Elettronica dei Sistemi Digitali
Lab 06

ASM Chart

Gruppo: A11

Pronesti, Massimiliano	S245831
Oropallo, Maria Vittoria	S245999
Nicolicchia, Riccardo	S244728
Miceli, Michelangelo	S245478

10 maggio 2020

Indice

Capitolo 1

Progettazione

1.1 Introduzione

Lo scopo dell'attività laboratoriale è stato la realizzazione di una versione semplificata del controllore PID (Proporzionale-Integrale-Derivativo) discreto tramite macchina a stati algoritmica, descritta in linguaggio VHDL. In particolare, a partire da una prima memoria (MEM_A) di 1024 celle contenente la funzione d'errore $e[k]$ campionata agli istanti k sotto forma di numeri espressi in complemento a 2 su 8 bit, la macchina valuta la funzione di controllo memorizzando gli output nelle celle di una seconda memoria della stessa capienza, secondo la relazione

$$u[k] = K_p \cdot e[k] + K_i \sum_{j=0}^{k} e[j] + K_d \left(e[k] - e[k-1]\right) \tag{1.1}$$

con K_p, K_i, K_d costanti note.

1.2 Pseudo codice

Di seguito è riportato lo pseudo codice dell'algoritmo(C-like syntax)

```
1  #define PAR 1024
2  #define Kp 3.75
3  #define Ki 2
4  #define Kd 0.5
5
6  int data_in [PAR], memA[PAR], memB [PAR];
7  int partial = 0;
```

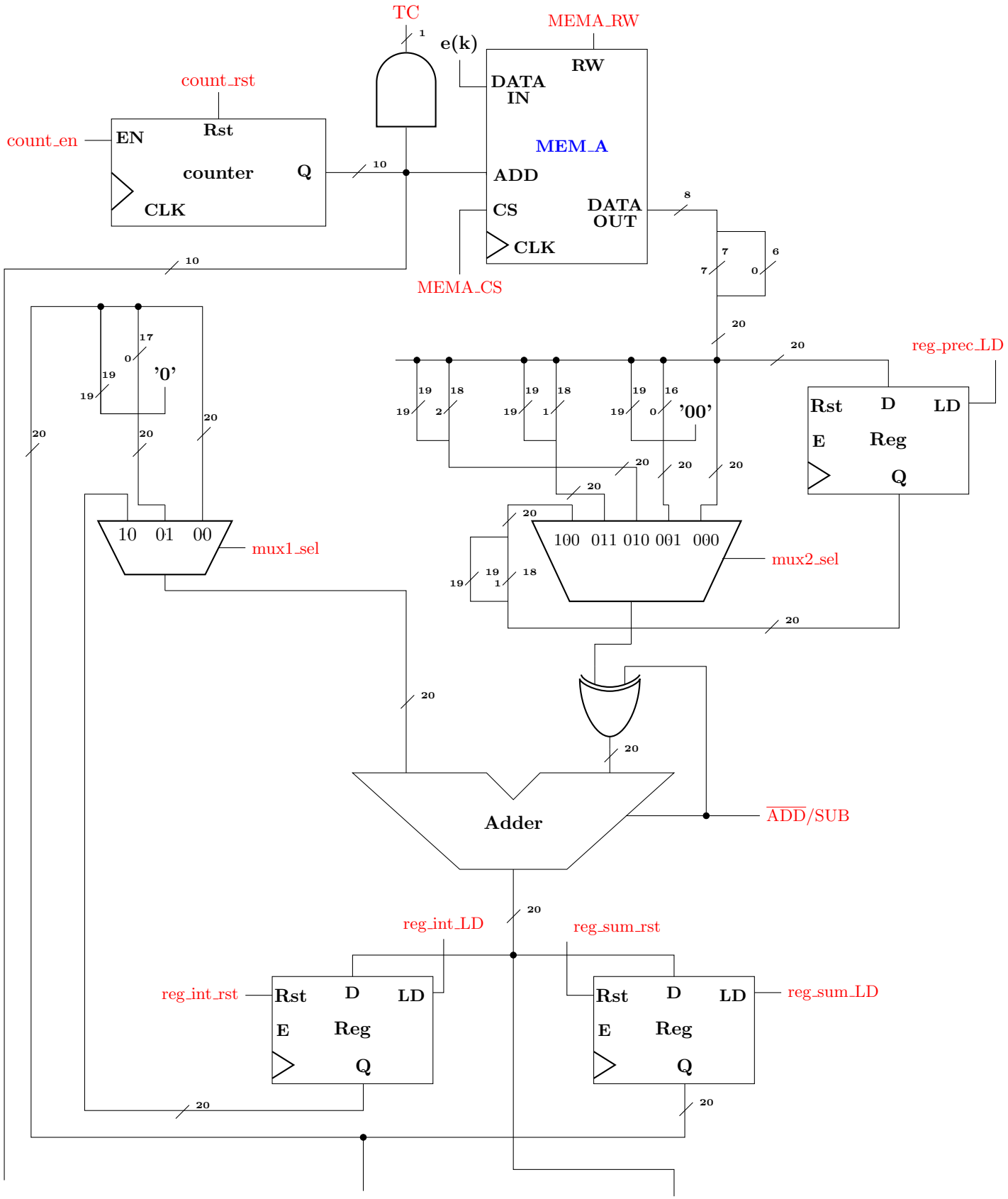
```
8
9  // getting data
10 for (int i = 0; i < PAR ; ++i)
11    memA[i] = data_in [i];
12
13
14 // processing data
15 int tmp;
16 int memA_prec = 0;
17 for (int i = 0; i< PAR; ++i){
18     partial += memA[i];
19     tmp = Kp * memA[i] + Ki * partial + Kd * (memA[i] - memA_prec );
20
21    // sat check
22     if (tmp > 127)
23        tmp = 127;
24     else if (tmp < -128)
25        tmp = -128;
26
27    // memB writing
28     memB[i] = tmp;
29     memA_prec = memA[i];
30 }
```

Code 1.1: pseudo-code

Come da specifiche, i dati in ingresso sono preliminarmente acquisiti e memorizzati nella prima memoria ed, in seguito, processati secondo la (1.1) e memorizzati nella corrispondente cella della seconda memoria, saturando al valore massimo o minimo rappresentabile su 8 bit in CA2 in caso di overflow. Si osservi che, sfruttando una variabile d'appoggio nella quale è salvato ad ogni ciclo il valore memorizzato precedentemente, non si è reso necessario includere alcun costrutto condizionale per distinguere la prima e l'ultima iterazione dalle restanti 1022.

1.3 Datapath

Il datapath dell'algoritmo è riportato in **figura 1.1**, laddove i segnali di controllo sono evidenziati in rosso.
Si osservi che, per evitare ridondanze grafiche nelle estensioni di parallelismo, si è preferito indicare esplicitamente quali bit restano invariati e solo una volta i bit "riempitivi".


TC
MEMA_RW
e(k)
RW
DATA IN
MEM_A
ADD
CS
DATA OUT
CLK
MEMA_CS
count_rst
count_en
EN
Rst
counter
Q
CLK
'0'
'00'
reg_prec_LD
Rst
D
LD
E
Reg
Q
10 01 00
mux1_sel
100 011 010 001 000
mux2_sel
Adder
ADD/SUB
reg_int_LD
reg_sum_rst
reg_int_rst
reg_sum_LD

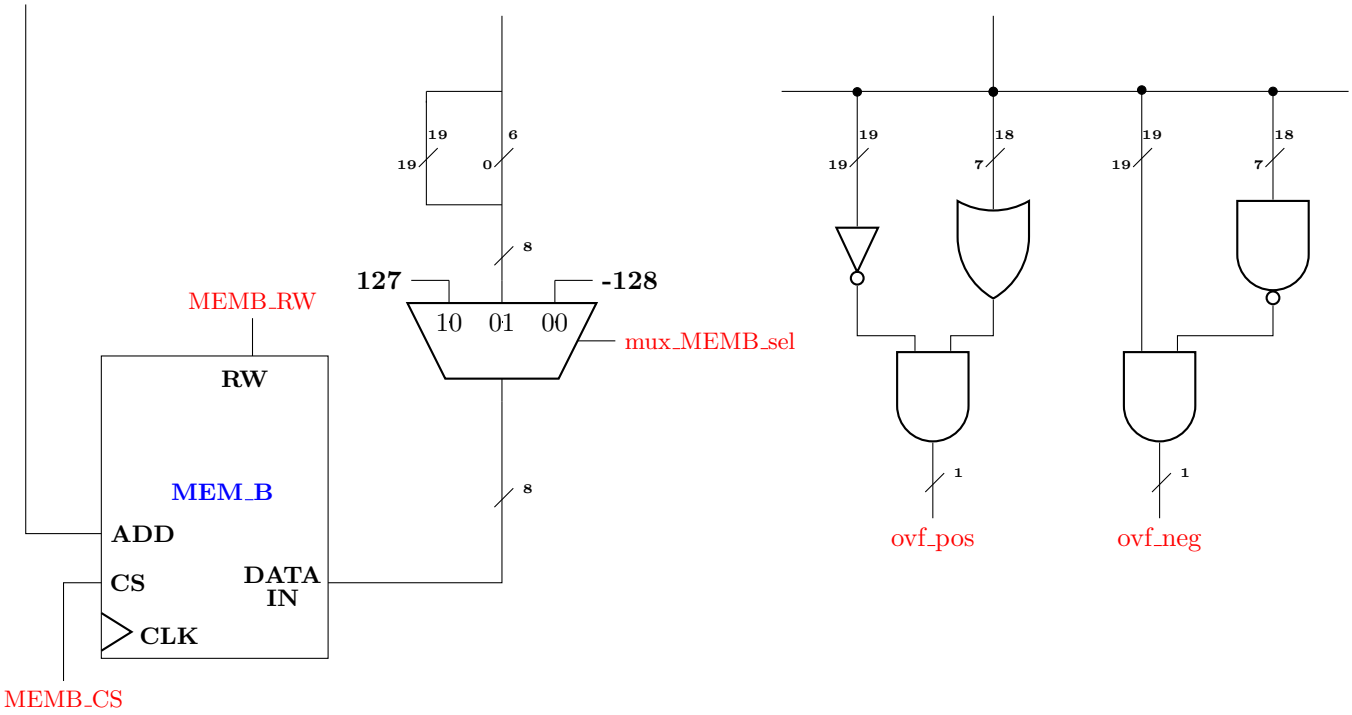


Figura 1.1: Datapath

Ad esempio, l'estensione del segnale d'uscita della memoria A (di parallelismo 8) su 20 bit conserva i 7 bit più a destra e presenta tutti i bit successivi pari al bit di segno.
Come da specifiche, si è fatto uso di un unico sommatore e di nessun moltiplicatore, sicché le operazioni aritmetiche sono state effettuate tramite shift di bit, come descritto a **1.3.2**. Di seguito è riportata una descrizione più dettagliata del datapath:

- counter: fornisce gli indirizzi della memoria A e, dopo essere stato resettato, quelli della memoria B;
- REG_PREC, nel quale si memorizza e[k-1], i.e. il valore precedente letto dalla memoria A;
- REG_INT, nel quale si memorizza, aggiornandola ad ogni colpo di clock, la somma di tutti i termini di errore letti;
- REG_SUM nel quale, nei diversi stati, si aggiorna la somma totale per la computazione di u[k];
- Adder, utilizzato per svolgere tutte le operazioni di somma e sottrazione. Riceve due ingressi signed ed un segnale di controllo ($\overline{\text{ADD}}$/SUB), attivo a '0' per la somma ed ad '1' per la differenza;

- MUX1, il quale, a seconda del segnale di selezione, sceglie la prima entrata dell'adder;
- MUX2, il quale, a seconda del segnale di selezione, sceglie la seconda entrata dell'adder;
- MUX_MEMB, il quale, a seconda del segnale di selezione, sceglie il dato da scrivere nella memoria B. In particolare, è utilizzato per la gestione dell'overflow.

Si osservi che i tre registri adempiono alla funzione delle tre variabili tmp, partial e memA_prec dello pseudo-codice. Tutti i segnali di reset sono di tipo high-active, i.e. agiscono quando assumono valore logico '1'.

1.3.1 Parallelismo massimo

Si realizza il circuito estendendo i dati in ingressi su un numero di bit tale per cui non ci sia mai overflow. A questo scopo, si stima il massimo valore che può assumere la funzione di controllo. Eseguendo un'analisi di caso peggiore, i.e. ponendoci nella condizione nella quale tutti i valori e[k] siano pari al minimo valore esprimibile su 8 bit in CA2 (-128) e considerando l'operazione per la computazione di u[1023] si ha

$$
\begin{aligned}
&2(e[0] + e[1] + \ldots + e[1023]) = -128 \cdot 1023 \cdot 2 = -261888 \\
&3.75 \cdot e[1023] = 3.75 \cdot (-128) = -480 \\
&0.5(e[1023] - e[1022]) = 0 \\
&u[1023] = -262368
\end{aligned}
$$

per la rappresentazione del quale occorrono 19 bit ed un ulteriore bit di segno. Ne segue che tutte le operazioni sono state svolte su 20 bit.

1.3.2 Operazioni aritmetiche

Per le operazioni di filling, moltiplicazione e divisione sono stati opportunamente concatenati i segnali, come mostrato nel datapath. In particolare:

- divisione per 4: si procede a shiftare il numero di due bit a destra, mantenendo il bit di segno. Di conseguenza, si concatena il 19esimo bit, ripetuto tre volte, con i bit da 18 a 2.

- divisione per 2: analogamente, si procede a shiftare il numero di un bit a destra. Si concatena il bit di segno, ripetuto due volte, con i bit da 18 a 1.
- moltiplicazione per 4: si procede a shiftare il numero di due bit a sinistra: al bit di segno vengono concatenati i bit da 16 a 0 e il bit '00'.
- moltiplicazione per 2: si procede a shiftare il numero di un bit a sinistra: al bit di segno vengono concatenati i bit da 17 a 0 e i bit '0'.

1.3.3 Gestione dell'overflow

Per le operazioni di confronto e saturazione non è stato utilizzato l'adder, bensì una logica combinatoria, in modo da utilizzare un numero minore di stati e colpi di clock. In particolare, si ha overflow se i bit da 7 a 19 sono diversi. In questo caso, si satura al valore massimo se il bit di segno è 0, a quello minimo altrimenti.

1.4 ASM Chart e Control ASM Chart

Di seguito è riportata l'ASM chart dell'algoritmo (**figura 1.2**). Per una maggior chiarezza, gli stati sono stati raggruppati secondo un codice a colori.
Inizialmente, la macchina si trova in uno stato di IDLE (azzurro) nel quale sia i registri che il contatore sono resettati. Ricevuto il segnale di START dall'esterno, inizia l'esecuzione dell'algoritmo che, sul modello di **code 1.1**, passa per una fase di acquisizione dei dati nelle celle della memoria A (rosso), scandita dall'incremento del contatore ed, in seguito, alla lettura della stessa (marrone) e all'esecuzione delle operazioni aritmetiche (giallo) con relativo controllo sull'overflow ed eventuale saturazione (verde). Al termine, si rimane nello stato di done fino all'handshake (arancione) che riporta allo stato di IDLE per un nuovo ciclo d'esecuzione. Le operazioni aritmetiche, essendo scandite dal contatore, richiedono 5 colpi di clock per essere portate a termine.
In ultimo, l'ASM chart della Control Unit è mostrata in **figura 1.3**.

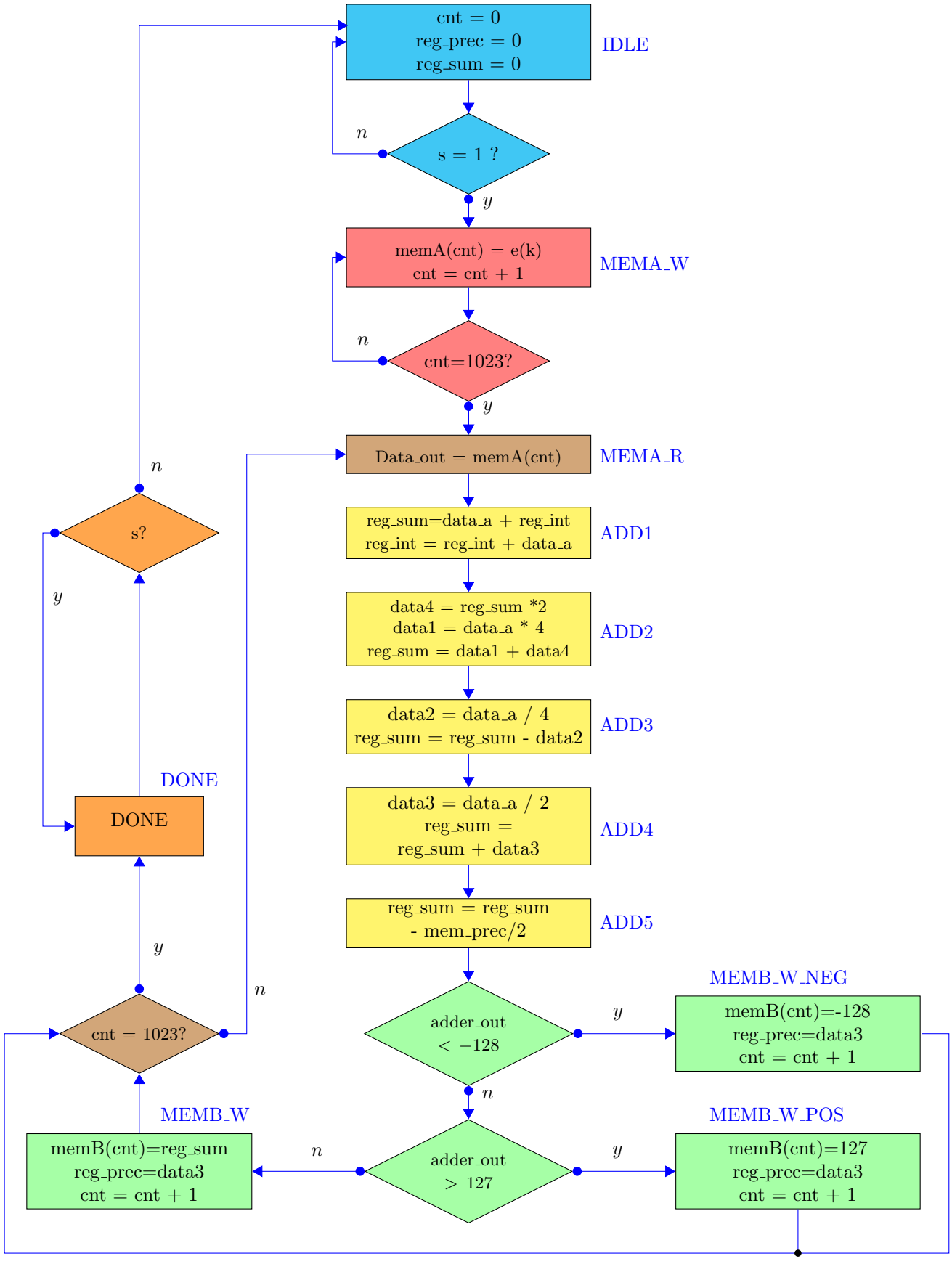


Figura 1.2: ASM chart

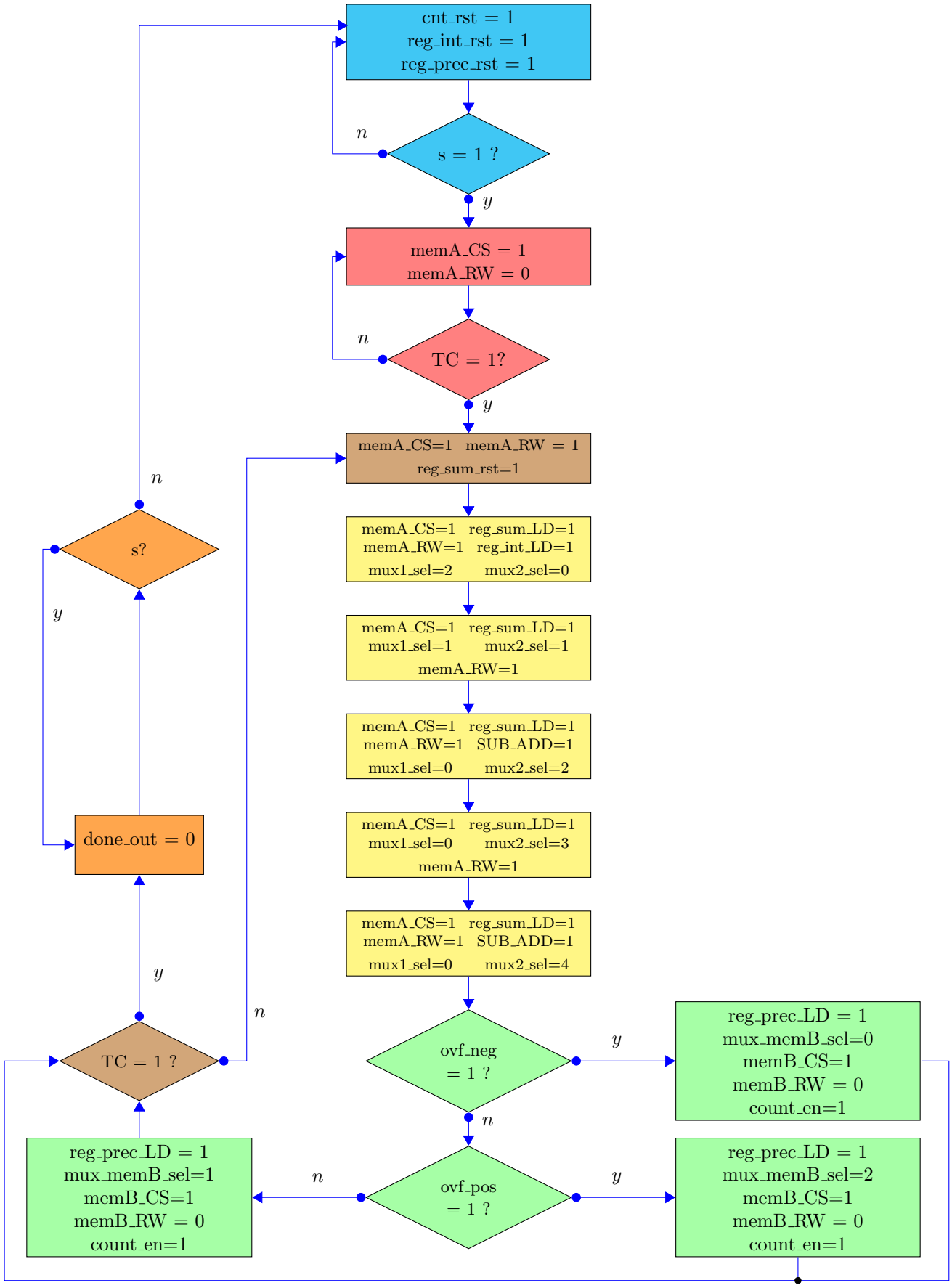


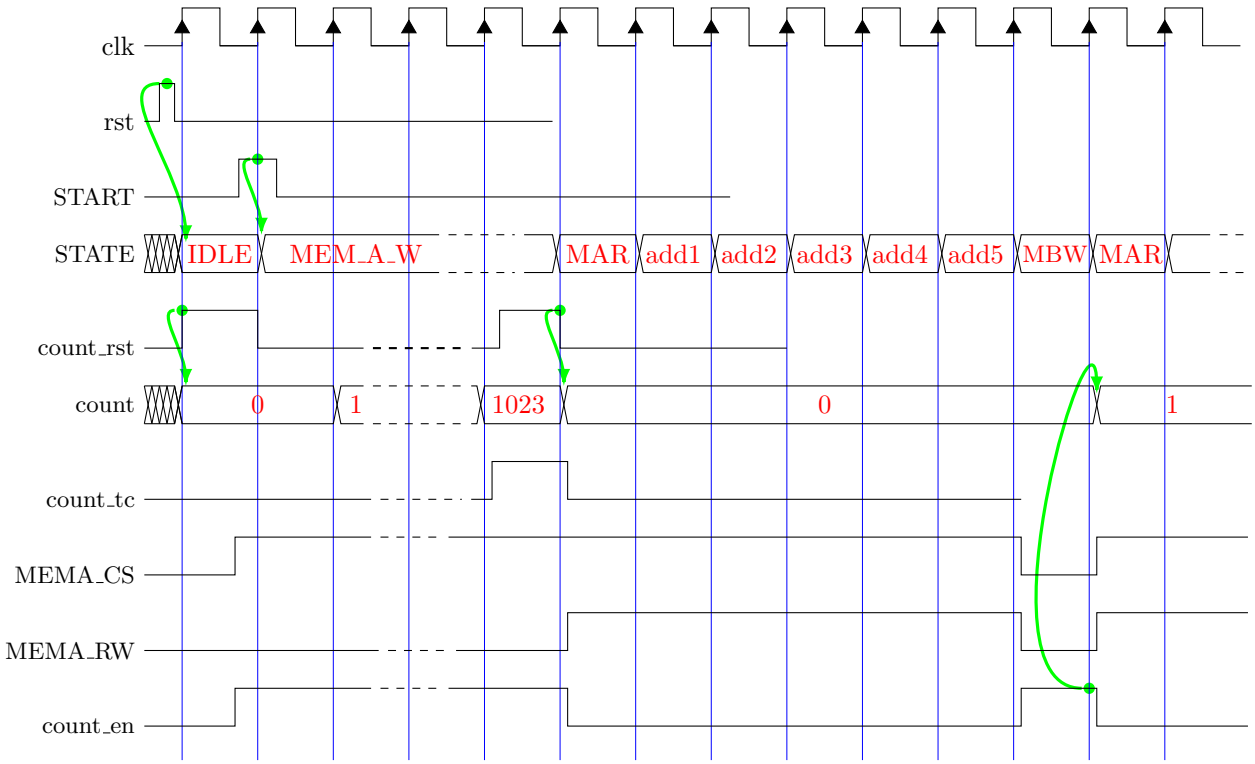
Figura 1.3: Control ASM chart

Capitolo 2

Simulazione

2.1 Time diagram

Di seguito è riportato un frammento significativo del time diagram del circuito durante le fasi di acquisizioni e processamento dei dati. Per questioni grafiche, i nomi di alcuni stati sono stati abbreviati.



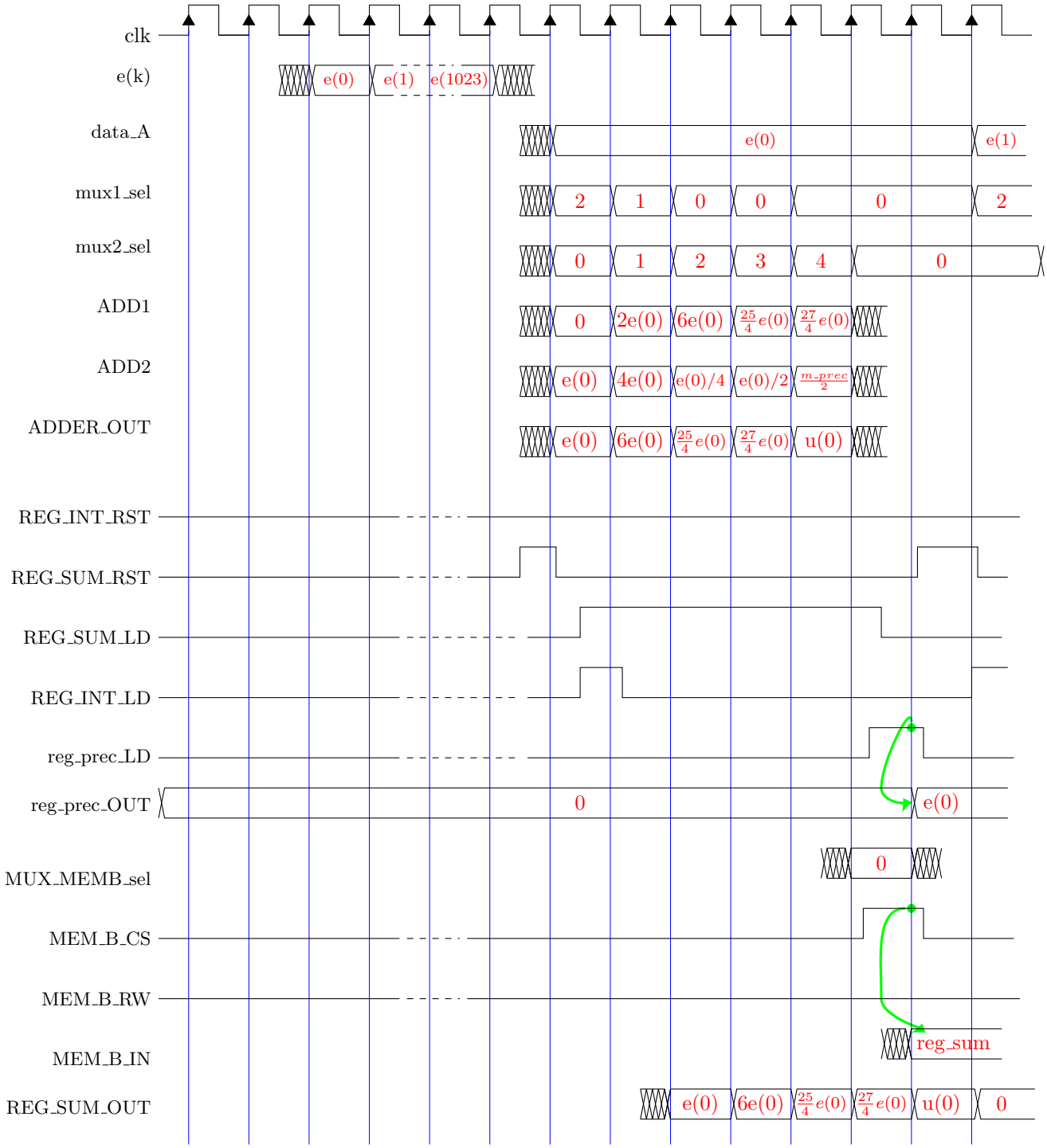


Figura 2.1: Time diagram

Di seguito se ne riporta una descrizione: un impulso di reset porta la macchina nel suo stato di IDLE, nel quale si attiva il segnale count_rst che resetta il contatore. Il segnale di START viene campionato sul secondo fronte di salita del clock, portando la macchina nello stato MEM_A_W, nel quale si attiva il segnale MEMA_CS, il segnale MEMA_RWn va a zero, abilitando la scrittura, e il segnale count_en viene attivato. In questo stato, ad ogni colpo di clock, viene incrementata l'uscita count del contatore e, nel successivo colpo di clock, è disponibile il dato e(k), e.g. al secondo colpo count = 0 sicchè e(0) è disponibile al terzo colpo.
Quando count = 1023 (al 1025 colpo di clock), si attiva il segnale di terminal count (TC) e la macchina transita nello stato MEMA_RD, nel quale il contatore torna a zero e si attiva il segnale reg_sum_rst. Nello stato successivo, ADD1, è disponibile il dato e(0), si attiva il segnale reg_sum_LD, e sono istantaneamente disponibili i segnali di selezione dei multiplexer e le entrate e l'uscita dell'Adder. Similmente, negli stati successivi (ADD2, ADD3, ADD4, ADD5) variano i segnali di selezione dei multiplexer e i dati in entrata e uscita dell'adder. Si noti che i dati reg_sum_out sono la replica dei dati adder_out shiftati a destra di un colpo di clock.
Nello stato MEMB_WR, si attiva il segnale di count_en, dimodoché, nello stato successivo, il contatore sia incrementato, si attivi il segnale MEMB_CS (con ritardo delta) ed il ciclo riparta per la computazione di u[1]. Si noti, inoltre, che, per la saturazione del risultato u(k), quest'ultimo deve già essere disponibile nello stato ADD5, nel quale si verifica se sia necessario saturarlo o meno. Di conseguenza, alla logica combinatoria che si occupa di fare i confronti nello stato ADD5, viene collegato il segnale adder_out, già disponibile in tale stato, invece che il segnale reg_sum_out, il cui contenuto sarebbe, invece, disponibile nello stato successivo. Infine, è da notare che tutti segnali gestiti dalla CU, tra cui i segnali di load, reset, count_tc, count_en, si attivano sempre con un certo ritardo corrispondente al delta delay; e.g il segnale reg_int_LD non si attiva esattamente nello stesso momento in cui si entra nello stato ADD1, ma con qualche istante di ritardo.

2.2 Testbenching

Per ciò che concerne il testbenching della top entity, ci si è serviti di uno script MATLAB (**code 1.2**) -preferito ad altri linguaggi per la presenza di funzioni di conversione predefinite da decimale a binary signed-, il quale genera, casualmente, un file di input di 1024 valori binari. che sono stati utilizzati per stimolare il circuito in fase di sintesi tramite le librerie *std.textio* e *ieee.std_logic_textio*, utilizzate, precisamente, per gestire la lettura riga per riga delle stringhe contenute nel file e interpretarle come std_logic_vector (poi opportunamente convertite in signed come da specifica di progetto).

```
1  clear
2  close all
3  clc
4
5  n=1024;
6  random_num = int8(randi([-128 127],1,n));
7
8  for i=1:n
9      %binary random numbers on 8 bits
10     binary_in{i}=dec2bin(typecast(int8(random_num(i)),'uint8'),8);
11 end
12
13 fileID1 = fopen('input.txt','w');
14 for i=1:n
15     %formatted write on file
16     fprintf(fileID1,'%s\n',binary_in{i})
17 end
18
19 fclose(fileID1);
```

Code 2.1: filegen.m

Data la notevole mole di risultati prodotti, si è optato per l'acquisizione di un frammento significativo del timing, contenente solo i dati e i passaggi più rilevanti per mostrare l'adeguato funzionamento della macchina a stati. In aggiunta per vericare l'effettiva correttezza delle operazione si è preferito stampare su file i risultati in uscita in maniera sincrona al processo di scrittura sulla memoria B, sfruttando un process di write, per poi processarli attraverso un secondo script MATLAB (**code 2.2**), nel quale si esegue una scansione del file di input, si procede col cast su 8 bit dei dati letti, per poi convertirli in double in modo da avere la massima precisione nelle operazioni di divisione.

```
clear
close all
clc

n=1024;
partial =0;
memA_prec = 0;

%% Reading inputs file
fileID1 = fopen('input.txt','r');
input = strsplit(fileread('input.txt'), { '\r','\n'});
fclose(fileID1);


empties = find(cellfun(@isempty,input));
input(empties) = [];

%% binary to double
for i=1:n
    tmp=typecast(uint8(bin2dec(char(input{i}))),'int8');
    input_dec{i}=double(tmp);
end

%% algorithm
for i=1:n
    %ADD1
    partial = approx(partial) + approx(input_dec{i});
    %ADD2
    u_add2=approx(2*partial)+approx(4*input_dec{i});
    %ADD3
    u_add3=u_add2 - approx(input_dec{i}/4);
    %ADD4
    u_add4= u_add3 + approx(input_dec{i}/2);
    %ADD5
    u_add5=approx(u_add4 - approx(memA_prec/2));

%% ovf check
    if (u_add5 > 127)
        u_add5 = 127;
    elseif (u_add5 < -128)
        u_add5 = -128;
    end

    result(i) = approx(u_add5);
    memA_prec = input_dec{i};
end

% reading VHDL output file
fileID2 = fopen('output.txt','r');
```

```
50 VHDL = strsplit(fileread('output.txt'), { '\r','\n'});
51 fclose(fileID2);
52
53 empties = find(cellfun(@isempty,VHDL));
54 VHDL(empties) = [] ;
55
56 for i=1:n
57     tmp=str2double(VHDL{i});
58     result_vhdl(i)=approx(tmp);
59 end
60
61 %% comparing results
62 for i=1:n
63     check(i) = isequal(result(i),result_vhdl(i));
64 end
65
66 %error check
67 error = find(check==0);
68 error_vhdl = result_vhdl(error);
69 error_matlab = result(error);
70
71
72
73 function result=approx(value)
74     result=int32(floor(value));
75 end
```

Code 2.2: res_check.m

In seguito, vengono svolte le stesse operazioni della FSM, nello stesso ordine e con la medesima aritmetica, i.e. compiendo le medesime approssimazioni. Infine, si confrontano i risultati dati dallo script e quelli in uscita dal PID_controller descritto in hardware. In **figura 2.2** sono mostrati due estratti iniziali dei file di input e output.

```
1  00100000
2  00101001
3  11011010
4  00101010
5  00001101
6  11010111
7  11101010
8  00000101
9  00101110
10 00101111
```

```
1  127
2  127
3  -111
4  127
5  127
6  -82
7  -18
8  96
9  127
10 127
```

Figura 2.2: estratti dei file di input ed output

Di seguito è inoltre riportato un grafico che mostra la corrispondenza tra i valori attesi e quelli effettivamente prodotti.

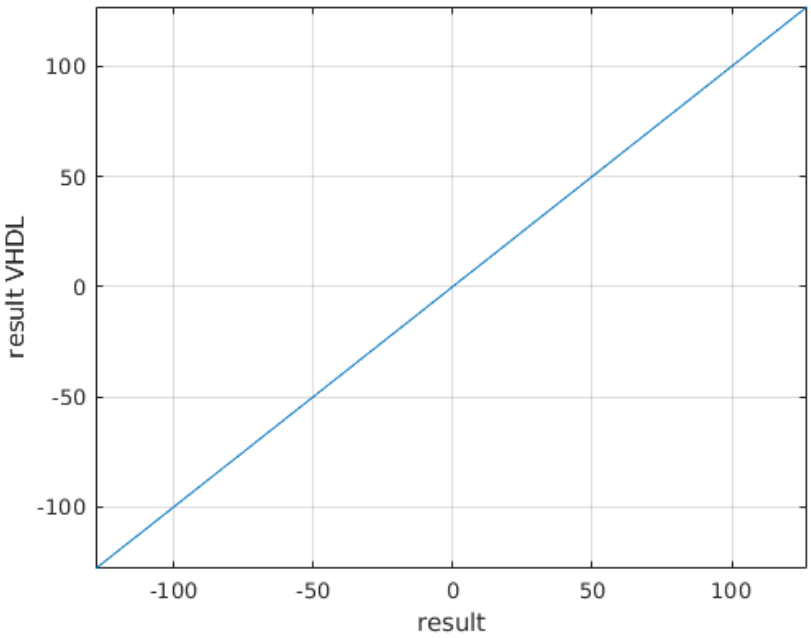


Figura 2.3: Results comparison

Il testbench ed il relativo risultato della simulazione sono riportati, rispettivamente, in **code 2.3** e **figure 2.4**.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use std.textio.all;
5  use ieee.std_logic_textio.all;
6
7  entity PID_controller_tb is
8  end PID_controller_tb;
9
10 architecture ASM of PID_controller_tb is
11
12   component PID_controller
13    port(
14         rst,clk,s : in std_logic;
15         ext_data : in signed(7 downto 0);
16         done_out : out std_logic;
```

```
17          memB_out : out signed(7 downto 0);
18          memB_CS_ftb: out std_logic
19      );
20    end component;
21
22    constant T_clk : time := 10 ns;
23    signal rst,clk,start,done,memB_CS: std_logic;
24    signal ext_data,memB_out : signed ( 7 downto 0);
25
26
27  begin
28
29    rst <= '1','0' after 1 ns;
30    start <= '0', '1' after 7 ns,'0' after 17 ns;
31
32    CLK_GEN: process begin
33      clk <= '0','1' after 5 ns;
34      wait for T_clk;
35    end process CLK_GEN;
36
37    PID: PID_controller port map
          (rst,clk,start,ext_data,done,memB_out,memB_CS) ;
38
39
40   -- reading input file
41    read_file: process
42
43    file file_input : text ;
44    variable buf : line;
45    variable input_data : std_logic_vector (7 downto 0);
46    begin
47
48        file_open(file_input , "input.txt", read_mode);
49        if(start = '1') then
50          wait for T_clk;
51          while endfile(file_input) loop
52
53              readline(file_input,buf);
54              read(buf, input_data);
55              ext_data <= signed(input_data);
56
57            wait for T_clk;
58          end loop;
59
60        else
61          ext_data <= "ZZZZZZZZ";
62          wait for T_clk;
63        end if;
64
65        file_close(file_input);
```

```
66      end process;
67
68
69  -- writing output on new file
70    write_file: process
71     file file_output:text;
72     variable fstatus : File_open_status;
73     variable buf: line; -- buffer to file
74    begin
75
76      file_open(fstatus, file_output, "output.txt",write_mode);
77
78          wait until rising_edge(clk) and memB_CS = '1';
79          write(buf,integer'image(to_integer(memB_out)));
80          writeline(file_output, buf);
81
82    if (done ='1') then
83      file_close(file_output); -- flush the buffer to the file
84    end if;
85  end process;
86
87  end architecture;
```

Code 2.3: PID_controller_tb.vhd

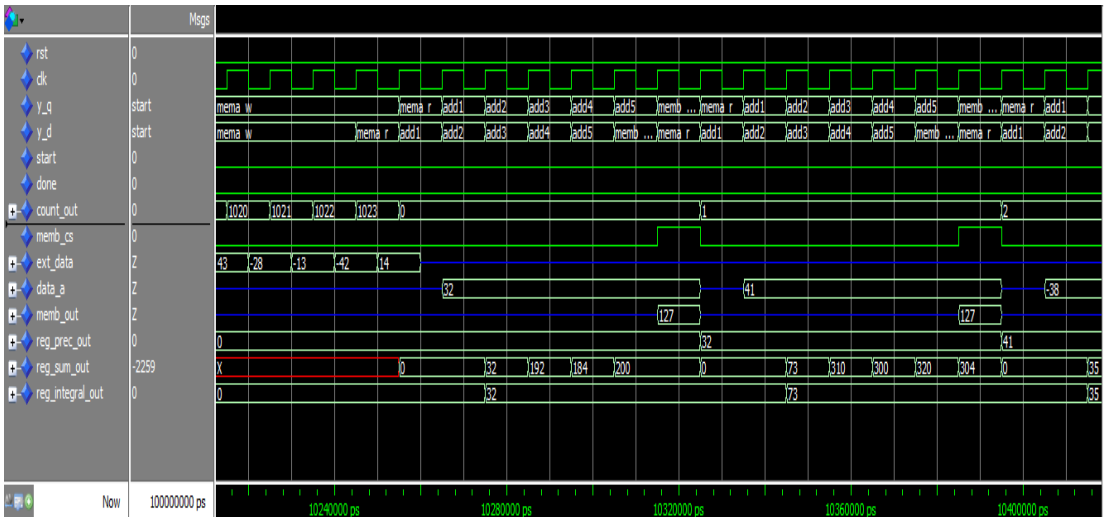


Figura 2.4: wave top entity

Capitolo 3

Appendice

3.1 Top entity

Si osservi che per la top entity si è preferito non separare la FSM e la CU per evitare un'interfaccia con una moltitudine di segnali di controllo.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity PID_controller is
7    port(
8           rst,clk,s : in std_logic;
9           ext_data : in signed(7 downto 0);
10          done_out : out std_logic;
11          memB_out : out signed(7 downto 0);
12          memB_CS_ftb: out std_logic
13       );
14 end entity;
15
16 architecture ASM of PID_controller is
17
18   -- components
19   component multiple_AND
20     generic (N: positive := 12 );
21     port (
22         bit_vect: in signed(N-1 downto 0);
23         res : out std_logic
24         );
25   end component;
26
```

```
27
28  component multiple_OR
29    generic (N: positive := 12 );
30    port (
31        bit_vect: in signed(N-1 downto 0);
32        res : out std_logic
33        );
34  end component;
35
36
37  component counter
38    generic (N : integer := 10);
39    port(
40        en, clk, clear : in std_logic;
41        Q : buffer unsigned (N-1 downto 0)
42        );
43  end component;
44
45
46  component mux3to1
47    generic (n: integer := 20);
48    port
49        (
50          a, b, c: in signed(n-1 downto 0);
51          sel    : in unsigned(1 downto 0);
52          mux_out: out signed(n-1 downto 0)
53        );
54    end component;
55
56
57    component mux5to1
58    generic (n: integer := 20);
59    port (
60           a, b, c, d, e : in signed(n-1 downto 0);
61           sel           : in unsigned(2 downto 0);
62           mux_out       : out signed(n-1 downto 0)
63         );
64    end component;
65
66
67    component memory
68    port (
69          Clk, CS, WR_RD : in std_logic;
70           ADDRESS_MEM : in unsigned(9 downto 0);
71           DATA_IN : in signed(7 downto 0);
72           DATA_OUT : out signed(7 downto 0)
73
74         );
75    end component;
76
```

```
77
78      component reg
79      generic (n : integer := 20);
80      port (
81              clk, rst,en : in std_logic;
82              R : in signed(n-1 downto 0);
83              Q : out signed(n-1 downto 0)
84          );
85
86      end component;
87
88
89      component adder
90      generic (n : integer := 20);
91      port(
92           A, B :    in signed(n-1 downto 0);
93           sub_addn : in std_logic;
94           sum :     out signed (n-1 downto 0)
95          );
96      end component;
97
98    -- states definition
99    Type state_type is (IDLE,MEMA_W,MEMA_R,ADD1,ADD2,ADD3,ADD4,
100                        ADD5,MEMB_W_NEG,MEMB_W_POS,MEMB_W,DONE);
101
102
103   signal y_Q : state_type; --present state
104   signal Y_D : state_type; -- next state
105
106   --control signals
107   signal count_en, count_rst, count_tc : std_logic;
108   signal count_out : unsigned ( 9 downto 0 );
109   signal MEMA_CS, MEMA_R_Wn : std_logic;
110   signal reg_sum_rst, reg_integral_rst,reg_prec_rst, reg_sum_LD,
111          reg_integral_LD,reg_prec_LD : std_logic;
112   signal mux1_sel : unsigned( 1 downto 0 );
113   signal mux2_sel : unsigned( 2 downto 0 );
114   signal sub_add, ovf_pos, ovf_neg,vector_or,vector_and: std_logic;
115   signal mux_memB_sel : unsigned( 1 downto 0 );
116   signal memB_CS, memB_R_Wn : std_logic;
117
118   --data signals
119   signal data1, data2, data3, data4, data5, data_A_fill : signed (19
          downto 0);
120   signal data_A : signed (7 downto 0);
121   signal reg_prec_out,reg_sum_out,reg_integral_out : signed (19 downto
          0);
122   signal mux1_out,mux2_out,adder_in, adder_out : signed (19 downto 0);
123   signal memB_in,reg_sum_out_unfilled : signed (7 downto 0);
124   signal nor_in,and_in : signed(12 downto 0);
```

```
125
126 begin
127   -----------------------------------------------
128   --------- COMPONENTS INSTANTIATION -----------
129   -----------------------------------------------
130
131   COUNT: counter port map(count_en,clk,count_rst,count_out);
132   TC : multiple_and generic map(10)
133       port map (signed(count_out), count_tc);
134   MEMA: memory port map(clk,MEMA_CS,mema_R_Wn,count_out,ext_data,data_A);
135
136   --bit filling
137   data_A_fill(19 downto 7) <= (others => data_A(7));
138   data_A_fill(6 downto 0) <= data_A(6 downto 0);
139
140   --mux addends entries
141   data1 <=data_A_fill(19) & data_A_fill(16 downto 0) & "00";
142   data2 <= data_A_fill(19) & data_A_fill(19) & data_A_fill(19) &
          data_A_fill(18 downto 2);
143   data3 <= data_A_fill(19) & data_A_fill(19) & data_A_fill(18 downto 1);
144   data4 <= reg_sum_out(19) & reg_sum_out(17 downto 0) & '0';
145   data5 <= reg_prec_out(19) & reg_prec_out(19) & reg_prec_out(18 downto
          1);
146
147   --regs inst
148   REG_PREC: reg port map ( clk, reg_prec_rst,reg_prec_LD, data_A_fill,
          reg_prec_out);
149   REG_INT: reg port map ( clk, reg_integral_rst, reg_integral_LD,
          adder_out, reg_integral_out);
150   REG_SUM: reg port map ( clk, reg_sum_rst, reg_sum_LD, adder_out,
          reg_sum_out);
151
152   --mux addends inst
153   MUX2: mux5to1 port map (
          data_A_fill,data1,data2,data3,data5,mux2_sel,mux2_out);
154   MUX1: mux3to1 port map (
          reg_sum_out,data4,reg_integral_out,mux1_sel,mux1_out);
155
156   adder_in <= mux2_out xor (mux2_out'range => sub_add);
157   ADD: adder port map ( mux1_out,adder_in,sub_add,adder_out);
158
159   --data on 8 bit
160   reg_sum_out_unfilled <= reg_sum_out(19) & reg_sum_out(6 downto 0);
161
162   --mux ovf inst
163   MUX_MEMB: mux3to1 generic map(8)
164                   port map (to_signed(-128,8), reg_sum_out_unfilled,
                        to_signed(127,8),mux_memB_sel,memB_in);
165
166   MEMB: memory port map(clk,MEMB_CS,memb_R_Wn,count_out,memB_in);
```

```
167
168   memB_out <= memB_in when y_Q = MEMB_W_NEG or y_Q = MEMB_W_POS or y_Q =
          MEMB_W else "ZZZZZZZZ";
169   memB_CS_ftb<=memB_CS;
170
171   OR_MULT: multiple_or port map (adder_out(18 downto 7),vector_or) ;
172   AND_MULT: multiple_and port map (adder_out(18 downto 7),vector_and);
173
174   --combinational ovf flags
175   ovf_neg <= adder_out(19) and not vector_and;
176   ovf_pos <= (not adder_out(19)) and vector_or;
177
178   ------------------------------------------------
179   -------------- CONTROL FSM -------------------
180   ------------------------------------------------
181
182   -- state transition process
183   STATE_TRANSITION: process (s,count_tc,ovf_pos,ovf_neg,y_Q)
184   begin
185     case y_Q is
186      when IDLE =>   if s = '0' then Y_D <= IDLE;
187                     else Y_D <= MEMA_W; end if;
188       when MEMA_W => if count_tc = '0' then Y_D <= MEMA_W;
189                      else Y_D <= MEMA_R; end if;
190       when MEMA_R => Y_D <= ADD1;
191       when ADD1 =>  Y_D <= ADD2;
192       when ADD2 =>  Y_D <= ADD3;
193       when ADD3 =>  Y_D <= ADD4;
194       when ADD4 =>  Y_D <= ADD5;
195       when ADD5 => if ovf_neg = '1' then Y_D <= MEMB_W_neg;
196                    elsif ovf_pos = '1' then Y_D <= MEMB_W_POS;
197                    else Y_D <= MEMB_W; end if;
198       when MEMB_W_neg => if count_tc = '0' then Y_D <= MEMA_R;
199                          else Y_D <= DONE; end if;
200       when MEMB_W_POS => if count_tc = '0' then Y_D <= MEMA_R;
201                          else Y_D <= DONE; end if;
202       when MEMB_W =>    if count_tc = '0' then Y_D <= MEMA_R;
203                          else Y_D <= DONE; end if;
204       when DONE =>      if s = '1' then Y_D <= DONE;
205                          else Y_D <= IDLE; end if;
206       when others => Y_D <= IDLE; --return to rst state
207     end case;
208   end process;
209
210
211   --
212   FFs: process (clk)
213   begin
214     if Rst = '1' then
215         y_Q <= IDLE;
```

```
216       elsif clk'event and clk = '1' then
217           y_Q <= Y_D;
218       end if;
219     end process;
220
221
222     -- output evaluation process
223     OUT_DEC: process (y_Q)
224     begin
225       -- default values
226       count_en <= '0';
227       count_rst <= '0';
228       memA_CS <= '0';
229       MEMA_R_Wn <= '0';
230       reg_sum_rst <= '0';
231       reg_integral_rst <= '0';
232       reg_integral_LD <= '0';
233       reg_sum_LD <= '0';
234       reg_prec_LD <= '0';
235       mux1_sel <= "00";
236       mux2_sel <= "000";
237       sub_add <= '0';
238       mux_memB_sel <= "00";
239       memB_CS <= '0';
240       memB_R_Wn <= '0';
241       done_out <= '0';
242       reg_prec_rst <= '0';
243
244
245       case y_Q is
246        when IDLE => count_rst <= '1';
247                     reg_integral_rst <= '1';
248                     reg_prec_rst <= '1';
249         when MEMA_W => memA_CS <= '1';
250                        memA_R_Wn <= '0';
251                        count_en <= '1';
252         when MEMA_R => memA_CS <= '1';
253                        memA_R_Wn <= '1';
254                        reg_sum_rst <= '1';
255
256         when ADD1 =>   memA_CS <= '1';
257                        memA_R_Wn <= '1';
258                        mux1_sel <= "10";
259                        mux2_sel <= "000";
260                        reg_sum_LD <= '1';
261                        reg_integral_LD <= '1';
262         when ADD2 =>  memA_CS <= '1';
263                       memA_R_Wn <= '1';
264                       mux1_sel <= "01";
265                       mux2_sel <= "001";
```

```
266                     reg_sum_LD <= '1';
267       when ADD3 => memA_CS <= '1';
268                    memA_R_Wn <= '1';
269                    mux1_sel <= "00";
270                    mux2_sel <= "010";
271                    reg_sum_LD <= '1';
272                    sub_add <= '1';
273       when ADD4 => memA_CS <= '1';
274                    memA_R_Wn <= '1';
275                    mux1_sel <= "00";
276                    mux2_sel <= "011";
277                    reg_sum_LD <= '1';
278       when ADD5 => memA_CS <= '1';
279                    memA_R_Wn <= '1';
280                    mux1_sel <= "00";
281                    mux2_sel <= "100";
282                    reg_sum_LD <= '1';
283                    sub_add <= '1';
284       when MEMB_W_neg => mux_memB_sel <= "00";
285                          memB_CS <= '1';
286                          memB_R_Wn <= '0';
287                          count_en <= '1';
288                          reg_prec_LD <= '1';
289
290       when MEMB_W_POS => mux_memB_sel <= "10";
291                          memB_CS <= '1';
292                          memB_R_Wn <= '0';
293                          count_en <= '1';
294                          reg_prec_LD <= '1';
295
296       when MEMB_W => mux_memB_sel <= "01";
297                      memB_CS <= '1';
298                      memB_R_Wn <= '0';
299                      count_en <= '1';
300                      reg_prec_LD <= '1';
301
302       when DONE => done_out <= '1';
303     end case;
304
305   end process;
306 end architecture;
```

Code 3.1: PID_controller.vhd

3.2 Components

```
1 library ieee;
```

```
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity memory is
6     port(
7         Clk, CS, WR_RD : in std_logic;
8          ADDRESS_MEM : in unsigned(9 downto 0);
9          DATA_IN : in signed(7 downto 0);
10         DATA_OUT : out signed(7 downto 0)
11   );
12 end entity;
13
14 architecture behaviour of memory is
15   type MEM is array(0 to 1024) of signed(7 downto 0);
16   signal memory : MEM;
17
18   begin
19     READ_WRITE : process(Clk)
20     begin
21       if Clk'event and Clk = '1' then
22         if CS = '1' then
23
24           if WR_RD = '0' then
25             memory(to_integer(ADDRESS_MEM)) <= DATA_IN;
26                  DATA_OUT<=(others=>'Z');
27           else
28             DATA_OUT <= memory(to_integer(ADDRESS_MEM));
29           end if;
30             else
31                DATA_OUT<=(others=>'Z');
32       end if;
33       end if;
34   end process;
35
36 end architecture;
```

Code 3.2: memory.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity counter is
6    generic (N : integer);
7    port(
8        en, clk, clear : in std_logic;
9        Q : buffer unsigned (N-1 downto 0)
10   );
11 end counter;
```

```
12
13 architecture behavior of counter is
14
15   begin
16     process(clear, clk) begin
17
18       if (clear='1') then
19         Q <= (others => '0');
20       elsif clk'event and clk = '1' then
21         if (en='1') then
22           Q <= Q + 1;
23         end if;
24       end if;
25
26     end process;
27
28 end architecture;
```

Code 3.3: counter.vhd

```
1
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  entity adder is
7    generic (n : integer);
8    port(
9      A, B :    in signed(n-1 downto 0);
10     sub_addn : in std_logic;
11     sum :     out signed (n-1 downto 0)
12   );
13 end entity;
14
15 architecture beh of adder is
16   begin
17     sum <= A + B + ('0'& sub_addn);
18 end beh;
```

Code 3.4: adder.vhd

```
1
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  entity mux3to1 is
7  generic (n: integer);
8  port
```

```
(
  a, b, c: in signed(n-1 downto 0);
  sel    : in unsigned(1 downto 0);
  mux_out: out signed(n-1 downto 0)
);
end mux3to1;

architecture behavior of mux3to1 is

begin
mux_out<= a when sel = "00" else
          b when sel = "01" else
          c;

end architecture;
```

Code 3.5: mux3to1.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux5to1 is
  generic (n : integer);
  port (
    a, b, c, d, e : in signed(n-1 downto 0);
    sel           : in unsigned(2 downto 0);
    mux_out       : out signed(n-1 downto 0)
  );
end mux5to1;

architecture behavior of mux5to1 is

begin
  mux_out <= a when sel = "000" else
             b when sel = "001" else
             c when sel = "010" else
             d when sel = "011" else
             e;
end behavior;
```

Code 3.6: mux5to1.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

```

```
6
7  entity reg is
8    generic (n : integer); --parallelism
9    port(
10     clk, rst,en : in std_logic;
11     R           : in signed(n-1 downto 0);
12     Q           : out signed(n-1 downto 0)
13   );
14 end reg;
15
16
17
18 architecture behavioral of reg is
19  begin
20    process (rst, clk) begin
21      if rst = '1' then
22        Q <= (others => '0');
23      elsif clk'event and clk = '1' then
24        if en = '1' then
25          Q <= R;
26        end if;
27      end if;
28    end process;
29
30 end architecture;
```

Code 3.7: register.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity multiple_AND is
7      generic (N: positive ); --array size
8      port (
9          bit_vect: in signed(N-1 downto 0);
10         res : out std_logic
11         );
12 end entity;
13
14 architecture behavior of multiple_AND is
15
16   function vectorial_and (vec : in signed) return std_logic is
17   variable res : std_logic := '1'; -- 1 and X = X
18   begin
19     for i in vec'range loop
20       res := res and vec(i);
21     end loop;
```

```
22      return res;
23    end function;
24
25  begin
26    res <= vectorial_and(bit_vect);
27  end architecture;
```

Code 3.8: multiple_AND.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5
6   entity multiple_OR IS
7       generic (N: positive); --array size
8       port (
9           bit_vect: in signed(N-1 downto 0);
10          res : out std_logic
11          );
12  end entity;
13
14  architecture behavior of multiple_OR is
15    function vectorial_or (vec : in signed) return std_logic is
16    variable res : std_logic := '0'; -- 0 or X = X
17    begin
18      for i in vec'range loop
19        res := res or vec(i);
20      end loop;
21      return res;
22    end function;
23
24  begin
25    res <= vectorial_or(bit_vect);
26
27  end architecture;
```

Code 3.9: multiple_OR.vhd