Implementazione di un moltiplicatore in virgola mobile su FPGA

Tommaso Zanotti

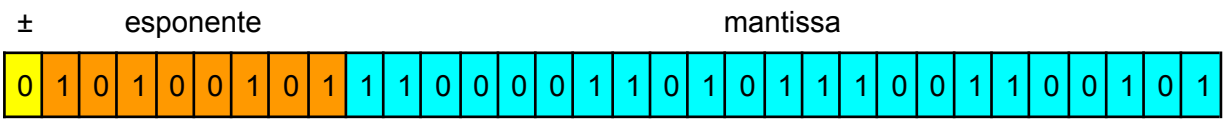
2021-2022

Indice

Introduzione

Un numero in virgola mobile a singola precisione, secondo lo standard IEEE 754 è rappresentato su parole di 32 bit divise nel seguente modo:

1. un bit di *segno*
2. un campo *esponente* formato dai successivi 8 bit
3. un campo *mantissa* formato dai 23 bit rimanenti



Il valore del numero rappresentato nel caso dei numeri normalizzati è calcolabile mediante:

$$(-1)^{S} * 2^{E} * 1.M$$

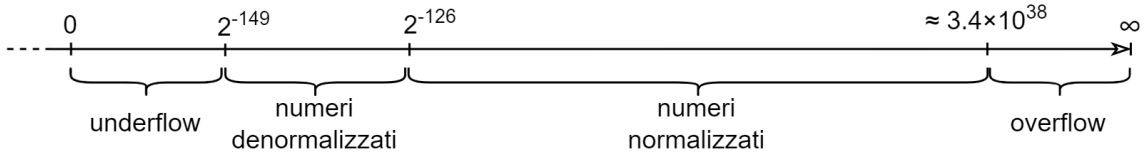
dove S è il bit di *segno*, E il campo *esponente* decrementato di 127 ed infine M il campo *mantissa* che è preceduto dal cosiddetto *bit implicito*.

Combinando opportunamente i bit di esponente e mantissa è inoltre possibile rappresentare le seguenti categorie:

- ±*infinito*, ottenuto ponendo ad 1 tutti i bit di *esponente* e *mantissa*. Tramite il bit di *segno* se ne specifica il segno.
- zero, codificato assegnando il valore 0 ai campi *esponente* e *mantissa*. Si noti che in questo modo si ottengono due codifiche per lo zero, +0 e -0.
- NaN, un valore speciale chiamato “Not a Number” restituito nel caso di operazioni in virgola mobile non valide quali $0 \div 0$ oppure $\infty \times 0$. È codificato ponendo a 1 i bit di *esponente* e ad un qualsiasi valore diverso da 0 il campo *mantissa*.

Lo standard IEEE 754 specifica infine la codifica dei numeri denormalizzati, numeri in virgola mobile che consentono di rappresentare, con una graduale perdita di precisione, i valori compresi tra 0 ed il più piccolo numero normalizzato rappresentabile.
Questi ultimi si distinguono dai numeri normalizzati perché il campo *esponente* assume un valore nullo ed il *bit implicito* è posto a 0.
Il valore del numero rappresentato in questo caso è calcolabile come:

$$(-1)^{S} * 2^{-126} * 0.M$$



Retta dei numeri positivi rappresentabili in virgola mobile a singola precisione.
Per i numeri negativi si ottiene una retta speculare.

Algoritmo di moltiplicazione

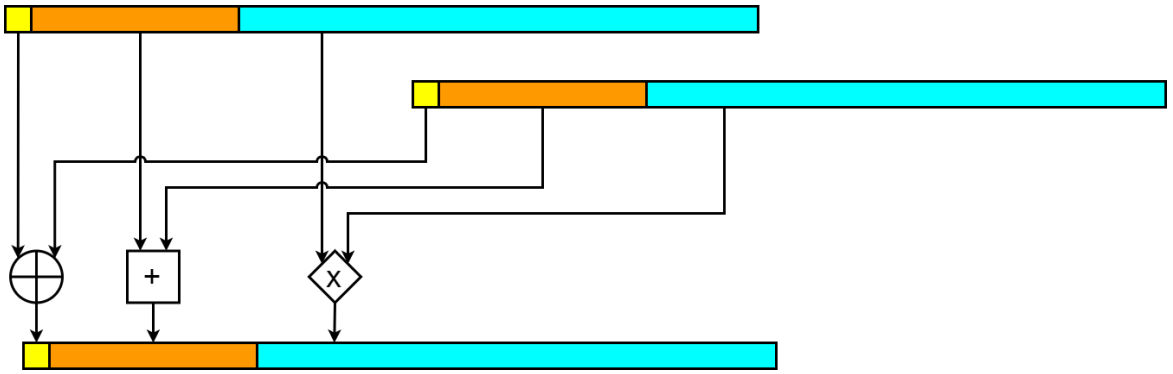
Il prodotto tra due numeri in virgola mobile è così descritto:

$$Z = A * B = \left[(-1)^{S_A} * 2^{E_A} * m_A\right] * \left[(-1)^{S_B} * 2^{E_B} * m_B\right] = (-1)^{S_A+S_B} * 2^{E_A+E_B} * \left(m_A * m_B\right)$$

dove *m* è la concatenazione del *bit implicito* con il campo *mantissa* M.

Nella pratica tutto ciò si traduce nel seguente algoritmo:

1. Ricavare i valori di segno, esponente e mantissa dai numeri in input.
2. Controllare se uno o entrambi gli operandi appartengono ai valori ±0, ±∞ o NaN e se il prodotto rientra nei casi $\infty * 0 = NaN$ o $NaN * x = NaN$.
3. Per ognuno dei numeri in input impostare il *bit implicito* a 1 nel caso sia in forma normalizzata, altrimenti impostarlo a 0 e correggere l'esponente.
4. Calcolare il bit di segno finale attraverso un'operazione di XOR tra i bit di segno degli operandi, la mantissa eseguendo la moltiplicazione tra i valori ottenuti al passo precedente ed infine l'esponente sommando i relativi esponenti tenendo conto del bias.
5. Normalizzare il valore ottenuto eseguendo uno shift a destra ed un incremento dell'esponente o uno shift a sinistra ed un decremento dell'esponente.
6. Arrotondare il risultato.
7. Verificare che non si sia verificato underflow o overflow. In tal caso collassare il risultato rispettivamente a ± 0 o $\pm \infty$, dove il segno è determinato al punto 4.



Nei numeri a virgola mobile la mantissa è codificata su 23+1 bit ma la moltiplicazione produce un risultato di 48 bit e pertanto è necessario applicare un arrotondamento.
La tecnica di arrotondamento *round to nearest (ties to even)* consiste nel:

- Verificare se il numero originale è equidistante dall'arrotondamento per difetto e da quello per eccesso e arrotondare il risultato verso il numero pari più vicino.
- In alternativa arrotondare il risultato verso il numero più vicino

Dal punto di vista pratico equivale ad applicare il seguente algoritmo:

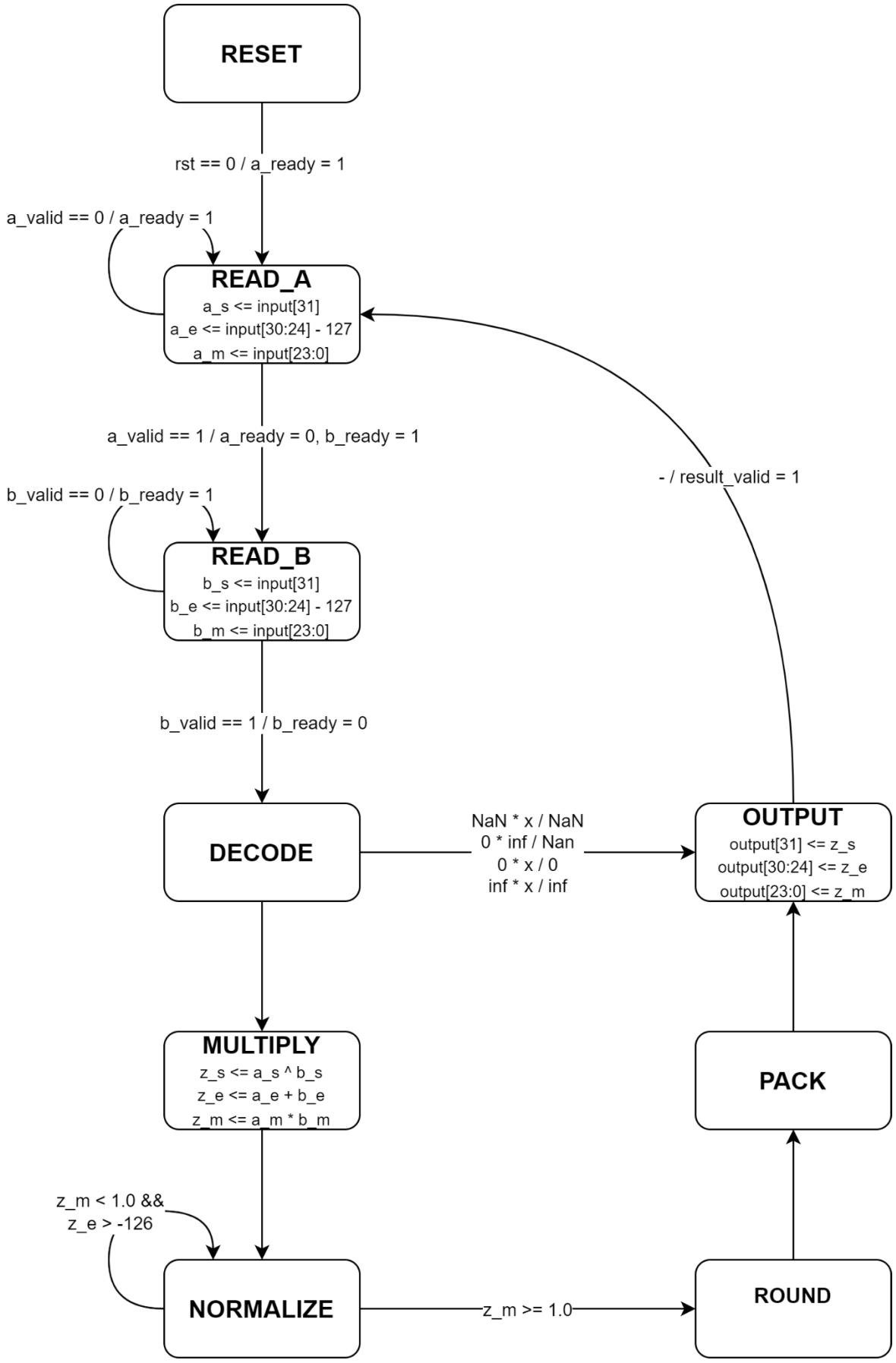
- se la cifra di guardia è pari a 1 e o il resto dei bit che seguono assumono valore non nullo o l'ultima cifra significativa è dispari si arrotonda per eccesso
- altrimenti si arrotonda per difetto.

Implementazione Verilog

Nel file src/fpm.v è possibile trovare l'implementazione del moltiplicatore in Verilog.
All'interno del file è presente il modulo fpm che riceve in input i numeri da moltiplicare (e relativi flag per il coordinamento) e ne calcola il prodotto.
La macchina a stati implementata esegue le seguenti operazioni:

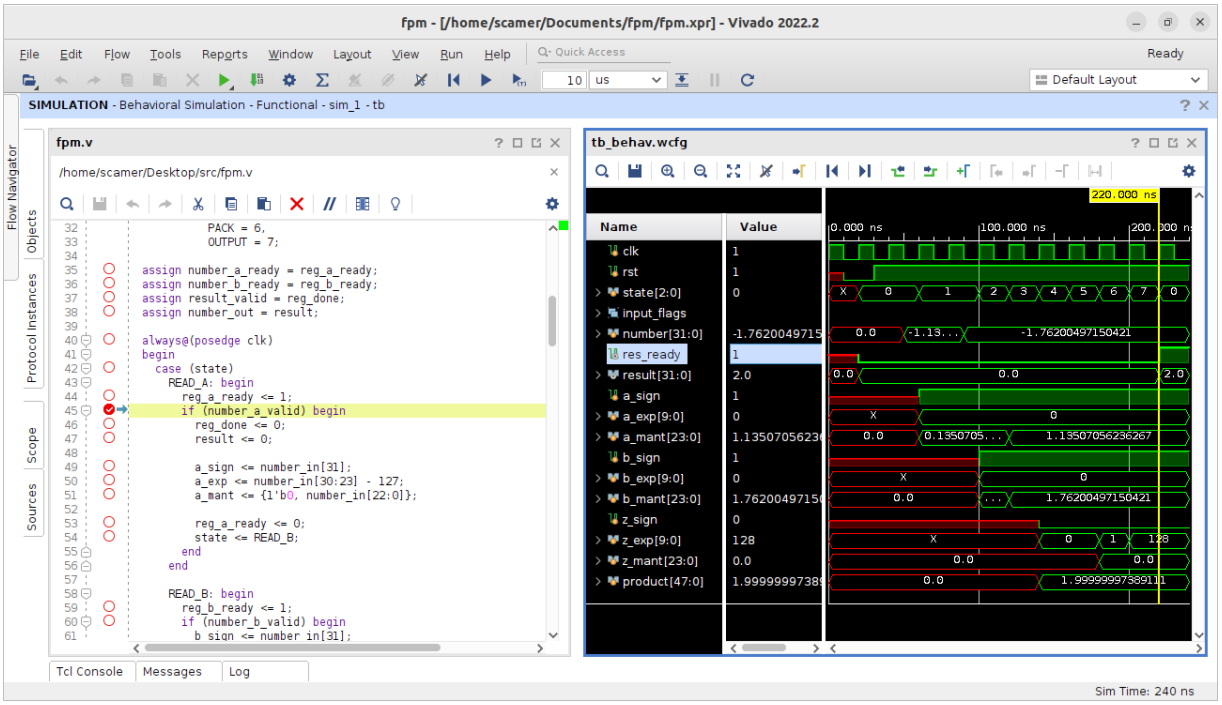
- Ripristina i registri interni fintanto che in ingresso il segnale di reset è impostato a 0.
- Comunica che è pronta a ricevere il primo numero ponendo a 1 il segnale number_a_ready. Quando il segnale number_a_valid viene posto a 1 legge il numero, ricava i valori di *segno*, *esponente* (a meno del bias) e *mantissa* e passa allo stato successivo.
- Similmente pone a 1 il bit number_b_ready ed in seguito memorizza i campi *segno*, *esponente* e *mantissa* del moltiplicatore. La FSM riconosce che il numero e' valido quando number_b_valid assume valore 1.
- Valuta i campi dei numeri in ingresso per distinguere le combinazioni di input non validi o il cui risultato è banale ($0 * x = 0$ e $\infty * x = \infty$). Inoltre riconosce eventuali numeri in forma denormalizzata e imposta adeguatamente *esponente* e *bit implicito*.
- Esegue la moltiplicazione determinando il segno del risultato tramite uno xor tra i bit di segno, somma gli esponenti e moltiplica le mantisse.
- Normalizza il numero ottenuto in output eseguendo uno shift a destra se $m_{OUT} \geq 2$ oppure esegue uno shift a sinistra fintanto che $m_{OUT} < 1.0$.
- Arrotonda il risultato finale applicando la tecnica "round to nearest, ties to even".
- Valuta il risultato per rilevare eventuali under/overflow e prepara il registro contenente il risultato finale concatenando bit di segno, esponente codificato in eccesso 127 e parte frazionaria della mantissa.
- Mostra in output il risultato, pone ad 1 il segnale result_valid e rimane in attesa di un nuovo numero.

Graficamente il processo appare come segue:



Simulazione e verifica dei moduli

Per la verifica ed il debugging dei moduli è stato utilizzato l'IDE Vivado di Xilinx.
Tramite la funzionalità "Behavioral Simulation" è possibile eseguire una simulazione comportamentale ed osservare l'evoluzione dei segnali senza la necessità di sintetizzare in hardware i moduli.
Per eseguire la simulazione è stata descritta nel file *src/fpm_tb.v* una testbech in cui è istanziato il moltiplicatore e che si occupa di generare il segnale di clock, ripristinare il modulo ed infine inviare i due numeri da moltiplicare.
Nella directory *sim* è presente il file per configurare la finestra waveform come di seguito.
Il software permette inoltre di eseguire un debugging passo-passo dell'hardware descritto nei file sorgente e di specificare la codifica dei segnali osservati.



Sintesi logica dei moduli

Per programmare la FPGA presente sulla scheda PYNQ-Z1 è necessario implementare il Block Design e generare il bitstream tramite il software Xilinx Vivado.
In particolare:

- Definire una Intellectual Property (IP) di tipo periferica AXI4 Lite ed istanziare al suo interno il modulo fpm, propagando adeguatamente i segnali.
- Creare il Block Design per interfacciare la IP con il Processing System (PS) della scheda.
- Creare un wrapper HDL: questo wrapper al suo interno gestirà PS, IP (con all'interno il moltiplicatore) ed il modulo per la sincronizzazione.
- Generare ed esportare Bitstream e Block Design.

L'interfaccia della periferica AXI generata da Vivado deve essere modificata nei seguenti punti:

- Dopo la definizione dell'interfaccia del modulo bisogna dichiarare i wire per gestire gli output del modulo.

```
wire[31:0] result;
wire[2:0] out_flags;
```

- Nel processo *(always @(posedge S_AXI_ACLK))* si memorizzano gli output del modulo nei relativi registri.
  Modificare le seguenti righe:

```
else begin
    if (slv_reg_wren)
```

  E rimpiazzarle con:

```
else begin
    slv_reg2 < = result;
    slv_reg3 < = 32'b0 + out_flags;
    if (slv_reg_wren)
```

- Nel medesimo processo è necessario inibire la sovrascrittura dei registri di output eliminando i rami *2'h2* e *2'h3* e nel ramo *default* modificare come segue:

```
slv_reg0 < = slv_reg0;
slv_reg1 < = slv_reg1;
// slv_reg2 < = slv_reg2;
// slv_reg3 < = slv_reg3;
```

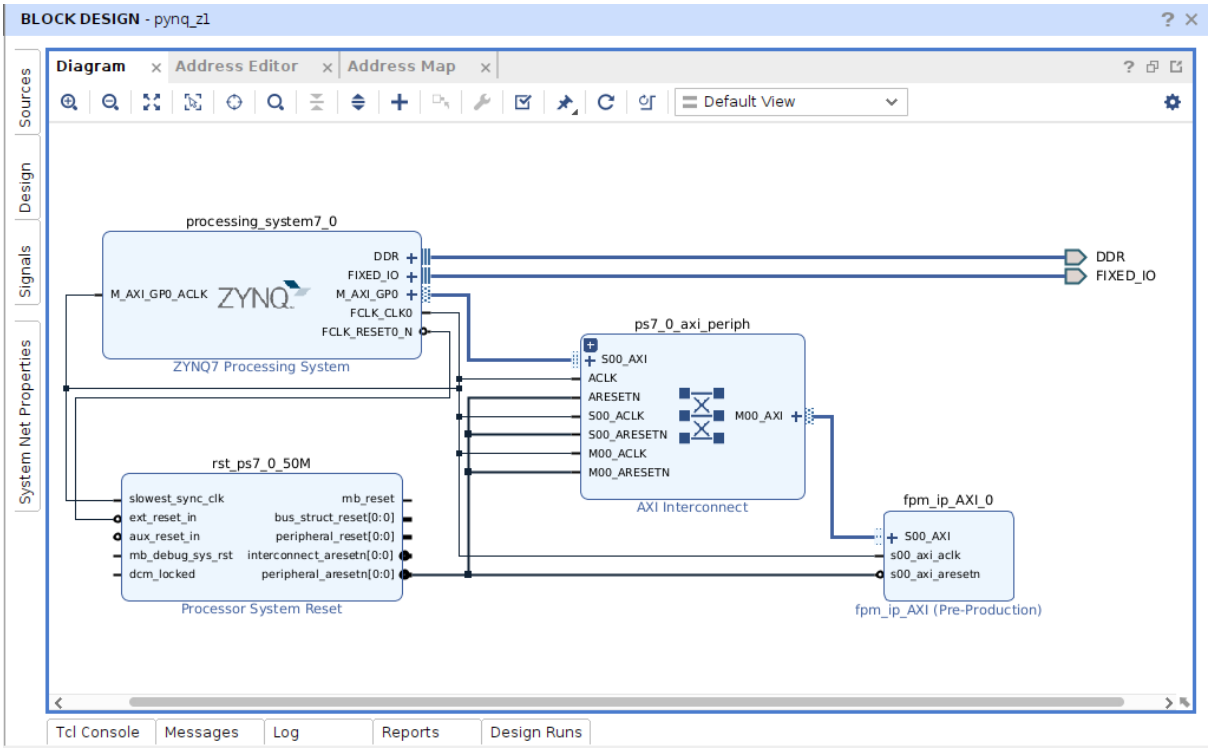
- Istanziare il moltiplicatore e propagare i segnali dell'interfaccia dopo il commento "add user logic here":

```
fpm multiplier(
    .clk(S_AXI_ACLK),
    .rst(S_AXI_ARESETN),

    .number_in(slv_reg0),
    .number_a_valid(slv_reg1[0]),
    .number_b_valid(slv_reg1[1]),

    .number_out(result),
    .number_a_ready(out_flags[0]),
    .number_b_ready(out_flags[1]),
    .result_valid(out_flags[2])
);
```

Attraverso uno script Python è possibile utilizzare il Bitstream e il Block Design per programmare e controllare la FPGA. I registri che verranno utilizzati dall'interfaccia AXI sono i seguenti:

Nome registro	Segnali associati
slv_reg0	number_in
slv_reg1	{ number_b_valid, number_a_valid }
slv_reg2	number_out
slv_reg3	{ result_valid, number_b_ready, number_a_ready }

Il diagramma del Block Design ottenuto dovrebbe essere il seguente:



Controllo della FPGA da riga di comando

È possibile interagire con la FPGA tramite command line utilizzando lo script fpm.py nella cartella *dist* del repository.
Per funzionare lo script necessita dell'interprete Python in versione 3.8 o superiore e della coppia di file Bitstream e Block Design per programmare la Programmable Logic. Essi vanno posizionati nella working directory, rinominati rispettivamente come system.bit e system.tcl, o in alternativa è possibile specificarne il percorso tramite il flag –bitstream <path>

Lo script Python fpm.py:

- Riceve come argomento da linea di comando i due numeri in virgola mobile da moltiplicare.
- Programma la FPGA con il file bitstream (e relativo block diagram).
- Codifica e sottomette i due numeri alla FPGA per eseguirne la moltiplicazione.
- Infine legge il risultato e lo mostra in codifica decimale.

Tramite il flag -v (abbreviazione di –verbose) è possibile visualizzare la codifica in binario dei valori scambiati tra Processing System o Programmable Logic.

```
~$ sudo ./fpm.py -h
usage: main.py [-h] [-v] [--bitstream BITSTREAM] a b

Multiply a pair of floating point numers

positional arguments:
  a                     multiplicand
  b                     multiplier

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         increase output verbosity (default: False)
  --bitstream BITSTREAM
                        path of the bitstream file (default: system.bit)
```

Per utilizzare lo script occorre:

- Copiare con il comando scp i file sulla board:
  `scp system.* <user>@<board>:~/`
  `scp fpm.py <user>@<board>:~/`
- Collegarsi tramite protocollo SSH alla board:
  `ssh <user>@<board>`
- Assegnare i permessi di esecuzione allo script:
  `chmod +x fmp.py`
- Eseguire lo script Python passando come argomento due numeri decimali:
  `sudo ./fpm.py <moltiplicando> <moltiplicatore>`

Scelte progettuali

1. Per ridurre il numero di input necessari per comunicare con il moltiplicatore si è optato per ricevere i numeri in input in due passaggi differenti. In questo modo sono necessari solo 32 bit di input e 4 bit di controllo anziché 64 bit di input (32 per numero) e 2 di controllo. Inoltre la sottomissione di entrambi i numeri alla periferica AXI nello stesso ciclo di clock avrebbe richiesto una comunicazione sincrona e timing precisi tra Programmable Logic e Processing System oppure ulteriori segnali di controllo per evitare che l'output venga immediatamente nascosto e venga ri-eseguita la medesima operazione.

2. Per l'implementazione descritta nel seguente documento vengono utilizzati i seguenti registri:
   - slv_reg0: registro a 32 bit contenente il numero ricevuto in input dalla FSM.
   - slv_reg1: registro contenente due flag che indicano alla macchina a stati quando memorizzare il numero in input.
   - slv_reg3: registro a 32 bit contenente il risultato della moltiplicazione.
   - slv_reg4: registro contenente 3 bit di flag impostati dalla macchina a stati per indicare quando accetta nuovi input e quando ha terminato la moltiplicazione.

   Si è optato per combinare i bit di controllo in due registri, uno per i segnali in input ed uno per quelli in output, in modo tale da minimizzare i registri utilizzati e mappati in memoria. Nello script sono state inoltre definite delle maschere che, usate assieme agli operatori logici AND e OR, permettono di estrarre dai registri il valore dei singoli flag.

3. Le tecniche di arrotondamento banali quali arrotondamento per difetto (o troncamento) e arrotondamento per eccesso prevedono che l'arrotondamento venga effettuato sempre verso una precisa direzione, sia essa lo 0 o l'infinito.
   Nonostante risultino di facile implementazione, introducono però un bias che su grandi distribuzioni può portare a incongruenze.
   L'arrotondamento *round to nearest (ties to even)* invece preserva l'equilibrio statistico tra i due arrotondamenti imponendo che nel caso in cui il valore originale cada esattamente nel mezzo tra arrotondamento per eccesso e per difetto esso avvenga in base alla parità del bit che precede la cifra di arrotondamento.