[2]

$$T = \frac{d}{v_P} + \frac{d}{v_S} = \frac{2 \cdot 2^{10}}{2 \cdot 2^{10}} + \frac{2 \cdot 2^{10}}{9600} = 1,21\text{s}$$

Durante le trasmissioni/ricezioni la CPU è in stato di *HALT*, ovvero viene sospesa in attesa della prossima interruzione esterna.

0.5 Esercizio 5

0.5.1 Punto (a)

Siano:

[2]Si faccia bene attenzione a non calcolare tre volte il tempo di ricezione. Le trasmissioni avvengono *concorrentemente* e perciò conta il valore più alto tra tutti quelli in ricezione. Nel nostro caso sono tutti e tre uguali quindi uno qualsiasi andrà bene.

Tabella 6: Campi di cache per i vettori A1, A2, A3 e RIS

	Hex value	*TAG*	*SET_ID*	*OFFSET*
A1(0)	0001 0000 H	0000 0000 0000 0001 0000	0000 000	0 0000
A1(2K-1)	0001 07FF H	0000 0000 0000 0001 0000	0111 111	1 1111
A2(0)	0001 0800 H	0000 0000 0000 0001 0000	1000 000	0 0000
A2(2K-1)	0001 0FFF H	0000 0000 0000 0001 0000	1111 111	1 1111
A3(0)	0001 1000 H	0000 0000 0000 0001 0001	0000 000	0 0000
A3(2K-1)	0001 17FF H	0000 0000 0000 0001 0001	0111 111	1 1111
RIS(0)	0001 1800 H	0000 0000 0000 0001 0001	1000 000	0 0000
RIS(2K-1)	0001 1FFF H	0000 0000 0000 0001 0001	1111 111	1 1111

N_O Il numero di bit del campo *OFFSET*

N_{SID} Il numero di bit del campo *SET_ID*

N_T Il numero di bit del campo *TAG*

$|l|$ La lunghezza, *in byte*, di una linea di cache

$|C|$ La dimensione totale, *in byte*, della cache

N_S Il numero di *set* della cache.

Valgono le seguenti relazioni:

$$\begin{aligned} N_O &= \log_2 |l| \\ N_{\text{SID}} &= \log_2 \left\lceil \frac{|C|}{|l| N_S} \right\rceil \\ N_T &= 32 - N_O - N_{\text{SID}} \end{aligned}$$

Nel nostro caso abbiamo quindi:

$$\begin{aligned} N_O &= 5 \\ N_{\text{SID}} &= 7 \\ N_T &= 20 \end{aligned}$$

In tabella 6 vengono riportati tutti i campi per il primo e l'ultimo byte di A1, A2, A3 e RIS: è facile notare che A1 e A2 riempiono tutti le linee di una via di cache. A3 e RIS hanno gli stessi *SET_ID* di A1 e A2, ma essendo la nostra cache a *due vie*, saranno memorizzata nella seconda via di cache senza sovrascrivere il contenuto della via precedentemente riempita da A1 e A2.

0.5.2 Punto (b)

La risposta a questa domanda è semplice: **la cache è tutta vuota e in stato *I (Invalid)***. Questo è vero **indipendentemente dal tipo di politica adottata dalla cache**. Il motivo è che i vettori A1, A2 e A3 vengono scritti in DMA, e il DMAC *non può accedere in alcun modo alla cache interna della CPU*. Perciò, se assumiamo che il sistema sia stato appena avviato, la cache permane nel suo stato *I*.

0.5.3 Punto (c)

I problemi iniziano qui. Di primo acchito, il codice *DLX-assembly*, in cui con $*$ indicheremo esplicitamente un accesso in memoria, per eseguire le operazioni vettoriali (o, per meglio dire, per fare l'operazione sul primo byte, supponendo le altre 2047 praticamente uguali) è il seguente:

```
lb  ax, A1(0)*
or  bx, ax, A2(0)*
and cx, bx, A3(0)*
sb  RIS(0)*, cx
```

Analizziamo il codice per calcolarne la *miss rate*. Le istruzioni `lb`, `or` e `and` causano *3 read miss*. Quello che accade è semplice, cioè vengono caricate in cache le rispettive linee, quindi tutti gli elementi dei vettori con *gli indici da 0 a 31*. Fin qui, **tutto ok**, quando giungeremo all'indice 32 si ripeterà la stessa operazione con gli indici da 32 a 63 e così via fino alla fine dei vettori. **Per ora**, abbiamo una miss ogni 32 letture, mantenendo una *read miss rate* $M_R(R) = \frac{1}{32} = 3.125\%$.

L'operazione `sb` è una trappola (figura 14).

Figura 14: **It's a trap!**

Saremmo tentati di pensare che la CPU, non trovando RIS(0) in cache, importi nella linea i byte da RIS(0) a RIS(32). **Ma non è così!**. Infatti, se controlliamo il testo dell'esame, è stato specificato che stiamo lavorando in politica *write-around* in caso di *write miss*. Questo significa che ***se un dato da scrivere non viene trovato in cache, viene aggiornato direttamente il dato in memoria. Nessuna linea di cache viene caricata.*** Questo, ahinoi, è un problema. Perché significa che *tutte* le istruzioni `sb` generano una *write miss*. Su 32 operazioni, avremo 32 scritture e 32 miss, quindi il *write miss rate* sarà, ovviamente, $M_R(W) = 100\%$. Il *miss rate totale* sarà la media pesata del *read miss rate* e del *write miss rate*. Ogni quattro accessi in memoria, avremo 3 accessi in lettura e 1 in scrittura, perciò:

$$M_R = \frac{3M_R(R) + M_R(W)}{4} = \frac{9.375\% + 100\%}{4} \approx 27.34\%$$

Più di una volta su quattro, abbiamo una miss, una situazione molto indesiderabile. **Come risolviamo il problema?**

Una soluzione potrebbe essere quella di cambiare la politica in caso di *write miss*. Questo funzionerebbe ma potrebbe non essere possibile se, ad esempio, la macchina per la quale dobbiamo scrivere il programma non è in nostro possesso oppure se per altri motivi progettuali dobbiamo tenerci la politica di *write around*. Ci serve una strada alternativa.

La nostra via d'uscita risiede nel fatto che *non ci è stato specificato alcun vincolo sulle politiche in caso di write hit*. Questo è un bene, perché abbiamo libertà di scelta. E la nostra scelta ricadrà su una politica di tipo *write back*, cioè scriveremo un dato solo in cache nel caso questo vi sia già presente. Il nostro compito sarà quindi di **fare in modo che il vettore RIS sia presente in cache quando andiamo a sovrascriverlo**. L'unico modo, è quello di **forzare una read sul vettore RIS costringendo la macchina a importarlo in cache.** Questo è fattibile aggiungendo un'istruzione `lb cx, RIS(0)`. Il codice diventa il seguente:

```
lb  ax, A1(0)*
```

```
or  bx, ax, A2(0)*
lb  cx, RIS(0)*
and cx, bx, A3(0)*
sb  RIS(0)*, cx
```

Come è reso evidente dal codice, questa volta abbiamo 5 accessi in memoria per ogni operazione su byte. In compenso, avremo solo *4 miss ogni 32 operazioni su byte*, ovvero il più piccolo numero di miss possibile. Questo perché anche nella più ottimistica della cache, dobbiamo avere una miss per ogni linea di cache, cioè la miss che causerà il caricamento della linea stessa. Ricordando che ogni vettore è composto da 2^{11}B, il miss rate totale, calcolato come il *rapporto tra il numero di miss e il numero di accessi totali* è:

$$M_R = \frac{4\frac{2^{11}}{32}}{5 \cdot 2^{11}} = \frac{4 \cdot 2^6}{2^{11}} = \frac{2^8}{5 \cdot 2^1 1} = \frac{1}{5} 2^{-3} = \frac{1}{40} = 2.5\%$$

Ed ecco a noi il miss rate richiesto dalla traccia che, in base alle precedenti osservazioni, è anche il minimo possibile. Senza fare le semplificazioni aritmetiche, il numero di miss N_M e il numero di accessi totali in memoria N_A sono:

$$N_M = 4\frac{2^{11}}{32} = 4 \cdot 2^6 = 2^8 = 256$$
$$N_A = 5 \cdot 2^{11} = 10240$$

0.5.4 Punto (d)

Il principio di località temporale in questo caso è rispettato ma molto poco influente. Quando effettuiamo la `lb cx, RIS(0)`, il principio di località temporale ci permette di riutilizzare la cache poco dopo con l'istruzione `sb RIS(0), cx`. La natura del problema in realtà nasconde questa proprietà, che è stata introdotta da noi artificialmente per permettere un uso più efficiente della cache riducendo gli accessi in memoria. Senza questo artificio, la località temporale sarebbe stata completamente ininfluente in quanto ogni vettore viene utilizzato solo una volta.

N.B.: se tutta l'operazione viene ripetuta una seconda volta, le linee di cache che contengono RIS sono ancora valide e l'istruzione `lb cx, RIS(0)` *non genera più una miss*. Rifacendo i calcoli, si può verificare che la miss rate, restando così il codice, scende all'1.875% proprio grazie al principio di località temporale. Se invece volessimo sfruttare, dalla seconda operazione in poi, un secondo programma che elimina l'istruzione `lb cx, RIS(0)`, la miss rate salirebbe al 2.34% ma in realtà le prestazioni aumenterebbero perché diminuirebbero sia il numero di miss che il numero di accessi totali. Questa seconda soluzione è comunque **sconsigliata** perché richiede un secondo programma e perché nel caso in cui le linee di cache in cui risiede il vettore RIS andassero sovrascritte da un altro programma esterno, si riavrebbe il caso di partenza con una serie continua di write miss.

Il principio di località spaziale, invece, vale e viene ampliamente sfruttato. Tutti i byte su cui andiamo ad operare si trovano a indirizzi contigui, perciò ogniqualvolta importiamo una linea di cache, tutti i byte che stiamo importando *saranno effettivamente utilizzati* in un futuro prossimo.

0.5.5 Punto (e)

Rispetto ai punti precedenti, questo è di una semplicità disarmante. L'unico caso in cui si renderanno necessari cicli di write back è all'inizio della trasmissione di RIS sulla porta seriale. Quando il DMAC cerca di trasferire il dato, la CPU riporta in memoria tutte le linee di cache in cui risiede RIS. Ogni ciclo di write back porta in memoria 8B e risiedendo tutti i 2KB del vettore RIS in cache, saranno necessari $N_{\text{WB}} = \frac{2^{11}}{2^3} = 2^8 = 256$ cicli di write back.

0.5.6 Punto (f)

Con una cache da 16KB la situazione non avrebbe subito alcuna modifica. 8KB di cache sono già sufficiente a gestire tutti e 4 i vettori senza alcuna sovrascrittura.

Diverso sarebbe il caso con una cache da 4KB. Le sovrascritture avrebbero reso praticamente inutile la cache, perché si potrebbero mantenere contemporaneamente in cache solo 2 vettori alla

Tabella 7: Evoluzione della cache per le prime 33 operazioni su byte

Code	M/H	Set	Set_{ID}	Offset	TAG	L/WB	MESI
`lb ax, A1(0)`	RM	0	00 H	00 H	20 H	L[A1(0-31)]	I →E
`or bx, ax, A2(0)`	RM	1	00 H	00 H	20 H	L[A2(0-31)]	I →E
`lb cx, RIS(0)`	RM	0	00 H	00 H	20 H	L[RIS(0-31)]	E
`and cx, bx, A3(0)`	RM	1	00 H	00 H	20 H	L[A3(0-31)]	E
`sb RIS(0), cx`	WH	0	00 H	00 H	20 H		E →M
`lb ax, A1(1)`	RM	0	00 H	01 H	20 H	WB(00 H,0)	M →S
						L[A1(0-31)]	S →E
`or bx, ax, A2(1)`	RM	1	00 H	01 H	20 H	L[A2(0-31)]	E
`lb cx, RIS(1)`	RM	0	00 H	01 H	20 H	L[RIS(0-31)]	E
`and cx, bx, A3(1)`	RM	1	00 H	01 H	20 H	L[A3(0-31)]	E
`sb RIS(1), cx`	WH	0	00 H	01 H	20 H		E →M
`lb ax, A1(2)`	RM	0	00 H	02 H	20 H	WB(00 H,0)	M →S
						L[A1(0-31)]	S →E
`or bx, ax, A2(2)`	RM	1	00 H	02 H	20 H	L[A2(0-31)]	E
`lb cx, RIS(2)`	RM	0	00 H	02 H	20 H	L[RIS(0-31)]	E
`and cx, bx, A3(2)`	RM	1	00 H	02 H	20 H	L[A3(0-31)]	E
`sb RIS(2), cx`	WH	0	00 H	02 H	20 H		E →M
`lb ax, A1(1)`	RM	0	00 H	01 H	20 H	WB(00 H,0)	M →S
						L[A1(0-31)]	S →E
...	...	...	...	...	...	...	...
`lb ax, A1(31)`	RM	0	00 H	1F H	20 H	WB(00 H,0)	M →S
						L[A1(0-31)]	S →E
`or bx, ax, A2(31)`	RM	1	00 H	1F H	20 H	L[A2(0-31)]	E
`lb cx, RIS(31)`	RM	0	00 H	1F H	20 H	L[RIS(0-31)]	E
`and cx, bx, A3(31)`	RM	1	00 H	1F H	20 H	L[A3(0-31)]	E
`sb RIS(31), cx`	WH	0	00 H	1F H	20 H		E →M
`lb ax, A1(32)`	RM	0	01 H	00 H	20 H	L[A1(32-63)]	I →E
`or bx, ax, A2(32)`	RM	1	01 H	00 H	20 H	L[A2(0-31)]	I →E
`lb cx, RIS(32)`	RM	0	01 H	00 H	20 H	L[RIS(0-31)]	I →E
`and cx, bx, A3(32)`	RM	1	01 H	00 H	20 H	L[A3(0-31)]	I →E
`sb RIS(32), cx`	WH	0	01 H	00 H	20 H		E →M
`lb ax, A1(33)`	RM	0	01 H	01 H	20 H	WB(01 H,0)	M →S
						L[A1(32-63)]	S →E
`or bx, ax, A2(33)`	RM	1	01 H	01 H	20 H	L[A2(32-63)]	E
...	...	...	...	...	...	...	...

volta (a causa della presenza di due sole vie), diciamo A1 e A2, dovendo sovrascrivere ogni linea ogni volta che si caricano i due successivi vettori A3 e RIS. Questo nel corso della stessa operazione. Al byte successivo A1 e A2 sovrascriverebbero i precedenti A3 e RIS perché sarebbero sulla stessa linea di cache. Avremmo una miss rate del 100% o quasi. L'ultima istruzione di `sb` in realtà potrebbe andare a buon fine in cache, ma andrebbe fatto poco dopo un write back perché quella linea andrebbe sovrascritta poco dopo. Solo una volta su 32 non sarà richiesto il write back, cioè quando si opera sull'ultimo byte in linea di cache e si passa alla successiva. Tanto varrebbe utilizzare il codice assembly originale per ridurre le istruzioni, visto che gli accessi in memoria passerebbero da circa 5 per operazione a 4 per operazione. Ancora meglio sarebbe disabilitare la cache, visto la sua completa inutilità in questo caso.

Per un'idea più precisa dell'evoluzione della cache in questo caso, si faccia riferimento alla tabella 7 che esprime, anche se non richiesto dall'esame, anche l'evoluzione degli stati *MESI*.