Calcolo SAD tra due blocchi di immagini

Carmine Benedetto, Silvio Bianchi

22 settembre 2011

Indice

Elenco delle figure

1 Specifiche Formali

1.1 Prima fase

Progettare un circuito digitale sincrono che realizzi il calcolo della *SAD*, definita come la somma delle differenze in valore assoluto pixel a pixel, tra due blocchi di immagini monocromatiche A e B. Si considerino blocchi di immagine di dimensioni 16 pixel x 16 pixel, ogni pixel è un numero intero tra 0 e 255 rappresentato su 8 bit. Il circuito ha come ingressi il segnale di *clock*, un segnale di *reset*, un segnale di *enable* e due segnali *PA* e *PB* su cui si ipotizza vengono forniti dall'esterno, in cicli successivi, i 256 pixel dei due blocchi di immagini A e B. In uscita il circuito ha un segnale *SAD* a 16 bit ed un segnale *data_valid* a 1 bit. In condizioni di reset $SAD = 0$ e *Data_valid* $= 0$. *Data_valid* viene settato alla fine del calcolo della *SAD*. Se *enable* $= 0$ il circuito conserva il suo stato indipendentemente dal valore dei segnali di ingresso.

1.2 Seconda fase

Terminata la pima fase paramatrizzare la descrizione VHDL estendola al caso generico di blocchi di dimensione NxN con N potenza di 2.
Nota: i segnali di ingresso e di uscita rimangono gli stessi a meno di *SAD* la cui dimensione sarà n bit, con $2^n \geq (255 * N^2)$; la temporizzazione viene modificata in quanto vengono ricevuti dall'esterno N^2 coppie di pixel.

2 Descrizione dell'algoritmo

Per realizzare il calcolo della *SAD* che rispettasse le specifiche indicate, è stato progettato, utilizzando il linguaggio di programmazione *VHDL*, un circuito sincrono che ad ogni passo prende in ingresso 1 pixel (rappresentato su 8 bit) per ogni immagine, ne effettua la sottrazione, si ricava poi il valore assoluto del risultato della sottrazione e lo invia ad un buffer che tiene memoria delle precedenti somme. Il calcolo della *SAD* si considera completato (con conseguente settaggio della variabile *data_valid*) quando sono stati effettuati NxN passi (con NxN numero dei pixel di un'immagine).

3 Architettura selezionata per la realizzazione

3.1 Modello architetturale

Il modello architetturale scelto per la realizzazione dell'applicazione è stato quello di tipo *behavioural*. In seguito viene mostrato il flusso di progettazione relativo al caso specifico con ingressi ad 8 bit ed uscite a 16 bit. Naruralmente il modello è facilmente estendibile al caso generale.

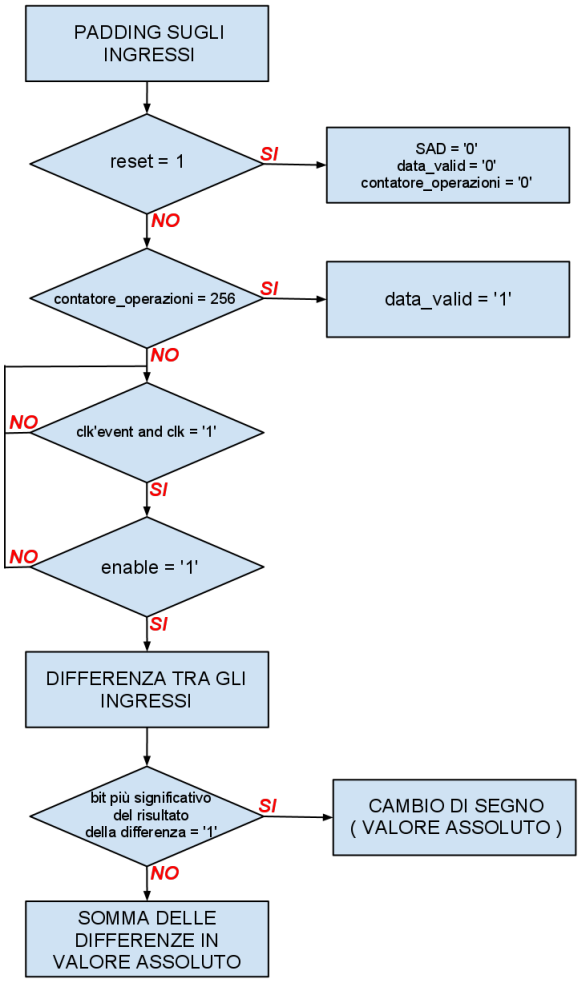

Figura 1: Flusso di progettazione

3.2 Caso specifico

Nel caso specifico i blocchi di immagine monocromatiche hanno dimensioni 16x16 pixel (256 pixel per immagine) ed ogni pixel è un numero intero compreso tra 0 e 255. Analizzando i dati forniti sono stati dimensionati gli ingressi a 8 bit ($2^8 = 256$), mentre per il dimensionamento dell'uscita si è reso necessario un calcolo preventivo del valore massimo ottenibile come somma dei valori assoluti delle differenze. Precisamente sono necessari, al fine del calcolo, 256 differenze, dove ognuna può avere un valore massimo (in modulo) di 255. Il valore massimo ottenibile in uscita dal circuito con questa configurazione è quindi di 256 (numero di pixel, quindi di differenze)* 255 (valore massimo in modulo di ogni differenza) = 65280, valore che necessita di 16 bit per poter essere rappresentato ($2^{16} = 65536$). Il circuito quindi è così strutturato: riceve 2 ingressi *PA* e *PB* a 8 bit, esegue il padding su di essi per uniformarli alla dimensione dell'uscita, esegue la differenza, facendone poi il valore assoluto e somma ad ogni ciclo di clock il valore ottenuto alla somma delle differenze precedenti. Alla 256ima differenza calcolata, il bit *data_valid* viene settato per segnalare che il calcolo *SAD* è terminato.

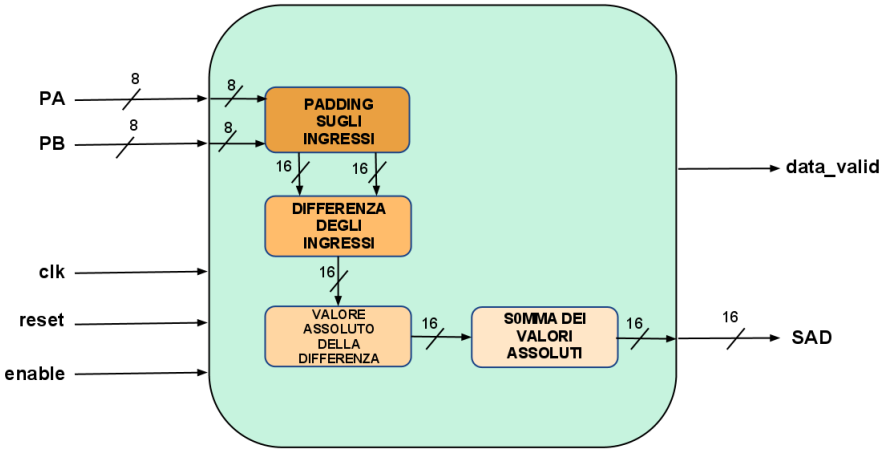

Figura 2: Architettura SAD con ingressi ed uscite specifici

3.3 Caso generale

Nel caso generale invece, ogni immagine è composta da un numero variabile di pixel, precisamente da NxN pixel. I pixel sono anche in questo caso interi

compresi tra 0 e 255, di conseguenza gli ingressi sono nuovamente a 8 bit (come prima $2^8 = 256$). Il numero di pixel che compone un'immagine va a variare però il numero di differenze da calcolare, e di conseguenza il numero massimo ottenibile in uscita. Per dimensionare l'uscita *SAD* quindi, si è calcolato il numero massimo ottenibile in funzione di N, secondo la relazione $2^n \geq (255 * N^2)$. Questa relazione sta ad indicare che l'uscita deve avere n bit, tali che il numero massimo (2^n) sia maggiore o uguale al numero massimo ottenuto dalla somma dei moduli delle differenze ovvero 255 (valore massimo del modulo della differenza) * N^2 (numero di pixel, quindi di differenze). Facendo una piccola manipolazione matematica della formula sopra descritta, si nota che $n \geq log_2(255 * N^2)$. E' sufficiente quindi fare preventivamente il precedente calcolo, e impostare i valori corretti nel codice relativo al caso specifico, per ottenere il calcolo della *SAD* nel caso generale di immagini composte da NxN pixel.

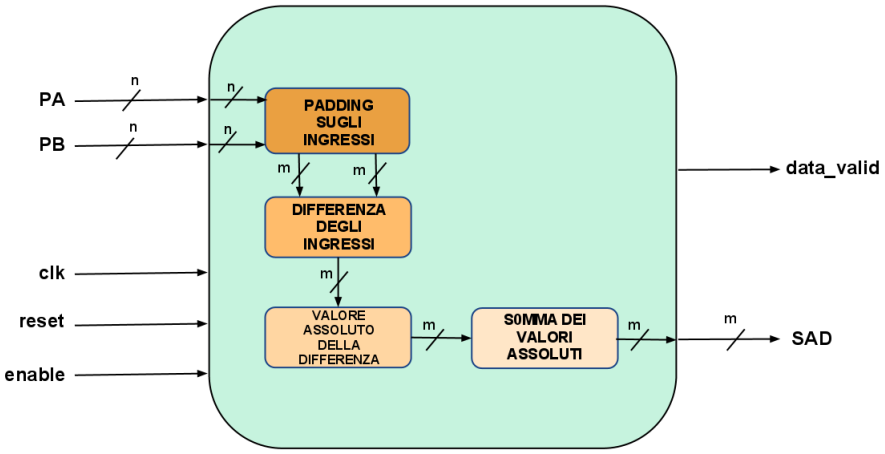

Figura 3: Architettura SAD con ingressi ed uscite generici

4 Analisi VHDL

4.1 Codice caso specifico

```
--------------------------------------------------------------------------------
-- (Behavioral)
--
-- File name : sad.vhd
-- Purpose : Calcolo SAD
--: somma delle differenze in valore assoluto pixel a pixel
--: tra due blocchi di immagini monocromatiche A e B
--
--
-- Library    : IEEE
-- Author(s)  : Carmine Benedetto, Silvio Bianchi
-- Copyrigth  : Creative Commons Attribution-ShareAlike 3.0 Unported License
--            : http://creativecommons.org/licenses/by-sa/3.0/
--
-- Simulator  : GHDL 0.29 (20100109) [Sokcho edition]
--      : GTKWave Analyzer v3.3.10 (w)1999-2010 BSI
--------------------------------------------------------------------------------
-- Revision List
-- Version Author Date Changes
--
-- 1.0 Carmine Benedetto, Silvio Bianchi 21/09/2011 New version
--------------------------------------------------------------------------------

library IEEE;
use IEEE.numeric_std.all;
use IEEE.std_logic_1164.all;

entity sad_c is

port(
PA : in std_logic_vector(7 downto 0); -- pixel immagine A
PB : in std_logic_vector(7 downto 0); -- pixel immagine B
clk : in std_logic; -- segnale di clock
reset : in std_logic; -- segnale di reset
enable : in std_logic; -- segnale di enable
SAD : out std_logic_vector(15 downto 0); -- uscita del circuito
data_valid : out std_logic -- segnale data_valid
);

end sad_c;

architecture behavioural of sad_c is

begin

sad_proc : process(clk, reset)
variable pap  : std_logic_vector(15 downto 0); -- variabile di appoggio per l'ingresso A
variable pbp  : std_logic_vector(15 downto 0); -- variabile di appoggio per l'ingresso B
variable cont : integer; -- contatore di operazioni
variable app1  : std_logic_vector(15 downto 0); -- variabile di appoggio
variable app2  : std_logic_vector(15 downto 0); -- variabile di appoggio
variable app3  : std_logic_vector(15 downto 0); -- variabile di appoggio

begin

-- padding sugli ingressi per renderli uniformi all'uscita
pap := "00000000" & PA;
pbp := "00000000" & PB;

--  se il segnale di reset e' attivo, azzera le variabili interne e le uscite
if (reset = '1') then

for i in 0 to 15 loop
app1(i) := '0';
app2(i) := '0';
app3(i) := '0';
SAD(i) <= '0';
```

```
end loop;

data_valid <= '0';
cont := 0;

-- se sono state effettuate 256 operazioni (16x16 pixel), il calcolo della SAD risulta concluso
-- l'uscita data_valid viene settata ad 1
elsif (cont = 256) then
data_valid <= '1';

-- per ogni fronte in salita del clock
elsif (clk'event and clk = '1') then

-- se il segnale di enable e' attivo, effettua la somma delle differenze in valore assoluto
-- pixel per pixel, altrimenti mantiene lo stato attuale delle variabili e delle uscite
if(enable = '1') then

app1 := std_logic_vector( unsigned (pap) - unsigned (pbp) );

-- se la differenza assume un valore negativo, si effettua un cambio si segno...
if (app1(8) = '1') then

app2 := std_logic_vector( -signed (app1) );
else

-- ... altrimenti si memorizza il valore della differenza
app2 := app1;

end if;

-- somma del valore assoluto del passo attuale con il valore assoluto dei passi precedenti
app3 := std_logic_vector( unsigned (app2) + unsigned (app3) );
SAD <= app3;
-- si incrementa il contatore di operazioni effettuate
cont := cont + 1;

end if;

end if;

end process sad_proc;

end behavioural;
```

File contenente il codice illustrato: *src/sad.vhd*
File di testbench connesso: *src/sad_test.vhd*

4.2 Codice caso generale

```
--------------------------------------------------------------------------------
-- (Behavioral)
--
-- File name : sad_n.vhd
-- Purpose : Calcolo SAD
--: somma delle differenze in valore assoluto pixel a pixel
--: tra due blocchi di immagini monocromatiche A e B
--
--
-- Library    : IEEE
-- Author(s)  : Carmine Benedetto, Silvio Bianchi
-- Copyrigth  : Creative Commons Attribution-ShareAlike 3.0 Unported License
--            : http://creativecommons.org/licenses/by-sa/3.0/
--
-- Simulator  : GHDL 0.29 (20100109) [Sokcho edition]
--      : GTKWave Analyzer v3.3.10 (w)1999-2010 BSI
--------------------------------------------------------------------------------
-- Revision List
-- Version Author Date Changes
--
-- 1.0 Carmine Benedetto, Silvio Bianchi 21/09/2011 New version
--------------------------------------------------------------------------------

library IEEE;
use IEEE.numeric_std.all;
use IEEE.std_logic_1164.all;

entity sad_c is

generic (
npix  : integer := 32 * 32; -- numero di pixel dell'immagine
n  : integer := 8; -- dimensione in bit dei pixel in ingresso
m  : integer := 18 -- dimensione in bit dell'uscita SAD
);
port(
PA : in std_logic_vector(n-1 downto 0); -- pixel immagine A
PB : in std_logic_vector(n-1 downto 0); -- pixel immagine B
clk : in std_logic; -- segnale di clock
reset : in std_logic; -- segnale di reset
enable : in std_logic; -- segnale di enable
SAD : out std_logic_vector(m-1 downto 0); -- uscita del circuito
data_valid : out std_logic -- segnale data_valid
);

end sad_c;

architecture behavioural of sad_c is

begin

sad_proc : process(clk, reset)
variable pap  : std_logic_vector(m-1 downto 0); -- variabile di appoggio per l'ingresso A
variable pbp  : std_logic_vector(m-1 downto 0); -- variabile di appoggio per l'ingresso B
variable cont : integer; -- contatore di operazioni
variable app1 : std_logic_vector(m-1 downto 0); -- variabile di appoggio
variable app2  : std_logic_vector(m-1 downto 0); -- variabile di appoggio
variable app3  : std_logic_vector(m-1 downto 0); -- variabile di appoggio
variable padding  : std_logic_vector(m-n-1 downto 0); -- variabile per il padding

begin

for i in 0 to m-n-1 loop
padding(i) := '0';
end loop;

pap := padding & PA;
pbp := padding & PB;

--  se il segnale di reset e' attivo, azzera le variabili interne e le uscite
if (reset = '1') then
```

```
for i in 0 to m-1 loop
app1(i) := '0';
app2(i) := '0';
app3(i) := '0';
SAD(i) <= '0';
end loop;

data_valid <= '0';
cont := 0;

-- se sono state effettuate 256 operazioni (16x16 pixel), il calcolo della SAD risulta concluso
-- l'uscita data_valid viene settata ad 1
elsif (cont = npix) then
data_valid <= '1';

-- per ogni fronte in salita del clock
elsif (clk'event and clk = '1') then

-- se il segnale di enable e' attivo, effettua la somma delle differenze in valore assoluto
-- pixel per pixel, altrimenti mantiene lo stato attuale delle variabili e delle uscite
if(enable = '1') then

app1 := std_logic_vector( unsigned (pap) - unsigned (pbp) );

-- se la differenza assume un valore negativo, si effettua un cambio si segno...
if (app1(n) = '1') then

app2 := std_logic_vector( -signed (app1) );
else

-- ... altrimenti si memorizza il valore della differenza
app2 := app1;

end if;

-- somma del valore assoluto del passo attuale con il valore assoluto dei passi
app3 := std_logic_vector( unsigned (app2) + unsigned (app3) );
SAD <= app3;
-- si incrementa il contatore di operazioni effettuate
cont := cont + 1;

end if;

end if;

end process sad_proc;

end behavioural;
```

In questo caso è stata considerata un'immagine 32 pixel x 32 pixel avente quindi due ingressi ad 8 bit ed un'uscita a 18 bit.
File contenente il codice illustrato: *src/sad_n.vhd*
File di testbench connesso: *src/sad_n_test.vhd*

Analizziamo adesso ogni singola componente del circuito.

4.3 Padding sugli ingressi

La prima operazione che viene eseguita è quella di uniformare la dimensione degli ingressi alla dimensione dell'uscita. Si aggiungono quindi in testa ad ogni ingresso (PA e PB) tanti 0 quanti sono i bit di differenza tra ingresso e uscita.

```
...

for i in 0 to m-n-1 loop
padding(i) := '0';
end loop;
pap := padding & PA;
pbp := padding & PB;

...
```

Dove m è il numero di bit dell'uscita, e n il numero di bit degli ingressi.

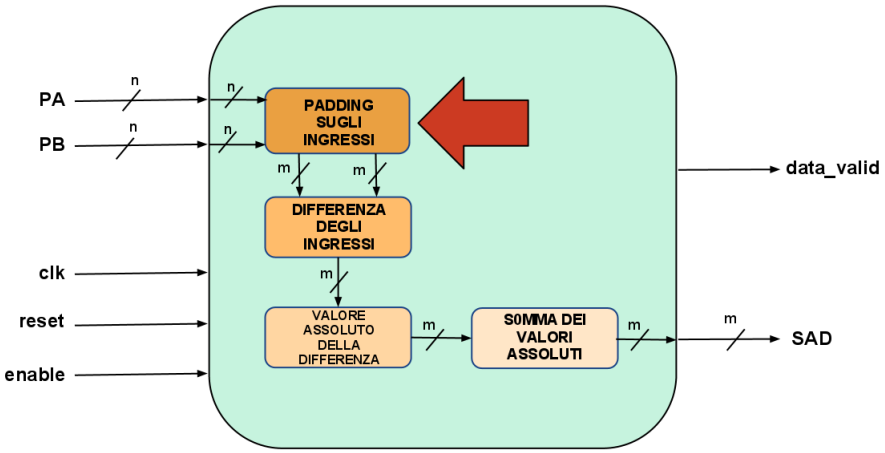

Figura 4: Modulo Padding

4.4 Differenza degli ingressi

In seguito i due ingressi vengono sottratti tra loro, con una semplice operazione di sottrazione, e il risultato messo in una variabile di appoggio *app1*.

```
...
app1 := std_logic_vector( unsigned (pap) - unsigned (pbp) );
...
```

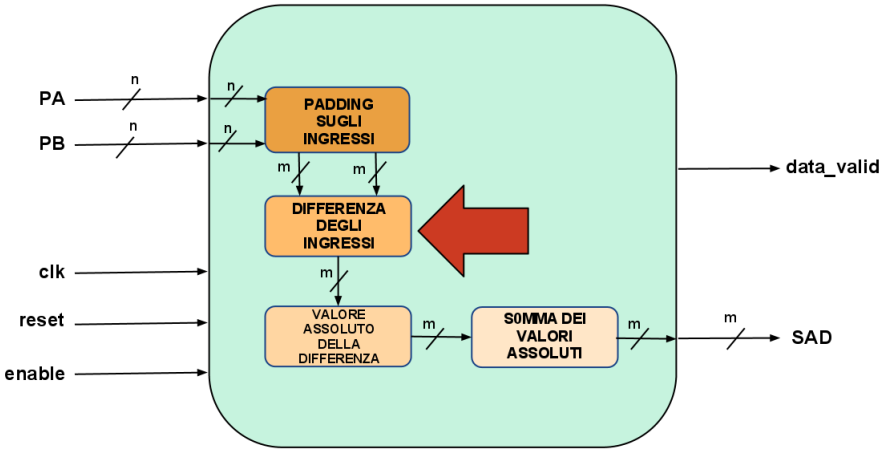

Figura 5: Modulo Differenza

4.5 Valore assoluto della differenza

La variabile di appoggio *app1* viene analizzata per capire se il risultato della differenza è positivo o negativo (se il *MSB* è settato a 1 il risultato è negativo, positivo altrimenti). Nel caso di risultato negativo è necessario effettuare il cambio di segno del valore ottenuto.

```
...

-- se la differenza assume un valore negativo, si effettua un cambio si segno...
if (app1(n) = '1') then
app2 := std_logic_vector( -signed (app1) );
else
-- ... altrimenti si memorizza il valore della differenza
app2 := app1;
end if;

...
```

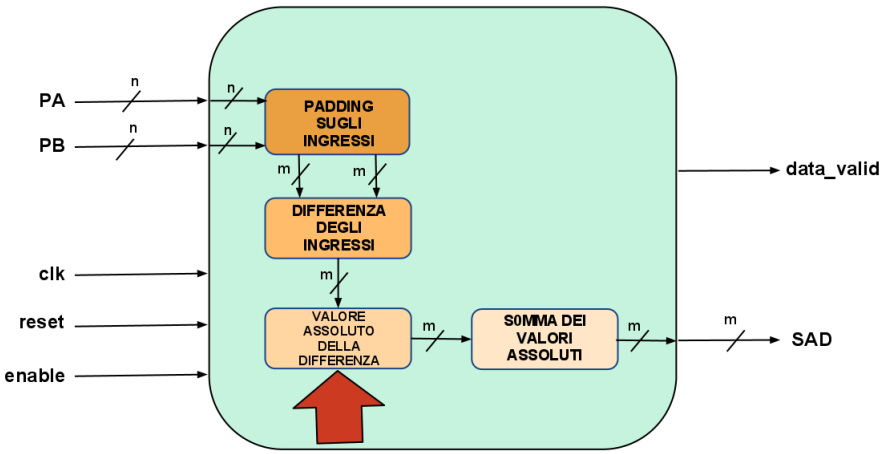

Figura 6: Modulo Valore Assoluto

4.6 Somma dei valori assoluti

L'ultimo elemento del circuito è quello che somma ad ogni passo il modulo della differenza ottenuta in quel passo con la somma dei moduli delle differenze di tutti passi precedenti, mettendo poi tale valore nell'uscita *SAD*.

```
...

-- somma del valore assoluto del passo attuale con il valore assoluto dei passi precedenti
app3 := std_logic_vector( unsigned (app2) + unsigned (app3) );
SAD <= app3;

...
```

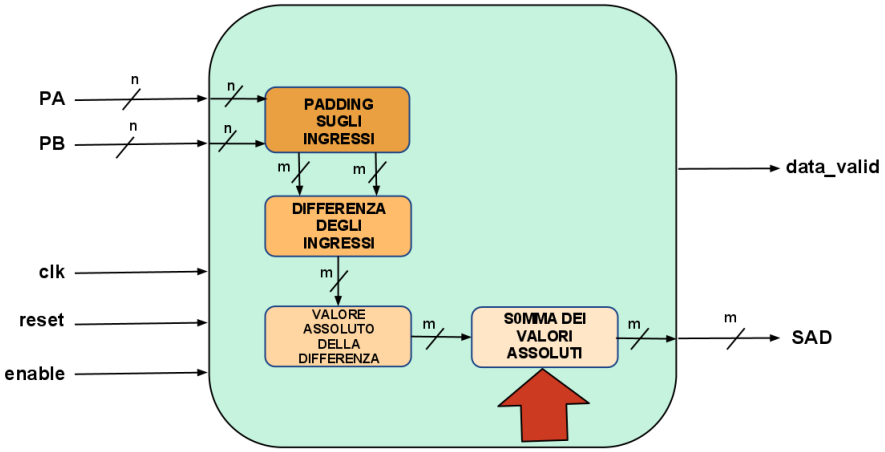

Figura 7: Modulo Somma

5 Testbench

Per controllare il corretto funzionamento del circuito sono stati realizzati due *testbench* (uno per il caso specifico e uno per il caso generale) settando gli ingressi come segue: $PA = 5$ e $PB = 10$. Come si vede dai grafici riportati in basso (relativi al caso specifico) tutti i segnali funzionano in maniera corretta e il calcolo della SAD a conclusione delle operazioni corrisponde a quello atteso.

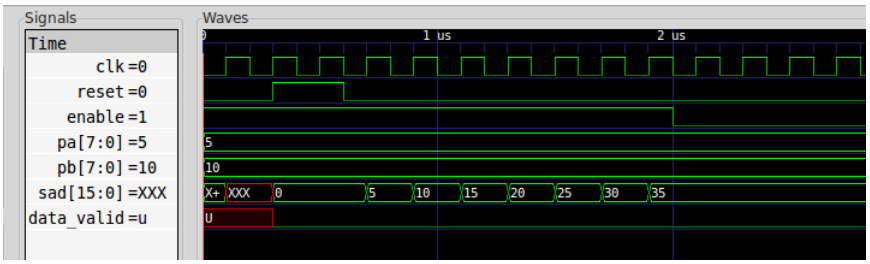

Figura 8: Testbench - Parte iniziale

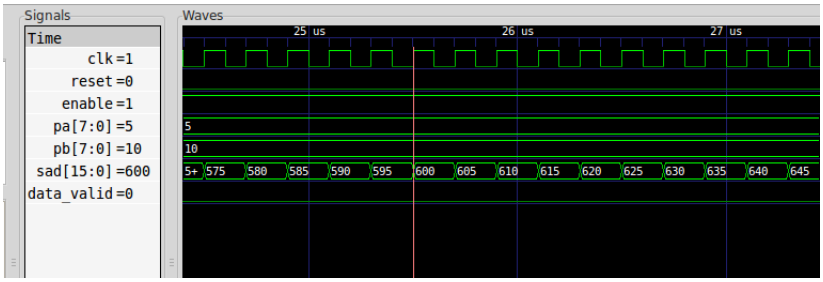

Figura 9: Testbench - Parte centrale

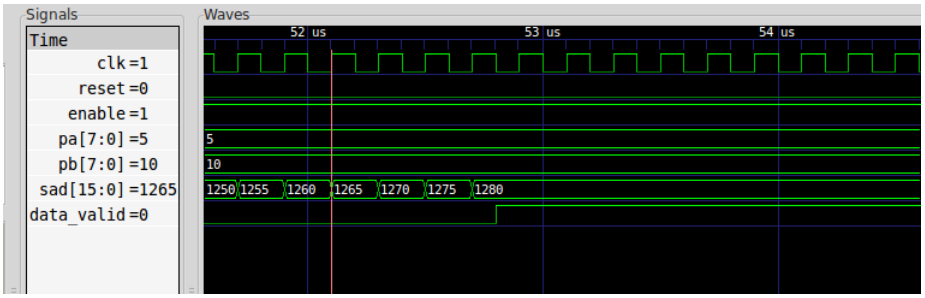

Figura 10: Testbench - Parte finale

6 Istruzioni di compilazione ed esecuzione

Il progetto è stato realizzato in ambiente GNU/Linux utilizzando i tool GHDL (per la compilazione e la creazione degli eseguibili) e GTKWave (per la visualizzazione delle temporizzazioni).
Per compilare l'applicazione posizionarsi nella directory */src* e digitare da shell:

- *make sad* - nel caso si voglia compilare il codice relativo al caso specifico;
- *make test* - nel caso si voglia compilare il codice relativo al test del caso specifico;
- *make sad_n* - nel caso si voglia compilare il codice relativo al caso generale;
- *make test_n* - nel caso si voglia compilare il codice relativo al test del caso generale;
- *make* - nel caso si vogliano compilare tutte le parti di codice sopra elencate.

Per visualizzare il testbench posizionarsi nella directory */testbench* e digitare da shell:

- *gtkwave sad.vcd* - nel caso si voglia visualizzare la temporizzazione relativa al caso specifico;
- *gtkwave sad_n.vcd* - nel caso si voglia visualizzare la temporizzazione relativa al caso generale.