Progetto Finale Reti Logiche

Politecnico di Milano

Anno accademico 2019/20

Scuola di Ingegneria Industriale e dell'Informazione

"Working Zone"

Codifica a bassa dissipazione di potenza

Docente: Fabio Salice
Consegna: 1 Aprile

Nome	Cognome	Cod. Persona	Matricola
Davide	Malmusi	10613338	891752
Giorgio	Lorini	10565790	889893

POLITECNICO
MILANO 1863

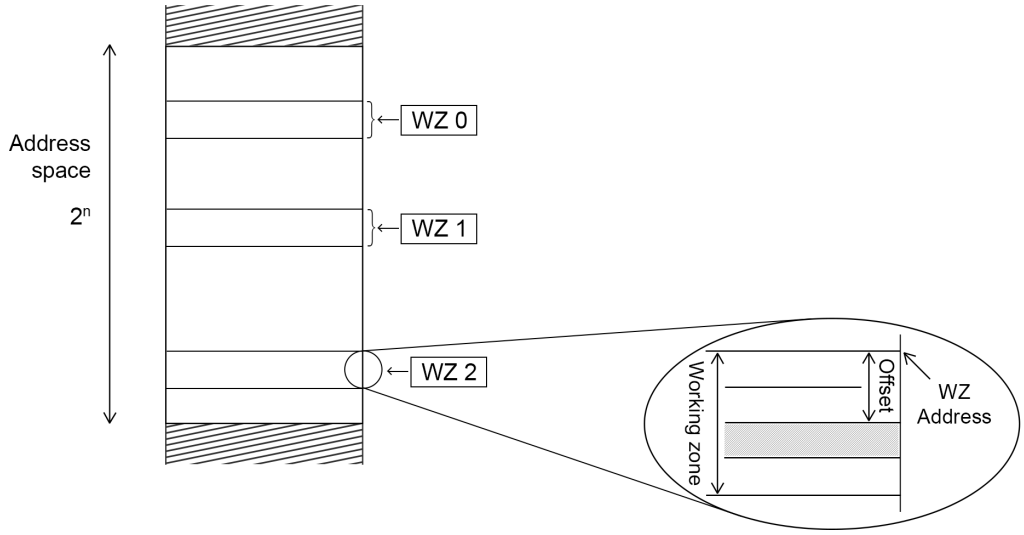
Indice

1. Introduzione 3
   1.1 Working Zone Encoding 3
   1.2 Specifica 4
2. Architettura 6
   2.1 Scelte architetturali 6
   2.2 Descrizione degli Stati 7
3. Risultati Sperimentali 8
   3.1 Record di Sintesi 8
   3.2 Schematic 9
4. Simulazioni 10
5. Conclusioni 11

Introduzione

Working Zone Encoding

La tecnica denominata “**Working Zone Encoding**” (WZE) è un metodo per ridurre la dissipazione di potenza dovuta alla trasmissione di indirizzi sui pin dell’address bus di un microprocessore.



Tale metodo si basa sull’ipotesi che all’interno di un range di indirizzi assegnato ad un programma, esso utilizzi maggiormente alcune zone di memoria rispetto ad altre.

Gli indirizzi appartenenti a tali zone, dette appunto Working Zones, vengono quindi identificati in maniera più energeticamente efficiente tramite un opportuno uso della codifica one-hot che ne indica l’offset rispetto all’indirizzo base.

Questo approccio permette di ridurre radicalmente il numero di transizioni sulle linee esterne al chip, caratterizzate da un’alta capacità, e quindi responsabili di una parte sostanziale del consumo di energia del processore.

La codifica one-hot, infatti, fa si che la variazione di un indirizzo porti alla modifica di due soli bit, mentre in codifica binaria una variazione costringe in media ad invertire metà dei bit totali della linea.

Specifica

La specifica del progetto richiede di descrivere in VHDL un componente in grado di simulare il funzionamento di un encoder atto ad effettuare una traduzione di indirizzi secondo il principio sopra descritto.

Più precisamente, dato un indirizzo da trasmettere (*addr*), la codifica viene effettuata nel modo seguente.

Se l'indirizzo appartiene ad una WZ la sua traduzione sarà composta da tre parti:

1. Un bit con valore '1' in testa, posto ad indicare l'appartenenza ad una WZ.
2. Il numero della WZ alla quale esso appartiene, codificato in binario naturale.
3. L'offset rispetto all'indirizzo base della WZ, codificato in one-hot.

Se invece l'indirizzo non appartiene a nessuna WZ, ad esso viene aggiunto un bit con valore 0 in testa.

Per il nostro progetto, supponiamo di ricevere in ingresso indirizzi a 7 bit, ovvero valori da 0 a 127, e di disporre di 8 WZ da 4 indirizzi ciascuna.
Tramite la nostra codifica, l'indirizzo che forniremo in uscita sarà ad 8 bit:

WZ_BIT	INDIRIZZO
0	Addr (7 bit)
1	XXX XXXX (NUM WZ, OFFSET)

Esempio:

Indirizzi base Working Zone = [29; 43; 34; 4; 66; 48; 60; 102]

Indirizzi da codificare = [35; 70]

35: *Appartiene alla WZ con indirizzo base 34.*

WZ_bit = 1
Num WZ = 2 → 010 (binario naturale)
Offset = 1 → 0010 (one-hot)

35 → 1 010 0010

70: *Non appartiene a nessuna WZ.*

WZ_bit = 0
Addr = 70 → 1000110

70 → 0 1000110

Il modulo realizzato interagirà con una RAM simulata dal Testbench e formata da parole di 8 bit, posizionate come segue:

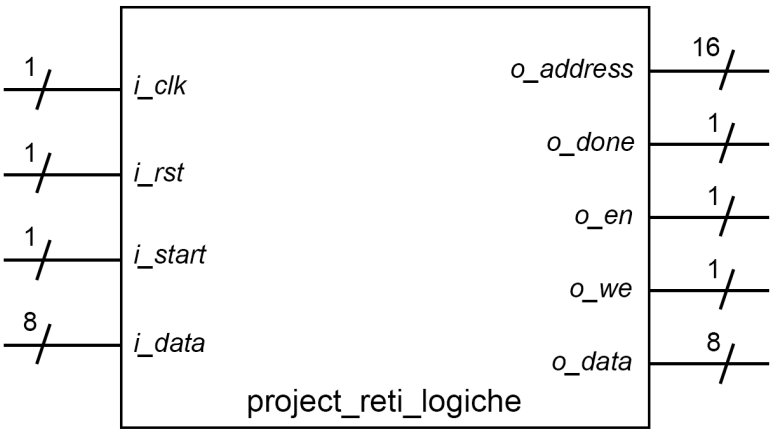
0
⋮ → indirizzi base di ogni Working Zone.
7
8 → indirizzo di lettura valore da codificare.
9 → indirizzo di scrittura valore codificato.

L'intefaccia del nostro componente è la seguente: (I = input; O = output)

I	*Clock*,	generato dal Testbench
I	*Start*,	segnale di partenza generato dal Testbench
I	*Reset*,	per riportare la macchina allo stato iniziale in attesa dello *start*
I	*DataInput*,	vettore proveniente dalla memoria in seguito a una richiesta di lettura
O	*Address,*	vettore per comunicare un indirizzo alla memoria
O	*Done,*	segnale per comunicare la disponibilità di un dato in output
O	*Enable,*	segnale per poter comunicare con la memoria
O	*WriteEnable,*	segnale per passare da lettura a scrittura su memoria
O	*DataOutput,*	vettore verso la memoria per la scrittura dati

Per la realizzazione di questo progetto faremo uso della piattaforma *Xilinx Vivado Webpack*, tramite la quale, mediante linguaggio VHDL, programmeremo la FPGA xc7a200tfbg484-1.

Questo software ci permette di simulare il comportamento di un componente descritto tramite VHDL, di sintetizzare tale componente (ove possibile), e di eseguire su di esso test per verificarne un funzionamento adeguato.



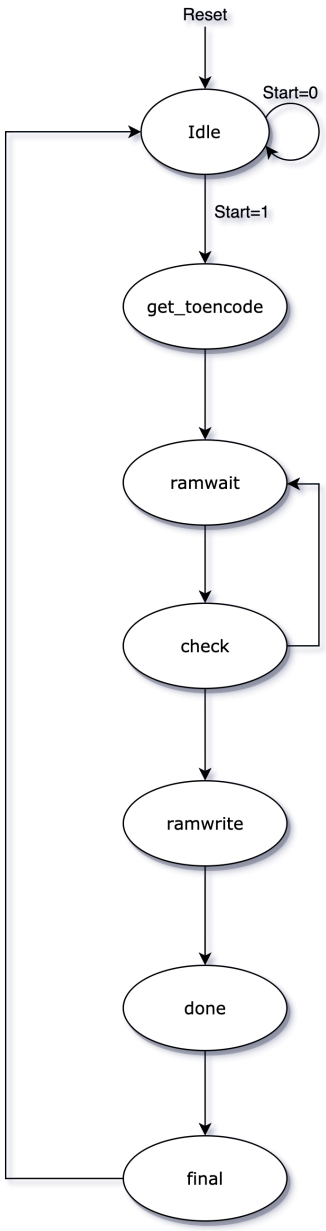
Architettura

Scelte Architetturali

Per la realizzazione erano possibili due strade implementative: salvare tutti gli indirizzi base delle WZ in registri da utilizzare successivamente per i confronti con l'indirizzo da codificare, oppure, come nell'implementazione da noi scelta, salvare in un registro il solo indirizzo da codificare e confrontarlo, ad ogni ciclo, con una WZ diversa.

Il primo approccio risulta più dispendioso in termini di numero di flip-flop in seguito al salvataggio di tutte le WZ, ma più efficiente nel caso di multi-start, ovvero quando si necessita di codificare più indirizzi senza variare le WZ (senza reset).
Il secondo approccio, invece, risulta vantaggioso in caso di un frequente aggiornamento delle Working Zone permettendo inoltre un risparmio di FF.

Per l'implementazione si è realizzata una macchina a stati che descrive il funzionamento logico del processo.



N.B.
In un qualsiasi istante la ricezione di un *reset* riporta la macchina allo stato di *idle* dopo un ripristino dei segnali

Descrizione degli stati

- **Idle**: Stato iniziale, nel quale si attende la ricezione di un segnale di *start*.
- **Get_toencode**: Stato nel quale si provvede al salvataggio nel segnale *toencode* del valore dell'indirizzo da codificare (*addr*) proveniente dalla cella 8 della RAM, e alla richiesta di lettura della prima WZ.
- **Ramwait**: Stato di attesa della disponibilità del dato richiesto in lettura. Si incrementa inoltre il valore del segnale relativo alla WZ attuale.
- **Check**: Stato di verifica di appartenenza di *addr* alla WZ presa in esame, e se necessario di abilitazione della scrittura su memoria (*o_we* = '1').
- **Ramwrite**: Dove si codifica l'indirizzo secondo le specifiche e si procede a scriverlo sulla memoria.
- **Done**: Si carica il valore '1' sul bit *done* per segnalare il termine della computazione, successivamente si mette a '0' il bit *o_we*.
- **Final**: Esegue un soft reset per ritornare allo stato di *idle*.

Lo stato più significativo del processo da noi realizzato è quello denominato *check*: al suo interno avvengono i confronti tra l'indirizzo da codificare e gli indirizzi appartenenti alla WZ attualmente in esame.
Rilevata l'appartenenza ad una Working Zone o terminate le WZ da analizzare, da questo stato si procede all'effettiva codifica dell'indirizzo in ingresso e alla sua scrittura in memoria.

Il process è anche sensibile al segnale *i_rst* che, se ricevuto, riporta i segnali in output ai seguenti valori di default:

o_done = '0'
o_en = '1'
o_we = '0'
o_data = '0000 0000'
o_addr = '0000 0000 0000 1000' (8)

e i segnali interni:

inwz = '1'
lettura = '0'
next_state = idle
toencode = '0'

Quelli sopra indicati sono segnali che abbiamo deciso di introdurre come ausilio alla progettazione: tengono traccia di parametri come il numero della WZ attualmente in analisi, l'indirizzo da codificare o quale sia lo stato successivo.

Risultati Sperimentali

Report di Sintesi

Dalla sintesi del componente possiamo estrarre le seguenti informazioni:

La codifica degli stati viene automaticamente effettuata in one-hot per velocizzare le transizioni.

State	Encoding
idle	0 0 0 0 0 0 1
get_toencode	0 0 0 0 0 1 0
ramwait	0 0 0 0 1 0 0
check	0 0 0 1 0 0 0
ramwrite	0 0 1 0 0 0 0
done	0 1 0 0 0 0 0
final	1 0 0 0 0 0 0

Dalla seguente tabella possiamo invece ricavare il numero di Flip Flop e LookUp Tables (LUT) necessari per la realizzazione del nostro componente.

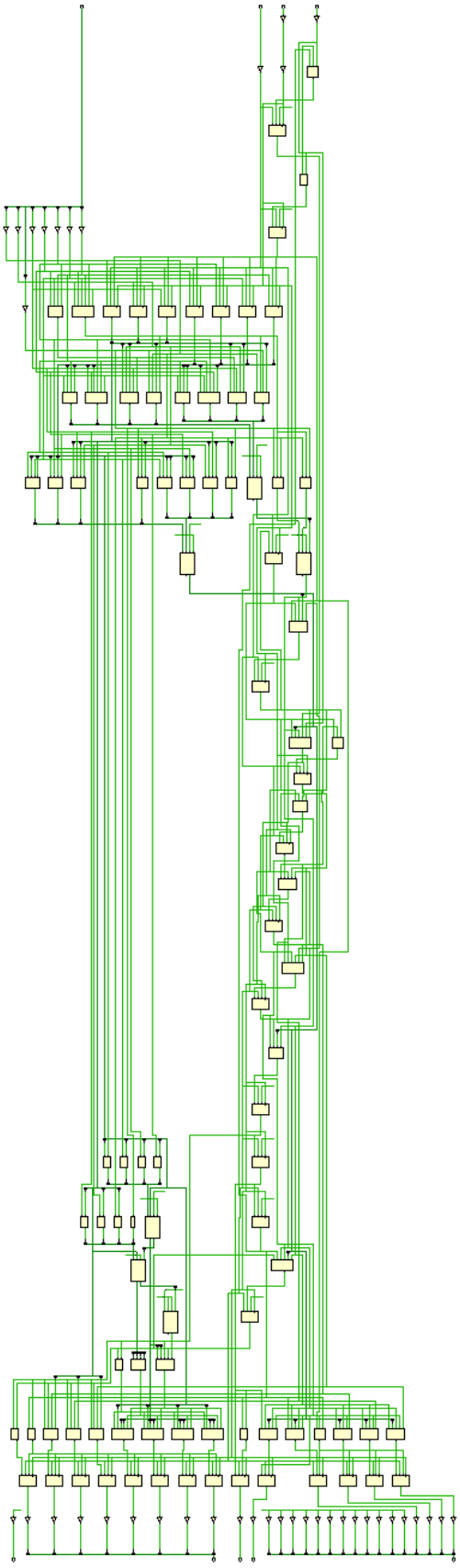
Site Type	Used	Fixed	Available	Util%
Slice LUTs	45	0	134600	0.03
LUT as Logic	45	0	134600	0.03
LUT as Memory	0	0	46200	0.00
Slice Registers	33	0	269200	0.01
Register as FF	33	0	269200	0.01
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Durante la sintesi viene generato il seguente warning:

> “ *inferring latch for variable 'o_en_reg'* ”.

Questo è dovuto alla scelta di mantenere il segnale *o_en* al valore ‘1’ durante tutta l’esecuzione del process. La nostra implementazione ci permette di risparmiare l’uso di un ulteriore registro in post-sintesi, a scapito di eventuali limitazioni degli accessi concorrenti alla RAM.

Schematic



In figura è riportato lo schematic generato in fase di sintesi.

Simulazioni

Per verificare il corretto funzionamento del nostro componente, abbiamo effettuato dei test su di esso.

Inizialmente tramite un generatore automatico di test, che grazie ad un gran numero di prove casuali ci ha permesso di avere una buona copertura sul funzionamento generale. In secondo luogo, per ottenere una coverage più completa, abbiamo pensato a dei casi di funzionamento specifici che potessero non essere correttamente gestiti. In particolare:

- Tutti i casi in cui le WZ avessero indirizzo base in 0 e 124
- Indirizzo da codificare in 0 e 127.

} Corner Cases

- Ricezione di segnali di *start* multipli durante l'esecuzione.
- Ricezione di segnali di *reset* durante l'esecuzione.

Per effettuare i primi due casi di test è stato sufficiente modificare i valori presenti nella RAM dei testbench di esempio forniti dal professore.

Per simulare i segnali di *start* multipli è stato invece necessario dichiarare due oggetti di tipo *ram_type* (RAM1 e RAM2) ed un contatore che, in seguito al primo segnale di *done* in uscita dal componente, sostituisse la prima RAM con la seconda, nella quale sono presenti le medesime WZ ma un diverso indirizzo da codificare. *(Fig.1)*

Per il testing dei *reset* si è ritenuto opportuno analizzare il comportamento del componente in caso di ricezione del segnale durante varie fasi dell'esecuzione, effettuando o meno un cambio degli indirizzi forniti dalla RAM. *(Fig. 2)*

```
if enable_wire = '1' then
    if mem_we = '1' then
        case counter is
            when 1 =>
                RAM1(conv_integer(mem_address))  <= mem_i_data;
                mem_o_data                      <= mem_i_data after 1 ns;
            when 2 =>
                RAM2(conv_integer(mem_address))  <= mem_i_data;
                mem_o_data                      <= mem_i_data after 1 ns;
            when others =>
                report "ERRORE";
        end case;
    else
        case counter is
            when 1 =>
                mem_o_data <= RAM1(conv_integer(mem_address)) after 1 ns;
            when 2 =>
                mem_o_data <= RAM2(conv_integer(mem_address)) after 1 ns;
            when others =>
                report "ERRORE";
        end case;
    end if;
end if;
```

Figura 1: Costrutto case per scambiare RAM1 e RAM2 durante l'esecuzione

```
wait for 100 ns;
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 mem_we ='1';
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 until tb_done = '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '0';
wait for 100 ns;
```

Figura 2: Reset dopo che il componente ha portato ad '1' il segnale o_we

Conclusioni

Ricapitolando, la linea evolutiva da noi seguita per la realizzazione del progetto è stata la seguente:

- Analisi accurata delle specifiche fornite, con chiarimenti rispetto ad eventuali dubbi inerenti alle stesse.
- Acquisizione di una dimestichezza di base relativa all'ambiente di sviluppo.
- Ideazione di uno schema della macchina a stati rappresentativa del processo logico.
- Traduzione da macchina a stati a linguaggio VHDL.
- Testing e simulazioni sul componente, prima in behavioral e successivamente in post-sintesi.

Tutto ciò ci ha portato alla realizzazione di un componente funzionante ed in linea con le specifiche fornite.

Tramite lo svolgimento di questo progetto abbiamo quindi potuto acquisire esperienza in campo pratico, interfacciandoci con il mondo della progettazione di componenti Hardware.

Lorini Giorgio e Malmusi Davide