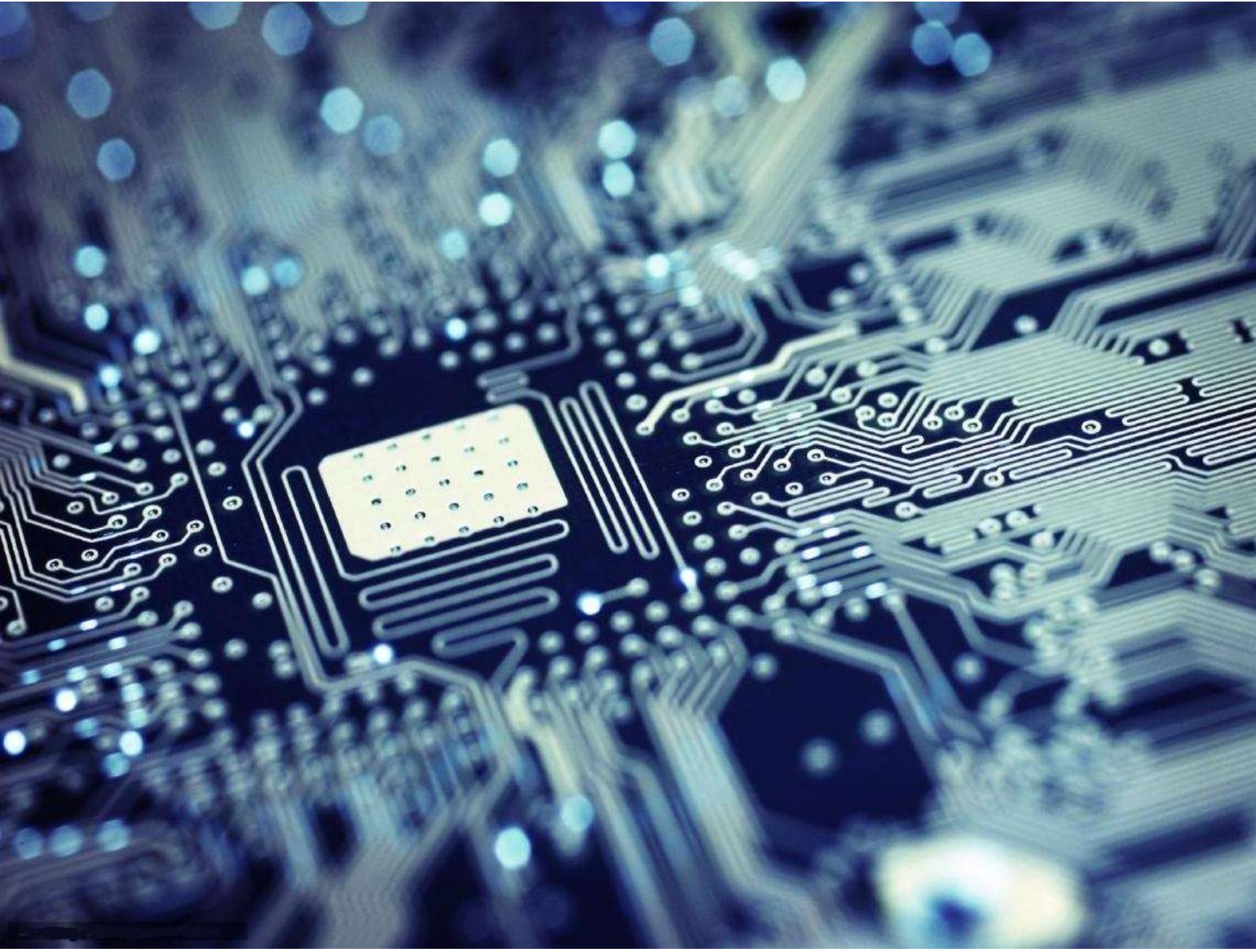


Prova Finale di Reti Logiche

15/05/2020

Autore : Andrea Sestito

Matricola: 869695

Codice persona: 10559959

POLITECNICO
MILANO 1863

Introduzione

Obiettivo della Prova Finale (Progetto di Reti Logiche) consiste nella descrizione di un componente logico in VHDL, il cui comportamento simuli una soluzione pensata per il trasferimento degli indirizzi all'interno dei microprocessori, con lo scopo di minimizzare le attività dei bus interni ed avere quindi una riduzione della dissipazione di potenza.

La simulazione del funzionamento prevede che dato un indirizzo (ADDR) da 7 bit, letto dall'indirizzo 8 della RAM, esso venga trasmesso attraverso due tipologie di codifica diverse, dipendentemente dal fatto che appartenga oppure no ad una Working Zone.

Le Working Zone (WZ) sono in totale 8, WZ_NUM sarà quindi espresso su 3 bit (000-111), e l'indirizzo base di ciascuna di esse è situato in memoria RAM rispettivamente nei primi 8 indirizzi. Ogni Working Zone è composta da 4 indirizzi consecutivi da 7 bit e per identificare un indirizzo, si utilizza il suo indirizzo base (WZ_BASE) assieme all'offset (WZ_OFFSET), espresso in notazione *one-hot* su 4 bit, permettendo la rappresentazione dei valori da 0 a 3.

Risultato finale del processo, sarà la scrittura del risultato della codifica attraverso l'interfaccia d'uscita `o_data: out std_logic_vector (7 downto 0)`, codifica Little-endian, nella cella `RAM(9)`, seguendo la politica sotto riportata:

- se ADDR appartiene ad una Working Zone,
  RAM(9) conterrà l'indirizzo così codificato:
  - Bit 7 : valore del bit WZ_BIT=1;
  - Bit 6-4: valore codificato binario di WZ_NUM;
  - Bit 3-0: valore codificato one-hot di WZ_OFFSET;
- se ADDR non appartiene ad alcuna Working Zone,
  RAM(9) conterrà la seguente sequenza di bit:
  - Bit 7 : WZ_BIT=0;
  - Bit 6-0: valore binario di ADDR;

1 - Architettura

In questo capitolo analizzeremo l'Architettura del progetto, iniziando con una presentazione dell'interfaccia del componente utilizzato, per poi descriverne il processo e infine la logica di funzionamento.

1.1 Interfaccia

Le specifiche del progetto vengono interamente svolte da un unico componente, `project_reti_logiche`, in quanto non si è ritenuto necessario aggiungerne ulteriori a supporto dell'algoritmo, sebbene tale scelta non permetta ulteriore astrazione e modularità dell'Architettura.

Il componente si occupa dunque di leggere i segnali in input e gestire quelli di ouput tramite l'interfaccia riportata in basso:

```
entity project_reti_logiche is
port (
      i_clk : in std_logic;
      i_start : in std_logic;
      i_rst : in std_logic;
      i_data : in std_logic_vector(7 downto 0);
      o_address : out std_logic_vector(15 downto 0);
      o_done : out std_logic;
      o_en : out std_logic;
      o_we : out std_logic;
      o_data : out std_logic_vector(7 downto 0)
);
end project_reti_logiche;
```

in particolare, sono riportati in basso i segnali di IO:

- **`i_clk`** riceve dal TestBench il segnale di `CLOCK`, con periodo impostabile; nello specifico il processo `p_CLK_GEN` si occupa di fornire la temporizzazione.
- **`i_start`** è il segnale di `START`, che quando assume valore `1`, indica l'inizio della codifica dell'`addr`.
- **`i_rst`** è il segnale di `RESET`, che quando assume valore `1`, permette di inizializzare i segnali per garantire l'esecuzione corrente e preparare quella successiva.
- **`i_data`** contiene il valore del registro espresso in 8 bit proveniente dalla `RAM`.
- **`o_address`** contiene l'indirizzo del registro della `RAM` dove si intende leggere o scrivere.

- **o_done** è il segnale di DONE, che quando assume valore 1, indica la fine della sequenza di comandi principale.

- **o_en** è il segnale di ENABLE, necessario all'attivazione del processo MEM del TestBench, al fine di comunicare alla RAM l'intenzione di accesso alla memoria.

- **o_we** è il segnale di WRITE ENABLE, inviato al processo MEM del TestBench, al fine di comunicare alla RAM l'intenzione di scrittura in memoria (0 per leggere, 1 per scrivere).

- **o_data** contiene il valore elaborato espresso in 8 bit da scrivere nella RAM all'indirizzo specificato da o_address.

In sintesi, i segnali possono essere categorizzati in tre tipologie: la prima (i_clk) è quella che realizza la temporizzazione, la seconda (i_rst, i_start, o_done) si occupa della gestione del ciclo vita dell'esecuzione del progetto ed infine una terza categoria (o_en, o_address, o_we, i_data, o_data) viene impiegata dal processo MEM per permettere il processo di lettura e scrittura dei dati nella RAM.

1.2 Processo

Il processo viene utilizzato per gestire i segnali in entrata e in uscita dal componente, impiegando la discretizzazione temporale del clock (lista di sensibilità), parallelamente alle condizioni che scandiscono gli eventi.

Esso si occupa di gestire opportunamente i segnali in entrata dal processo test (i_rst, i_start) del Test Bench e produrre in uscita il segnale o_done in modo da portare a termine correttamente la specifica e preparare una nuova esecuzione.

La prima condizione **[1]** permette al componente di resettare i valori non appena viene inviato il segnale i_rst, e con l'aggiunta del segnale alla sensitivity list, si rende il processo di RESET istantaneo, asincrono e indipendente dalle altre parti.

La seconda condizione **[2]** permette di temporizzare i segnali in maniera tale che quando i_clock commuta da 1 a 0, si inneschino le operazioni su di essi, mentre nel restante periodo, vengono effettivamente modificati e messi a disposizione per il ciclo di clock successivo.

Questo scandire ed effettuare operazioni diverse ad ogni ciclo, combacia perfettamente con l'idea di una macchina a stati finiti, motivo per il quale si è scelto di utilizzare questa tipologia di struttura per rappresentare la logica del componente e della quale verrà descritto il suo funzionamento nel dettaglio nei prossimi paragrafi.

Successivamente se il segnale `i_start` viene alzato ad `1`, si entra nella condizione **[3]**, e l'esecuzione della codifica può finalmente avere inizio.

Nel codice sottostante si evidenziano le parti principali che caratterizzano il ciclo vita del processo:

```
processo: process(i_clk, i_rst)

...
begin
     if i_rst = '1' then    [1]
          curr_state <= rst;

     else if (i_clk'event and i_clk='0') then   [2]

               case curr_state is

                    when rst =>
                         ...
                         if i_start = '1' then   [3]
                              ...
                              curr_state <= read_input_addr;

                         end if;

                    ...

                    when final_state =>
                         if i_start = '0' then   [4]
                              o_done <= '0';
                              curr_state <= rst;
                         else   [5]
                              o_done <= '1';
                         end if;

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

La terminazione della codifica e la predisposizione dei segnali ad una nuova è gestita dalle condizioni **[5]** e **[4]**, nelle quali si controlla che `i_start` si trovi ad `1`, per poter segnalare la terminazione della codifica impostando `o_done` ad `1`. Successivamente, dal processo `test` del TestBench viene inviato `i_start = 0` che grazie alla **[4]** porta il segnale `DONE` a `0` per poi tornare allo stato di `RESET` nell'attesa di un nuovo segnale di `START` e quindi di una nuova codifica.

1.3 Segnali interni e variabili

In questo paragrafo vengono descritti i segnali interni al componente utilizzati a supporto dello sviluppo del codice e dell'architettura in generale, per permettere di memorizzare segnali temporanei e non interferire con i segnali di interfaccia, ma fornire ad essi i risultati solo ad elaborazione completata.

- **`addr`** segnale nel quale viene memorizzato l'indirizzo da codificare, letto della cella 8 della `RAM`.
- **`wz_bit`** segnale costituito da un solo bit utilizzato nella codifica finale per indicare l'appartenenza (`1`) o non appartenenza (`0`) ad una specifica Working Zone.
- **`wz_base`** segnale utilizzato per memorizzare l'indirizzo base delle Working Zone durante l'esecuzione.
- **`wz_num`** segnale utilizzato per memorizzare il numero della Working Zone specifica nel caso in cui l'`addr` vi appartenga.
- **`wz_offset_binary`** segnale utilizzato per salvare la distanza in binario di un indirizzo da una Working Zone relativa.
- **`wz_offset`** segnale utilizzato per salvare l'offset in notazione *one-hot*.
- **`i`** variabile utilizzata come contatore incrementale per il ciclo della FSM.
- **`curr_state`** segnale utilizzato per indicare lo stato corrente della FSM; può valere uno dei 9 stati definiti dal tipo `type_states`.
- **`type_states`** tipo contentente tutti gli stati della FSM che `curr_state` potrà assumere nel corso dell'esecuzione della specifica; in particolare, la definizione risulta

```
type type_states is (rst, read_input_addr, find_wz_base,
get_wz_base, scan_wz, not_in_wz, in_wz, set_output,
final_state);
```

1.4 Logica e stati della FSM

Nel seguente paragrafo viene descritto il funzionamento della macchina a stati finiti che governa la logica del progetto; ogni cambiamento di stato è dato dall'assegnamento di uno specifico stato al segnale `curr_state`, così che al ciclo di clock successivo si possa entrare nello stato assegnato al segnale.

L'implementazione della FSM in VHDL è stata realizzata con uno *switch-case*, dove ogni *case* corrisponde ad uno stato di `type_states`.

La codifica comincia dallo stato `rst`, il cui scopo è quello di inizializzare i segnali interni e di interfaccia, onde evitare valori di partenza inaspettati; nel momento in cui viene inviato il segnale di `START` la codifica può avere inizio.

Si passa quindi allo stato `read_input_addr`: in questo stato si legge all'indirizzo 8 della `RAM` il valore dell'indirizzo da codificare e lo si salva nel segnale interno `addr`.

Successivamente, si entra nel ciclo principale composto dagli stati `find_wz_base`, `get_wz_base`, `scan_wz`, il cui scopo è scorrere tutte le Working Zone arrivando ad un punto in cui si prenderà una delle due diramazioni, dipendentemente dall'appartenenza dell'indirizzo alla WZ, `not_in_wz` oppure `in_wz`.

- La funzione di `find_wz_base` è quella di impostare in `o_address` l'indirizzo della i-esima Working Zone che si vuole andare a controllare.

- La funzione di `get_wz_base`, è quella di leggere dalla memoria l'indirizzo base della i-esima WZ e garantire la permanenza all'interno del ciclo **[1]** a patto che non siano finite le Working Zone da controllare (condizione `else`), e in quest'ultimo caso si entra nello stato `not_in_wz`. **[2]**.

```
if i < 8 then                    // i inizializzata a 0 nello stato rst e
                                 // incrementata in scan_wz
     wz_base <= i_data (6 downto 0);
       curr_state <= scan_wz; [1]

else
     ...

     curr_state <= not_in_wz;   [2]

end if;
```

- Nel caso `[1]` si procede con l'esecuzione dello stato `scan_wz`, nel quale si controlla l'appartenenza di `addr` alla WZ numero `i` `[3]`, il cui indirizzo base è stato salvato nel segnale `wz_base` nello stato precedente.

In caso affermativo si entra nello stato `in_wz`, **[4]** in caso negativo si torna allo stato `find_wz_base` collegamento che identifica l'esistenza di un loop all'interno della FSM, e si incrementa la i per controllare la `i+1` – esima WZ.

```
if   addr >= wz_base and

     std_logic_vector(unsigned(addr)-unsigned(wz_base)) < "0000100"
     [3]

then

     ...

     curr_state <= in_wz; [4]

else
     i := i+1;
     curr_state <= find_wz_base;

end if;
```

In altre parole, se la variabile `i` raggiunge il valore 8 `[2]` significa che le WZ da scorrere sono finite e la condizione di appartenenza ad una WZ `[3]` non si è mai verificata.

In questo caso si procede quindi con la codifica come da specifica: viene scritto nella cella 9 della `RAM`, attraverso l'interfaccia di uscita del componente `o_data`, la concatenazione di `wz_bit & addr`, dove `wz_bit = 0`.

Nel caso contrario invece viene fornito a `o_data` la concatenazione di `wz_bit = 1`, `wz_num` (ottenuto convertendo in binario la variabile `i`) e `wz_offset`.

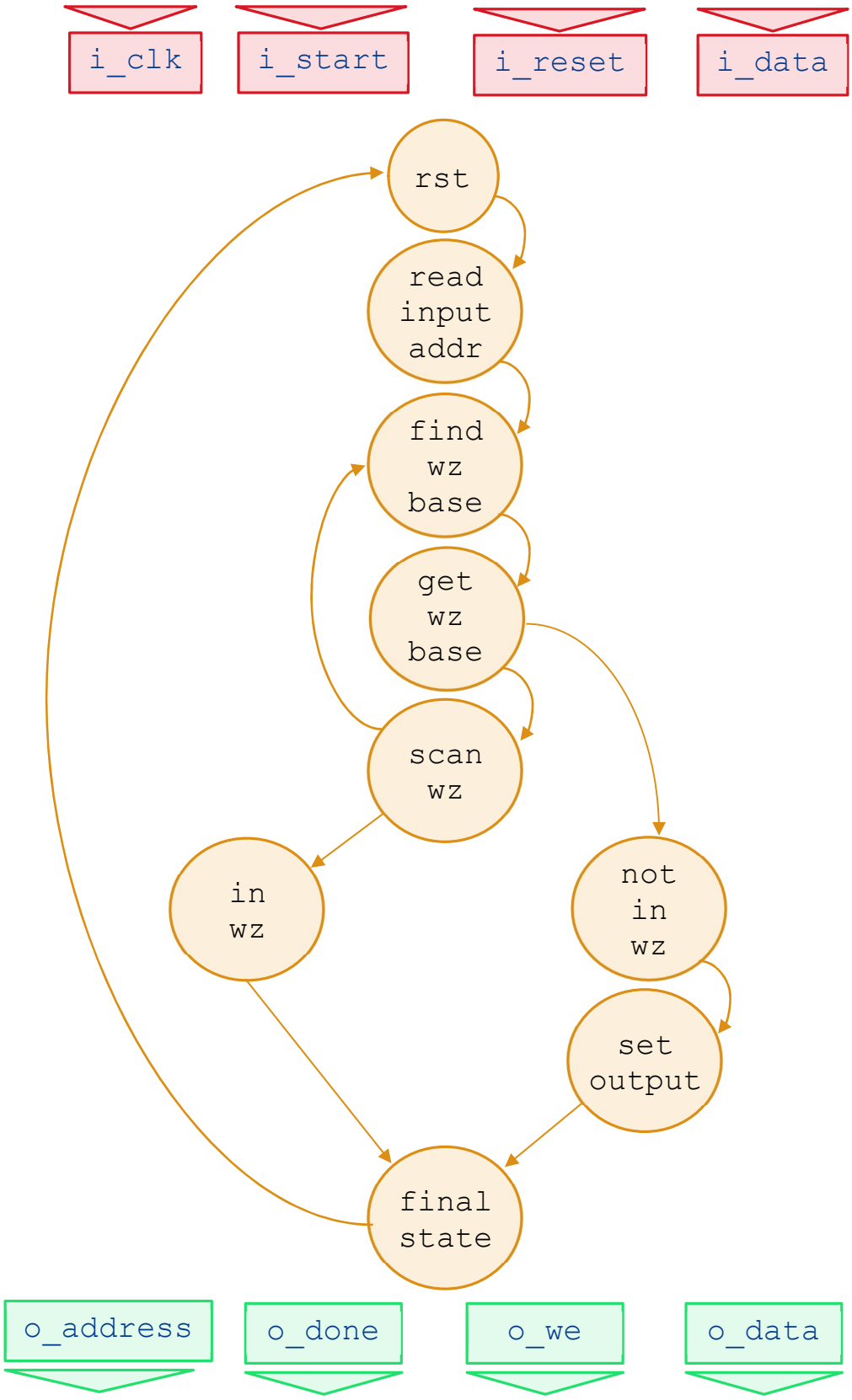
Quest'ultimo segnale viene ottenuto eseguendo lo shift verso sinistra della sequenza `0001` di un numero di volte pari a `wz_offset_binary`, dove `wz_offset_binary` è la distanza espressa su 3 bit dell'`addr` dalla WZ, ottenuto dalla sottrazione binaria di questi due segnali.

Ad esempio, se wz_offset_binary fosse `011`, si deve effettuare uno shift a sinistra per 3 volte dei bit `0001` ottenendo quindi `1000` (notazione *one-hot* del numero 3).

Poiché nello stato `in_wz` viene effettuata questa operazione, il valore di `wz_offset` non sarebbe pronto per la codifica finale prima del ciclo di clock successivo, quindi e si è utilizzato uno stato aggiuntivo `set_output` nel quale si concatena `wz_bit`, `wz_num` e `wz_offset` e si assegna la sequenza di bit così ottenuta a `o_data`.

Infine, dagli stati `not_in_wz` e `set_ouput` si giunge nello stato `final_state` dove si termina la codifica inviando il segnale di `o_done` impostato ad `1` al TestBench e si lascia la gestione della terminazione della codifica e la predisposizione ad una nuova al `processo` e a quest'ultimo stato della FSM.

Viene riportato sotto lo schema ad alto livello del funzionamento della FSM.



➢ segnali di input ➢ segnali di output ● stati FSM

2 – Test Bench

Viene ora descritta la struttura del TestBench, analiazzando alcune delle sue parti fondamentali.
Grazie alla seguente interfaccia si mappa la comunicazione fra il componente e il TestBench, in modo da far corrispondere i segnali in entrata/uscita e permettere il passaggio di informazioni da un modulo all'altro.

```
UUT: project_reti_logiche
port map (
          i_clk     => tb_clk,
          i_start   => tb_start,
          i_rst     => tb_rst,
          i_data    => mem_o_data,
          o_address => mem_address,
          o_done    => tb_done,
          o_en      => enable_wire,
          o_we      => mem_we,
          o_data    => mem_i_data
        );
```

Per scelta progettuale, la `RAM` viene fornita dal Test Bench e vengono inizializzati i primi 8 indirizzi con gli indirizzi base delle Working Zone:

```
type ram_type is array (65535 downto 0)
                 of std_logic_vector (7 downto 0);
signal RAM: ram_type := (
     0 => std_logic_vector(to_unsigned(4 ,8)),
     1 => std_logic_vector(to_unsigned(13,8)),
     :
     8 => std_logic_vector(to_unsigned(ADDR,8)),
     others => (others =>'0'))
```

La lettura e la scrittura su di essa avviene grazie al seguente processo:

```
MEM : process(tb_clk)
begin
    if tb_clk'event and tb_clk = '1' then
        if enable_wire = '1' then
           if mem_we = '1' then
              RAM(conv_integer(mem_address) <= mem_i_data;
              mem_o_data <=  mem_i_data after 1 ns;
           else
               mem_o_data <= RAM(conv_integer(mem_address))
               after 1 ns;
            end if;
        end if;
    end if;
end process;
```

Esso si attiva nel momento in cui il segnale di clock assume valore alto; se l'operazione da effettuare è una scrittura il segnale `enable_wire` dovrà essere impostato a `1`, mentre se è una lettura a `0`.
Tutti i processi sono sincronizzati con il segnale di clock che nel Test Bench è gestito dal processo `p_CLK_GEN` riportato in basso, dove `c_CLOCK_PERIOD` è impostato a 100ns.

```
p_CLK_GEN : process is
   begin
        wait for c_CLOCK_PERIOD/2;
        tb_clk <= not tb_clk;
   end process p_CLK_GEN;
```

Infine, vi è un ultimo processo che si occupa di inviare i segnali di `RST`, `START` e aspettare fino a quando il segnale di `DONE` non viene portato a `1`, per poi poter avviare un'altra esecuzione con nuovi valori.

```
test: process
     begin
          wait for c_CLOCK_PERIOD;
          tb_rst <= '1';
          wait for c_CLOCK_PERIOD;
          tb_rst <= '0';
          wait for c_CLOCK_PERIOD;
          tb_start <= '1';
          wait for c_CLOCK_PERIOD;
          wait until tb_done = '1';
          wait for c_CLOCK_PERIOD;
          tb_start <= '0';
          wait until tb_done = '0';
```

Da notare come questo processo potrebbe essere modificato in base al tipo di esecuzione che si vuole ottenere, e in ogni caso il modulo dovrebbe processare i valori in maniera opportuna. Ad esempio, si potrebbe volere eseguire due codifiche all'interno della stessa esecuzione, quindi occorerebbe modificare il processo in maniera tale da fornire i segnali opportunamente.

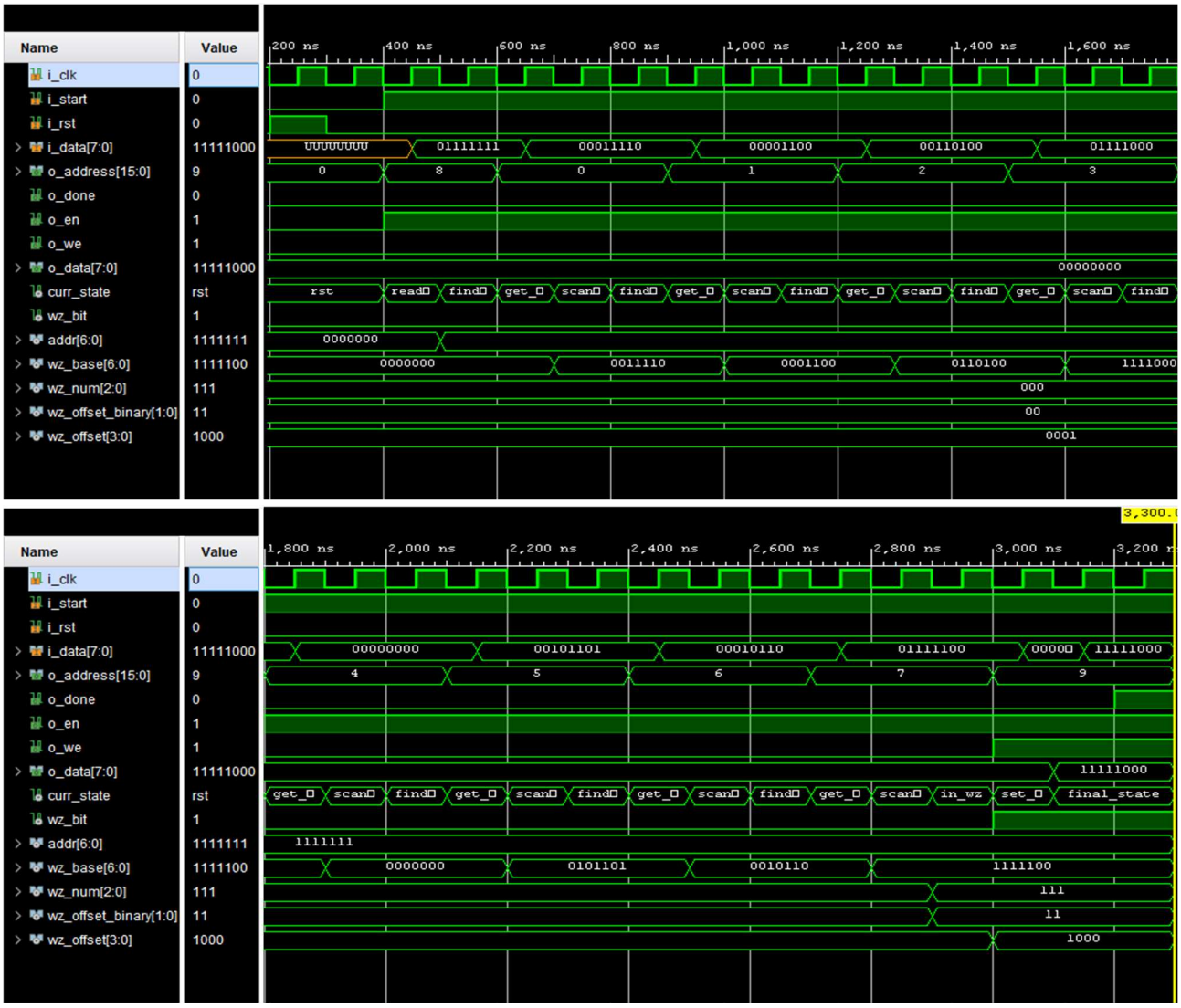
In generale, quindi il processo `test` si occupa di impartire il ciclo-vita che il modulo dovrà seguire nel momento in cui viene fatta partire la simulazione.

3 – Risultati Sperimentali

In questo capitolo verranno commentati i risultati ottenuti dalle esecuzioni delle simulazioni *Behavioural* e *Post-Functional Synthesis*.

3.1 TestBench – Caso limite massimo

Obiettivo di questo TestBench è quello di fornire i valori in ingresso in modo da avere il caso massimo da codificare: l'indirizzo da cercare è il massimo previsto su 7 bit e si trova nell'ultima Working Zone avente indirizzo base 124, che è il massimo previsto, poiché gli indirizzi della WZ in questione partono da 124 ed arrivano fino a 127.



Vengono rappresentati di seguito primi 9 indirizzi della `RAM`, i segnali interni e di interfaccia che entrano in gioco nel corso dell'esecuzione; in particolare, nella tabella si leggono gli indirizzi base delle WZ (`0-7`), l'`addr` (`8`) e infine il valore codificato (`9`).

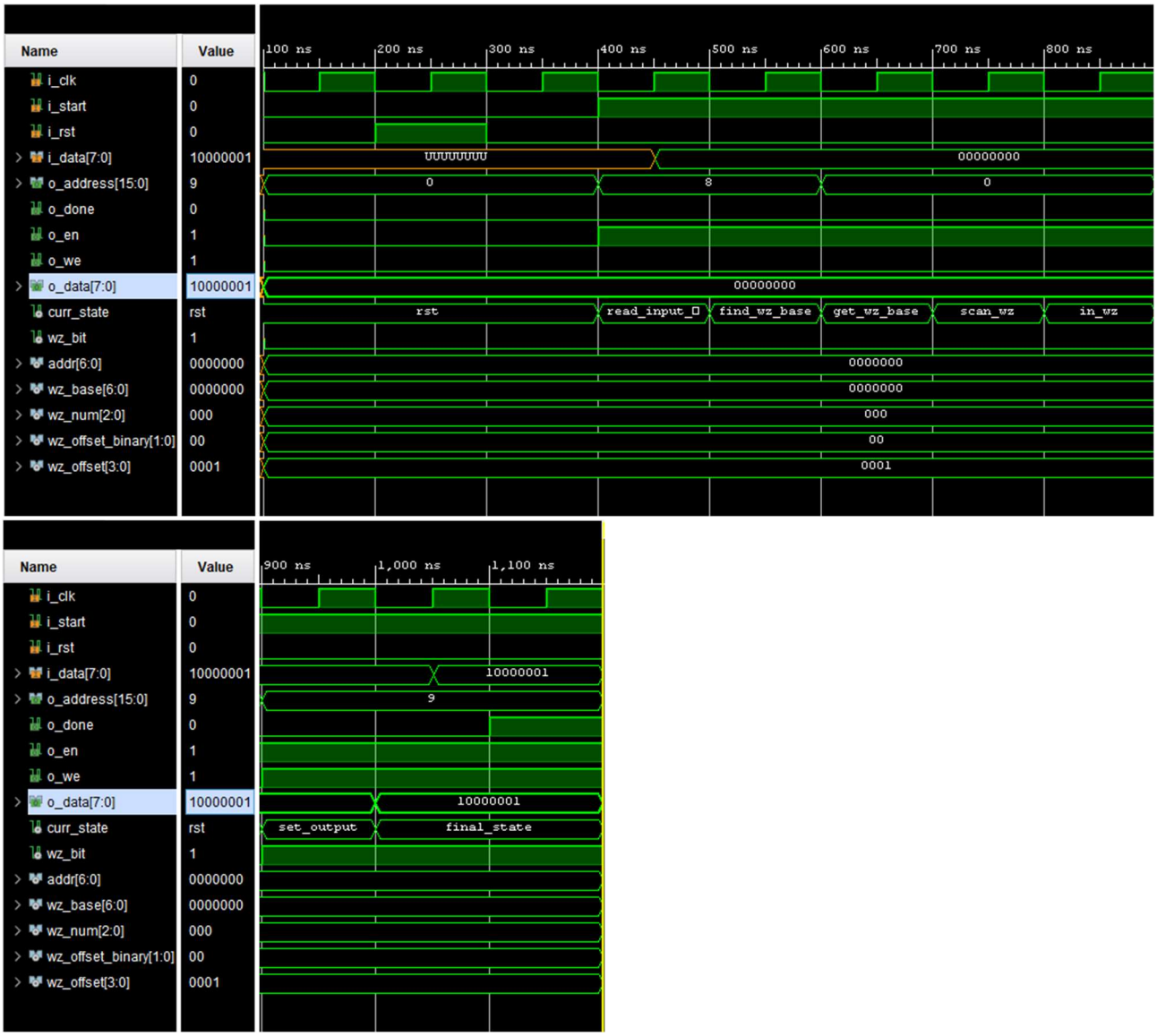
A fianco della tabella si evidenziano i valori dei segnali interni nel `final_state`.

RAM	value
0	30
1	12
2	52
3	120
4	0
5	45
6	22
7	124
8	127
9	248

```
addr:                1111111
wz_bit:              1
wz_base:             1111100
wz_num:              111
wz_offset_binary:    11
wz_offset:           1000
codifica finale:     1 111 1000
```

3.2 – TestBench – Caso limite minimo

Obiettivo di questo TestBench è quello di fornire in ingresso l'indirizzo da codificare appartenente al primo indirizzo della Working Zone 0 (caso limite minimo).



RAM	value
0	0
1	64
2	70
3	10
4	15
5	99
6	4
7	122
8	0
9	129

```
addr:                0000000
wz_bit:              1
wz_base:             0000000
wz_num:              000
wz_offset_binary:    00
wz_offset:           0001
codifica finale:     1 000 0001
```

Viene qui di seguito riportato il log di esecuzione della *Behavioural Simulation* con esito positivo

```
launch_simulation
INFO: [Vivado 12-5682] Launching behavioral simulation in 'D:/RetiLogiche/10559959.sim/sim_1/behav/xsim'
INFO: [SIM-utils-51] Simulation object is 'sim_1'
INFO: [SIM-utils-54] Inspecting design source files for 'project_tb' in fileset 'sim_1'...
INFO: [USF-XSim-97] Finding global include files...
INFO: [USF-XSim-98] Fetching design files from 'sim_1'...
INFO: [USF-XSim-2] XSim::Compile design
INFO: [USF-XSim-61] Executing 'COMPILE and ANALYZE' step in 'D:/RetiLogiche/10559959.sim/sim_1/behav/xsim'
"xvhdl --relax -prj project_tb_vhdl.prj"
INFO: [VRFC 10-163] Analyzing VHDL file "D:/RetiLogiche/10559959.vhd" into library xil_defaultlib
INFO: [VRFC 10-3107] analyzing entity 'project_reti_logiche'
INFO: [VRFC 10-163] Analyzing VHDL file "D:/RetiLogiche/TestBench Prof/tb_pfrl_2020.vhd" into library xil_defaultlib
INFO: [VRFC 10-3107] analyzing entity 'project_tb'
INFO: [USF-XSim-69] 'compile' step finished in '1' seconds
INFO: [USF-XSim-3] XSim::Elaborate design
INFO: [USF-XSim-61] Executing 'ELABORATE' step in 'D:/RetiLogiche/10559959.sim/sim_1/behav/xsim'
"xelab -wto 605a88c343664b378a55d9271a01a903 --debug typical --relax --mt 2 -L xil_defaultlib -L secureip --snapshot
Vivado Simulator 2019.1
Copyright 1986-1999, 2001-2019 Xilinx, Inc. All Rights Reserved.

# run 15000ns
Failure: Simulation Ended!, TEST PASSATO
Time: 1200 ns  Iteration: 2  Process: /project_tb/test  File: D:/RetiLogiche/TestBench Prof/tb_pfrl_2020.vhd
$finish called at time : 1200 ns : File "D:/RetiLogiche/TestBench Prof/tb_pfrl_2020.vhd" Line 103
INFO: [USF-XSim-96] XSim completed. Design snapshot 'project_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 15000ns
launch_simulation: Time (s): cpu = 00:00:05 ; elapsed = 00:00:13 . Memory (MB): peak = 788.492 ; gain = 0.000
```

L'esecuzione non porta a nessun tipo di errore o warning, e viene eseguita nel tempo descritto nell'ultima riga del log.

3.3 – Sintesi

Il componente supera correttamente le simulazioni *Post-Synthesis Functional* insieme alla *Post-Synthesis Timing*. Di seguito vengono riportati i log della prima:

```
simulation -mode post-synthesis -type functional
Vivado 12-5682] Launching post-synthesis functional simulation in 'D:/RetiLogiche/10559959.sim/sim_1/synth/func/xsim'
SIM-utils-51] Simulation object is 'sim_1'
SIM-utils-24] Writing simulation netlist file for design 'synth_1'...
SIM-utils-26] write_vhdl -mode funcsim -nolib -force -file "D:/RetiLogiche/10559959.sim/sim_1/synth/func/xsim/project_tb_func_synth.vhd"
SIM-utils-36] Netlist generated:D:/RetiLogiche/10559959.sim/sim_1/synth/func/xsim/project_tb_func_synth.vhd
SIM-utils-54] Inspecting design source files for 'project_tb' in fileset 'sim_1'...
USF-XSim-2] XSim::Compile design
USF-XSim-61] Executing 'COMPILE and ANALYZE' step in 'D:/RetiLogiche/10559959.sim/sim_1/synth/func/xsim'
--relax -prj project_tb_vhdl.prj"
VRFC 10-163] Analyzing VHDL file "D:/RetiLogiche/10559959.sim/sim_1/synth/func/xsim/project_tb_func_synth.vhd" into library xil_defaultlib
VRFC 10-3107] analyzing entity 'project_reti_logiche'
VRFC 10-163] Analyzing VHDL file "D:/RetiLogiche/TestBench Prof/tb_pfrl_2020_no_wz.vhd" into library xil_defaultlib
VRFC 10-3107] analyzing entity 'project_tb'
USF-XSim-69] 'compile' step finished in '2' seconds
USF-XSim-3] XSim::Elaborate design
USF-XSim-61] Executing 'ELABORATE' step in 'D:/RetiLogiche/10559959.sim/sim_1/synth/func/xsim'
-wto 605a88c343664b378a55d9271a01a903 --debug typical --relax --mt 2 -L xil_defaultlib -L secureip --snapshot project_tb_func_synth xil_def
Simulator 2019.1
ht 1986-1999, 2001-2019 Xilinx, Inc. All Rights Reserved.

# run 15000ns
Failure: Simulation Ended!, TEST PASSATO
Time: 1300100 ps  Iteration: 2  Process: /project_tb/test  File: D:/RetiLogiche/TestBench Prof/tb_pfrl_2020_no_wz.vhd
$finish called at time : 1300100 ps : File "D:/RetiLogiche/TestBench Prof/tb_pfrl_2020_no_wz.vhd" Line 103
INFO: [USF-XSim-96] XSim completed. Design snapshot 'project_tb_func_synth' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 15000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:16 . Memory (MB): peak = 2510.445 ; gain = 0.762
```

Nel report di sintesi non sono presenti né *errors*, né *critical warnings*. Gli unici *warnings* che vengono riportati riguardano il segnale `i_data`, del quale si utilizzano solo 7 bit su 8, in quanto i segnali `addr` e `wz_base` non richiedono bit ulteriori per essere rappresentati; per verifica empirica, non risulta comunque essere un rilevante problema per l'esecuzione della specifica.

```
Synthesis finished with 0 errors, 0 critical warnings and 2 warnings.
Synthesis Optimization Runtime : Time (s): cpu = 00:00:14 ; elapsed = 00:00:14 . Memory (MB): peak = 727.109 ; gain = 428.438
Synthesis Optimization Complete : Time (s): cpu = 00:00:14 ; elapsed = 00:00:14 . Memory (MB): peak = 727.109 ; gain = 428.438
```

Infatti, al fine di eliminare questi *warning,* si è provato ad utilizzare 8 bit per `addr` ed altrettanti per `wz_base`, e, nonostante il bit aggiuntivo non fosse strettamente necessario, si è riscontrato aumento non significativo dei `LUT` e dei `FF`, quindi si è preferito evitare di implementare la modifica.

Viene di sotto riportato il `synth_report_utilization` generato da *Vivado*, dal quale si ricava il numero di componenti elementari usati per la sintesi.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	69	0	134600	0.05
LUT as Logic	69	0	134600	0.05
LUT as Memory	0	0	46200	0.00
Slice Registers	87	0	269200	0.03
Register as Flip Flop	87	0	269200	0.03
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Come si può notare dalla tabella, viene utilizzata una parte molto piccola dell'`FPGA`, solo il 0.05% di `LUT` ed il 0.03% di `FF`; i risultati citati si considerano accettabili e non si è ritenuto necessario ridurre il numero `FF` e `LUT`; questa scelta avrebbe portato a rendere il codice meno semplice e lineare, con la perdita della divisione netta degli stati, che invece caratterizza il progetto in questione.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!								69	87	0.0	0	0
✓ impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	8.359	0	69	87	0.0	0	0

Conclusioni

Con questo progetto si è voluta dare una possibile soluzione alla specifica richiesta, concentrandosi sull'utilizzo di un'architettura compatta, composta da un singolo processo e allo stesso tempo da una logica estesa su più stati, in modo da separare le operazioni sintatticamente e semanticamente, offrendo una visione più semplice, e allo stesso tempo funzionale del componente. In definitiva, il componente passa correttamente i test *Behavioural Simulation*, *Post-Synthesis Functional Simulation* con i risultati descritti nei paragrafi precedenti, mantendosi essenziale nella struttura.