UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato Calcolatori Elettronici II:

Architettura e funzionamento dell'interfaccia parallela PIA

Prof.ssa Casola Valentina

Anno Accademico 2019/2020

Studenti:

Coppola Vincenzo	Matr. M63/1000
Della Torca Salvatore	Matr. M63/1011

Indice

Capitolo 1

PIA: Peripheral Interface Adapter

Il modello generale semplificato di un calcolatore è costituito da:

- memoria centrale (RAM);
- unità di controllo (CPU);
- unità logico-aritmetica (ALU);
- periferiche di I/O.

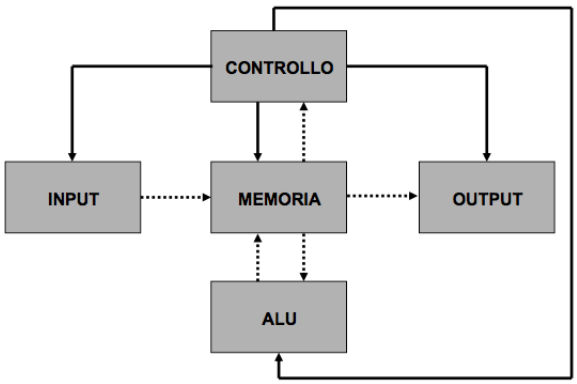

Fig. 1.1: Modello generale

Le periferiche hanno due diversi meccanismi per poter comunicare col processore, interruzioni e polling, che sono necessari in quanto le periferiche sono *asincrone* rispetto all'esecuzione del processore.
Nel caso delle interruzioni bisogna capire qual è la periferica che ha generato l'interruzione, qual è la priorità, qual è la ISR e come gestire il context switch.
Il processore presenta 3 linee per le interruzioni, che sono collegate ad un decoder, e ogni periferica è collegata ad una linea di interruzione ed è costituita da un registro di *modo*

(controllo), un registro di *stato*, e un registro *dati*.
Il collegamento tra processore e periferica è realizzato mediante un insieme di linee dato e linee indirizzo.

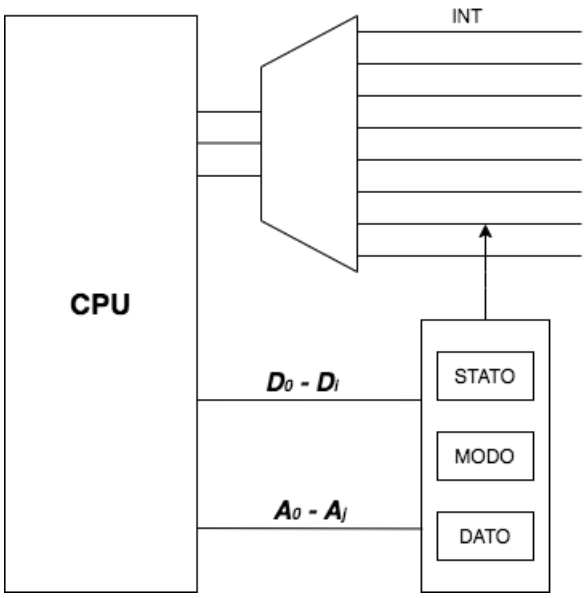

Fig. 1.2: Comunicazione processore-periferica

Una volta collegata la periferica al processore è necessario programmarla mediante due operazioni: *inizializzazione*, dove si setta il registro di modo della periferica, e *scrittura della ISR* associata alla periferica.
Le periferiche non vengono però collegate direttamente al processore, ma necessitano di un **adattatore** che gestisca il *protocollo di sincronizzazione* con la CPU.
La **PIA** *(Peripheral Interface Adapter)* è quindi responsabile di garantire l'interfacciamento tra periferica e CPU; l'interfacciamento tra adattatore e periferica e tra periferica e processore è sempre lo stesso e ciò che cambia è solamente la periferica che deve interfacciarsi.

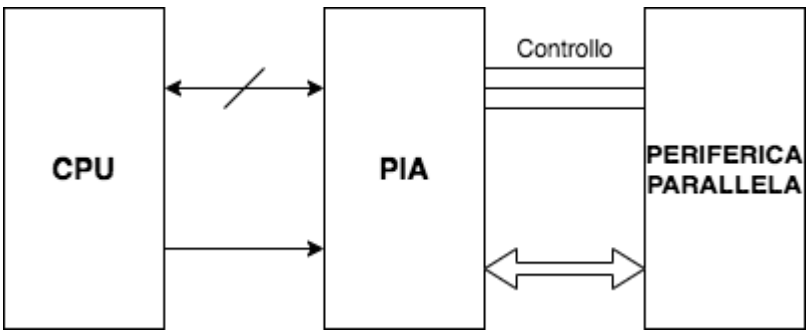

Fig. 1.3: CPU-PIA-PERIFERICA

La **PIA MC6821** è un dispositivo parallelo con parallelismo a 8 bit, ossia il bus di collegamento tra *processore/adattatore* e *adattatore/periferica* è a 8 bit.
Questa periferica presenta due sezioni quasi identiche, denominate *PORTO A* e *PORTO B*, di 8 bit dati configurabili come linee di ingresso o di uscita e alle quali è possibile collegare sia due dispositivi con parallelismo a 8 bit, sia un unico dispositivo con parallelismo a 16 bit.
Dal lato del processore possiamo distinguere diversi segnali:

- *Data Bus bidirezionale*, costituito dalle linee $D_0 - D_7$ di tipo *three-state* e tale che la direzione dei dati sul bus dipenda dal segnale di R/W;
- *Enable*, che è l'unico segnale di sincronizzazione che viene fornito alla PIA;
- *Read/Write*, per controllare la direzione del trasferimento dei dati sul data bus; se $\mathbf{RW = 0}$ l'operazione è di scrittura, altrimenti è di lettura;
- *Reset*, che opera in logica negata;
- *Chip Select* (CS_0, CS_1, CS_2), per selezionare la PIA $(CS_0 = 1, CS_1 = 1, CS_2 = 0)$;
- *Register Select* (RS_0, RS_1), per poter selezionare i registri interni alla PIA (3 per porto); a tal proposito, si usa la tecnica del *memory mapped* per mappare i registri interni della PIA e queste due linee sono utilizzate insieme al *Control Register* per selezionare un particolare registro che deve essere letto o scritto;
- *Interrupt Request (IRQA, IRQB)*, che agiscono per interrompere la CPU direttamente o attraverso il circuito di priorità di interrupt. Queste linee sono "open drain" e questo permette di legare tutte le linee di richiesta di interruzione in una configurazione wire-OR. Ogni linea di Interrupt Request ha due flag bit interni di interrupt che possono causare l'abbassamento della linea di interrupt request e ogni flag bit è associato ad una particolare linea di interrupt. Dopo la richiesta di un interruzione ci sarà una routine software che legge e controlla tutti i registri di controllo secondo un criterio prestabilito e dopo aver letto i dati dai registri la CPU cancella automaticamente il segnale di richiesta di interruzione disabilitando i bit.

Dal lato della periferica invece troviamo:

- *Data Bus bidirezionali* $(PA_0 - PA_7)$ per il porto A; ognuna di queste linee può essere programmata per operare come input o output settando rispettivamente a **0** o a **1** i corrispondenti bit del registro *DDR_A (Data Direction Register)*;

- *Data Bus bidirezionali ($PB_0 - PB_7$)* per il porto B; ognuna di queste linee può essere programmata per operare come input o output settando rispettivamente a **0** o a **1** i corrispondenti bit del registro *DDR_B (Data Direction Register)*;
- *Interrupt Input ($CA_1 - CB_1$)*, sono linee che operano solo come input e settano gli interrupt flag del *control register*;
- *Peripheral Control (CA_2)*, che può essere programmata per operare come interrupt input o come peripheral control output in funzione del valore che assume il bit 5 del *control register*;
- *Peripheral Control (CB_2)*, che può essere programmata per operare come interrupt input o come peripheral control output in funzione del valore che assume il bit 5 del *control register*.

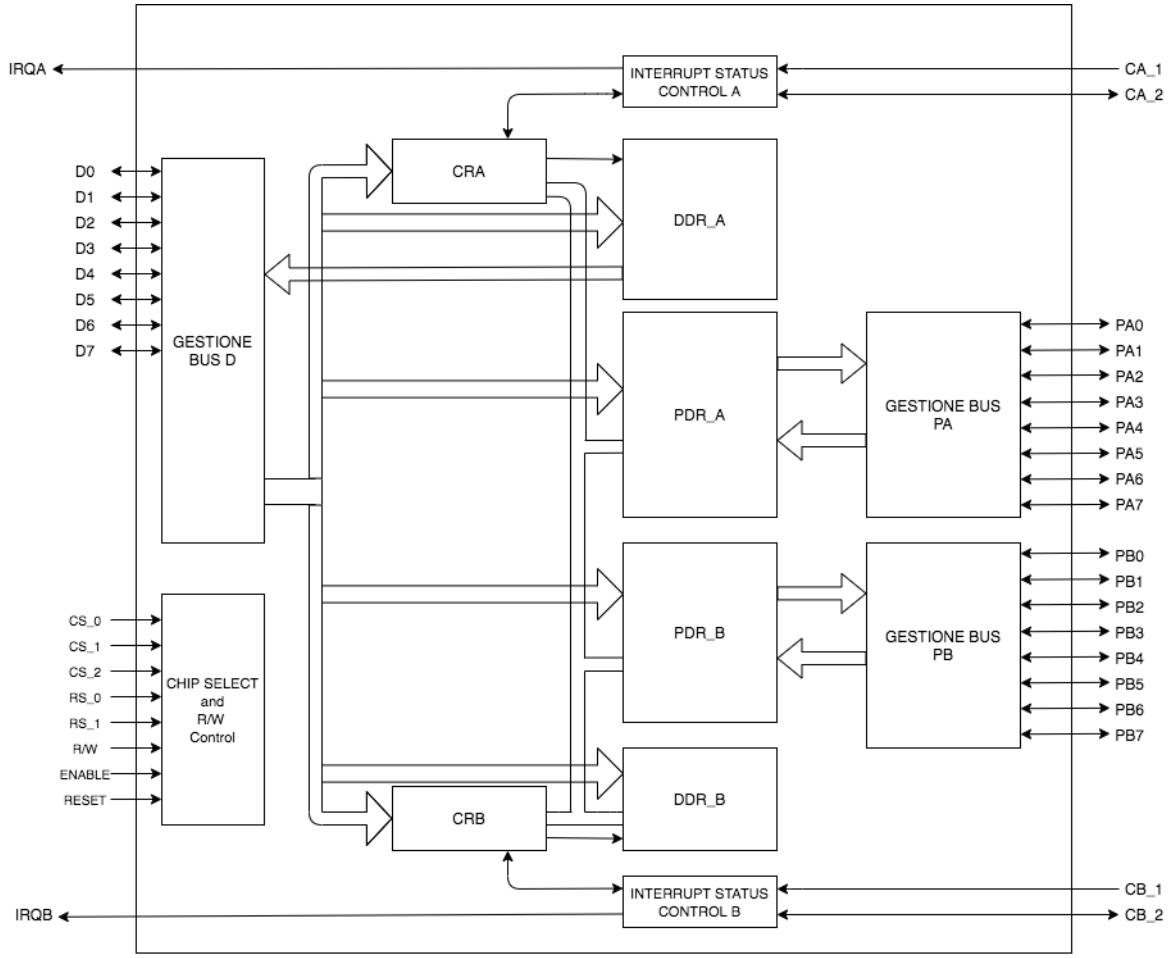

Fig. 1.4: Architettura PIA

Il segnale di **reset** funziona in logica negata e ha l'effetto di azzerare tutti i registri della PIA; successivamente al reset, la PIA deve essere configurata.
La PIA presenta 6 registri interni, 3 per ogni porto, e la selezione di questi registri è gestita da RS_0, RS_1 e dal bit 2 del *control register*:

- se $RS_1 = 0$
  - se $RS_0 = 0$ allora
    - se $CRA_2 = 1$ si accede al PDR_A;
    - se $CRA_2 = 0$ si accede al DDR_A;
  - altrimenti se $RS_0 = 1$ allora si accede al **CRA**;
- altrimenti se $RS_1 = 1$ allora
  - se $RS_0 = 0$ allora
    - se $CRB_2 = 1$ si accede al PDR_B;
    - se $CRB_2 = 0$ si accede al DDR_B;
  - altrimenti se $RS_0 = 1$ allora si accede al **CRB**.

RS1	RS0	Control Register Bit CRA-2	Control Register Bit CRB-2	Location Selected
0	0	1	X	Peripheral Register A
0	0	0	X	Data Direction Register A
0	1	X	X	Control Register A
1	0	X	1	Peripheral Register B
1	0	X	0	Data Direction Register B
1	1	X	X	Control Register B

X = Don't Care

Fig. 1.5: Internal Addressing

Control Registers

I due *control registers CRA e CRB* permettono al processore di controllare le operazioni delle linee CA1, CA2, CB1 e CB2. Inoltre permettono al processore di abilitare le linee di interruzione e monitorare lo stato dei flag di interrupt.
In particolare, come vedremo nel dettaglio successivamente, i bit da 0 a 5 possono essere scritti o letti dal processore, mentre i bit 6 e 7 possono essere solamente letti e vengono modificati da *external interrupts* che si presentano sulle linee di controllo CA1, CA2, CB1 o CB2.

b7	b6	b5	b4	b3	b2	b1	b0
IRQA(B)1 Flag	IRQA(B)2 Flag	CA2 (CB2) Control			DDR Access	CA1 (CB1) Control	

Fig. 1.6: Control Register

BIT 0 - BIT 1

Questi due bit permettono di abilitare/disabilitare l'interrupt request:

- se b0 = 0 la PIA non è sensibile alle interruzioni;
- se b0 = 1 la PIA è sensibile alle interruzioni.

Se la PIA è sensibile alle interruzioni, $b1$ determina su quale fronte della variazione di CA1 (CB1) bisogna interrompere:

- se b1 = 0 la PIA è sensibile sul fronte di discesa di CA1(CB1);
- se b1 = 1 la PIA è sensibile sul fronte di salita di CA1(CB1).

BIT 2

Determina a quale registro accedere:

- se b2 = 0 allora si accede al DDR;
- se b2 = 1 allora si accede al PDR.

BIT 3 - BIT 4 - BIT 5

Questi 3 bit servono per determinare il comportamento delle linee CA2 e CB2 che possono operare sia come input che come output.
In funzione del valore di $b5$ possiamo distinguere due casi:

- se b5 = 0 la linea CA2 (CB2) è impostata come input e i bit $b4$ e $b3$ operano esattamente come i bit $b1$ e $b0$, cioè determinano se la PIA è sensibile o meno alle interruzioni e su che fronte della variazione di CA2 (CB2) è sensibile;
- se b5 = 1 la linea CA2 (CB2) è impostata per operare come output. In funzione del valore di $b4$ e $b3$ possiamo distinguere diversi casi:
  - se b4 = 0
    - se b3 = 0 allora si ha la modalità di *HANDSHAKING*; supponiamo che la periferica voglia inviare un dato alla PIA: per prima cosa mette i dati sul BUS DATI e per comunicare alla PIA che il dato è pronto per essere

letto abbassa la linea CA1 e sulla transizione alto-basso si alza la linea CA2. Internamente alla PIA, il flag IRQA(B)1 si alza e viene inviata al processore una richiesta di interruzione che porterà all'esecuzione della ISR e alla lettura del registro dati. Quando terminerà la lettura il flag di interruzione verrà abbassato e con esso anche il segnale CA2. Nel caso del porto B, il flag di interruzione si abbassa con un'operazione di lettura ma, a differenza del segnale CA2, il segnale CB2 si abbassa con un'operazione di scrittura;

  - ∗ se b3 = 1 allora CA2 si abbassa a seguito di un'operazione di lettura del PDR avvenuta sul fronte di discesa del segnale di enable;

- se b4 = 1 allora in funzione di $b3$ si setta o si resetta la linea CA2 (CB2): la linea CA2 viene abbassata se la CPU scrive 0 nel $b3$, altrimenti viene alzata se la CPU scrive 1 nel $b3$.

BIT 6

É un *interrupt flag (IRQA(B)2)*: quando CA2 (CB2) è settata come input, IRQA(B) viene alzata sulla transizione attiva di CA2 (CB2); viene resettato automaticamente a seguito di una lettura del PDR da parte del processore, oppure attraverso un hardware reset.
Se CA2 (CB2) è settata come output allora b6 = 0.

BIT 7

É un *interrupt flag (IRQA(B)1)*: viene messo a 1 sulla transizione attiva di CA1 (CB1) e viene resettato automaticamente a seguito di una lettura del PDR da parte del processore oppure via hardware.

Capitolo 2

Soluzione

In questo capitolo viene analizzato il codice sviluppato per implementare la PIA e il relativo funzionamento per la modalità che abbiamo voluto analizzare, ovvero l'handshaking.

2.1 Gestione bus Bidirezionali

Innanzitutto analizzeremo un component realizzato appositamente per gestire i bus bidirezionali presenti nella PIA come se fossero dei buffer three-state.
In particolare utilizzeremo il component solamente quando il bus deve essere usato come output.

Componente per gestire i bus bidirezionali

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Bidirezionali is
    generic (N: integer);
    port(Bidi:      inout std_logic_vector(N-1 downto 0);
         Input:     in std_logic_vector(N-1 downto 0);
         Enable:    in std_logic;
         Interr:    in std_logic
        );
end Bidirezionali;

architecture Behavioural of Bidirezionali is
    signal temp1: std_logic_vector(N-1 downto 0) := (others => 'Z');

begin
    process (ENABLE)
    begin
        if(ENABLE'event and ENABLE='0') then
                temp1 <= Input;
        end if;
    end process;

    process(Interr, bidi, temp1)
    begin
        if(Interr = '1') then
            Bidi <= temp1;
        else
            Bidi <= (others => 'Z');
        end if;

    end process;
end Behavioural;
```

I pin del component *Bidirezionali* sono:

- *Bidi*, che rappresenta il bus bidirezionale da gestire;
- *Input*, che rappresenta la sorgente che vuole scrivere in *Bidi* nel caso in cui questo ultimo funzioni come output,;
- *Enable*, che rappresenta il clock che permette il funzionamento di questo component (coincide con l'Enable della PIA);
- *Interr*, ossia l'interruttore che permette il funzionamento del component.

Questo componente funziona come un buffer three-state il cui interruttore è chiuso se assume valore '1': i dettagli implementativi verranno spiegati successivamente.

2.2 PIA

In questa sezione verrà analizzata la PIA partendo dalla sua interfaccia e dalla definizione dei segnali interni fino ad analizzare in modo dettagliato il funzionamento di ogni sua parte.

2.2.1 Interfaccia della PIA

Partiamo dai vari pin della PIA.

Interfaccia PIA

```
entity PIA is
    port(
    ENABLE:     in std_logic;
    CS :        in std_logic_vector(2 downto 0);
    RESET:      in std_logic;
    PA:         inout std_logic_vector(7 downto 0);
    PB:         inout std_logic_vector(7 downto 0);
    CA1:        in std_logic;
    CA2:        inout std_logic;
    CB1:        in std_logic;
    CB2:        inout std_logic;
    D:          inout std_logic_vector(7 downto 0);
    RW:         in std_logic;
    RS:         in std_logic_vector(0 to 1);
    IRQA:       out std_logic;
    IRQB:       out std_logic
    );
end PIA;
```

- *ENABLE* è il clock con il quale la PIA funziona;
- *CS* è il chip select, ovvero i segnali usati dalla CPU per selezionare e avviare la PIA (devono assumere valore 011);
- *RESET* serve a resettare tutti i registri interni della PIA;
- *PA* è il porto bidirezionale che utilizza il dispositivo A per scambiare dati con la PIA, e la sua direzione è gestita dal registro DDRA;
- *PB* è il porto bidirezionale che utilizza il dispositivo B per scambiare dati con la PIA, e la sua direzione è gestita dal DDRB;
- *CA1* e *CA2* sono linee che servono a istanziare un protocollo di comunicazione: in particolare *CA1* sarà solo di input e serve al dispositivo A per interrompere, *CA2* è bidirezionale e il suo comportamento viene stabilito dal registro di controllo;
- *CB1* e *CB2* si comportano allo stesso modo di *CA1* e *CA2*, ma per il dispositivo B;
- *D* è il bus bidirezionale attraverso il quale PIA e CPU scambiano dati;

- *RW* è la linea che usa la CPU per scrivere o leggere dalla PIA: in particolare, se RW=1 la CPU sta leggendo dati da uno dei registri interni della PIA, se RW=0 sta scrivendo;
- *RS* sono le linee per indirizzare i registri interni della PIA;
- *IRQA* è la linea che la PIA usa per segnalare alla CPU che il dispositivo A sta interrompendo;
- *IRQB* è la linea che la PIA usa per segnalare alla CPU che il dispositivo B sta interrompendo;

2.2.2 Definizione dei segnali interni

In questa sezione verranno mostrati i segnali interni, definiti nell'architecture.

Segnali interni

```
signal CRA: std_logic_vector(7 downto 0) := (others => '0');
signal CRB: std_logic_vector(7 downto 0) := (others => '0');

signal DDRA: std_logic_vector(7 downto 0) := (others => '0');
signal DDRB: std_logic_vector(7 downto 0) := (others => '0');

signal PDRA: std_logic_vector(7 downto 0) := (others => '0');
signal PDRB: std_logic_vector(7 downto 0) := (others => '0');


signal d_in: std_logic_vector(7 downto 0) := (others => 'Z');
signal interr: std_logic := 'Z';

signal pa_in: std_logic_vector(7 downto 0) := (others => 'Z');
signal pa_interr: std_logic := 'Z';

signal pb_in: std_logic_vector(7 downto 0) := (others => 'Z');
signal pb_interr: std_logic := 'Z';


component Bidirezionali is
    generic (N: integer);
    port(Bidi:            inout std_logic_vector(N-1 downto 0);
        Input:            in std_logic_vector(N-1 downto 0);
        Enable:         in std_logic;
        Interr:           in std_logic
        );
    end component;
```

- *CRA* e *CRB* sono i registri di controllo utilizzati dalla CPU per stabilire il comportamento nei confronti dei dispositivi sul porto A e sul porto B;
- *DDRA* e *DDRB* sono i registri direzione che servono a settare la direzione rispettivamente dei bus *PA* e *PB*;
- *PDRA* e *PDRB* sono i registri dati rispettivamente dei dispositivi A e B: essi possono essere acceduti in lettura e in scrittura sia dalla CPU mediante il bus *D*, sia dai dispositivi A e B mediante i bus *PA* e *PB* rispettivamente;

- *d_in, interr* servono a gestire il bus bidirezionale *D*;
- *pa_in, pa_interr* servono a gestire il bus bidirezionale *PA*;
- *pb_in, pb_iterr* servono a gestire il bus bidirezionale *PB*;
- *Bidirezionali* è il component per gestire i bus bidirezionali come output.

2.2.3 Funzionamento della PIA

In questa sezione verranno analizzati i dettagli implementativi che si trovano nella sezione **begin**.

Gestione bus bidirezionali

```
Gestione_bus_D: Bidirezionali generic map (8)
                              port map(D, d_in, ENABLE, interr);
Gestione_bus_PA: Bidirezionali generic map(8)
                               port map(PA, pa_in, ENABLE, pa_interr);
Gestione_bus_PB: Bidirezionali generic map(8)
                               port map(PB, pb_in, ENABLE, pb_interr);

Interr <= RW;
pa_interr <= DDRA(0) and DDRA(1) and DDRA(2) and DDRA(3) and DDRA(4)
             and DDRA(5) and DDRA(6) and DDRA(7);
pb_interr <= DDRB(0) and DDRB(1) and DDRB(2) and DDRB(3) and DDRB(4)
             and DDRB(5) and DDRB(6) and DDRB(7);
```

- *Interr* assume lo stesso valore di *RW*, quindi l'interruttore del buffer three-state per il bus *D* si chiude quando *RW*='1', ovvero quando la CPU vuole leggere;
- *pa_interr* e *pb_interr* assumono valore pari a 1 solo quando DDRA e DDRB hanno tutti i bit alti: questo significa che i bus *PA* e *PB* sono di output, ovvero i dispositivi A e B stanno leggendo rispettivamente dal PDRA e PDRB.
  La scelta di assegnare agli interruttori la **and** bit a bit tra i bit dei rispettivi DDR è stata fatta per mostrare la dipendenza tra gli interruttori e tali registri.

Questa sezione di codice chiarisce invece come la modalità di funzionamento che abbiamo voluto analizzare influenza i registri di controllo e lo scatenarsi delle interruzioni.

Scelte

```
IRQA <= CRA(7);
IRQB <= CRB(7);

CRA(6) <= '0';
CRB(6) <= '0';
```

Avendo posto a priori i bit CRA(6) e CRB(6) a 0 abbiamo scelto di analizzare il caso in cui la linee *CA2* e *CB2* si comportano come output, quindi non influenzano i bit 6 dei relativi registri di controllo, e questo significa anche che non influenzano le linee di interruzione.

Infatti, a *IRQA* e *IRQB* sono stati assegnati rispettivamente CRA(7) e CRB(7), in quanto le interruzioni vengono scatenate solo dalle linee di input *CA1* e *CB1* che modificano solo i bit 7 dei rispettivi registri di controllo.
In particolare la modalità che abbiamo voluto analizzare è quella di **handshaking**, in cui le linee C*1 (CA1 o CB1) vengono abbassate dal dispositivo per interrompere, mentre le linee C*2 in quell'istante vengono alzate; verranno poi abbassate dalla PIA quando l'interruzione è stata gestita, così da segnalarlo al dispositivo. La modalità di handshaking prevede di scrivere 100 nei bit 5,4 e 3 del registro di controllo.

Le varie funzionalità sono state inserite nel seguente process che si attiva ad ogni fronte di discesa del segnale di *ENABLE*.

Process

```
funzionamento:process(ENABLE)
begin
    if(ENABLE'event and ENABLE='0') then
```

Analizziamo nel dettaglio le varie porzioni di codice.

Reset

Reset

```
if (RESET='0') then
    CRA <= (others => '0');
    CRB <= (others => '0');
    DDRA <= (others => '0');
    DDRB <= (others => '0');
    PDRA <= (others => '0');
    PDRB <= (others => '0');

else
```

Questa porzione di codice si occupa di resettare il dispositivo mettendo il contenuto di tutti i registri interni a 0; il segnale di *RESET* funziona in logica negata, quindi il resettaggio avverrà quando tale segnale assume valore pari a 0, altrienti la PIA può eseguire le proprie funzionalità. La CPU è responsabile di eseguire il reset.

A interrompe

A interrompe

```
if (CA1='0') then
    if(DDRA <=    "00000000")    then
        PDRA <= PA;
    end if;
    if(CRA(1 downto 0) = "01") then
        CRA(7) <= '1';
        if(CRA(5 downto 3) <= "100") then
            CA2 <= '1';
        end if;
    end if;
end if;
```

Questa porzione di codice mostra invece cosa accade quando il dispositivo A (legato al porto A) vuole comunicare con il processore. L'interruzione si attiva quando il dispositivo abbassa *CA1*, e in questo caso vengono eseguiti diversi controlli:

- si controlla se il *DDRA* contiene tutti 0: ciò significa che le linee *PA* sono di input alla PIA e quindi il dato che manda il dispositivo A può essere inserito nel *PDRA*;
- si controllano poi i bit 1 e 0 del *CRA*: essi devono assumere rispettivamente valore 0 e 1 in quanto il bit 0 dice se l'interruzione generata dal dispositivo A mediante la linea *CA1* viene vista o meno; il bit 1 dice invece su che fronte viene vista l'interruzione generata su *CA1* (nella modalità di handshaking il dispositivo cerca di interrompere abbassando *CA1*, ecco perché è stato messo il bit a 0). Se entrambi i bit sono stati settati correttamente, allora viene alzato il CRA(7) che, essendo legato a *IRQA*, genera l'interruzione verso la CPU.
- infine si controlla se è stata impostata la modalità handshaking mettendo 100 rispettivamente nei bit 5, 4 e 3 del *CRA*: se ciò accade, allora nello stesso istante in cui viene visto *CA1* basso, si alza *CA2*. Ciò serve a instaurare il protocollo di handshaking, ovvero facendo così la PIA segnala al dispositivo A che l'interruzione che egli ha generato è stata accettata, e non può generarne un'altra, né inviare un altro dato, fino a quando *CA2* non ritornerà basso (ossia fin quando la CPU non termina la gestione dell'interruzione). Ovviamente *CA2* resta basso non solo se non è stata impostata la modalità handshaking, ma anche se non si alza il bit CRA(7).

B interrompe

B interrompe

```
if (CB1='0') then
    if(DDRB <=    "00000000")    then
        PDRB <= PB;
    end if;
    if(CRB(1 downto 0) = "01") then
        CRB(7) <= '1';
        if(CRB(5 downto 3) <= "100") then
            CB2 <= '1';
        end if;
    end if;
end if;
```

Questa sezione gestisce il caso in cui il dispositivo B vuole interrompere la CPU e scrivere nel *PDRB*.
Viene mostrato solo il codice poiché le azioni che la PIA esegue quando il dispositivo A interrompe sono le stesse che vengono eseguite quando il dispositivo B interrompe, con l'unica differenza che stavolta vengono trattati i registri *PDRB*, *DDRB* e *CRB*.

Nelle seguenti sezioni viene analizzato cosa accade quando la CPU accede ai registri interni alla PIA; tutti i vari comportamenti verranno gestiti all'interno di un *case* dove si analizzano i diversi valori che assume *RS*, che permette di controllare a quale registro sta accedendo la CPU.

Controllo di RS

```
case RS is
```

CPU accede ai Control Register

CPU accede al CRA

```
when "10" =>
    if(RW='0') then
        CRA(5 downto 0) <= D(5 downto 0);
    elsif(RW='1') then
        d_in(5 downto 0) <= CRA(5 downto 0);
    end if;
```

La CPU accede al registro *CRA* scrivendo 10 in *RS*: in particolare può accedere solo ai bit da 5 a 0 (non ai bit 6 e 7). Se accede in scrittura (*RW*=0) allora scrive nei 6 bit meno significativi del registro, con *RW*=1 li legge. L'accesso in lettura o in scrittura è gestito dal component *Gestione_bus_D*.

CPU accede al CRB

```
when "11" =>
    if(RW='0') then
        CRB(5 downto 0) <= D(5 downto 0);
    elsif(RW='1') then
        d_in(5 downto 0) <= CRB(5 downto 0);
    end if;
```

Le stesse azioni vengono eseguite nel caso in cui la CPU acceda al *CRB* ponendo *RS* pari a 11.

CPU accede a DDRA o PDRA

CPU accede a DDRA o PDRA

```
when "00" =>
    if(CRA(2)='0')then
        if(RW='0') then
            DDRA <= D;
        elsif(RW='1') then
            d_in <= DDRA;
        end if;
    elsif(CRA(2)='1') then
        if(RW='1') then
            CRA(7) <= '0';
            d_in <= PDRA;
            if(CA2='1') then
                CA2 <= '0';
            end if;
        elsif(RW='0') then
            PDRA <= D;
        end if;
    end if;
```

Questa sezione gestisce invece l'accesso ai registri *DDRA* o *PDRA*. Ad entrambi i registri si accede ponendo in *RS* il valore 00, e la selezione di uno dei due avviene settando il bit 2 del registro di controllo:

- se CRA(2) assume valore 0 allora la CPU può accedere in lettura o in scrittura, a seconda del valore di *RW*, al *DDRA*;
- se CRA(2) assume valore 1 allora la CPU accede al *PDRA*, e:
  - se RW=0 accede in scrittura;
  - se accede in lettura, allora il dato del *PDRA* viene messo in *pa_in* (il quale verrà messo poi in *D* dal component), la PIA abbassa il bit CRA(7) per terminare l'interruzione, e quindi abbassa anche la linea *CA2* per segnalare al dispositivo A che l'interruzione è stata gestita.

CPU accede a DDRB o PDRB

CPU accede a DDRB o PDRB

```
when "01" =>
    if(CRB(2)='0') then
        if(RW='0') then
            DDRB <= D;
        elsif(RW='1') then
            d_in <= DDRB;
        end if;
    elsif(CRB(2)='1') then
        if(RW='0') then
            PDRB <= D;
            if(CB2 = '1') then
                CB2 <= '0';
            end if;
        elsif(RW='1') then
            CRB(7) <= '0';
            d_in <= PDRB;
        end if;
    end if;
```

Questa sezione gestisce l'accesso ai registri *DDRB* o *PDRB*. Le considerazioni fatte per l'accesso ai registri *DDRA* e *PDRA* valgono anche in questo caso, ma con 2 differenze sostanziali:

- si controlla il bit CRB(2) per disciplinare l'accesso ad uno dei 2 registri;
- la linea *CB2*, che si alza quando il dispositivo B interrompe (alzando *CB1*), si abbassa quando la CPU **scrive** nel *PDRB*.

I dispositivi A e B leggono dai PDR

Lettura dai PDR

```
if(DDRA = "11111111") then
    pa_in <= PDRA;
end if;

if(DDRB <=    "11111111")    then
    pb_in <= PDRB;
end if;
```

Questa semplice sezione analizza la lettura da parte dei dispositivi A e B dei rispettivi PDR: in particolare, il dato presente nei PDR viene mandato a PA o PB se i DDR sono impostati in modo tale che le linee siano di output. Il passaggio dei dati alle linee PA e PB avviene attraverso il component *Bidirezionali* usando pa_in e pb_in.

Chiusura process

Chiusura

```
            when others =>
                null;

        end case;
    end if;
end if;
```

Questa sezione chiude il process in cui è stato implementato il funzionamento: se *RS* assume valori diversi da 00, 01, 10, 11 non bisogna eseguire nessuna operazione (può assumere anche i valori Indefinito o Indeterminato); si chiudono quindi il costrutto *case*, l'if di controllo del *RESET* e l'if di controllo del fronte di discesa di *ENABLE*.

Capitolo 3

Testing

In questo capitolo verrà mostrato il testing della PIA.
É stato utilizzato il simulatore **ghdl** per compilare, simulare e testare, e il programma **gtkwave** per mostrare le forme d'onda.

3.1 Testbench

Testbench

```
entity testbench is

end testbench;

architecture behavioural of testbench is

    component PIA is
        port(
        ENABLE:     in std_logic;
        CS:            in std_logic_vector(2 downto 0);
        RW:            in std_logic;
        RESET:        in std_logic;
        PA:            inout std_logic_vector(7 downto 0);
        PB:            inout std_logic_vector(7 downto 0);
        CA1:        in std_logic;
        CA2:        inout std_logic;
        CB1:        in std_logic;
        CB2:        inout std_logic;
        D:            inout std_logic_vector(7 downto 0);
        RS:            in std_logic_vector(0 to 1);
        IRQA:        out std_logic;
        IRQB:        out std_logic
        );
    end component;

    signal enable, reset, rw, ca1, ca2, cb1, cb2, irqa, irqb: std_logic  := 'Z';
    signal cs: std_logic_vector(2 downto 0) := (others => 'Z');
    signal rs: std_logic_vector(0 to 1) := (others => 'Z');
    signal pa, pb, d: std_logic_vector(7 downto 0) := (others => 'Z');
    constant period: time := 10 ns;

begin

    pia_test: PIA port map (enable, cs, rw, reset, pa, pb, ca1, ca2, cb1, cb2, d, rs,
                            irqa, irqb);
```

Questa sezione mostra:

- definizione dell'entity vuoto per il testbench;

- un'architettura di tipo behavioural che contiene diverse sezioni:
  - il component **PIA** da testare;
  - definizione dei segnali del testbench da collegare alla PIA;
  - un'istanza del component da testare;
  - due process che verranno analizzati in seguito.

3.2 Generazione del clock

Process che genera la forma d'onda del clock

```
clock: process
    variable count: integer;
begin
    for count in 0 to 55 loop
        enable <= '0';
        wait for period/2;
        enable <= '1';
        wait for period/2;
    end loop;
    wait;
end process;
```

Questo process si occupa di generare un clock il cui periodo è 10 nano secondi: come clock è stato usato il segnale di *ENABLE*, cioè lo stesso della PIA.

3.3 Codice per il testing

In quest'ultima sezione ci occuperemo di mostrare alcuni casi di test che simulano il corretto funzionamento della PIA implementata con modalità handshaking. La sezione è suddivisa in più sotto-sezioni, ognuna delle quali ha il compito di simulare un particolare comportamento della PIA.

Fase iniziale

```
test: process
begin
    wait for 17 ns;
    cs <= "011";
    reset <= '0';
    wait for 30 ns;
    reset <= '1';
    wait for 11 ns;
```

In questa sezione iniziale si imposta il chip-select pari a 011, che è il valore che la CPU usa per selezionare la PIA, e viene abbassato per 30 ns il reset semplicemente per mostrarne il funzionamento.
In seguito vengono mostrati casi più interessanti.

Prima scrittura nel CRA

Scrittura CRA

```
d <= "10100001";
rs <= "10";
wait for 3 ns;
rw <= '0';
wait for 10 ns;
rs <= "ZZ";
d <= (others => 'Z');
rw <= 'Z';
wait for 40 ns;
```

Questa sezione mostra la prima scrittura che la CPU esegue nel *CRA*. Bisogna notare alcune cose: il processore fornisce in ingresso alla PIA comunque 8 bit, ma non può accedere ai bit 6 e 7 del *CRA* (così come nel CRB), e quindi nel *CRA* verranno copiati solo i 6 bit meno significativi. Imposta quindi la modalità handshaking per il dispositivo A e decide di accettarne le interruzioni sul fronte di discesa. Inoltre scrive 0 in CRA(2) così da accedere al *DDRA*. Per comodità, alla fine della scrittura vengono messi a Z i bit sulle linee *D*, mentre *RS* e *RW* vengono messi a Z per evitare di eseguire altre operazioni poiché per qualsiasi altro valore dei due segnali si eseguirebbe comunque un'operazione. La scrittura viene effettuata sul fronte di discesa di *ENABLE*, dopo che il *RESET* si è alzato.

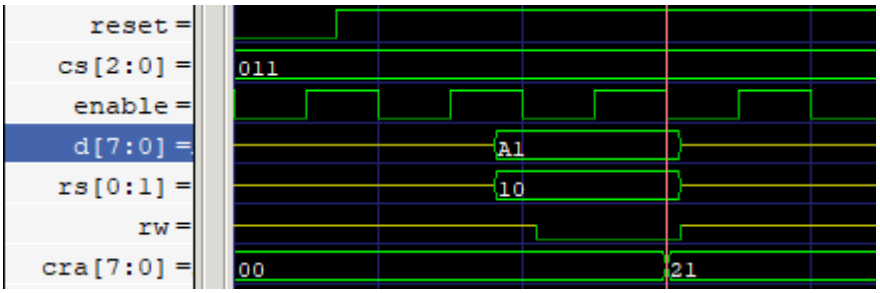

Fig. 3.1: Prima scrittura in PDRA

Scrittura nel DDRA

Scrittura nel DDRA

```
wait for 7 ns;
d <= "00000000";
rs <= "00";
wait for 6 ns;
rw <= '0';
wait for 11 ns;
rs <= "ZZ";
d <= (others => 'Z');
rw <= 'Z';
```

La seguente sezione mostra invece una scrittura nel *DDRA*, in cui vengono messi i bit tutti pari a 0 cosicché il dispositivo A possa scrivere qualcosa nel *PDRA*. Il DDRA viene indirizzato con RS=00, come il PDRA, e la CPU accede al DDRA poiché nella scrittura

precedente aveva settato a 0 il bit CRA(2). In questo caso la scrittura non è visibile poiché precedentemente è stato effettuato il reset e quindi il *DDRA* conteneva già tutti i bit pari a 0.

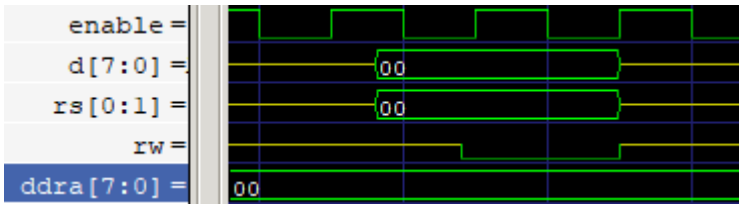

Fig. 3.2: Scrittura nel DDRA

Seconda scrittura nel CRA

Seconda scrittura nel CRA

```
wait for 30 ns;
d <= "00100101";
wait for 6 ns;
rs <= "10";
wait for 3 ns;
rw <= '0';
wait for 10 ns;
rs <= "ZZ";
d <= (others => 'Z');
rw <= 'Z';
wait for 40 ns;
```

In questa sezione la CPU effettua una seconda scrittura nel *CRA* al fine di modificare il solo bit CRA(2) che viene messo ad 1 cosicché la CPU possa accedere al *PDRA*. Da notare che la scrittura avviene sul fronte di discesa di *ENABLE* quando ancora *RW* è basso e *RS* assume valore 10.

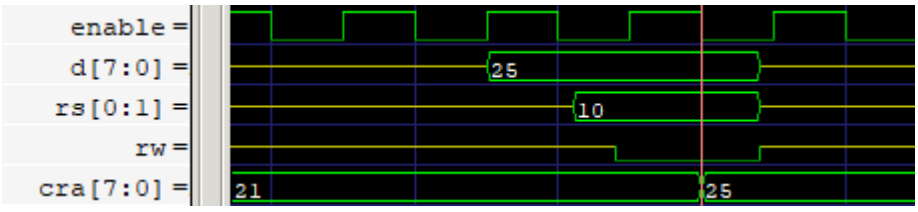

Fig. 3.3: Seconda scrittura in CRA

Dispositivo A interrompe

Dispositivo A interrompe

```
pa <= "10101111";
wait for 7 ns;
ca1 <= '0';
wait for 12 ns;
ca1 <= '1';
```

Questa sezione mostra invece un'interruzione da parte del dispositivo A: egli abbassa per 5 ns la linea *CA1* e mediante la linea *PA* tenta di scrivere un dato in *PDRA*. Analizzando più nel dettaglio questa parte, vediamo che la scrittura nel *PDRA* avviene sul primo fronte di discesa di *ENABLE* che intercetta la linea *CA1* bassa, e la scrittura è permessa poiché precedentemente la CPU, scrivendo nel *DDRA* tutti 0, ha configurato le linee *PA* come input. Inoltre, nel momento in cui *CA1* si abbassa, si alza il bit 7 del CRA, e quindi anche IRQA a cui è legato, per segnalare alla CPU che il dispositivo A ha generato un'interruzione.
Ciò è dovuto al fatto che precedentemente la CPU ha scritto 01 negli ultimi 2 bit meno significativi di *CRA*, indicando quindi che accetta le interruzioni di A ed è sensibile sul fronte di discesa. Inoltre, sempre nel momento in cui *ENABLE* intercetta *CA1* basso sul suo fronte di discesa, si alza anche la linea *CA2*: ciò è dovuto invece all'handshaking, che è stato configuarto scrivendo 100 nei bit 5,4,3 del *CRA*.

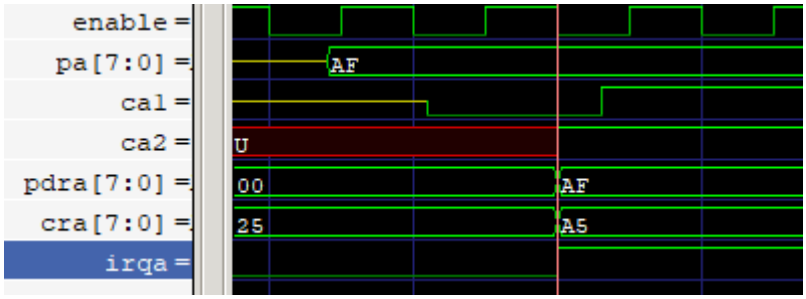

Fig. 3.4: Interruzione da parte di A

CPU legge il PDRA

CPU legge i PDRA

```
wait for 19 ns;
rs <= "00";
rw <= '1';
wait for 29 ns;
rw <= 'Z';
rs <= "ZZ";
```

In questa sezione è stata testata la lettura del *PDRA* da parte della CPU mediante le linee *PA*; la lettura avviene semplicemente indirizzando il PDRA, ponendo 00 in *RS*, con l'opportuno valore del bit CRA(2), e ponendo RW=1.
Il dato preso dal *PDRA* verrà mostrato in *D* un colpo di clock dopo che la lettura è stata intercettata, sul fronte di discesa dell'*ENABLE* successivo, in quanto la lettura passa attraverso il component Bidirezionali. Quando però viene intercettata la lettura, viene abbassato il bit 7 del *CRA* (*che passa da A5 (10100101) a 25 (00100101)*), e quindi anche

IRQA, per segnalare che l'interruzione è stata gestita. Inoltre viene abbassata anche la linea *CA2* per segnalare al dispositivo A che la CPU ha letto il dato e che quindi può riceverne un altro.

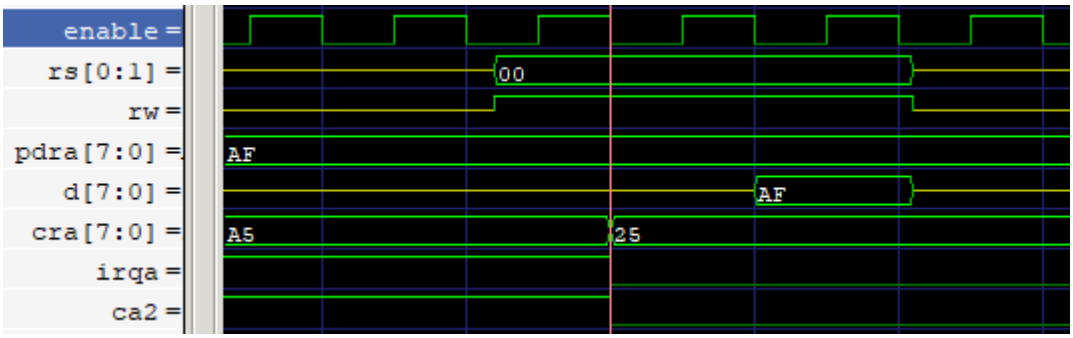

Fig. 3.5: CPU lege il PDRA

Scrittura in CRB

Scrittura in CRB

```
wait for 25 ns;
d <= "00100001";
rs <= "11";
wait for 3 ns;
rw <= '0';
wait for 10 ns;
rs <= "ZZ";
d <= (others => 'Z');
rw <= 'Z';
wait for 40 ns;
```

É stato poi testato anche il comportamento relativo al porto B, che descriveremo più brevemente in quanto ci sono molti punti in comune con l'interazione con A. In questa sezione la CPU configura per la prima volta il *CRB* accedendovi ponendo 11 in *RS*. Anche in questo caso verranno scritti solo i 5 bit meno significativi.

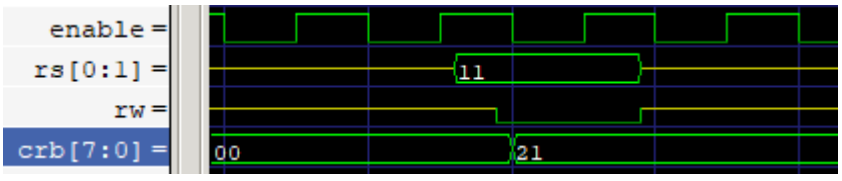

Fig. 3.6: CPU scrive nel CRB

Scrittura del PDRB

Scrittura del PDRB

```
wait for 7 ns;
d <= "11111111";
rs <= "01";
wait for 6 ns;
rw <= '0';
wait for 11 ns;
rs <= "ZZ";
d <= (others => 'Z');
rw <= 'Z';
```

In questa sezione la CPU vuole accedere al *DDRB*, e lo indirizza ponendo 01 in *RS*; accede al *DDRB* (e non al *CRB*) poiché precedentemente in CRB(2) è stato scritto 0. In *DDRB* vengono scritti tutti 1 per rendere output le linee *PB*; infatti dal momento in cui viene intercettata la scrittura sul *DDRB*, sulle linee *PB* si troveranno i dati contenuti in *PDRB*.

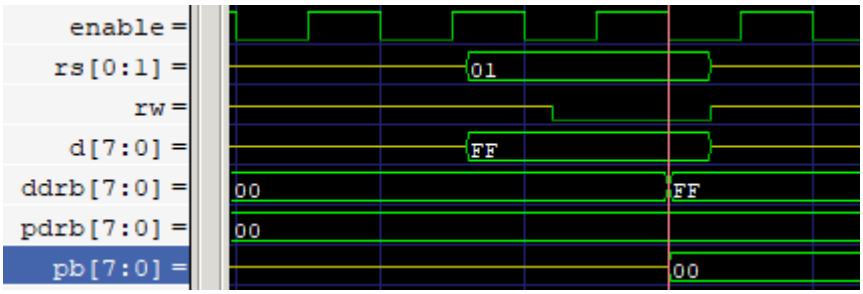

Fig. 3.7: Scrittura del DDRB

Seconda scrittura CRB

Seconda scrittura CRB

```
wait for 25 ns;
cb1 <= '0';                    --Intanto B interrompe
d <= "00100101";
rs <= "11";
wait for 3 ns;
rw <= '0';
wait for 10 ns;
cb1 <= '1';
rw <= 'Z';
wait for 40 ns;
```

In questa sezione vengono mostrati diversi comportamenti. Innanzitutto, la linea *CB1* si abbassa per generare un' interruzione e, avendo settato precedentemente a 01 i bit CRB(1) e CRB(0), si alza il bit CRB(7), e quindi la linea *IRQB*, per segnalare l'interruzione al processore; quindi il contenuto di CRB passerà da 21 (00100001) a A1 (10100001) e, avendo impostato la modalità handshaking, si alza anche *CB2*. Nello stesso momento la CPU vuole scrivere in *CRB* ponendo il dato sul bus e indirizzando il registro, ma non avendo ancora dato il segnale di write, la scrittura non avviene. Il dato viene scritto in

CRB solo 3 ns dopo, quando viene intercettato *RW* basso, e tale scrittura non modifica il bit 7 ma modifica solamente il bit CRB(2) cosicché la CPU possa accedere al *PDRB*.

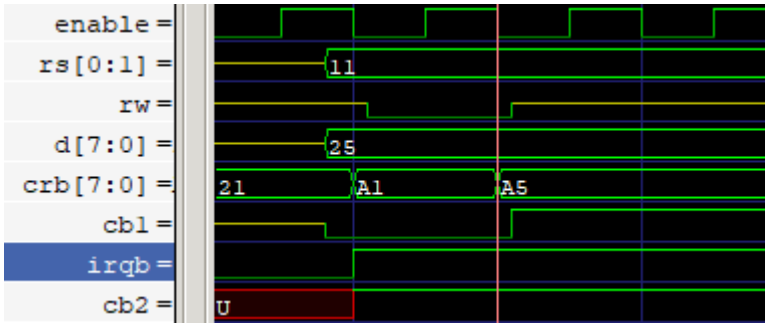

Fig. 3.8: Seconda scrittura in CRB

CPU scrive in PDRB

CPU scrive in PDRB

```
wait for 7 ns;
d <= "11111111";
rs <= "01";
wait for 6 ns;
rw <= '0';
wait for 11 ns;
rs <= "ZZ";
d <= (others => 'Z');
rw <= 'Z';
```

In quest'ultima sezione la CPU scrive un dato nel *PDRB*, indirizzandolo con 01, e vi accede poiché in precedenza aveva settato a 1 il bit CRB(2). Nel momento in cui avviene la scrittura, ossia quando viene intercettata sul fronte di discesa di *ENABLE*, si abbassa la linea *CB2* (al contrario di *CA2* che si abbassava in seguito alla lettura del *PDRA*). Dato che le linee *PB* sono di output, il dato appena scritto in *PDRB* viene mandato su *PB*, ma sempre un colpo di clock dopo (sul successivo fronte di discesa di *ENABLE*) poiché la scrittura su un bus bidirezionale passa per il component Bidirezionali.

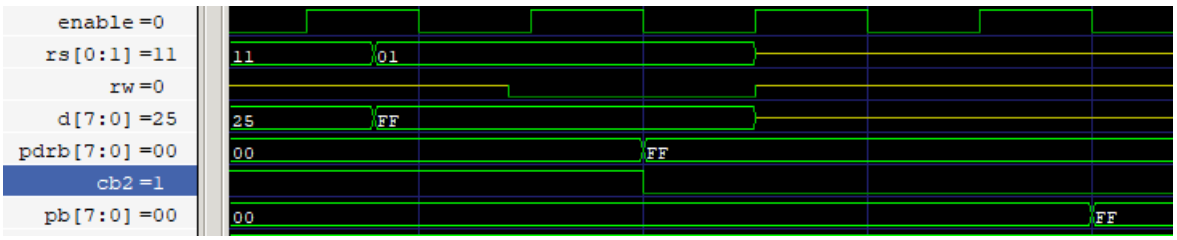

Fig. 3.9: CPU scrive nel PDRB

3.3.1 Testbench complessivo

Mostriamo infine le forme d'onda dell'intero testbench che abbiamo realizzato.

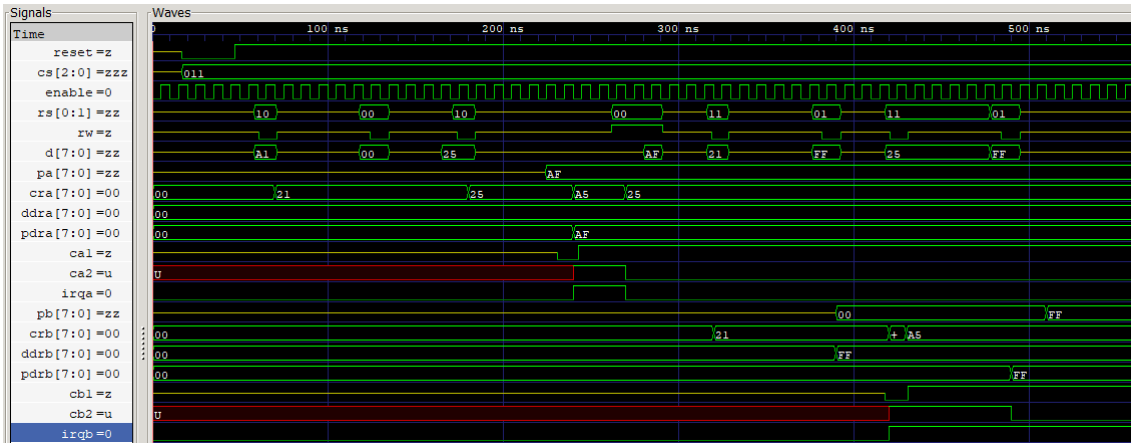

Fig. 3.10: Testbench complessivo