ARCHITETTURA DEI SISTEMI DI ELABORAZIONE

Progetto di una ALU a 16 bit

Alessandro Arrichiello – M63/43
Flavio Battimo – M63/38

Sommario

Progetto di una ALU a 16 bit 1
- Introduzione 2
- Specifiche 3
- Implementazione 3
  - Button Manager 7
  - Terminale 8
  - Uart 9
  - Lcd Manager 12
  - Sommatore/Sottrattore 13
  - Moltiplicatore 14
  - Divisore 16
  - Elevatore a potenza 17
  - Statistiche 18
- Funzionamento 19
  - Funzionamento tramite terminale 19
  - Funzionamento tramite pulsanti 19
- Test 20

Introduzione

Questo progetto consiste nell'implementazione di una ALU a 16 bit capace di eseguire le operazioni logiche e aritmetiche più comuni. L'unità è stata inserita all'interno di un circuito integrato digitale FPGA (Field-programmable gate array). Oltre alla ALU sono state aggiunte ulteriori unità per consentire l'inserimento degli operandi e dell'operazione e la lettura del risultato.

Per lo sviluppo del progetto sono stati usati i seguenti strumenti:

- Ambiente di programmazione e simulazione: **Xilinxs ISE (versione 12.3)**
- Scheda di sviluppo: **Digilent BASYS (100K gate)**

L'inserimento dei dati può avvenire tramite:

- Terminale seriale UART (nel nostro caso è stato usato un ricetrasmettitore Bluetooth)
- Levette e pulsanti presenti sulla scheda di sviluppo

Il progetto è stato realizzato in moduli secondo l'ordine descritto di seguito

1. Sommatore
2. Sottrattore
3. Moltiplicatore
4. ALU
5. Gestore display 7 segmenti
6. Divisore
7. Elevatore a potenza
8. UART
9. Terminale

Specifiche

- ALU a 16 bit
- Inserimento operandi e operazione dalla scheda
- Lettura del risultato attraverso i display a 7 segmenti
- Operazioni
    - Addizione (Signed , Unsigned)
    - Sottrazione (Signed, Unsigned)
    - Moltiplicazione (Unsigned)
    - Divisione (Unsigned)
    - Elevazione a potenza (Unsigned)
    - NOT
    - AND
    - OR
    - XOR

A queste specifiche di base è stato aggiunto:
- Terminale UART
- operazione aritmetica X+Y+1

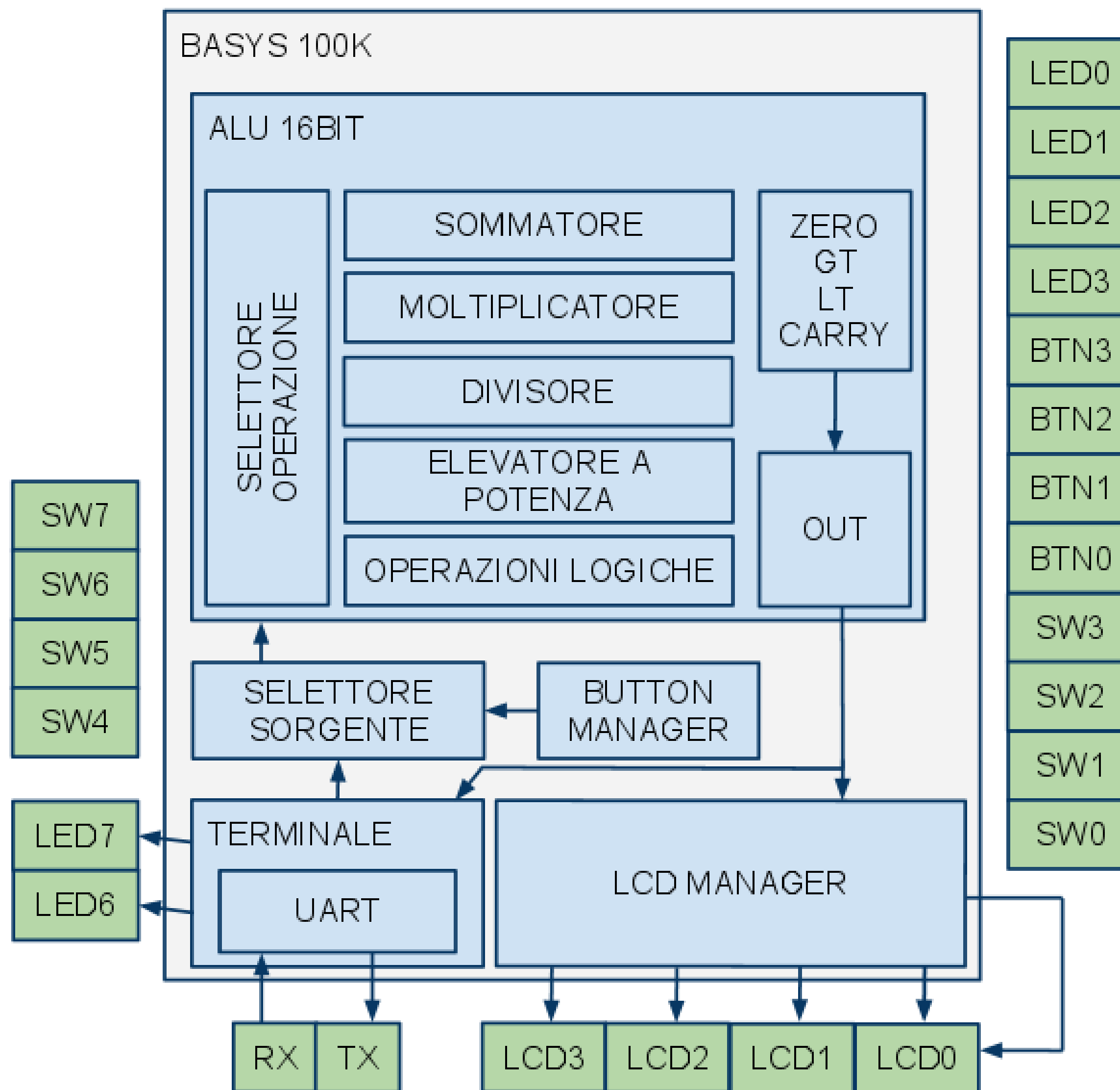
Implementazione

L'implementazione di tutti i componenti è stato realizzato in VHDL. Durante la fase di unione i componenti così realizzati occupavano più spazio di quello disponibile sulla scheda di sviluppo a nostra disposizione. L'unità ALU è combinatoria per tutte le operazioni eccetto l'elevazione a potenza, ciò ha richiesto una elevata quantità di spazio. Per evitare di eliminare funzionalità richieste dalle specifiche, sono state eseguite le seguenti modifiche o limitazioni:
- Sostituzione del moltiplicatore MAC con uno dei quattro moltiplicatori disponibili sulla scheda. L'unità è stata generata dallo strumento IPCORE.
- Implementazione dell'elevazione a potenza sfruttando 3 moltiplicatori 18X18 unsigned disponibili sulla scheda (che quindi non contribuiscono all'occupazione di spazio aggiuntivo).
- L'operando esponente dell'operazione elevazione a potenza è limitato a 5 bit.
- L'operando base dell'operazione elevazione a potenza è limitato a 8 bit.
- Il terminale non gestisce gli errori in ricezione (overrun error e framing error)

Alcuni moduli utilizzano il clock per gestire dei timer e le sequenze. Nel caso del button manager e del modulo LCD MANAGER non è necessario che il clock sia impostato ad una frequenza prestabilita. Per la UART invece la precisione del clock è fondamentale. L'implementazione è stata realizzata impostando una frequenza di clock pari a **50MHZ**. Nel caso in cui si voglia variare la frequenza di clock è necessario ricalcolare la variabile SPBRG nel modulo TERMINALE che gestisce il divisore del clock per la UART. Se la frequenza di clock è impostata a 25MHZ potrebbero essere visualizzati alcuni lampeggiamenti nella gestione dei display. In questo caso

basta modificare il valore massimo della variabile CONTATORE nel modulo LCD MANAGER. Di seguito sono riportati i diversi moduli implementati:

- **SOMMATORE**: Esegue la somma o la sottrazione di due numeri a 16 bit (signed o unsigned). Il sommatore è implementato in logica ripple carry.
- **MOLTIPLICATORE**: Esegue la moltiplicazione senza segno di due operandi a 16 bit e ritorna il risultato della moltiplicazione a 32 bit. Inizialmente il moltiplicatore è stato realizzato attraverso l'uso di celle MAC in logica di somma per righe. Ma è stato successivamente sostituito dal moltiplicatore generato attraverso il tool IPCORE generator per motivi di spazio.
- **DIVISORE**: Esegue l'operazione di divisione degli operandi dividendo (16 bit) e divisore (16 bit) ritornando il quoziente (16 bit) e il resto (16 bit). Il divisore è implementato in VHDL senza fare uso di clock esterni.
- **ELEVATORE A POTENZA**: Calcola l'elevazione a potenza di un operando a 8bit (base) elevato ad un operando a 5 bit (esponente). Il risultato è a precisione massima su 32 bit. Inizialmente l'elevazione a potenza è stata realizzata in vhdl senza l'utilizzo di clock esterni. Ciò però generava una unità troppo grande in termini di spazio. Si è scelto quindi di usare una macchina sequenziale gestita da un clock esterno. La scrittura del codice dell'algoritmo dell'elevazione a potenza, attraverso la scrittura di codice sequenziale porta il compilatore a sfruttare 3 moltiplicatori 18X18 disponibili sulla scheda, diminuendo così lo spazio utilizzato. Tuttavia ciò non è stato sufficiente per inserire tutti e 16 i bit della base, che sono stati quindi limitati ad 8. In particolare usa i primi 8 bit.
- **ALU**: La ALU si occupa di attivare l'operazione richiesta a seconda dell'input operazione ricevuto dal bus in ingresso. La ALU è capace di eseguire operazioni aritmetiche e logiche.
- **BUTTON MANAGER**: Il button manager si occupa di gestire la pressione dei pulsanti disponibili sulla scheda da parte dell'utente. In particolare gestisce il fenomeno di BOUNCE del segnale e la pressione singola o prolungata del pulsante stesso. Questa unità utilizza il clock della scheda.
- **LCD MANAGER**: Questo componente gestisce i 4 display lcd a 7 segmenti presenti sulla scheda. In particolare traduce il segnale ricevuto in ingresso per ogni singolo display. Questa unità inoltre gestisce la visualizzazione e l'abilitazione dei singoli led del display multiplexato. Questa unità utilizza il clock della scheda.
- **UART**: Si occupa della ricezione e la trasmissione di singoli byte secondo il protocollo standard RS232. La velocità di trasferimento è selezionabile dall'utente mentre la codifica è fissa e non può essere modifica: 1 bit di stop, nessuna parità, 8 bit di dati (8N1)
- **TERMINALE**: Il terminale riceve i dati dalla UART (operazione, operando 1, operando 2). Alla pressione del tasto invio esegue l'operazione specificata e invia il risultato dell'operazione attraverso la porta seriale. Per motivi implementativi di spazio, il terminale non gestisce i codici di errore (frame error e overrun error) della UART.



Schema generale della scheda

LED7: spia funzionamento RX
LED6: spia funzionamento TX
RX: pin di ricezione dei dati dalla periferica seriale
TX: pin di invio dei dati alla periferica seriale
(SW7:SW4): selezione dell'operazione. Nel caso in cui sia impostato a "1111" l'operazione è selezionata in base al valore ricevuto tramite terminale. Negli altri casi fare riferimento all'allegato con le operazioni possibili.
(LCD3:LCD0): display led 7 segmenti
LED0: bit di test per ZERO

LED1: bit di test per GT
LED2: bit di test per LT
LED3: bit di test per CARRYOUT
(BTN3:BTN0): pulsante per l'incremento del valore della i-esima cifra visualizzata sul display
SW3: se alto imposta il valore visualizzato sul display nella variabile operando A
SW2: se alto imposta il valore visualizzato sul display nella variabile operando B
SW1: se alto visualizza sul display il risultato di OUT_HIGH in uscita dalla ALU
SW0: se alto visualizza sul display il risultato di OUT_LOW in uscita dalla ALU

Button Manager

Il button manager gestisce i quattro pulsanti della scheda. In particolare questa unità è composta da quattro moduli, ognuna delle quali gestisce un pulsante in modo indipendente. Il button manager non ha una propria interfaccia ed è mappato direttamente nel modulo radice (Scheda.vhd). Ogni singolo modulo, chiamato DebounceButton, gestisce la pressione breve e la pressione prolungata, filtrando gli errori dovuti al rimbalzo del segnale durante il passaggio dallo stato di pulsante premuto a quello di pulsante rilasciato e viceversa.

```
entity DebounceButton is
    Port ( btn : in  STD_LOGIC;  --btn: pin del pulsante
           rst : in  STD_LOGIC;  --rst: reset della cifra
           value  :  out   STD_LOGIC_VECTOR  (3  downto  0);    --value:  cifra
esadecimale
           CLK : in  STD_LOGIC   --CLK: clock della scheda
               );
end DebounceButton;

architecture DebounceButtonARCH of DebounceButton is
   constant max_value_contatore : integer := 15000000; --circa 300ms a 50mhz
      shared variable contatore : integer range 0 to max_value_contatore;
      shared variable mvalue : integer range 0 to 15;
      signal long_clicked : std_logic := '0';
begin
    value <= conv_std_logic_vector(mvalue,4);
    processo_debounce : process (CLK,rst)
       begin
         if rst='1' then
             --reset della cifra
               contatore := 0;
               mvalue := 0;
            elsif CLK'event and CLK='1' then
                  if btn='0' and contatore > 983040 and long_clicked='0' then
                      --se il pulsante è stato rilasciato
                        --ma è stato premuto per un tempo non troppo breve
                        --circa 20ms a 50mhz e non è scattata la pressione
                        --prolungata allora incrementa la cifra di 1
                        mvalue := mvalue + 1;
                        contatore := 0;
                  elsif contatore = max_value_contatore then
                      --pressione prolungata
                        mvalue := mvalue + 1;
                        contatore := 0;
                        long_clicked <= '1';
                  elsif btn='1' then
                     contatore := contatore + 1;
                  elsif btn='0' then
                     contatore := 0;
                     long_clicked <= '0';
                  end if;
            end if;
       end process;

end DebounceButtonARCH;
```


Alessandro Arrichiello – M63/43
Flavio Battimo – M63/38

Terminale

Il terminale riceve il comando e gli operandi della uart, attiva l'ALU per calcolare il risultato e ritrasmette il risultato e il registro di stato al mittente. Il terminale inoltre ritrasmette anche il carattere ricevuto, consentendo all'utente che usa il terminale di visualizzare il dato trasmesso.

```
entity Terminale is
    Port (
      --valori in uscita dalla ALU
      alu_status : in  STD_LOGIC_VECTOR (7 downto 0);
      risultato_high : in  STD_LOGIC_VECTOR (15 downto 0);
      risultato_low : in  STD_LOGIC_VECTOR (15 downto 0);

      CLK : in  STD_LOGIC;
      rst : in std_logic;

      --valori in entrata alla ALU
      operando_A : out  STD_LOGIC_VECTOR (15 downto 0);
      operando_B : out  STD_LOGIC_VECTOR (15 downto 0);
      operazione : out  STD_LOGIC_VECTOR (3 downto 0);

      --canale di ricezione e trasmissione seriale
      RX : in std_logic;
      TX : out std_logic
      );
end Terminale;
```

I valori operando_A, operand_B e operazione sono in configurazione latch, e vengono modificati solo alla ricezione dal terminale del carattere LF o del carattere CR. I bit RX e TX sono usati solo dal modulo interno UART e vanno collegati direttamente ai pin di I/O della scheda di sviluppo. Il clock deve avere necessariamente una frequenza di clock pari a 50mhz. Nel caso in cui si debba modificare la frequenza di clock, si deve impostare il valore di ingresso SPBRG del modulo UART secondo la seguente equazione: SPBRG = CLOCK_HZ / BAUD. Il terminale non gestisce gli errori di overrun e di framing, in questi casi il risultato potrebbe essere differente da quello effettivo. Per convertire i caratteri 0-F in vettori di 4 bit, sono stati implementati due moduli:

```
entity ConvertitoreCharHex is
    Port ( carattere : in  STD_LOGIC_VECTOR (7 downto 0);
           valore : out  STD_LOGIC_VECTOR (3 downto 0);
                    valido : out std_logic
                    );
end ConvertitoreCharHex;

entity ConvertitoreHexChar is
    Port ( valore : in  STD_LOGIC_VECTOR (3 downto 0);
           carattere : out  STD_LOGIC_VECTOR (7 downto 0));
end ConvertitoreHexChar;
```

Il terminale è implementato nel file Terminale.vhd.

Alessandro Arrichiello – M63/43
Flavio Battimo – M63/38

Uart

Il modulo UART si occupa della ricezione e dell'invio di stringhe in seriale rispettivamente attraverso un canale RX e un canale TX. Il modulo è diviso in due parti, una dedicata alla ricezione (rxUART) e una all'invio (txUART).

```
entity Uart is
    Port (
      -- Abilitazione trasmettitore
       TXEN : in std_logic;
      -- Abilitazione ricevitore
       RXEN : in std_logic;
      -- Stato Shift Register di Trasmissione
      -- 1 = TSR vuoto
      -- 0 = TSR pieno
       TRMT : out std_logic;

      -- Errore Frame
       FERR : out std_logic;
      -- Errore Overrun
       OERR : out std_logic;
       -- non implementato
       RX9D : out std_logic;

       -- array di bit necessario per la modifica del Baud Rate
       SPBRG : in std_logic_vector(15 downto 0);
       -- clock
       CLK : in std_logic;

      -- Buffer Ricevitore
       RCREG : out std_logic_vector(7 downto 0);
      -- Bit di Lettura effettuata
       RCREGR : in std_logic;

       -- Flag di Interruzione Buffer Ricezione Pieno
       RCIF : out std_logic;
       -- Flag di Interruzione Buffer Trasmissione Vuoto
       TXIF : out std_logic;

       -- uscita invertita
       INVERTED : in std_logic;

       -- Buffer Trasmettitore
       TXREG : in std_logic_vector(7 downto 0);
       -- Bit di Scrittura effettuata
       TXREGW : in std_logic;
       -- Canale di trasmissione
       TX : out std_logic;
      -- Canale di ricezione
       RX : in std_logic

      );
end Uart;
```

L'implementazione consiste in un semplice mapping di porte con i sottomoduli. I sottomoduli condividono solo i seguenti segnali: SPBRG, CLK e INVERTED. Il segnale INVERTED se posto a 1 inverte i livelli logici dell'uscita ed interpreta quelli in ingresso all'inverso dello standard (bit di start basso, bit di stop alto). Questo modulo invia e riceve i dati seguendo il formato 8N1, cioè 8 bit di dati, nessun bit di parità e 1 bit di stop ed è asincrono. La velocità invece è variabile e può essere configurata opportunamente attraverso la variabile SPBRG. Questo modulo segue le specifiche e i nomi dei segnali del modulo USART presente nei chip PIC serie 16 della Microchip.

In invio è presente un buffer di 8 bit, la uart è capace di memorizzare un dato in invio mentre l'invio del valore precedente è ancora in corso. Una scrittura del buffer di invio, se pieno, sostituisce il valore memorizzato. L'invio di un nuovo valore avviene innanzitutto attivando la seriale in uscita alzando il bit `TXEN`. La scrittura di un valore nel buffer di invio avviene tramite l'assegnazione del segnale `TXREG` e la commutazione a livello alto e poi basso del segnale `TXREGW`. Il bit `TXIF` segnala che il buffer di invio è vuoto, ed è quindi possibile inviare un nuovo valore. Il bit `TRMT` indica, quando alto, che il valore in uscita è stato inviato. Il trasmettitore è stato implementato usando un contatore per generare il clock di baud alla velocità specificata tramite il valore `SPBRG` e uno shift register per l'invio del bit di start, vettore di bit dei dati e bit di stop.

```vhdl
entity txUART is
    Port (
      TXEN : in std_logic;

      SPBRG : in std_logic_vector(15 downto 0);
      CLK : in std_logic;

      INVERTED : in std_logic;

      TXREG : in std_logic_vector(7 downto 0);
      TXREGW : in std_logic;

      TRMT : out std_logic;
      TXIF : out std_logic;
      TX : out std_logic

                    );
end txUART;
```

Per l'implementazione di txUart confronta txUART.vhd.

Anche nel caso della ricezione è presente un buffer da cui può essere letto il valore durante la ricezione di una nuova stringa di dati. Nel caso in cui il valore presente nel buffer non venga letto prima della ricezione del nuovo valore il bit di frame error (FERR) viene posto a 1. Nel caso in cui, inoltre, il bit di stop non sia valido il bit di overrun error (OERR) viene posto a 1. I bit di errore vengono resettati al momento della lettura del valore presente nel buffer (RCREG) alzando il bit RCREGR.
Il ricevitore è leggermente più complesso del trasmettitore in quanto deve sincronizzarsi con il bit di start e poi posizionarsi al centro di quest'ultimo per iniziare la campionatura dei bit. Per effettuare ciò utilizza un contatore per generare il clock di baud (o metà) e uno shift register per la memorizzazione della stringa di bit ricevuta. Alla fine della campionatura inoltre verifica se il bit di stop è corretto e se è presente un valore non letto nel buffer.

```
entity rxUART is
    Port (
      RXEN : in std_logic;

      SPBRG : in std_logic_vector(15 downto 0);
      CLK : in std_logic;

      INVERTED : in std_logic;

      FERR : out std_logic;
      OERR : out std_logic;

      RCREG : out std_logic_vector(7 downto 0);
      RCREGR : in std_logic;

      RCIF : out std_logic;
      RX : in std_logic
                    );
end rxUART;
```

Per l'implementazione di rxUart confronta rxUART.vhd.

Lcd Manager

Questo modulo si occupa di gestire la visualizzazione di 4 cifre esadecimale sul display presente sulla scheda di sviluppo. Il display è composto da quattro display a 7 segmenti multiplexati tra loro, può quindi essere visualizzato solo un valore diverso per ogni display alla volta. Una scansione abbastanza veloce di accensione e spegnimenti di ogni singolo display crea un effetto che consente ad un occhio umano di visualizzare il display come se fossero accese contemporaneamente tutte e quattro le cifre. Una velocità di scansione troppo bassa creerebbe degli sfarfallii fastidiosi, una velocità di scansione troppo alta porta ad un abbassamento della luminosità dei singoli display e ad un accavallamento delle cifre contigue visualizzate. Il modulo utilizza un contatore per generare un clock a frequenza più bassa di quello della scheda di sviluppo e attiva al momento opportuno ogni display, mostrando la cifra memorizzata da visualizzare per quel display. Lcd manager utilizza 4 sottomoduli chiamati BcdToLcd per convertire un valore bcd nel corrispettivo vettore di accensione del display a led.

```
entity LcdManager is
    Port (
           --cifra esadecimale da visualizzare per ogni display
           bcd3 : in  STD_LOGIC_VECTOR (3 downto 0);
           bcd2 : in  STD_LOGIC_VECTOR (3 downto 0);
           bcd1 : in  STD_LOGIC_VECTOR (3 downto 0);
           bcd0 : in  STD_LOGIC_VECTOR (3 downto 0);
           --vettore di abilitazione di ogni display
           --en=1 (acceso), en=0 (spento)
           en : in  STD_LOGIC_VECTOR (3 downto 0);
           --vettore dei punti DP
           points : in  STD_LOGIC_VECTOR (3 downto 0);
           --connessione ai catodi dei display (in comune)
           lcdrow : out  STD_LOGIC_VECTOR (7 downto 0);
           --connessione agli anodi dei singoli display
           lcdcol : out  STD_LOGIC_VECTOR (3 downto 0);
           --clock della scheda
           CLOCK : in STD_LOGIC
                );
end LcdManager;
```

Sommatore/Sottrattore

Il sommatore implementato consente di eseguire sia somme sia sottrazioni. Gli operandi sono su 16 bit sia signed sia unsigned. Il risultato è su 16 bit più un bit di carry out. L'operazione di sottrazione può avvenire a patto di settare il bit di carry in a 1 e negare il secondo operando in ingresso. Non è stato implementato un sommatore con bit di ingresso per selezionare l'operazione di addizione o sottrazione poiché l'ALU implementata oltre a supportare già l'operazione di sottrazione, supporta anche l'operazione A+B+1 che necessita di poter inserire in ingresso al sommatore il carry in. Il sommatore è implementato in logica ripple carry ed è completamente combinatorio. Questo modulo è stato implementato mediante la generazione automatica di più full adder a 1 bit.

```
entity Sommatore is
       generic(L: integer := N);
    Port ( in1 : in  STD_LOGIC_VECTOR (L-1 downto 0);
           in2 : in  STD_LOGIC_VECTOR (L-1 downto 0);
           carryin : in  STD_LOGIC;
           result : out  STD_LOGIC_VECTOR (L-1 downto 0);
           carryout : out  STD_LOGIC);
end Sommatore;

architecture SommatoreARCH of Sommatore is

component FullAdder Port (
           a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           cin : in  STD_LOGIC;
           cout : out  STD_LOGIC;
           s : out  STD_LOGIC);
      end component;

SIGNAL im : std_logic_vector(0 TO L-2);

begin

ciclo: FOR i IN 0 TO L-1 GENERATE

      c_f: IF (i = 0) GENERATE
      primo: FullAdder PORT MAP (in1(0),in2(0),carryin,im(0),result(0));
      END GENERATE;

      c_i: IF (i>0 AND i<L-1) GENERATE
      intermedi:  FullAdder PORT MAP (in1(i),in2(i),im(i-1),im(i),result(i));
      END GENERATE;

      c_l: IF (i = L-1) GENERATE
      ultimo:FullAdder PORT MAP (in1(i),in2(i),im(i-1),carryout,result(i));
      END GENERATE;

END GENERATE;


end SommatoreARCH;
```


Alessandro Arrichiello – M63/43
Flavio Battimo – M63/38

Moltiplicatore

Di seguito viene riportata l'implementazione del moltiplicatore a celle MAC inizialmente utilizzato e poi sostituito dal moltiplicatore interno presente sulla scheda di sviluppo. I due moltiplicatori hanno la medesima interfaccia e sono completamente intercambiabili.

Il moltiplicatore consente di eseguire la moltiplicazione di due operandi unsigned a 16 bit. Il risultato è il prodotto dei due operandi, su 32 bit. Questo modulo è stato implementato genericamente per funzionare per qualsiasi valore di N. Nel caso dell'implementazione della ALU a 16 bit è stato posto N=16.

```
entity MoltiplicatoreMacNbit is
    Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
           b : in  STD_LOGIC_VECTOR (N-1 downto 0);
           p : out  STD_LOGIC_VECTOR (2*N-1 downto 0));
end MoltiplicatoreMacNbit;
```

Dall'interfaccia si vede che a e b sono i due operandi e p è il prodotto. Il modulo non utilizza nessun clock, è puramente combinatorio. Sfrutta inoltre l'interconnessione di più celle MAC (Multiply-Accumulator) per generare il prodotto. Il MAC a sua volta utilizza un full adder.

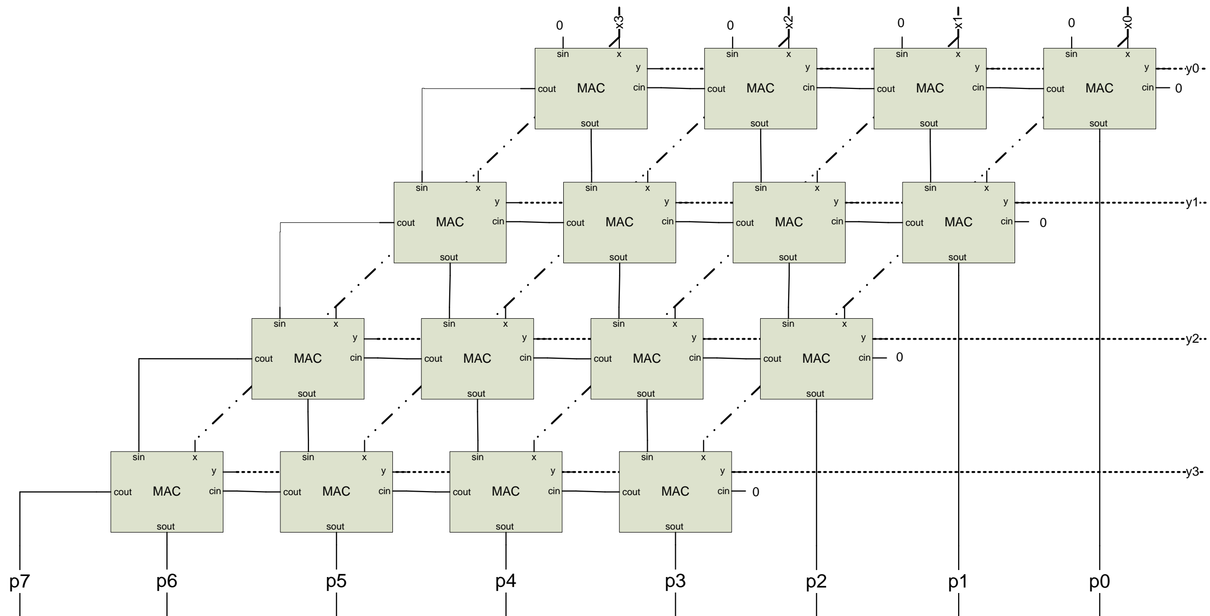
```
component Mac Port (
      x : in  STD_LOGIC;
      y : in  STD_LOGIC;
      cin : in  STD_LOGIC;
      sin : in  STD_LOGIC;
      cout : out  STD_LOGIC;
      sout : out  STD_LOGIC);
end component;

architecture MacARCH of Mac is
      component FullAdder Port (
           a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           cin : in  STD_LOGIC;
           cout : out  STD_LOGIC;
           s : out  STD_LOGIC);
      end component;

      signal xy : std_logic;
begin
   adder : FullAdder port map ( sin , xy , cin , cout , sout);
   xy <= x and y;
end MacARCH;
```

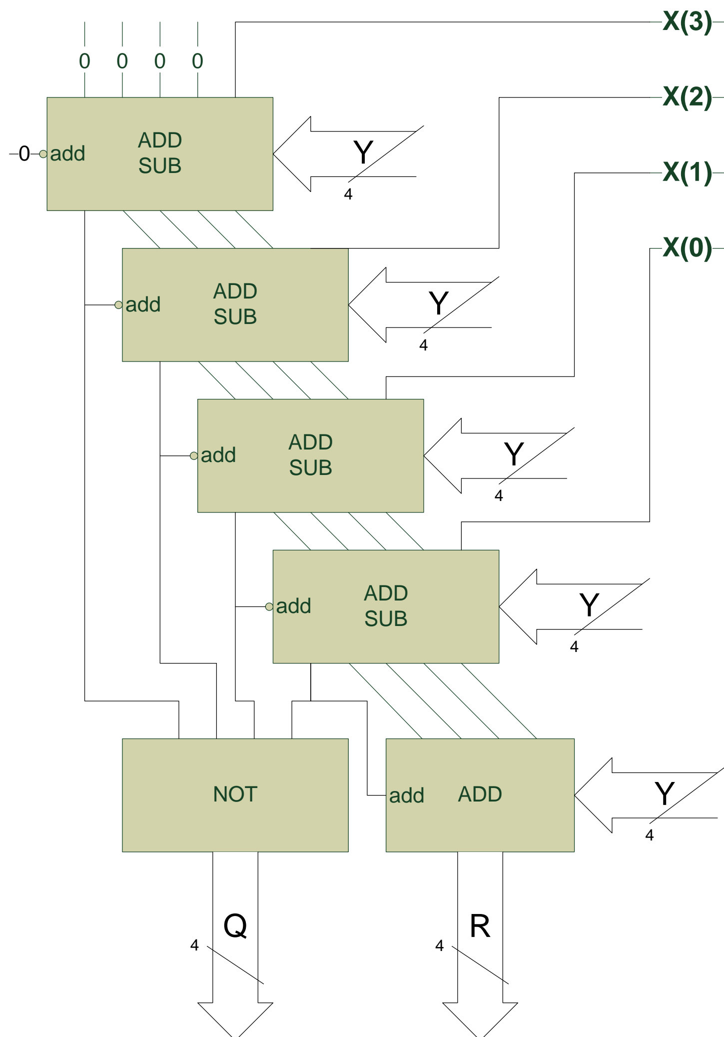

Alessandro Arrichiello – M63/43
Flavio Battimo – M63/38

La cella MAC genera sia il prodotto parziale sia la somma. La combinazione di più celle MAC consente di creare un moltiplicatore a N bit. Di seguito è riportato lo schema di un moltiplicatore a 4 bit. Le righe sono tutte simili tranne la prima in alto.



Divisore

Questo modulo esegue la divisione di due numeri unsigned a 16 bit (dividendo e divisore) e ritorna in uscita il quoziente e il resto, entrambi a 16 bit. L'operazione di divisione è implementata sulla base dell'algoritmo di divisione non restoring, cioè ad ogni ciclo viene eseguita una somma o una sottrazione e viene effettuata una sola correzione finale, se necessario. Di seguito è riportato uno schema a 4 bit, del divisore implementato. Il modulo è puramente combinatorio ed è stato implementato attraverso un modello architetturale di combinazione di N ADD/SUB a N+1 bit e un addizionatore di correzione finale a N bit. Il primo ADD/SUB in alto esegue sempre una sottrazione, mentre quelli successivi eseguono una sottrazione solo se il bit più significativo dell'ADD/SUB precedente è 0, negli altri casi eseguono una somma. L'addizionatore finale del resto esegue una somma tra gli N bit meno significativi dell'uscita dell'ultimo ADD/SUB e Y solo se il bit più significativo dell'ultimo ADD/SUB vale 1 (cioè se il resto è negativo), altrimenti ritorna semplicemente l'uscita degli N bit meno significativi dell'ultimo ADD/SUB. Il quoziente, pari ai bit più significativi delle uscite degli N ADD/SUB deve essere negata.



Alessandro Arrichiello – M63/43
Flavio Battimo – M63/38

Elevatore a potenza

Questo modulo effettua l'elevazione a potenza di m elevato ad e, con m ed e entrambi valori unsigned. L'algoritmo di elevazione consiste in una sequenza di moltiplicazioni effettuate solo se necessario e di un'altra sequenza di moltiplicazioni sempre effettuate. Di seguito viene riportato l'algoritmo utilizzato per l'elevazione a potenza:

```
1. z:=1
2. y:=m
3. t:=length(e)
4. for i=0 to t-1
5.    if e(i)=1 then z:=z * y
6.    y:=y * y
7. end for
8. return z
```

Questo algoritmo è stato poi migliorato per evitare calcoli inutili. Poiché la nostra ALU è a 16 bit e poiché il risultato massimo rappresentabile, per come è stata implementata, è a 32 bit (usata nel caso della moltiplicazione), si è voluto limitare anche il risultato della elevazione a 32 bit. A questo punto però avere un esponente di 16 bit produrrebbe in molti casi un valore superiore a quello massimo consentito, nonché una elevata perdita di tempo per eseguire un calcolo non rappresentabile. L'esponente è quindi stato limitato a 5 bit, consentendo una elevazione massima di m^31. Infatti prendendo m=2 risulta 2^31=2147483648 (HEX=80.00.00.00), che è ancora a 32 bit. Un esponente maggiore produrrebbe un risultato non rappresentabile, infatti 2^32=x1.00.00.00.00, > 32bit). Inoltre la moltiplicazione y:=y*y nell'ultimo ciclo è inutile in quanto non viene utilizzata. L'algoritmo è stato quindi modificato nel seguente:

```
1. z:=1
2. y:=m
3. for i=0 to 5-1
4.    if e(i)=1 then z:=z * y
5.    if i=5-1 then return z
6.    y:=y * y
7. end for
8. return z
```

L'algoritmo consiste nella moltiplicazione di y per se stesso ad ogni passo. Il valore z viene moltiplicato per y al passo i solo se il bit nella posizione i di e vale 1.
Prendiamo per esempio m=2 e e=21 (0b10101).

$$2^{21} = 2^{1^1} * 2^{2^0} * 2^{4^1} * 2^{8^0} * 2^{16^1} = 2 * 1 * 16 * 1 * 65536 = 2097152$$

Come altro esempio prendiamo m=3 e e=5 (0b00101).

$$3^5 = 3^{1^1} * 3^{2^0} * 3^{4^1} * 3^{8^0} * 3^{16^0} = 3 * 1 * 81 * 1 * 1 = 243$$

L'algoritmo è stato implementato come macchina a stati finiti, ma all'esterno appare come una macchina combinatoria (nel senso che non è presente nessun segnale di reset o di done

Alessandro Arrichiello – M63/43
Flavio Battimo – M63/38

all’esterno). Questo modulo esegue continuamente l’elevazione a potenza degli operandi in ingresso, ma nel momento in cui un operando cambia, il calcolo viene rieseguito. L’operazione impiega al massimo 11 cicli di clock da quando si immettono in ingresso gli operandi `([reset] , init , molt_z1, molt_y1, molt_z2 , molt_y2 , molt_z3 , molt_y3 , molt_z4 , molt_y4 , return)` mentre impiega 4 cicli di clock se l’esponente è nullo `([reset] , init , molt_z1 , return)`. Si potrebbe ulteriormente velocizzare il calcolo eliminando gli stati `return` e `init`, integrandoli negli altri stati, risparmiando quindi 2 cicli di clock.

```
entity ElevazionePotenza is
    generic ( NN : integer := 16 );
    Port ( m : in  STD_LOGIC_VECTOR (NN-1 downto 0);
           e : in  STD_LOGIC_VECTOR (NN-1 downto 0);
           pot : out  STD_LOGIC_VECTOR (2*NN-1 downto 0);
           CLK : in  STD_LOGIC
                 );
end ElevazionePotenza;
```

Per l’implementazione confronta ElevazionePotenza.vhd.

Statistiche

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Total Number Slice Registers	465	1,920	24%
Number used as Flip Flops	389		
Number used as Latches	76		
Number of 4 input LUTs	1,662	1,920	86%
Number of occupied Slices	958	960	99%
Number of Slices containing only related logic	958	958	100%
Number of Slices containing unrelated logic	0	958	0%
Total Number of 4 input LUTs	1,805	1,920	94%
Number used as logic	1,662		
Number used as a route-thru	143		
Number of bonded IOBs	33	108	30%
Number of BUFGMUXs	4	24	16%
Number of MULT18X18SIOs	4	4	100%
Average Fanout of Non-Clock Nets	3.19		

Funzionamento

Funzionamento tramite terminale

1. Collegare il pin RX della porta JA(1) al pin di uscita TX del trasmettitore seriale.
2. Collegare il pin TX della porta JA(2) al pin di ingresso RX del trasmettitore seriale.
3. Accendere la scheda.
4. Impostare gli switch SW7-SW4 a "1111" in modo da attivare i comandi tramite terminale.
5. (OPZIONALE) Impostare a "1" lo switch SW0 per visualizzare OUT_LOW oppure SW1 per visualizzare OUT_HIGH.
6. Attivare la comunicazione seriale inviando la stringa nel seguente formato "OAAAABBBB" seguito o dal carattere CR (hex=0D) odal carattere LF (hex=0A). i caratteri validi sono (0-9,a-f,A-F), ad un carattere non riconosciuto viene associato il codice 0. Con il carattere "O" si specifica l'operazione (vedere specifiche ALU allegato). Con la stringa "AAAA" si specifica l'operando A a 16 bit (4 caratteri ascii 0-F). Con la stringa "BBBB" si specifica l'operando B a 16 bit (4 caratteri ascii 0-F).
    a. Digitare l'operazione (1 carattere)
    b. Digitare l'operando A (4 caratteri)
    c. Digitare l'operando B (4 caratteri)
    d. Premere invio (carattere "0A" o "0D")
7. Leggere il risultato. Il risultato segue il formato "SHHHHLLLL" seguito da un carattere CR e un carattere LF. Il carattere S è espresso in esadecimale con S="0 0 0 0 C LT GT Z. HHHH=OUT_HIGH, LLLL=OUT_LOW.

Funzionamento tramite pulsanti

1. Accendere la scheda
2. Impostare gli switch SW7-SW4 a seconda dell'operazione da eseguire (vedi specifiche ALU)
3. Impostare gli switch SW3-SW0 a "0000".
4. Con i pulsanti impostare l'operando A desiderato
5. Portare SW3 a "1" e poi successivamente a "0" per memorizzare l'operando A
6. Con i pulsanti impostare l'operando B desiderato
7. Portare SW2 a "1" e poi successivamente a "0" per memorizzare l'operando B
8. Attivare SW1 per visualizzare OUT_HIGH o SW0 per visualizzare OUT_LOW

Test

I test sono stati eseguiti prima sul simulatore e poi sulla scheda. Per simulare il corretto funzionamento delle operazioni aritmetiche sono stati realizzati dei testbench che generassero errore nel caso in cui il risultato non fosse stato corretto. Negli altri casi sono stati simulati manualmente visualizzando la variazione degli stati logici.

- **Sommatore e Sottrattore**
  I test del sommatore e del sottrattore sono stati eseguiti una prima volta a 4 bit per valutarne il buon funzionamento. Una volta eseguito il test a 4 bit è stato ricompilato a 16 bit e valutato in modo automatico dalla procedura realizzata nel testbench.
- **Moltiplicatore**
  Test a 4 bit manuale. Ricompilato a 16bit e test automatico eseguito.
- **ALU**
  Test della ALU eseguito direttamente sulla scheda poiché composto per la maggior parte di componenti già simulati in precedenza.
- **Gestore display 7 segmenti**
  Test eseguito direttamente sulla scheda poiché molte variabili non erano di facile simulazione. In particolare sulla scheda sono state verificate le giuste connessioni ai display a 7 segmenti, la visualizzazione corretta delle cifre e il giusto valore del divisore di clock in modo da evitare lampeggiamenti eccessivi.
- **Divisore**
  Il divisore è stato testato prima a 4 bit e poi automaticamente a 16 bit.
- **Elevatore a potenza**
  Questo modulo è stato implementato sia attraverso componenti sia tramite blocco sequenziale. In tutti e due i casi è stato prima testato a 4 bit e poi a 8 bit per alcuni casi.
- **UART**

Nel caso della UART è stato implementato e testato prima il modulo di trasmissione e poi quello di ricezione.

- **Terminale**

  Il terminale è stato sia simulato sia testato sulla scheda.

```
 stim_proc: process
begin
   -- hold reset state for 100 ns.
   -- wait for 100 ns;

         --operazione 0 op="0000" out_low=X
         op <= "0000";
         for ix in 0 to 2**N-1 loop
                x <= conv_std_logic_vector(ix,N);
                wait for clk_period/2;

                if out_low=conv_std_logic_vector(ix,N) then
                       errore<='0';
                else
                       errore<='1';
                end if;
                wait for clk_period/2;
         end loop;
         --OK

................

         --operazione 0 op="0010" out_low=X+Y
         op <= "0010";
         for ix in 0 to 2**N-1 loop
            for iy in 0 to 2**N-1 loop
                   x <= conv_std_logic_vector(ix,N);
                       y <= conv_std_logic_vector(iy,N);
                       wait for clk_period/2;

                       if out_low=conv_std_logic_vector(ix+iy,N) then
                              errore<='0';
                       else
                              errore<='1';
                       end if;
                       wait for clk_period/2;
                end loop;
         end loop;
         --OK

         --operazione 0 op="0011" out_low=X+Y+1
         op <= "0011";
         for ix in 0 to 2**N-1 loop
            for iy in 0 to 2**N-1 loop
                   x <= conv_std_logic_vector(ix,N);
                       y <= conv_std_logic_vector(iy,N);
                       wait for clk_period/2;

                       if out_low=conv_std_logic_vector(ix+iy+1,N) then
                              errore<='0';
                       else
                              errore<='1';
                       end if;
                       wait for clk_period/2;
                end loop;
         end loop;
         --OK
```

```
                --operazione 0 op="0100" out_low=X-Y
                op <= "0100";
                for ix in 0 to 2**N-1 loop
                   for iy in 0 to 2**N-1 loop
                           x <= conv_std_logic_vector(ix,N);
                                y <= conv_std_logic_vector(iy,N);
                                wait for clk_period/2;

                                if out_low=conv_std_logic_vector(ix-iy,N) then
                                        errore<='0';
                                else
                                        errore<='1';
                                end if;
                                wait for clk_period/2;
                        end loop;
                end loop;
                --OK


                --operazione 0 op="0101" out_high=high(X*Y) out_low=low(X*Y)
                op <= "0101";
                for ix in 0 to 2**N-1 loop
                   for iy in 0 to 2**N-1 loop
                           x <= conv_std_logic_vector(ix,N);
                                y <= conv_std_logic_vector(iy,N);
                                wait for clk_period/2;

                                if out_high & out_low=conv_std_logic_vector(ix*iy,2*N) then
                                        errore<='0';
                                else
                                        errore<='1';
                                end if;
                                wait for clk_period/2;
                        end loop;
                end loop;
                --OK

                --operazione 0 op="0110" out_high=X modulo Y out_low=X/Y
                op <= "0110"; -- divisione
                for ix in 0 to 2**N-1 loop
                   for iy in 1 to 2**N-1 loop -- il divisore NON può iniziare da 0
                           x <= conv_std_logic_vector(ix,N);
                                y <= conv_std_logic_vector(iy,N);
                                wait for clk_period/2;

                                if iy=0 then
                                        errore<='U';
                                elsif out_high=conv_std_logic_vector(ix mod iy,N) and
out_low=conv_std_logic_vector(ix/iy,N) then
                                        errore<='0';
                                else
                                        errore<='1';
                                end if;
                                wait for clk_period/2;
                        end loop;
                end loop;
                --OK
....................


      wait;
   end process;
```